Boost.Intrusive

Olaf Krzikalla

Ion Gaztanaga

Copyright © 2005 Olaf Krzikalla
Copyright © 2006-2012 Ion Gaztanaga

Distributed under the Boost Software License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)

Table of Contents

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Introduction

Presenting Boost.Intrusive

Boost.Intrusive is a library presenting some intrusive containers to the world of C++. Intrusive containers are special containers that offer better performance and exception safety guarantees than non-intrusive containers (like STL containers).

The performance benefits of intrusive containers makes them ideal as a building block to efficiently construct complex containers like multi-index containers or to design high performance code like memory allocation algorithms.

While intrusive containers were and are widely used in C, they became more and more forgotten in C++ due to the presence of the standard containers which don't support intrusive techniques.**Boost.Intrusive** wants to push intrusive containers usage encapsulating the implementation in STL-like interfaces. Hence anyone familiar with standard containers can easily use **Boost.Intrusive**.

Building Boost.Intrusive

There is no need to compile anything to use **Boost.Intrusive**, since it's a header only library. Just include your Boost header directory in your compiler include path.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Intrusive and non-intrusive containers

Differences between intrusive and non-intrusive containers

The main difference between intrusive containers and non-intrusive containers is that in C++ non-intrusive containers store **copies** of values passed by the user. Containers use the `Allocator` template parameter to allocate the stored values:

```
#include <list>
#include <assert.h>

int main()
{
   std::list<MyClass> myclass_list;

   MyClass myclass(...);
   myclass_list.push_back(myclass);

   //The stored object is different from the original object
   assert(&myclass != &myclass_list.front());
   return 0;
}
```

To store the newly allocated copy of `myclass`, the container needs additional data: `std::list` usually allocates nodes that contain pointers to the next and previous node and the value itself. Something similar to:

```
//A possible implementation of a std::list<MyClass> node
class list_node
{
   list_node *next;
   list_node *previous;
   MyClass    value;
};
```

On the other hand, an intrusive container does not store copies of passed objects, but it stores the objects themselves. The additional data needed to insert the object in the container must be provided by the object itself. For example, to insert `MyClass` in an intrusive container that implements a linked list, `MyClass` must contain the needed *next* and *previous* pointers:

```
class MyClass
{
   MyClass *next;
   MyClass *previous;
   //Other members...
};

int main()
{
   acme_intrusive_list<MyClass> list;

   MyClass myclass;
   list.push_back(myclass);

   //"myclass" object is stored in the list
   assert(&myclass == &list.front());
   return 0;
}
```

As we can see, knowing which additional data the class should contain is not an easy task. **Boost.Intrusive** offers several intrusive containers and an easy way to make user classes compatible with those containers.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Properties of Boost.Intrusive containers

Semantically, a **Boost.Intrusive** container is similar to a STL container holding pointers to objects. That is, if you have an intrusive list holding objects of type `T`, then `std::list<T*>` would allow you to do quite the same operations (maintaining and navigating a set of objects of type T and types derived from it).

A non-intrusive container has some limitations:

- An object can only belong to one container: If you want to share an object between two containers, you either have to store multiple copies of those objects or you need to use containers of pointers: `std::list<Object*>`.
- The use of dynamic allocation to create copies of passed values can be a performance and size bottleneck in some applications. Normally, dynamic allocation imposes a size overhead for each allocation to store bookkeeping information and a synchronization to protected concurrent allocation from different threads.
- Only copies of objects are stored in non-intrusive containers. Hence copy or move constructors and copy or move assignment operators are required. Non-copyable and non-movable objects can't be stored in non-intrusive containers.
- It's not possible to store a derived object in a STL-container while retaining its original type.

Intrusive containers have some important advantages:

- Operating with intrusive containers doesn't invoke any memory management at all. The time and size overhead associated with dynamic memory can be minimized.
- Iterating an Intrusive container needs less memory accesses than the semantically equivalent container of pointers: iteration is faster.
- Intrusive containers offer better exception guarantees than non-intrusive containers. In some situations intrusive containers offer a no-throw guarantee that can't be achieved with non-intrusive containers.
- The computation of an iterator to an element from a pointer or reference to that element is a constant time operation (computing the position of `T*` in a `std::list<T*>` has linear complexity).
- Intrusive containers offer predictability when inserting and erasing objects since no memory management is done with intrusive containers. Memory management usually is not a predictable operation so complexity guarantees from non-intrusive containers are looser than the guarantees offered by intrusive containers.

Intrusive containers have also downsides:

- Each type stored in an intrusive container needs additional memory holding the maintenance information needed by the container. Hence, whenever a certain type will be stored in an intrusive container **you have to change the definition of that type** appropriately. Although this task is easy with **Boost.Intrusive**, touching the definition of a type is sometimes a crucial issue.
- In intrusive containers you don't store a copy of an object, **but rather the original object is linked with other objects in the container**. Objects don't need copy-constructors or assignment operators to be stored in intrusive containers. But you have to take care of possible side effects, whenever you change the contents of an object (this is especially important for associative containers).
- The user **has to manage the lifetime of inserted objects** independently from the containers.
- Again you have to be **careful**: in contrast to STL containers **it's easy to render an iterator invalid** without touching the intrusive container directly, because the object can be disposed before is erased from the container.
- **Boost.Intrusive** containers are **non-copyable and non-assignable**. Since intrusive containers don't have allocation capabilities, these operations make no sense. However, swapping can be used to implement move capabilities. To ease the implementation of copy constructors and assignment operators of classes storing **Boost.Intrusive** containers, **Boost.Intrusive** offers special cloning functions. See Cloning Boost.Intrusive containers section for more information.
- Analyzing the thread safety of a program that uses containers is harder with intrusive containers, because the container might be modified indirectly without an explicit call to a container member.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Table 1. Summary of intrusive containers advantages and disadvantages

Issue	Intrusive	Non-intrusive
Memory management	External	Internal through allocator
Insertion/Erasure time	Faster	Slower
Memory locality	Better	Worse
Can hold non-copyable and non-movable objects by value	Yes	No
Exception guarantees	Better	Worse
Computation of iterator from value	Constant	Non-constant
Insertion/erasure predictability	High	Low
Memory use	Minimal	More than minimal
Insert objects by value retaining polymorphic behavior	Yes	No (slicing)
User must modify the definition of the values to insert	Yes	No
Containers are copyable	No	Yes
Inserted object's lifetime managed by	User (more complex)	Container (less complex)
Container invariants can be broken without using the container	Easier	Harder (only with containers of pointers)
Thread-safety analysis	Harder	Easier

For a performance comparison between Intrusive and Non-intrusive containers see Performance section.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

How to use Boost.Intrusive

If you plan to insert a class in an intrusive container, you have to make some decisions influencing the class definition itself. Each class that will be used in an intrusive container needs some appropriate data members storing the information needed by the container. We will take a simple intrusive container, the intrusive list (`boost::intrusive::list`), for the following examples, but all **Boost.Intrusive** containers are very similar. To compile the example using `boost::intrusive::list`, just include:

```
#include <boost/intrusive/list.hpp>
```

Every class to be inserted in an intrusive container, needs to contain a hook that will offer the necessary data and resources to be insertable in the container. With **Boost.Intrusive** you just choose the hook to be a public base class or a public member of the class to be inserted. **Boost.Intrusive** also offers more flexible hooks for advanced users, as explained in the chapter Using function hooks, but usually base or member hooks are good enough for most users.

Using base hooks

For `list`, you can publicly derive from `list_base_hook`.

```
template <class ...Options>
class list_base_hook;
```

The class can take several options. **Boost.Intrusive** classes receive arguments in the form `option_name<option_value>`. You can specify the following options:

- **`tag<class Tag>`**: this argument serves as a tag, so you can derive from more than one `list_base_hook` and hence put an object in multiple intrusive lists at the same time. An incomplete type can serve as a tag. If you specify two base hooks, you **must** specify a different tag for each one. Example: `list_base_hook< tag<tag1> >`. If no tag is specified a default one will be used (more on default tags later).

- **`link_mode<link_mode_type LinkMode>`**: The second template argument controls the linking policy. **Boost.Intrusive** currently supports 3 modes: `normal_link`, `safe_link` and `auto_unlink`. By default, `safe_link` mode is used. More about these in sections Safe hooks and Auto-unlink hooks. Example: `list_base_hook< link_mode<auto_unlink> >`

- **`void_pointer<class VoidPointer>`**: this option is the pointer type to be used internally in the hook. The default value is `void *`, which means that raw pointers will be used in the hook. More about this in the section titled Using smart pointers with Boost.Intrusive containers. Example: `list_base_hook< void_pointer< my_smart_ptr<void> >`

For the following examples, let's forget the options and use the default values:

```
#include <boost/intrusive/list.hpp>

using namespace boost::intrusive;

class Foo
   //Base hook with default tag, raw pointers and safe_link mode
   :  public list_base_hook<>
{ /**/ };
```

After that, we can define the intrusive list:

```
template <class T, class ...Options>
class list;
```

`list` receives the type to be inserted in the container (`T`) as the first parameter and optionally, the user can specify options. We have 3 option types:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

- **`base_hook<class Hook>`** / **`member_hook<class T, class Hook, Hook T::* PtrToMember>`** / **`value_traits<class ValueTraits>`**: All these options specify the relationship between the type `T` to be inserted in the list and the hook (since we can have several hooks in the same `T` type). `member_hook` will be explained a bit later and `value_traits` will be explained in the Containers with custom ValueTraits section. **If no option is specified, the container will be configured to use the base hook with the default tag**. Some options configured for the hook (the type of the pointers, link mode, etc.) will be propagated to the container.

- **`constant_time_size<bool Enabled>`**: Specifies if a constant time `size()` function is demanded for the container. This will instruct the intrusive container to store an additional member to keep track of the current size of the container. By default, constant-time size is activated.

- **`size_type<class SizeType>`**: Specifies an unsigned type that can hold the size of the container. This type will be the type returned by `list.size()` and the type stored in the intrusive container if `constant_time_size<true>` is requested. The user normally will not need to change this type, but some containers can have a `size_type` that might be different from `std::size_t` (for example, STL-like containers use the `size_type` defined by their allocator). **Boost.Intrusive** can be used to implement such containers specifying the the type of the size. By default the type is `std::size_t`.

Example of a constant-time size intrusive list that will store Foo objects, using the base hook with the default tag:

```
typedef list<Foo> FooList;
```

Example of an intrusive list with non constant-time size that will store Foo objects:

```
typedef list<Foo, constant_time_size<false> > FooList;
```

Remember that the user must specify the base hook in the container declaration if the base hook has no default tag, because that usually means that the type has more than one base hook, and a container shall know which hook will be using:

```
#include <boost/intrusive/list.hpp>

using namespace boost::intrusive;

struct my_tag1;
struct my_tag2;

typedef list_base_hook< tag<my_tag>  > BaseHook;
typedef list_base_hook< tag<my_tag2> > BaseHook2;
class Foo   :  public BaseHook, public BaseHook2
{ /**/ };

typedef list< Foo, base_hook<BaseHook>  > FooList;
typedef list< Foo, base_hook<BaseHook2> > FooList2;
```

Once the list is defined, we can use it:

```
//An object to be inserted in the list
Foo foo_object;
FooList list;

list.push_back(object);

assert(&list.front() == &foo_object);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Using member hooks

Sometimes an 'is-a' relationship between list hooks and the list value types is not desirable. In this case, using a member hook as a data member instead of 'disturbing' the hierarchy might be the right way: you can add a public data member `list_member_hook<...>` to your class. This class can be configured with the same options as `list_base_hook` except the option `tag`:

```
template <class ...Options>
class list_member_hook;
```

Example:

```
#include <boost/intrusive/list.hpp>

class Foo
{
   public:
   list_member_hook<> hook_;
   //...
};
```

When member hooks are used, the `member_hook` option is used to configure the list:

```
//This option will configure "list" to use the member hook
typedef member_hook<Foo, list_member_hook<>, &Foo::hook_> MemberHookOption;

//This list will use the member hook
typedef list<Foo, MemberHookOption> FooList;
```

Now we can use the container:

```
//An object to be inserted in the list
Foo foo_object;
FooList list;

list.push_back(object);

assert(&list.front() == &foo_object);
```

Using both hooks

You can insert the same object in several intrusive containers at the same time, using one hook per container. This is a full example using base and member hooks:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
#include <boost/intrusive/list.hpp>
#include <vector>

using namespace boost::intrusive;

class MyClass : public list_base_hook<>
{
   int int_;

   public:
   list_member_hook<> member_hook_;

   MyClass(int i) :  int_(i)  {}
};

//Define a list that will store MyClass using the base hook
typedef list<MyClass> BaseList;

//Define a list that will store MyClass using the member hook
typedef member_hook
   < MyClass, list_member_hook<>, &MyClass::member_hook_> MemberOption;
typedef list<MyClass, MemberOption> MemberList;

int main()
{
   typedef std::vector<MyClass>::iterator VectIt;
   typedef std::vector<MyClass>::reverse_iterator VectRit;

   //Create several MyClass objects, each one with a different value
   std::vector<MyClass> values;
   for(int i = 0; i < 100; ++i)  values.push_back(MyClass(i));

   BaseList baselist;
   MemberList memberlist;

   //Now insert them in the reverse order in the base hook list
   for(VectIt it(values.begin()), itend(values.end())
      ; it != itend  ; ++it){
      baselist.push_front(*it);
   }

   //Now insert them in the same order as in vector in the member hook list
   for(VectIt it(values.begin()), itend(values.end()); it != itend; ++it)
      memberlist.push_back(*it);

   //Now test lists
   {
      BaseList::reverse_iterator rbit(baselist.rbegin());
      MemberList::iterator mit(memberlist.begin());
      VectIt  it(values.begin()), itend(values.end());

      //Test the objects inserted in the base hook list
      for(; it != itend; ++it, ++rbit)
         if(&*rbit != &*it)   return 1;

      //Test the objects inserted in the member hook list
      for(it = values.begin(); it != itend; ++it, ++mit)
         if(&*mit != &*it)    return 1;
   }

   return 0;
}
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Object lifetime

Even if the interface of `list` is similar to `std::list`, its usage is a bit different: You always have to keep in mind that you directly store objects in intrusive containers, not copies. The lifetime of a stored object is not bound to or managed by the container:

- When the container gets destroyed before the object, the object is not destroyed, so you have to be careful to avoid resource leaks.
- When the object is destroyed before the container, your program is likely to crash, because the container contains a pointer to an non-existing object.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

When to use?

Intrusive containers can be used for highly optimized algorithms, where speed is a crucial issue and:

- additional memory management should be avoided.
- the programmer needs to efficiently track the construction and destruction of objects.
- exception safety, especially the no-throw guarantee, is needed.
- the computation of an iterator to an element from a pointer or reference to that element should be a constant time operation.
- it's important to achieve a well-known worst-time system response.
- localization of data (e.g. for cache hit optimization) leads to measurable effects.

The last point is important if you have a lot of containers over a set of elements. E.g. if you have a vector of objects (say, `std::vector<Object>`), and you also have a list storing a subset of those objects (`std::list<Object*>`), then operating on an Object from the list iterator (`std::list<Object*>::iterator`) requires two steps:

- Access from the iterator (usually on the stack) to the list node storing a pointer to `Object`.
- Access from the pointer to `Object` to the Object stored in the vector.

While the objects themselves are tightly packed in the memory of the vector (a vector's memory is guaranteed to be contiguous), and form something like a data block, list nodes may be dispersed in the heap memory. Hence depending on your system you might get a lot of cache misses. The same doesn't hold for an intrusive list. Indeed, dereferencing an iterator from an intrusive list is performed in the same two steps as described above. But the list node is already embedded in the Object, so the memory is directly tracked from the iterator to the Object.

It's also possible to use intrusive containers when the objects to be stored can have different or unknown size. This allows storing base and derived objects in the same container, as shown in the following example:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
#include <boost/intrusive/list.hpp>

using namespace boost::intrusive;

//An abstract class that can be inserted in an intrusive list
class Window : public list_base_hook<>
{
   public:
   //This is a container those value is an abstract class: you can't do this with std::list.
   typedef list<Window> win_list;

   //A static intrusive list declaration
   static win_list all_windows;

   //Constructor. Includes this window in the list
   Window()             {  all_windows.push_back(*this);  }
   //Destructor. Removes this node from the list
   virtual ~Window()    {  all_windows.erase(win_list::s_iterator_to(*this));  }
   //Pure virtual function to be implemented by derived classes
   virtual void Paint() = 0;
};

//The static intrusive list declaration
Window::win_list Window::all_windows;

//Some Window derived classes
class FrameWindow :  public Window
{  void Paint(){/**/} };

class EditWindow :  public Window
{  void Paint(){/**/} };

class CanvasWindow :  public Window
{  void Paint(){/**/} };

//A function that prints all windows stored in the intrusive list
void paint_all_windows()
{
   for(Window::win_list::iterator i(Window::all_windows.begin())
                                , e(Window::all_windows.end())
       ; i != e; ++i)
       i->Paint();
}

//...

//A class derived from Window
class MainWindow  :  public Window
{
   FrameWindow   frame_;  //these are derived from Window too
   EditWindow    edit_;
   CanvasWindow  canvas_;

   public:
   void Paint(){/**/}
   //...
};

//Main function
int main()
{
   //When a Window class is created, is automatically registered in the global list
   MainWindow window;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
   //Paint all the windows, sub-windows and so on
   paint_all_windows();

   //All the windows are automatically unregistered in their destructors.
   return 0;
}
```

Due to certain properties of intrusive containers they are often more difficult to use than their STL-counterparts. That's why you should avoid them in public interfaces of libraries. Classes to be stored in intrusive containers must change their implementation to store the hook and this is not always possible or desirable.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Concept summary

Here is a small summary of the basic concepts that will be used in the following chapters:

Brief Concepts Summary

Node Algorithms	A class containing typedefs and static functions that define basic operations that can be applied to a group of `nodes`. It's independent from the node definition and configured using a NodeTraits template parameter that describes the node.
Node Traits	A class that stores basic information and operations to insert a node into a group of nodes.
Hook	A class that a user must add as a base class or as a member to make the user class compatible with intrusive containers. A Hook encapsulates a `node`
Intrusive Container	A class that stores user classes that have the needed hooks. It takes a ValueTraits template parameter as configuration information.
Semi-Intrusive Container	Similar to an intrusive container but a semi-intrusive container needs additional memory (e.g. an auxiliary array) to work.
Value Traits	A class containing typedefs and operations to obtain the node to be used by Node Algorithms from the user class and the inverse.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Presenting Boost.Intrusive containers

Boost.Intrusive offers a wide range of intrusive containers:

- **slist**: An intrusive singly linked list. The size overhead is very small for user classes (usually the size of one pointer) but many operations have linear time complexity, so the user must be careful if he wants to avoid performance problems.
- **list**: A `std::list` like intrusive linked list. The size overhead is quite small for user classes (usually the size of two pointers). Many operations have constant time complexity.
- **set/multiset/rbtree**: `std::set/std::multiset` like intrusive associative containers based on red-black trees. The size overhead is moderate for user classes (usually the size of three pointers). Many operations have logarithmic time complexity.
- **avl_set/avl_multiset/avltree**: A `std::set/std::multiset` like intrusive associative containers based on AVL trees. The size overhead is moderate for user classes (usually the size of three pointers). Many operations have logarithmic time complexity.
- **splay_set/splay_multiset/splaytree**: `std::set/std::multiset` like intrusive associative containers based on splay trees. Splay trees have no constant operations, but they have some interesting caching properties. The size overhead is moderate for user classes (usually the size of three pointers). Many operations have logarithmic time complexity.
- **sg_set/sg_multiset/sgtree**: A `std::set/std::multiset` like intrusive associative containers based on scapegoat trees. Scapegoat can be configured with the desired balance factor to achieve the desired rebalancing frequency/search time compromise. The size overhead is moderate for user classes (usually the size of three pointers). Many operations have logarithmic time complexity.

Boost.Intrusive also offers semi-intrusive containers:

- **unordered_set/unordered_multiset**: `std::tr1::unordered_set/std::tr1::unordered_multiset` like intrusive unordered associative containers. The size overhead is moderate for user classes (an average of two pointers per element). Many operations have amortized constant time complexity.

Most of these intrusive containers can be configured with constant or linear time size:

- **Linear time size**: The intrusive container doesn't hold a size member that is updated with every insertion/erasure. This implies that the `size()` function doesn't have constant time complexity. On the other hand, the container is smaller, and some operations, like `splice()` taking a range of iterators in linked lists, have constant time complexity instead of linear complexity.
- **Constant time size**: The intrusive container holds a size member that is updated with every insertion/erasure. This implies that the `size()` function has constant time complexity. On the other hand, increases the size of the container, and some operations, like `splice()` taking a range of iterators, have linear time complexity in linked lists.

To make user classes compatible with these intrusive containers **Boost.Intrusive** offers two types of hooks for each container type:

- **Base hook**: The hook is stored as a public base class of the user class.
- **Member hook**: The hook is stored as a public member of the user class.

Apart from that, **Boost.Intrusive** offers additional features:

- **Safe mode hooks**: Hook constructor initializes the internal `node` to a well-known safe state and intrusive containers check that state before inserting a value in the container using that hook. When erasing an element from the container, the container puts the `node` of the hook in the safe state again. This allows a safer use mode and it can be used to detect programming errors. It implies a slight performance overhead in some operations and can convert some constant time operations to linear time operations.
- **Auto-unlink hooks**: The hook destructor removes the object from the container automatically and the user can safely unlink the object from the container without referring to the container.
- **Non-raw pointers**: If the user wants to use smart pointers instead of raw pointers, **Boost.Intrusive** hooks can be configured to use any type of pointer. This configuration information is also transmitted to the containers, so all the internal pointers used by intrusive containers configured with these hooks will be smart pointers. As an example, **Boost.Interprocess** defines a smart

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

pointer compatible with shared memory, called `offset_ptr`. **Boost.Intrusive** can be configured to use this smart pointer to allow shared memory intrusive containers.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Safe hooks

Features of the safe mode

Boost.Intrusive hooks can be configured to operate in safe-link mode. The safe mode is activated by default, but it can be also explicitly activated:

```
//Configuring the safe mode explicitly
class Foo : public list_base_hook< link_mode<safe_link> >
{};
```

With the safe mode the user can detect if the object is actually inserted in a container without any external reference. Let's review the basic features of the safe mode:

- Hook's constructor puts the hook in a well-known default state.
- Hook's destructor checks if the hook is in the well-known default state. If not, an assertion is raised.
- Every time an object is inserted in the intrusive container, the container checks if the hook is in the well-known default state. If not, an assertion is raised.
- Every time an object is being erased from the intrusive container, the container puts the erased object in the well-known default state.

With these features, without any external reference the user can know if the object has been inserted in a container by calling the `is_linked()` member function. If the object is not actually inserted in a container, the hook is in the default state, and if it is inserted in a container, the hook is not in the default state.

Configuring safe-mode assertions

By default, all safe-mode assertions raised by **Boost-Intrusive** hooks and containers in are implemented using `BOOST_ASSERT`, which can be configured by the user. See http://www.boost.org/libs/utility/assert.html for more information about `BOOST_ASSERT`.

`BOOST_ASSERT` is globally configured, so the user might want to redefine intrusive safe-mode assertions without modifying the global `BOOST_ASSERT`. This can be achieved redefining the following macros:

- `BOOST_INTRUSIVE_SAFE_HOOK_DEFAULT_ASSERT`: This assertion will be used in insertion functions of the intrusive containers to check that the hook of the value to be inserted is default constructed.
- `BOOST_INTRUSIVE_SAFE_HOOK_DESTRUCTOR_ASSERT`: This assertion will be used in hooks' destructors to check that the hook is in a default state.

If any of these macros is not redefined, the assertion will default to `BOOST_ASSERT`. If `BOOST_INTRUSIVE_SAFE_HOOK_DEFAULT_ASSERT` or `BOOST_INTRUSIVE_SAFE_HOOK_DESTRUCTOR_ASSERT` is defined and the programmer needs to include a file to configure that assertion, it can define `BOOST_INTRUSIVE_SAFE_HOOK_DESTRUCTOR_ASSERT_INCLUDE` or `BOOST_INTRUSIVE_SAFE_HOOK_DEFAULT_ASSERT_INCLUDE` with the name of the file to include:

```
#define BOOST_INTRUSIVE_SAFE_HOOK_DESTRUCTOR_ASSERT          MYASSERT
#define BOOST_INTRUSIVE_SAFE_HOOK_DESTRUCTOR_ASSERT_INCLUDE <myassert.h>
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Auto-unlink hooks

What's an auto-unlink hook?

Boost.Intrusive offers additional hooks with unique features:

- When the destructor of the hook is called, the hook checks if the node is inserted in a container. If so, the hook removes the node from the container.
- The hook has a member function called `unlink()` that can be used to unlink the node from the container at any time, without having any reference to the container, if the user wants to do so.

These hooks have exactly the same size overhead as their analog non auto-unlinking hooks, but they have a restriction: they can only be used with non-constant time containers. There is a reason for this:

- Auto-unlink hooks don't store any reference to the container where they are inserted.
- Only containers with non constant-time `size()` allow removing an object from the container without referring to the container.

This auto-unlink feature is useful in certain applications but it must be used **very carefully**:

- If several threads are using the same container the destructor of the auto-unlink hook will be called without any thread synchronization so removing the object is thread-unsafe.
- Container contents change silently without modifying the container directly. This can lead to surprising effects.

These auto-unlink hooks have also safe-mode properties:

- Hooks' constructors put the hook in a well-known default state.
- Every time an object is inserted in the intrusive container, the container checks if the hook is in the well-known default state. If not, an assertion is raised.
- Every time an object is erased from an intrusive container, the container puts the erased object in the well-known default state.

Auto-unlink hook example

Let's see an example of an auto-unlink hook:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
#include <boost/intrusive/list.hpp>
#include <cassert>

using namespace boost::intrusive;

typedef list_base_hook<link_mode<auto_unlink> > auto_unlink_hook;

class MyClass : public auto_unlink_hook
               //This hook removes the node in the destructor
{
   int int_;

   public:
   MyClass(int i = 0)   :  int_(i)  {}
   void unlink()     {  auto_unlink_hook::unlink();  }
   bool is_linked()  {  return auto_unlink_hook::is_linked();  }
};

//Define a list that will store values using the base hook
//The list can't have constant-time size!
typedef list< MyClass, constant_time_size<false> > List;

int main()
{
   //Create the list
   List l;
   {
      //Create myclass and check it's linked
      MyClass myclass;
      assert(myclass.is_linked() == false);

      //Insert the object
      l.push_back(myclass);

      //Check that we have inserted the object
      assert(l.empty() == false);
      assert(&l.front() == &myclass);
      assert(myclass.is_linked() == true);

      //Now myclass' destructor will unlink it
      //automatically
   }

   //Check auto-unlink has been executed
   assert(l.empty() == true);

   {
      //Now test the unlink() function

      //Create myclass and check it's linked
      MyClass myclass;
      assert(myclass.is_linked() == false);

      //Insert the object
      l.push_back(myclass);

      //Check that we have inserted the object
      assert(l.empty() == false);
      assert(&l.front() == &myclass);
      assert(myclass.is_linked() == true);

      //Now unlink the node
      myclass.unlink();
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
      //Check auto-unlink has been executed
      assert(l.empty() == true);
   }
   return 0;
}
```

Auto-unlink hooks and containers with constant-time `size()`

As explained, **Boost.Intrusive** auto-unlink hooks are incompatible with containers that have constant-time `size()`, so if you try to define such container with an auto-unlink hook's value_traits, you will get a static assertion:

```
#include <boost/intrusive/list.hpp>

using boost::intrusive;

struct MyTag;

class MyClass : public list_base_hook< link_mode<auto_unlink> >
{/**/};

list <MyClass, constant_time_size<true> > bad_list;

int main()
{
   bad_list list;
   return 0;
}
```

leads to an error similar to:

```
error : use of undefined type 'boost::STATIC_ASSERTION_FAILURE<false>'
```

Pointing to code like this:

```
//Constant-time size is incompatible with auto-unlink hooks!
BOOST_STATIC_ASSERT(!(constant_time_size && ((int)value_traits::link_mode == (int)auto_unlink)));
```

This way, there is no way to compile a program if you try to use auto-unlink hooks in constant-time size containers.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Intrusive singly linked list: slist

`slist` is the simplest intrusive container of **Boost.Intrusive**: a singly linked list. The memory overhead it imposes is 1 pointer per node. The size of an empty, non constant-time size `slist` is the size of 1 pointer. This lightweight memory overhead comes with drawbacks, though: many operations have linear time complexity, even some that usually are constant time, like `swap`. `slist` only provides forward iterators.

For most cases, a doubly linked list is preferable because it offers more constant-time functions with a slightly bigger size overhead. However, for some applications like constructing more elaborate containers, singly linked lists are essential because of their low size overhead.

slist hooks

Like the rest of **Boost.Intrusive** containers, `slist` has two hook types:

```
template <class ...Options>
class slist_base_hook;
```

- `slist_base_hook`: the user class derives publicly from `slist_base_hook` to make it `slist`-compatible.

```
template <class ...Options>
class slist_member_hook;
```

- `slist_member_hook`: the user class contains a public `slist_member_hook` to make it `slist`-compatible.

`slist_base_hook` and `slist_member_hook` receive the same options explained in the section How to use Boost.Intrusive:

- **`tag<class Tag>`** (for base hooks only): This argument serves as a tag, so you can derive from more than one slist hook. Default: `tag<default_tag>`.
- **`link_mode<link_mode_type LinkMode>`**: The linking policy. Default: `link_mode<safe_link>`.
- **`void_pointer<class VoidPointer>`**: The pointer type to be used internally in the hook and propagated to the container. Default: `void_pointer<void*>`.

slist container

```
template <class T, class ...Options>
class slist;
```

`slist` receives the options explained in the section How to use Boost.Intrusive:

- **`base_hook<class Hook>`** / **`member_hook<class T, class Hook, Hook T::* PtrToMember>`** / **`value_traits<class ValueTraits>`**: To specify the hook type or value traits used to configure the container. (To learn about value traits go to the section Containers with custom ValueTraits.)
- **`constant_time_size<bool Enabled>`**: To activate the constant-time `size()` operation. Default: `constant_time_size<true>`
- **`size_type<bool Enabled>`**: To specify the type that will be used to store the size of the container. Default: `size_type<std::size_t>`.

`slist` can receive additional options:

- **`linear<bool Enable>`**: the singly linked list is implemented as a null-terminated list instead of a circular list. This allows `O(1)` swap, but losses some operations like `container_from_end_iterator`.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

- **`cache_last<bool Enable>`**: the singly linked also stores a pointer to the last element of the singly linked list. This allows `O(1)` swap, `splice_after(iterator, slist &)` and makes the list offer new functions like `push_back(reference)` and `back()`. Logically, the size an empty list is increased in `sizeof(void_pointer)` and the the cached last node pointer must be updated in every operation, and that might incur in a slight performance impact.

`auto_unlink` hooks are not usable if `linear<true>` and/or `cache_last<true>` options are used. If `auto_unlink` hooks are used and those options are specified, a static assertion will be raised.

Example

Now let's see a small example using both hooks:

```
#include <boost/intrusive/slist.hpp>
#include <vector>

using namespace boost::intrusive;

                  //This is a base hook
class MyClass : public slist_base_hook<>
{
   int int_;

   public:
   //This is a member hook
   slist_member_hook<> member_hook_;

   MyClass(int i)
      :  int_(i)
   {}
};

//Define an slist that will store MyClass using the public base hook
typedef slist<MyClass> BaseList;

//Define an slist that will store MyClass using the public member hook
typedef member_hook<MyClass, slist_member_hook<>, &MyClass::member_hook_> MemberOption;
typedef slist<MyClass, MemberOption> MemberList;

int main()
{
   typedef std::vector<MyClass>::iterator VectIt;
   typedef std::vector<MyClass>::reverse_iterator VectRit;

   //Create several MyClass objects, each one with a different value
   std::vector<MyClass> values;
   for(int i = 0; i < 100; ++i)  values.push_back(MyClass(i));

   BaseList baselist;
   MemberList memberlist;

   //Now insert them in the reverse order in the base hook list
   for(VectIt it(values.begin()), itend(values.end()); it != itend; ++it)
      baselist.push_front(*it);

   //Now insert them in the same order as in vector in the member hook list
   for(BaseList::iterator it(baselist.begin()), itend(baselist.end())
      ; it != itend; ++it){
      memberlist.push_front(*it);
   }

   //Now test lists
   {
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
      BaseList::iterator bit(baselist.begin());
      MemberList::iterator mit(memberlist.begin());
      VectRit rit(values.rbegin()), ritend(values.rend());
      VectIt  it(values.begin()), itend(values.end());

      //Test the objects inserted in the base hook list
      for(; rit != ritend; ++rit, ++bit)
         if(&*bit != &*rit)   return 1;

      //Test the objects inserted in the member hook list
      for(; it != itend; ++it, ++mit)
         if(&*mit != &*it)    return 1;
   }

   return 0;
}
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Intrusive doubly linked list: list

`list` is a doubly linked list. The memory overhead it imposes is 2 pointers per node. An empty, non constant-time size `list` also has the size of 2 pointers. `list` has many more constant-time operations than `slist` and provides a bidirectional iterator. It is recommended to use `list` instead of `slist` if the size overhead is acceptable:

list hooks

Like the rest of **Boost.Intrusive** containers, `list` has two hook types:

```
template <class ...Options>
class list_base_hook;
```

- `list_base_hook`: the user class derives publicly from `list_base_hook` to make it `list`-compatible.

```
template <class ...Options>
class list_member_hook;
```

- `list_member_hook`: the user class contains a public `list_member_hook` to make it `list`-compatible.

`list_base_hook` and `list_member_hook` receive the same options explained in the section How to use Boost.Intrusive:

- **`tag<class Tag>`** (for base hooks only): This argument serves as a tag, so you can derive from more than one list hook. Default: `tag<default_tag>`.
- **`link_mode<link_mode_type LinkMode>`**: The linking policy. Default: `link_mode<safe_link>`.
- **`void_pointer<class VoidPointer>`**: The pointer type to be used internally in the hook and propagated to the container. Default: `void_pointer<void*>`.

list container

```
template <class T, class ...Options>
class list;
```

`list` receives the same options explained in the section How to use Boost.Intrusive:

- **`base_hook<class Hook>`** / **`member_hook<class T, class Hook, Hook T::* PtrToMember>`** / **`value_traits<class ValueTraits>`**: To specify the hook type or value traits used to configure the container. (To learn about value traits go to the section Containers with custom ValueTraits.)
- **`constant_time_size<bool Enabled>`**: To activate the constant-time `size()` operation. Default: `constant_time_size<true>`
- **`size_type<bool Enabled>`**: To specify the type that will be used to store the size of the container. Default: `size_type<std::size_t>`

Example

Now let's see a small example using both hooks:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
#include <boost/intrusive/list.hpp>
#include <vector>

using namespace boost::intrusive;

class MyClass : public list_base_hook<>   //This is a derivation hook
{
   int int_;

   public:
   //This is a member hook
   list_member_hook<> member_hook_;

   MyClass(int i)
      :  int_(i)
   {}
};

//Define a list that will store MyClass using the public base hook
typedef list<MyClass>   BaseList;

//Define a list that will store MyClass using the public member hook
typedef list< MyClass
            , member_hook< MyClass, list_member_hook<>, &MyClass::member_hook_>
            > MemberList;

int main()
{
   typedef std::vector<MyClass>::iterator VectIt;
   typedef std::vector<MyClass>::reverse_iterator VectRit;

   //Create several MyClass objects, each one with a different value
   std::vector<MyClass> values;
   for(int i = 0; i < 100; ++i)  values.push_back(MyClass(i));

   BaseList baselist;
   MemberList memberlist;

   //Now insert them in the reverse order in the base hook list
   for(VectIt it(values.begin()), itend(values.end()); it != itend; ++it)
      baselist.push_front(*it);

   //Now insert them in the same order as in vector in the member hook list
   for(VectIt it(values.begin()), itend(values.end()); it != itend; ++it)
      memberlist.push_back(*it);

   //Now test lists
   {
      BaseList::reverse_iterator rbit(baselist.rbegin());
      MemberList::iterator mit(memberlist.begin());
      VectIt  it(values.begin()), itend(values.end());

      //Test the objects inserted in the base hook list
      for(; it != itend; ++it, ++rbit)
         if(&*rbit != &*it)   return 1;

      //Test the objects inserted in the member hook list
      for(it = values.begin(); it != itend; ++it, ++mit)
         if(&*mit != &*it)    return 1;
   }

   return 0;
}
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Intrusive associative containers: set, multiset, rbtree

Boost.Intrusive also offers associative containers that can be very useful when creating more complex associative containers, like containers maintaining one or more indices with different sorting semantics. Boost.Intrusive associative containers, like most STL associative container implementations are based on red-black trees.

The memory overhead of these containers is usually 3 pointers and a bit (with alignment issues, this means 3 pointers and an integer). This size can be reduced to 3 pointers if pointers have even alignment (which is usually true in most systems).

An empty, non constant-time size `set`, `multiset` or `rbtree` has also the size of 3 pointers and an integer (3 pointers when optimized for size). These containers have logarithmic complexity in many operations like searches, insertions, erasures, etc. `set` and `multiset` are the intrusive equivalents of standard `std::set` and `std::multiset` containers.

`rbtree` is a superset of `set` and `multiset` containers that offers functions to insert unique and multiple keys.

set, multiset and rbtree hooks

`set`, `multiset` and `rbtree` share the same hooks. This is an advantage, because the same user type can be inserted first in a `multiset` and after that in `set` without changing the definition of the user class.

```
template <class ...Options>
class set_base_hook;
```

- `set_base_hook`: the user class derives publicly from `set_base_hook` to make it `set`/`multiset`-compatible.

```
template <class ...Options>
class set_member_hook;
```

- `set_member_hook`: the user class contains a public `set_member_hook` to make it `set`/`multiset`-compatible.

`set_base_hook` and `set_member_hook` receive the same options explained in the section How to use Boost.Intrusive plus a size optimization option:

- **`tag<class Tag>`** (for base hooks only): This argument serves as a tag, so you can derive from more than one base hook. Default: `tag<default_tag>`.
- **`link_mode<link_mode_type LinkMode>`**: The linking policy. Default: `link_mode<safe_link>`.
- **`void_pointer<class VoidPointer>`**: The pointer type to be used internally in the hook and propagated to the container. Default: `void_pointer<void*>`.
- **`optimize_size<bool Enable>`**: The hook will be optimized for size instead of speed. The hook will embed the color bit of the red-black tree node in the parent pointer if pointer alignment is even. In some platforms, optimizing the size might reduce speed performance a bit since masking operations will be needed to access parent pointer and color attributes, in other platforms this option improves performance due to improved memory locality. Default: `optimize_size<false>`.

set, multiset and rbtree containers

```
template <class T, class ...Options>
class set;

template <class T, class ...Options>
class multiset;

template <class T, class ...Options>
class rbtree;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

These containers receive the same options explained in the section How to use Boost.Intrusive:

- **base_hook<class Hook>** / **member_hook<class T, class Hook, Hook T::* PtrToMember>** / **value_traits<class ValueTraits>**: To specify the hook type or value traits used to configure the container. (To learn about value traits go to the section Containers with custom ValueTraits.)
- **constant_time_size<bool Enabled>**: To activate the constant-time `size()` operation. Default: `constant_time_size<true>`
- **size_type<bool Enabled>**: To specify the type that will be used to store the size of the container. Default: `size_type<std::size_t>`

And they also can receive an additional option:

- **compare<class Compare>**: Comparison function for the objects to be inserted in containers. The comparison functor must induce a strict weak ordering. Default: `compare< std::less<T> >`

Example

Now let's see a small example using both hooks and both containers:

```
#include <boost/intrusive/set.hpp>
#include <vector>
#include <algorithm>
#include <cassert>

using namespace boost::intrusive;

                  //This is a base hook optimized for size
class MyClass : public set_base_hook<optimize_size<true> >
{
   int int_;

   public:
   //This is a member hook
   set_member_hook<> member_hook_;

   MyClass(int i)
      :  int_(i)
      {}
   friend bool operator< (const MyClass &a, const MyClass &b)
      {  return a.int_ < b.int_;  }
   friend bool operator> (const MyClass &a, const MyClass &b)
      {  return a.int_ > b.int_;  }
   friend bool operator== (const MyClass &a, const MyClass &b)
      {  return a.int_ == b.int_;  }
};

//Define a set using the base hook that will store values in reverse order
typedef set< MyClass, compare<std::greater<MyClass> > >     BaseSet;

//Define an multiset using the member hook
typedef member_hook<MyClass, set_member_hook<>, &MyClass::member_hook_> MemberOption;
typedef multiset< MyClass, MemberOption>   MemberMultiset;

int main()
{
   typedef std::vector<MyClass>::iterator VectIt;
   typedef std::vector<MyClass>::reverse_iterator VectRit;

   //Create several MyClass objects, each one with a different value
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
   std::vector<MyClass> values;
   for(int i = 0; i < 100; ++i)  values.push_back(MyClass(i));

   BaseSet baseset;
   MemberMultiset membermultiset;

   //Check that size optimization is activated in the base hook
   assert(sizeof(set_base_hook<optimize_size<true> >) == 3*sizeof(void*));
   //Check that size optimization is deactivated in the member hook
   assert(sizeof(set_member_hook<>) > 3*sizeof(void*));

   //Now insert them in the reverse order in the base hook set
   for(VectIt it(values.begin()), itend(values.end()); it != itend; ++it){
      baseset.insert(*it);
      membermultiset.insert(*it);
   }

   //Now test sets
   {
      BaseSet::reverse_iterator rbit(baseset.rbegin());
      MemberMultiset::iterator mit(membermultiset.begin());
      VectIt it(values.begin()), itend(values.end());

      //Test the objects inserted in the base hook set
      for(; it != itend; ++it, ++rbit)
         if(&*rbit != &*it)   return 1;

      //Test the objects inserted in the member hook set
      for(it = values.begin(); it != itend; ++it, ++mit)
         if(&*mit != &*it) return 1;
   }
   return 0;
}
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Semi-Intrusive unordered associative containers: unordered_set, unordered_multiset

Boost.Intrusive also offers hashed containers that can be very useful to implement fast-lookup containers. These containers (unordered_set and unordered_multiset) are semi-intrusive containers: they need additional memory apart from the hook stored in the value_type. This additional memory must be passed in the constructor of the container.

Unlike C++ TR1 unordered associative containers (which are also hashed containers), the contents of these semi-intrusive containers are not rehashed to maintain a load factor: that would require memory management and intrusive containers don't implement any memory management at all. However, the user can request an explicit rehashing passing a new bucket array. This also offers an additional guarantee over TR1 unordered associative containers: **iterators are not invalidated when inserting an element** in the container.

As with TR1 unordered associative containers, rehashing invalidates iterators, changes ordering between elements and changes which buckets elements appear in, but does not invalidate pointers or references to elements.

Apart from expected hash and equality function objects, **Boost.Intrusive** unordered associative containers' constructors take an argument specifying an auxiliary bucket vector to be used by the container.

unordered_set and unordered_multiset performance notes

The size overhead for a hashed container is moderate: 1 pointer per value plus a bucket array per container. The size of an element of the bucket array is usually one pointer. To obtain a good performance hashed container, the bucket length is usually the same as the number of elements that the container contains, so a well-balanced hashed container (bucket_count() is equal to size()) will have an equivalent overhead of two pointers per element.

An empty, non constant-time size unordered_set or unordered_multiset has also the size of bucket_count() pointers.

Insertions, erasures, and searches, have amortized constant-time complexity in hashed containers. However, some worst-case guarantees are linear. See unordered_set or unordered_multiset for complexity guarantees of each operation.

Be careful with non constant-time size hashed containers: some operations, like empty(), have linear complexity, unlike other **Boost.Intrusive** containers.

unordered_set and unordered_multiset hooks

unordered_set and unordered_multiset share the same hooks. This is an advantage, because the same user type can be inserted first in a unordered_multiset and after that in unordered_set without changing the definition of the user class.

```
template <class ...Options>
class unordered_set_base_hook;
```

- unordered_set_base_hook: the user class derives publicly from unordered_set_base_hook to make it unordered_set/unordered_multiset-compatible.

```
template <class ...Options>
class unordered_set_member_hook;
```

- unordered_set_member_hook: the user class contains a public unordered_set_member_hook to make it unordered_set/unordered_multiset-compatible.

unordered_set_base_hook and unordered_set_member_hook receive the same options explained in the section How to use Boost.Intrusive:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

- **`tag<class Tag>`** (for base hooks only): This argument serves as a tag, so you can derive from more than one base hook. Default: `tag<default_tag>`.
- **`link_mode<link_mode_type LinkMode>`**: The linking policy. Default: `link_mode<safe_link>`.
- **`void_pointer<class VoidPointer>`**: The pointer type to be used internally in the hook and propagated to the container. Default: `void_pointer<void*>`.

Apart from them, these hooks offer additional options:

- **`store_hash<bool Enabled>`**: This option reserves additional space in the hook to store the hash value of the object once it's introduced in the container. When this option is used, the unordered container will store the calculated hash value in the hook and rehashing operations won't need to recalculate the hash of the value. This option will improve the performance of unordered containers when rehashing is frequent or hashing the value is a slow operation. Default: `store_hash<false>`.
- **`optimize_multikey<bool Enabled>`**: This option reserves additional space in the hook that will be used to group equal elements in unordered multisets, improving significantly the performance when many equal values are inserted in these containers. Default: `optimize_multikey<false>`.

unordered_set and unordered_multiset containers

```
template<class T, class ...Options>
class unordered_set;

template<class T, class ...Options>
class unordered_multiset;
```

As mentioned, unordered containers need an auxiliary array to work. **Boost.Intrusive** unordered containers receive this auxiliary array packed in a type called `bucket_traits` (which can be also customized by a container option). All unordered containers receive a `bucket_traits` object in their constructors. The default `bucket_traits` class is initialized with a pointer to an array of buckets and its size:

```
#include <boost/intrusive/unordered_set.hpp>

using namespace boost::intrusive;

struct MyClass : public unordered_set_base_hook<>
{};

typedef unordered_set<MyClass>::bucket_type     bucket_type;
typedef unordered_set<MyClass>::bucket_traits   bucket_traits;

int main()
{
   bucket_type buckets[100];
   unordered_set<MyClass> uset(bucket_traits(buckets, 100));
   return 0;
}
```

Each hashed container needs **its own bucket traits**, that is, **its own bucket vector**. Two hashed containers **can't** share the same `bucket_type` elements. The bucket array **must** be destroyed **after** the container using it is destroyed, otherwise, the result is undefined.

`unordered_set` and `unordered_multiset` receive the same options explained in the section How to use Boost.Intrusive:

- **`base_hook<class Hook>`** / **`member_hook<class T, class Hook, Hook T::* PtrToMember>`** / **`value_traits<class ValueTraits>`**: To specify the hook type or value traits used to configure the container. (To learn about value traits go to the section Containers with custom ValueTraits.)

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

- **`constant_time_size<bool Enabled>`**: To activate the constant-time `size()` operation. Default: `constant_time_size<true>`
- **`size_type<bool Enabled>`**: To specify the type that will be used to store the size of the container. Default: `size_type<std::size_t>`

And they also can receive additional options:

- **`equal<class Equal>`**: Equality function for the objects to be inserted in containers. Default: `equal< std::equal_to<T> >`
- **`hash<class Hash>`**: Hash function to be used in the container. Default: `hash< boost::hash<T> >`
- **`bucket_traits<class BucketTraits>`**: A type that wraps the bucket vector to be used by the unordered container. Default: a type initialized by the address and size of a bucket array and stores both variables internally.
- **`power_2_buckets<bool Enabled>`**: The user guarantees that only bucket arrays with power of two length will be used. The container will then use masks instead of modulo operations to obtain the bucket number from the hash value. Masks are faster than modulo operations and for some applications modulo operations can impose a considerable overhead. In debug mode an assertion will be raised if the user provides a bucket length that is not power of two. Default: `power_2_buckets<false>`.
- **`cache_begin<bool Enabled>`**: **Note: this option is not compatible with `auto_unlink` hooks**. Due to its internal structure, finding the first element of an unordered container (`begin()` operation) is amortized constant-time. It's possible to speed up `begin()` and other operations related to it (like `clear()`) if the container caches internally the position of the first element. This imposes the overhead of one pointer to the size of the container. Default: `cache_begin<false>`.
- **`compare_hash<bool Enabled>`**: **Note: this option requires `store_hash<true>` option in the hook**. When the comparison function is expensive, (e.g. strings with a long common predicate) sometimes (specially when the load factor is high or we have many equivalent elements in an `unordered_multiset` and no `optimize_multikey<>` is activated in the hook) the equality function is a performance problem. Two equal values must have equal hashes, so comparing the hash values of two elements before using the comparison functor can speed up some implementations.
- **`incremental<bool Enabled>`**: Activates incremental hashing (also known as Linear Hashing). This option implies `power_2_buckets<true>` and the container will require power of two buckets. For more information on incremental hashing, see `Linear hash` on Wikipedia Default: `incremental<false>`

Example

Now let's see a small example using both hooks and both containers:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
#include <boost/intrusive/unordered_set.hpp>
#include <vector>
#include <algorithm>
#include <boost/functional/hash.hpp>

using namespace boost::intrusive;

class MyClass : public unordered_set_base_hook<>
{               //This is a derivation hook
   int int_;

   public:
   unordered_set_member_hook<> member_hook_; //This is a member hook

   MyClass(int i)
      :  int_(i)
   {}

   friend bool operator== (const MyClass &a, const MyClass &b)
   {  return a.int_ == b.int_;  }

   friend std::size_t hash_value(const MyClass &value)
   {  return std::size_t(value.int_); }
};

//Define an unordered_set that will store MyClass objects using the base hook
typedef unordered_set<MyClass>    BaseSet;

//Define an unordered_multiset that will store MyClass using the member hook
typedef member_hook<MyClass, unordered_set_member_hook<>, &MyClass::member_hook_>
   MemberOption;
typedef unordered_multiset< MyClass, MemberOption>  MemberMultiSet;

int main()
{
   typedef std::vector<MyClass>::iterator VectIt;
   typedef std::vector<MyClass>::reverse_iterator VectRit;

   //Create a vector with 100 different MyClass objects
   std::vector<MyClass> values;
   for(int i = 0; i < 100; ++i)  values.push_back(MyClass(i));

   //Create a copy of the vector
   std::vector<MyClass> values2(values);

   //Create a bucket array for base_set
   BaseSet::bucket_type base_buckets[100];

   //Create a bucket array for member_multi_set
   MemberMultiSet::bucket_type member_buckets[200];

   //Create unordered containers taking buckets as arguments
   BaseSet base_set(BaseSet::bucket_traits(base_buckets, 100));
   MemberMultiSet member_multi_set
      (MemberMultiSet::bucket_traits(member_buckets, 200));

   //Now insert values's elements in the unordered_set
   for(VectIt it(values.begin()), itend(values.end()); it != itend; ++it)
      base_set.insert(*it);

   //Now insert values's and values2's elements in the unordered_multiset
   for(VectIt it(values.begin()), itend(values.end()), it2(values2.begin())
      ; it != itend; ++it, ++it2){
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
      member_multi_set.insert(*it);
      member_multi_set.insert(*it2);
   }

   //Now find every element
   {
      VectIt it(values.begin()), itend(values.end());

      for(; it != itend; ++it){
         //base_set should contain one element for each key
         if(base_set.count(*it) != 1)           return 1;
         //member_multi_set should contain two elements for each key
         if(member_multi_set.count(*it) != 2)   return 1;
      }
   }
   return 0;
}
```

Custom bucket traits

Instead of using the default `bucket_traits` class to store the bucket array, a user can define his own class to store the bucket array using the **bucket_traits**<> option. A user-defined bucket-traits must fulfill the following interface:

```
class my_bucket_traits
{
   bucket_ptr        bucket_begin();
   const_bucket_ptr  bucket_begin() const;
   std::size_t bucket_count() const;
};
```

The following bucket traits just stores a pointer to the bucket array but the size is a compile-time constant. Note the use of the auxiliary `unordered_bucket` and `unordered_bucket_ptr` utilities to obtain the type of the bucket and its pointer before defining the unordered container:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
#include <boost/intrusive/unordered_set.hpp>
#include <boost/functional/hash.hpp>
#include <vector>

using namespace boost::intrusive;

//A class to be inserted in an unordered_set
class MyClass : public unordered_set_base_hook<>
{
   int int_;

   public:
   MyClass(int i = 0) : int_(i)
   {}

   friend bool operator==(const MyClass &l, const MyClass &r)
      {  return l.int_ == r.int_;   }
   friend std::size_t hash_value(const MyClass &v)
      {  return boost::hash_value(v.int_); }
};

//Define the base hook option
typedef base_hook< unordered_set_base_hook<> >     BaseHookOption;

//Obtain the types of the bucket and the bucket pointer
typedef unordered_bucket<BaseHookOption>::type     BucketType;
typedef unordered_bucket_ptr<BaseHookOption>::type BucketPtr;

//The custom bucket traits.
class custom_bucket_traits
{
   public:
   static const int NumBuckets = 100;

   custom_bucket_traits(BucketPtr buckets)
      :  buckets_(buckets)
   {}

   //Functions to be implemented by custom bucket traits
   BucketPtr   bucket_begin() const {  return buckets_;  }
   std::size_t bucket_count() const {  return NumBuckets;}

   private:
   BucketPtr buckets_;
};

//Define the container using the custom bucket traits
typedef unordered_set<MyClass, bucket_traits<custom_bucket_traits> > BucketTraitsUset;

int main()
{
   typedef std::vector<MyClass>::iterator VectIt;
   std::vector<MyClass> values;

   //Fill values
   for(int i = 0; i < 100; ++i)  values.push_back(MyClass(i));

   //Now create the bucket array and the custom bucket traits object
   BucketType buckets[custom_bucket_traits::NumBuckets];
   custom_bucket_traits btraits(buckets);

   //Now create the unordered set
   BucketTraitsUset uset(btraits);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
   //Insert the values in the unordered set
   for(VectIt it(values.begin()), itend(values.end()); it != itend; ++it)
      uset.insert(*it);

   return 0;
}
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Intrusive splay tree based associative containers: splay_set, splay_multiset and , splay_tree

C++ associative containers are usually based on red-black tree implementations (e.g.: STL, Boost.Intrusive associative containers). However, there are other interesting data structures that offer some advantages (and also disadvantages).

Splay trees are self-adjusting binary search trees used typically in caches, memory allocators and other applications, because splay trees have a "caching effect": recently accessed elements have better access times than elements accessed less frequently. For more information on splay trees see Wikipedia entry.

Boost.Intrusive offers 3 containers based on splay trees: `splay_set`, `splay_multiset` and `splaytree`. The first two are similar to `set` or `multiset` and the latter is a generalization that offers functions both to insert unique and multiple keys.

The memory overhead of these containers with Boost.Intrusive hooks is usually 3 pointers. An empty, non constant-time size splay container has also a size of 3 pointers.

Advantages and disadvantages of splay tree based containers

Splay tree based intrusive containers have logarithmic complexity in many operations like searches, insertions, erasures, etc., but if some elements are more frequently accessed than others, splay trees perform faster searches than equivalent balanced binary trees (such as red-black trees).

The caching effect offered by splay trees comes with a cost: the tree must be rebalanced when an element is searched. This disallows const versions of search functions like `find()`, `lower_bound()`, `upper_bound()`, `equal_range()`, `count()`, etc.

Because of this, splay-tree based associative containers are not drop-in replacements of `set`/ `multiset`.

Apart from this, if element searches are randomized, the tree will be rebalanced without taking advantage of the cache effect, so splay trees can offer worse performance than other balanced trees for some search patterns.

splay_set, splay_multiset and splaytree hooks

`splay_set`, `splay_multiset` and `splaytree` share the same hooks.

```
template <class ...Options>
class splay_set_base_hook;
```

- `splay_set_base_hook`: the user class derives publicly from this class to make it compatible with splay tree based containers.

```
template <class ...Options>
class splay_set_member_hook;
```

- `set_member_hook`: the user class contains a public member of this class to make it compatible with splay tree based containers.

`splay_set_base_hook` and `splay_set_member_hook` receive the same options explained in the section How to use Boost.Intrusive:

- **`tag<class Tag>`** (for base hooks only): This argument serves as a tag, so you can derive from more than one base hook. Default: `tag<default_tag>`.

- **`link_mode<link_mode_type LinkMode>`**: The linking policy. Default: `link_mode<safe_link>`.

- **`void_pointer<class VoidPointer>`**: The pointer type to be used internally in the hook and propagated to the container. Default: `void_pointer<void*>`.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

splay_set, splay_multiset and splaytree containers

```
template <class T, class ...Options>
class splay_set;

template <class T, class ...Options>
class splay_multiset;

template <class T, class ...Options>
class splaytree;
```

These containers receive the same options explained in the section How to use Boost.Intrusive:

- **`base_hook<class Hook>`** / **`member_hook<class T, class Hook, Hook T::* PtrToMember>`** / **`value_traits<class ValueTraits>`**: To specify the hook type or value traits used to configure the container. (To learn about value traits go to the section Containers with custom ValueTraits.)
- **`constant_time_size<bool Enabled>`**: To activate the constant-time `size()` operation. Default: `constant_time_size<true>`
- **`size_type<bool Enabled>`**: To specify the type that will be used to store the size of the container. Default: `size_type<std::size_t>`

And they also can receive an additional option:

- **`compare<class Compare>`**: Comparison function for the objects to be inserted in containers. The comparison functor must induce a strict weak ordering. Default: `compare< std::less<T> >`

Splay trees with BST hooks

Intrusive splay containers can also use plain binary search tree hooks `bs_set_base_hook` and `bs_set_base_hook`. These hooks can be used by other intrusive containers like intrusive scapegoat containers `sg_set` and `sg_multiset`. A programmer might prefer using a binary search tree hook so that the same type can be inserted in some situations in a splay container but also inserted in other compatible containers when the hook is not being used in a splay container.

`bs_set_base_hook` and `bs_set_member_hook` admit the same options as `splay_set_base_hook`.

Example

Now let's see a small example using both splay hooks, binary search tree hooks and `splay_set`/ `splay_multiset` containers:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
#include <boost/intrusive/splay_set.hpp>
#include <boost/intrusive/bs_set_hook.hpp>
#include <vector>
#include <algorithm>

using namespace boost::intrusive;

class MyClass
   : public splay_set_base_hook<>   //This is an splay tree base hook
   , public bs_set_base_hook<>      //This is a binary search tree base hook

{
   int int_;

   public:
   //This is a member hook
   splay_set_member_hook<> member_hook_;

   MyClass(int i)
      :  int_(i)
      {}
   friend bool operator< (const MyClass &a, const MyClass &b)
      {  return a.int_ < b.int_;  }
   friend bool operator> (const MyClass &a, const MyClass &b)
      {  return a.int_ > b.int_;  }
   friend bool operator== (const MyClass &a, const MyClass &b)
      {  return a.int_ == b.int_;  }
};

//Define a set using the base hook that will store values in reverse order
typedef splay_set< MyClass, compare<std::greater<MyClass> > >     BaseSplaySet;

//Define a set using the binary search tree hook
typedef splay_set< MyClass, base_hook<bs_set_base_hook<> > >      BaseBsSplaySet;

//Define an multiset using the member hook
typedef member_hook<MyClass, splay_set_member_hook<>, &MyClass::member_hook_> MemberOption;
typedef splay_multiset< MyClass, MemberOption>   MemberSplayMultiset;

int main()
{
   typedef std::vector<MyClass>::iterator VectIt;
   typedef std::vector<MyClass>::reverse_iterator VectRit;

   //Create several MyClass objects, each one with a different value
   std::vector<MyClass> values;
   for(int i = 0; i < 100; ++i)  values.push_back(MyClass(i));

   BaseSplaySet   baseset;
   BaseBsSplaySet bsbaseset;
   MemberSplayMultiset membermultiset;


   //Insert values in the container
   for(VectIt it(values.begin()), itend(values.end()); it != itend; ++it){
      baseset.insert(*it);
      bsbaseset.insert(*it);
      membermultiset.insert(*it);
   }

   //Now test sets
   {
      BaseSplaySet::reverse_iterator rbit(baseset.rbegin());
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
      BaseBsSplaySet::iterator bsit(bsbaseset.begin());
      MemberSplayMultiset::iterator mit(membermultiset.begin());
      VectIt it(values.begin()), itend(values.end());

      //Test the objects inserted in the base hook set
      for(; it != itend; ++it, ++rbit){
         if(&*rbit != &*it)   return 1;
      }

      //Test the objects inserted in member and binary search hook sets
      for(it = values.begin(); it != itend; ++it, ++bsit, ++mit){
         if(&*bsit != &*it)   return 1;
         if(&*mit != &*it)    return 1;
      }
   }
   return 0;
}
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Intrusive avl tree based associative containers: avl_set, avl_multiset and avltree

Similar to red-black trees, AVL trees are balanced binary trees. AVL trees are often compared with red-black trees because they support the same set of operations and because both take O(log n) time for basic operations. AVL trees are more rigidly balanced than Red-Black trees, leading to slower insertion and removal but faster retrieval, so AVL trees perform better than red-black trees for lookup-intensive applications.

Boost.Intrusive offers 3 containers based on avl trees: `avl_set`, `avl_multiset` and `avltree`. The first two are similar to `set` or `multiset` and the latter is a generalization that offers functions both to insert unique and multiple keys.

The memory overhead of these containers with Boost.Intrusive hooks is usually 3 pointers and 2 bits (due to alignment, this usually means 3 pointers plus an integer). This size can be reduced to 3 pointers if pointers have 4 byte alignment (which is usually true in 32 bit systems).

An empty, non constant-time size `avl_set`, `avl_multiset` or `avltree` also has a size of 3 pointers and an integer (3 pointers when optimized for size).

avl_set, avl_multiset and avltree hooks

`avl_set`, `avl_multiset` and `avltree` share the same hooks.

```
template <class ...Options>
class avl_set_base_hook;
```

- `avl_set_base_hook`: the user class derives publicly from this class to make it compatible with avl tree based containers.

```
template <class ...Options>
class avl_set_member_hook;
```

- `set_member_hook`: the user class contains a public member of this class to make it compatible with avl tree based containers.

`avl_set_base_hook` and `avl_set_member_hook` receive the same options explained in the section How to use Boost.Intrusive plus an option to optimize the size of the node:

- **`tag<class Tag>`** (for base hooks only): This argument serves as a tag, so you can derive from more than one base hook. Default: `tag<default_tag>`.
- **`link_mode<link_mode_type LinkMode>`**: The linking policy. Default: `link_mode<safe_link>`.
- **`void_pointer<class VoidPointer>`**: The pointer type to be used internally in the hook and propagated to the container. Default: `void_pointer<void*>`.
- **`optimize_size<bool Enable>`**: The hook will be optimized for size instead of speed. The hook will embed the balance bits of the AVL tree node in the parent pointer if pointer alignment is multiple of 4. In some platforms, optimizing the size might reduce speed performance a bit since masking operations will be needed to access parent pointer and balance factor attributes, in other platforms this option improves performance due to improved memory locality. Default: `optimize_size<false>`.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

avl_set, avl_multiset and avltree containers

```
template <class T, class ...Options>
class avl_set;

template <class T, class ...Options>
class avl_multiset;

template <class T, class ...Options>
class avltree;
```

These containers receive the same options explained in the section How to use Boost.Intrusive:

- **`base_hook<class Hook>`** / **`member_hook<class T, class Hook, Hook T::* PtrToMember>`** / **`value_traits<class ValueTraits>`**: To specify the hook type or value traits used to configure the container. (To learn about value traits go to the section Containers with custom ValueTraits.)
- **`constant_time_size<bool Enabled>`**: To activate the constant-time `size()` operation. Default: `constant_time_size<true>`
- **`size_type<bool Enabled>`**: To specify the type that will be used to store the size of the container. Default: `size_type<std::size_t>`

And they also can receive an additional option:

- **`compare<class Compare>`**: Comparison function for the objects to be inserted in containers. The comparison functor must induce a strict weak ordering. Default: `compare< std::less<T> >`

Example

Now let's see a small example using both hooks and `avl_set`/ `avl_multiset` containers:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
#include <boost/intrusive/avl_set.hpp>
#include <vector>
#include <algorithm>
#include <cassert>

using namespace boost::intrusive;

                  //This is a base hook optimized for size
class MyClass : public avl_set_base_hook<optimize_size<true> >
{
   int int_;

   public:
   //This is a member hook
   avl_set_member_hook<> member_hook_;

   MyClass(int i)
      :  int_(i)
      {}
   friend bool operator< (const MyClass &a, const MyClass &b)
      {  return a.int_ < b.int_;  }
   friend bool operator> (const MyClass &a, const MyClass &b)
      {  return a.int_ > b.int_;  }
   friend bool operator== (const MyClass &a, const MyClass &b)
      {  return a.int_ == b.int_;  }
};

//Define an avl_set using the base hook that will store values in reverse order
typedef avl_set< MyClass, compare<std::greater<MyClass> > >     BaseSet;

//Define an multiset using the member hook
typedef member_hook<MyClass, avl_set_member_hook<>, &MyClass::member_hook_> MemberOption;
typedef avl_multiset< MyClass, MemberOption>   MemberMultiset;

int main()
{
   typedef std::vector<MyClass>::iterator VectIt;
   typedef std::vector<MyClass>::reverse_iterator VectRit;

   //Create several MyClass objects, each one with a different value
   std::vector<MyClass> values;
   for(int i = 0; i < 100; ++i)  values.push_back(MyClass(i));

   BaseSet baseset;
   MemberMultiset membermultiset;

   //Check that size optimization is activated in the base hook
   assert(sizeof(avl_set_base_hook<optimize_size<true> >) == 3*sizeof(void*));
   //Check that size optimization is deactivated in the member hook
   assert(sizeof(avl_set_member_hook<>) > 3*sizeof(void*));

   //Now insert them in the sets
   for(VectIt it(values.begin()), itend(values.end()); it != itend; ++it){
      baseset.insert(*it);
      membermultiset.insert(*it);
   }

   //Now test avl_sets
   {
      BaseSet::reverse_iterator rbit(baseset.rbegin());
      MemberMultiset::iterator mit(membermultiset.begin());
      VectIt it(values.begin()), itend(values.end());
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
      //Test the objects inserted in the base hook avl_set
      for(; it != itend; ++it, ++rbit)
         if(&*rbit != &*it)   return 1;

      //Test the objects inserted in the member hook avl_set
      for(it = values.begin(); it != itend; ++it, ++mit)
         if(&*mit != &*it) return 1;
   }
   return 0;
}
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Intrusive scapegoat tree based associative containers: sg_set, sg_multiset and sgtree

A scapegoat tree is a self-balancing binary search tree, that provides worst-case O(log n) lookup time, and O(log n) amortized insertion and deletion time. Unlike other self-balancing binary search trees that provide worst case O(log n) lookup time, scapegoat trees have no additional per-node overhead compared to a regular binary search tree.

A binary search tree is said to be weight balanced if half the nodes are on the left of the root, and half on the right. An a-height-balanced tree is defined with defined with the following equation:

height(tree) <= log1/a(tree.size())

- **a == 1**: A tree forming a linked list is considered balanced.
- **a == 0.5**: Only a perfectly balanced binary is considered balanced.

Scapegoat trees are loosely *a-height-balanced* so:

height(tree) <= log1/a(tree.size()) + 1

Scapegoat trees support any a between 0.5 and 1. If a is higher, the tree is rebalanced less often, obtaining quicker insertions but slower searches. Lower a values improve search times. Scapegoat-trees implemented in **Boost.Intrusive** offer the possibility of **changing a at run-time** taking advantage of the flexibility of scapegoat trees. For more information on scapegoat trees see Wikipedia entry.

Scapegoat trees also have downsides:

- They need additional storage of data on the root (the size of the tree, for example) to achieve logarithmic complexity operations so it's not possible to offer `auto_unlink` hooks. The size of an empty scapegoat tree is also considerably increased.
- The operations needed to determine if the tree is unbalanced require floating-point operations like *log1/a*. If the system has no floating point operations (like some embedded systems), scapegoat tree operations might become slow.

Boost.Intrusive offers 3 containers based on scapegoat trees: `sg_set`, `sg_multiset` and `sgtree`. The first two are similar to `set` or `multiset` and the latter is a generalization that offers functions both to insert unique and multiple keys.

The memory overhead of these containers with Boost.Intrusive hooks is 3 pointers.

An empty, `sg_set`, `sg_multiset` or `sgtree` has also the size of 3 pointers, two integers and two floating point values (equivalent to the size of 7 pointers on most systems).

Using binary search tree hooks: bs_set_base_hook and bs_set_member_hook

`sg_set`, `sg_multiset` and `sgtree` don't use their own hooks but plain binary search tree hooks. This has many advantages since binary search tree hooks can also be used to insert values in splay and treap containers.

```
template <class ...Options>
class bs_set_base_hook;
```

- `bs_set_base_hook`: the user class derives publicly from this class to make it compatible with scapegoat tree based containers.

```
template <class ...Options>
class bs_set_member_hook;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

- set_member_hook: the user class contains a public member of this class to make it compatible with scapegoat tree based containers.

bs_set_base_hook and bs_set_member_hook receive the same options explained in the section How to use Boost.Intrusive:

- **tag<class Tag>** (for base hooks only): This argument serves as a tag, so you can derive from more than one base hook. Default: tag<default_tag>.
- **link_mode<link_mode_type LinkMode>**: The linking policy. Default: link_mode<safe_link>.
- **void_pointer<class VoidPointer>**: The pointer type to be used internally in the hook and propagated to the container. Default: void_pointer<void*>.

sg_set, sg_multiset and sgtree containers

```
template <class T, class ...Options>
class sg_set;

template <class T, class ...Options>
class sg_multiset;

template <class T, class ...Options>
class sgtree;
```

These containers receive the same options explained in the section How to use Boost.Intrusive:

- **base_hook<class Hook>** / **member_hook<class T, class Hook, Hook T::* PtrToMember>** / **value_traits<class ValueTraits>**: To specify the hook type or value traits used to configure the container. (To learn about value traits go to the section Containers with custom ValueTraits.)
- **size_type<bool Enabled>**: To specify the type that will be used to store the size of the container. Default: size_type<std::size_t>

And they also can receive additional options:

- **compare<class Compare>**: Comparison function for the objects to be inserted in containers. The comparison functor must induce a strict weak ordering. Default: compare< std::less<T> >
- **floating_point<bool Enable>**: When this option is deactivated, the scapegoat tree loses the ability to change the balance factor a at run-time, but the size of an empty container is reduced and no floating point operations are performed, normally increasing container performance. The fixed a factor that is used when this option is activated is *1/sqrt(2) ~ 0,70711*. Default: floating_point<true>

Example

Now let's see a small example using both hooks and sg_set/ sg_multiset containers:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
#include <boost/intrusive/sg_set.hpp>
#include <vector>
#include <algorithm>
#include <cassert>

using namespace boost::intrusive;

class MyClass : public bs_set_base_hook<>
{
   int int_;

   public:
   //This is a member hook
   bs_set_member_hook<> member_hook_;

   MyClass(int i)
      :  int_(i)
      {}
   friend bool operator< (const MyClass &a, const MyClass &b)
      {  return a.int_ < b.int_;  }
   friend bool operator> (const MyClass &a, const MyClass &b)
      {  return a.int_ > b.int_;  }
   friend bool operator== (const MyClass &a, const MyClass &b)
      {  return a.int_ == b.int_;  }
};

//Define an sg_set using the base hook that will store values in reverse order
//and won't execute floating point operations.
typedef sg_set
   < MyClass, compare<std::greater<MyClass> >, floating_point<false> >   BaseSet;

//Define an multiset using the member hook
typedef member_hook<MyClass, bs_set_member_hook<>, &MyClass::member_hook_> MemberOption;
typedef sg_multiset< MyClass, MemberOption>   MemberMultiset;

int main()
{
   typedef std::vector<MyClass>::iterator VectIt;
   typedef std::vector<MyClass>::reverse_iterator VectRit;

   //Create several MyClass objects, each one with a different value
   std::vector<MyClass> values;
   for(int i = 0; i < 100; ++i)  values.push_back(MyClass(i));

   BaseSet baseset;
   MemberMultiset membermultiset;

   //Now insert them in the reverse order in the base hook sg_set
   for(VectIt it(values.begin()), itend(values.end()); it != itend; ++it){
      baseset.insert(*it);
      membermultiset.insert(*it);
   }

   //Change balance factor
   membermultiset.balance_factor(0.9f);

   //Now test sg_sets
   {
      BaseSet::reverse_iterator rbit(baseset.rbegin());
      MemberMultiset::iterator mit(membermultiset.begin());
      VectIt it(values.begin()), itend(values.end());

      //Test the objects inserted in the base hook sg_set
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
      for(; it != itend; ++it, ++rbit)
         if(&*rbit != &*it)   return 1;

      //Test the objects inserted in the member hook sg_set
      for(it = values.begin(); it != itend; ++it, ++mit)
         if(&*mit != &*it) return 1;
   }
   return 0;
}
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Intrusive treap based associative containers: treap_set, treap_multiset and treap

The name *treap* is a mixture of *tree* and *heap* indicating that Treaps exhibit the properties of both binary search trees and heaps. A treap is a binary search tree that orders the nodes by a key but also by a priority attribute. The nodes are ordered so that the keys form a binary search tree and the priorities obey the max heap order property.

- If v is a left descendant of u, then key[v] < key[u];
- If v is a right descendant of u, then key[v] > key[u];
- If v is a child of u, then priority[v] <= priority[u];

If priorities are non-random, the tree will usually be unbalanced; this worse theoretical average-case behavior may be outweighed by better expected-case behavior, as the most important items will be near the root. This means most important objects will be retrieved faster than less important items and for items keys with equal keys most important objects will be found first. These properties are important for some applications.

The priority comparison will be provided just like the key comparison, via a function object that will be stored in the intrusive container. This means that the priority can be stored in the value to be introduced in the treap or computed on flight (via hashing or similar).

Boost.Intrusive offers 3 containers based on treaps: `treap_set`, `treap_multiset` and `treap`. The first two are similar to `set` or `multiset` and the latter is a generalization that offers functions both to insert unique and multiple keys.

The memory overhead of these containers with Boost.Intrusive hooks is 3 pointers.

An empty, `treap_set`, `treap_multiset` or `treap` has also the size of 3 pointers and an integer (supposing empty function objects for key and priority comparison and constant-time size).

Using binary search tree hooks: bs_set_base_hook and bs_set_member_hook

`treap_set`, `treap_multiset` and `treap` don't use their own hooks but plain binary search tree hooks. This has many advantages since binary search tree hooks can also be used to insert values in splay containers and scapegoat trees.

```
template <class ...Options>
class bs_set_base_hook;
```

- `bs_set_base_hook`: the user class derives publicly from this class to make it compatible with scapegoat tree based containers.

```
template <class ...Options>
class bs_set_member_hook;
```

- `set_member_hook`: the user class contains a public member of this class to make it compatible with scapegoat tree based containers.

`bs_set_base_hook` and `bs_set_member_hook` receive the same options explained in the section How to use Boost.Intrusive:

- **`tag<class Tag>`** (for base hooks only): This argument serves as a tag, so you can derive from more than one base hook. Default: `tag<default_tag>`.
- **`link_mode<link_mode_type LinkMode>`**: The linking policy. Default: `link_mode<safe_link>`.
- **`void_pointer<class VoidPointer>`**: The pointer type to be used internally in the hook and propagated to the container. Default: `void_pointer<void*>`.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

treap_set, treap_multiset and treap containers

```
template <class T, class ...Options>
class treap_set;

template <class T, class ...Options>
class treap_multiset;

template <class T, class ...Options>
class treap;
```

These containers receive the same options explained in the section How to use Boost.Intrusive:

- **`base_hook<class Hook>`** / **`member_hook<class T, class Hook, Hook T::* PtrToMember>`** / **`value_traits<class ValueTraits>`**: To specify the hook type or value traits used to configure the container. (To learn about value traits go to the section Containers with custom ValueTraits.)

- **`constant_time_size<bool Enabled>`**: To activate the constant-time `size()` operation. Default: `constant_time_size<true>`

- **`size_type<bool Enabled>`**: To specify the type that will be used to store the size of the container. Default: `size_type<std::size_t>`

And they also can receive additional options:

- **`compare<class Compare>`**: Comparison function for the objects to be inserted in containers. The comparison functor must induce a strict weak ordering. Default: `compare< std::less<T> >`

- **`priority<class PriorityCompare>`**: Priority Comparison function for the objects to be inserted in containers. The comparison functor must induce a strict weak ordering. Default: `priority< priority_compare<T> >`

The default `priority_compare<T>` object function will call an unqualified function `priority_order` passing two constant `T` references as arguments and should return true if the first argument has higher priority (it will be searched faster), inducing strict weak ordering. The function will be found using ADL lookup so that the user just needs to define a `priority_order` function in the same namespace as his class:

```
struct MyType
{
   friend bool priority_order(const MyType &a, const MyType &b)
   {...}
};
```

or

```
namespace mytype {

struct MyType{ ... };

bool priority_order(const MyType &a, const MyType &b)
{...}

}  //namespace mytype {
```

Exception safety of treap-based intrusive containers

In general, intrusive containers offer strong safety guarantees, but treap containers must deal with two possibly throwing functors (one for value ordering, another for priority ordering). Moreover, treap erasure operations require rotations based on the priority order

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

function and this issue degrades usual `erase(const_iterator)` no-throw guarantee. However, intrusive offers the strongest possible behaviour in these situations. In summary:

- If the priority order functor does not throw, treap-based containers, offer exactly the same guarantees as other tree-based containers.
- If the priority order functor throws, treap-based containers offer strong guarantee.

Example

Now let's see a small example using both hooks and `treap_set`/ `treap_multiset` containers:

```
#include <boost/intrusive/treap_set.hpp>
#include <vector>
#include <algorithm>
#include <cassert>

using namespace boost::intrusive;

class MyClass : public bs_set_base_hook<> //This is a base hook
{
   int int_;
   unsigned int prio_;

   public:
   //This is a member hook
   bs_set_member_hook<> member_hook_;

   MyClass(int i, unsigned int prio)  :  int_(i), prio_(prio)
      {}

   unsigned int get_priority() const
   {  return this->prio_;   }

   //Less and greater operators
   friend bool operator< (const MyClass &a, const MyClass &b)
      {  return a.int_ < b.int_;  }
   friend bool operator> (const MyClass &a, const MyClass &b)
      {  return a.int_ > b.int_;  }
   //Default priority compare
   friend bool priority_order (const MyClass &a, const MyClass &b)
      {  return a.prio_ < b.prio_;  }  //Lower value means higher priority
   //Inverse priority compare
   friend bool priority_inverse_order (const MyClass &a, const MyClass &b)
      {  return a.prio_ > b.prio_;  }  //Higher value means higher priority
};

struct inverse_priority
{
   bool operator()(const MyClass &a, const MyClass &b) const
   {  return priority_inverse_order(a, b); }
};


//Define an treap_set using the base hook that will store values in reverse order
typedef treap_set< MyClass, compare<std::greater<MyClass> > >     BaseSet;

//Define an multiset using the member hook that will store
typedef member_hook<MyClass, bs_set_member_hook<>, &MyClass::member_hook_> MemberOption;
typedef treap_multiset
   < MyClass, MemberOption, priority<inverse_priority> > MemberMultiset;

int main()
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
{
   typedef std::vector<MyClass>::iterator VectIt;

   //Create several MyClass objects, each one with a different value
   std::vector<MyClass> values;
   for(int i = 0; i < 100; ++i)  values.push_back(MyClass(i, (i % 10)));

   BaseSet baseset;
   MemberMultiset membermultiset;

   //Now insert them in the sets
   for(VectIt it(values.begin()), itend(values.end()); it != itend; ++it){
      baseset.insert(*it);
      membermultiset.insert(*it);
   }

   //Now test treap_sets
   {
      BaseSet::reverse_iterator rbit(baseset.rbegin());
      MemberMultiset::iterator mit(membermultiset.begin());
      VectIt it(values.begin()), itend(values.end());

      //Test the objects inserted in the base hook treap_set
      for(; it != itend; ++it, ++rbit)
         if(&*rbit != &*it)   return 1;

      //Test the objects inserted in the member hook treap_set
      for(it = values.begin(); it != itend; ++it, ++mit)
         if(&*mit != &*it) return 1;

      //Test priority order
      for(int i = 0; i < 100; ++i){
         if(baseset.top()->get_priority() != static_cast<unsigned int>(i/10))
            return 1;
         if(membermultiset.top()->get_priority() != 9u - static_cast<unsigned int>(i/10))
            return 1;
         baseset.erase(baseset.top());
         membermultiset.erase(membermultiset.top());
      }
   }
   return 0;
}
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Advanced lookup and insertion functions for associative containers

Advanced lookups

Boost.Intrusive associative containers offer the same interface as STL associative containers. However, STL and TR1 ordered and unordered simple associative containers (`std::set`, `std::multiset`, `std::tr1::unordered_set` and `std::tr1::unordered_multiset`) have some inefficiencies caused by the interface: the user can only operate with `value_type` objects. When using these containers we must use `iterator find(const value_type &value)` to find a value. The same happens in other functions like `equal_range`, `lower_bound`, `upper_bound`, etc.

However, sometimes the object to be searched is quite expensive to construct:

```
#include <boost/intrusive/set.hpp>
#include <boost/intrusive/unordered_set.hpp>
#include <cstring>

using namespace boost::intrusive;

// Hash function for strings
struct StrHasher
{
   std::size_t operator()(const char *str) const
   {
      std::size_t seed = 0;
      for(; *str; ++str)   boost::hash_combine(seed, *str);
      return seed;
   }
};

class Expensive : public set_base_hook<>, public unordered_set_base_hook<>
{
   std::string key_;
   // Other members...

   public:
   Expensive(const char *key)
      :  key_(key)
      {}  //other expensive initializations...

   const std::string & get_key() const
      {  return key_;   }

   friend bool operator <  (const Expensive &a, const Expensive &b)
      {  return a.key_ < b.key_;  }

   friend bool operator == (const Expensive &a, const Expensive &b)
      {  return a.key_ == b.key_;  }

   friend std::size_t hash_value(const Expensive &object)
      {  return StrHasher()(object.get_key().c_str());  }
};

// A set and unordered_set that store Expensive objects
typedef set<Expensive>           Set;
typedef unordered_set<Expensive> UnorderedSet;

// Search functions
Expensive *get_from_set(const char* key, Set &set_object)
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
{
   Set::iterator it = set_object.find(Expensive(key));
   if( it == set_object.end() )     return 0;
   return &*it;
}

Expensive *get_from_uset(const char* key, UnorderedSet &uset_object)
{
   UnorderedSet::iterator it = uset_object.find(Expensive (key));
   if( it == uset_object.end() )  return 0;
   return &*it;
}
```

`Expensive` is an expensive object to construct. If "key" c-string is quite long `Expensive` has to construct a `std::string` using heap memory. Like `Expensive`, many times the only member taking part in ordering issues is just a small part of the class. For example, with `Expensive`, only the internal `std::string` is needed to compare the object.

In both containers, if we call `get_from_set/get_from_unordered_set` in a loop, we might get a performance penalty, because we are forced to create a whole `Expensive` object to be able to find an equivalent one.

Sometimes this interface limitation is severe, because we **might not have enough information to construct the object** but we might **have enough information to find the object**. In this case, a name is enough to search `Expensive` in the container but constructing an `Expensive` might require more information that the user might not have.

To solve this, `set`/`multiset` offer alternative functions, which take any type comparable with the value and a functor that should be compatible with the ordering function of the associative container. `unordered_set`/`unordered_multiset` offers functions that take any key type and compatible hash and equality functions. Now, let's see the optimized search function:

```
// These compare Expensive and a c-string
struct StrExpComp
{
   bool operator()(const char *str, const Expensive &c) const
   {  return std::strcmp(str, c.get_key().c_str()) < 0;  }

   bool operator()(const Expensive &c, const char *str) const
   {  return std::strcmp(c.get_key().c_str(), str) < 0;  }
};

struct StrExpEqual
{
   bool operator()(const char *str, const Expensive &c) const
   {  return std::strcmp(str, c.get_key().c_str()) == 0;  }

   bool operator()(const Expensive &c, const char *str) const
   {  return std::strcmp(c.get_key().c_str(), str) == 0;  }
};

// Optimized search functions
Expensive *get_from_set_optimized(const char* key, Set &set_object)
{
   Set::iterator it = set_object.find(key, StrExpComp());
   if( it == set_object.end() )   return 0;
   return &*it;
}

Expensive *get_from_uset_optimized(const char* key, UnorderedSet &uset_object)
{
   UnorderedSet::iterator it = uset_object.find(key, StrHasher(), StrExpEqual());
   if( it == uset_object.end() )  return 0;
   return &*it;
}
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

This new arbitrary key overload is also available for other functions taking values as arguments:

- equal_range
- lower_bound
- upper_bound
- count
- find
- erase

Check `set`, `multiset`, `unordered_set`, `unordered_multiset` references to know more about those functions.

Advanced insertions

A similar issue happens with insertions in simple ordered and unordered associative containers with unique keys (`std::set` and `std::tr1::unordered_set`). In these containers, if a value is already present, the value to be inserted is discarded. With expensive values, if the value is already present, we can suffer efficiency problems.

`set` and `unordered_set` have insertion functions to check efficiently, without constructing the value, if a value is present or not and if it's not present, a function to insert it immediately without any further lookup. For example, using the same `Expensive` class, this function can be inefficient:

```
// Insertion functions
bool insert_to_set(const char* key, Set &set_object)
{
   Expensive *pobject = new Expensive(key);
   bool success = set_object.insert(*pobject).second;
   if(!success)   delete pobject;
   return success;
}

bool insert_to_uset(const char* key, UnorderedSet &uset_object)
{
   Expensive *pobject = new Expensive(key);
   bool success = uset_object.insert(*pobject).second;
   if(!success)   delete pobject;
   return success;
}
```

If the object is already present, we are constructing an `Expensive` that will be discarded, and this is a waste of resources. Instead of that, let's use `insert_check` and `insert_commit` functions:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
// Optimized insertion functions
bool insert_to_set_optimized(const char* key, Set &set_object)
{
   Set::insert_commit_data insert_data;
   bool success = set_object.insert_check(key, StrExpComp(), insert_data).second;
   if(success) set_object.insert_commit(*new Expensive(key), insert_data);
   return success;
}

bool insert_to_uset_optimized(const char* key, UnorderedSet &uset_object)
{
   UnorderedSet::insert_commit_data insert_data;
   bool success = uset_object.insert_check
      (key, StrHasher(), StrExpEqual(), insert_data).second;
   if(success) uset_object.insert_commit(*new Expensive(key), insert_data);
   return success;
}
```

`insert_check` is similar to a normal `insert` but:

- `insert_check` can be used with arbitrary keys
- if the insertion is possible (there is no equivalent value) `insert_check` collects all the needed information in an `insert_commit_data` structure, so that `insert_commit`:
  - **does not execute** further comparisons
  - can be executed with **constant-time complexity**
  - has **no-throw guarantee**.

These functions must be used with care, since no other insertion or erasure must be executed between an `insert_check` and an `insert_commit` pair. Otherwise, the behaviour is undefined. `insert_check` and `insert_commit` will come in handy for developers programming efficient non-intrusive associative containers. See `set` and `unordered_set` reference for more information about `insert_check` and `insert_commit`.

With multiple ordered and unordered associative containers (`multiset` and `unordered_multiset`) there is no need for these advanced insertion functions, since insertions are always successful.

Positional insertions

Some ordered associative containers offer low-level functions to bypass ordering checks and insert nodes directly in desired tree positions. These functions are provided for performance reasons when values to be inserted in the container are known to fulfill order (sets and multisets) and uniqueness (sets) invariants. A typical usage of these functions is when intrusive associative containers are used to build non-intrusive containers and the programmer wants to speed up assignments from other associative containers: if the ordering and uniqueness properties are the same, there is no need to waste time checking the position of each source value, because values are already ordered: back insertions will be much more efficient.

Note: These functions **don't check preconditions** so they must used with care. These are functions are low-level operations **will break container invariants if ordering and uniqueness preconditions are not assured by the caller.**

Let's see an example:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
#include <boost/intrusive/set.hpp>
#include <vector>
#include <algorithm>
#include <cassert>

using namespace boost::intrusive;

//A simple class with a set hook
class MyClass : public set_base_hook<>
{
   public:
   int int_;

   MyClass(int i) :  int_(i) {}
   friend bool operator< (const MyClass &a, const MyClass &b)
      {  return a.int_ < b.int_;  }
   friend bool operator> (const MyClass &a, const MyClass &b)
      {  return a.int_ > b.int_;  }
};

int main()
{
   //Create some ORDERED elements
   std::vector<MyClass> values;
   for(int i = 0; i < 100; ++i)  values.push_back(MyClass(i));

   {  //Data is naturally ordered in the vector with the same criteria
      //as multiset's comparison predicate, so we can just push back
      //all elements, which is more efficient than normal insertion
      multiset<MyClass> mset;
      for(int i = 0; i < 100; ++i)  mset.push_back(values[i]);

      //Now check orderd invariant
      multiset<MyClass>::const_iterator next(mset.cbegin()), it(next++);
      for(int i = 0; i < 99; ++i, ++it, ++next) assert(*it < *next);
   }
   {  //Now the correct order for the set is the reverse order
      //so let's push front all elements
      multiset<MyClass, compare< std::greater<MyClass> > > mset;
      for(int i = 0; i < 100; ++i)  mset.push_front(values[i]);

      //Now check orderd invariant
      multiset<MyClass, compare< std::greater<MyClass> > >::
         const_iterator next(mset.cbegin()), it(next++);
      for(int i = 0; i < 99; ++i, ++it, ++next) assert(*it > *next);
   }
   {  //Now push the first and the last and insert the rest
      //before the last position using "insert_before"
      multiset<MyClass> mset;
      mset.insert_before(mset.begin(), values[0]);
      multiset<MyClass>::const_iterator pos =
         mset.insert_before(mset.end(), values[99]);
      for(int i = 1; i < 99; ++i)  mset.insert_before(pos, values[i]);

      //Now check orderd invariant
      multiset<MyClass>::const_iterator next(mset.cbegin()), it(next++);
      for(int i = 0; i < 99; ++i, ++it, ++next) assert(*it < *next);
   }

   return 0;
}
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

For more information about advanced lookup and insertion functions see associative containers' documentation (e.g. `set`, `multiset`, `unordered_set` and `unordered_multiset` references).

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Erasing and disposing values from Boost.Intrusive containers

One of the most tedious tasks when using intrusive containers is the management of the erased elements. When using STL containers, the container itself unlinks and destroys the contained elements, but with intrusive containers, the user must explicitly destroy the object after erasing an element from the container. This makes STL-like functions erasing multiple objects unhelpful: the user can't destroy every erased element. For example, let's take the function `remove_if` from `list`:

```
template<class Pred>
void remove_if(Pred pred);
```

How can the user destroy the elements (say, using `operator delete`) that will be erased according to the predicate? **Boost.Intrusive** containers offer additional functions that take a function object that will be called after the element has been erased from the container. For example, `list` offers:

```
template<class Pred, class Disposer>
void remove_and_dispose_if(Pred pred, Disposer disposer)
```

With this function the user can efficiently remove and destroy elements if the disposer function destroys an object: `remove_and_dispose_if` will call the "disposer" function object for every removed element. `list` offers more functions taking a disposer function object as argument, like `erase_and_dispose`, `clear_and_dispose`, `remove_and_dispose`, etc.

Note that the disposing function does not need to just destroy the object. It can implement any other operation like inserting the remove object in another container. Let's see a small example:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
#include <boost/intrusive/list.hpp>

using namespace boost::intrusive;

//A class that can be inserted in an intrusive list
class my_class : public list_base_hook<>
{
   public:
   my_class(int i)
      :  int_(i)
   {}

   int int_;
   //...
};

//Definition of the intrusive list
typedef list<my_class> my_class_list;

//The predicate function
struct is_even
{
   bool operator()(const my_class &c) const
   {  return 0 == (c.int_ % 2);  }
};

//The disposer object function
struct delete_disposer
{
   void operator()(my_class *delete_this)
   {  delete delete_this;  }
};

int main()
{
   const int MaxElem = 100;

   //Fill all the nodes and insert them in the list
   my_class_list list;

   try{
      //Insert new objects in the container
      for(int i = 0; i < MaxElem; ++i) list.push_back(*new my_class(i));

      //Now use remove_and_dispose_if to erase and delete the objects
      list.remove_and_dispose_if(is_even(), delete_disposer());
   }
   catch(...){
      //If something throws, make sure that all the memory is freed
      list.clear_and_dispose(delete_disposer());
      throw;
   }

   //Dispose remaining elements
   list.erase_and_dispose(list.begin(), list.end(), delete_disposer());
   return 0;
}
```

All **Boost.Intrusive** containers offer these "erase + dispose" additional members for all functions that erase an element from the container.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Cloning Boost.Intrusive containers

As previously mentioned, **Boost.Intrusive** containers are **non-copyable and non-assignable**, because intrusive containers don't allocate memory at all. To implement a copy-constructor or assignment operator, the user must clone one by one all the elements of the container and insert them in another intrusive container. However, cloning by hand is usually more inefficient than a member cloning function and a specialized cloning function can offer more guarantees than the manual cloning (better exception safety guarantees, for example).

To ease the implementation of copy constructors and assignment operators of classes containing **Boost.Intrusive** containers, all **Boost.Intrusive** containers offer a special cloning function called `clone_from`.

Apart from the container to be cloned, `clone_from` takes two function objects as arguments. For example, consider the `clone_from` member function of `list`:

```
template <class Cloner, class Disposer>
void clone_from(const list &src, Cloner cloner, Disposer disposer);
```

This function will make `*this` a clone of `src`. Let's explain the arguments:

- The first parameter is the list to be cloned.
- The second parameter is a function object that will clone `value_type` objects and return a pointer to the clone. It must implement the following function: `pointer operator()(const value_type &)`.
- The second parameter is a function object that will dispose `value_type` objects. It's used first to empty the container before cloning and to dispose the elements if an exception is thrown.

The cloning function works as follows:

- First it clears and disposes all the elements from *this using the disposer function object.
- After that it starts cloning all the elements of the source container using the cloner function object.
- If any operation in the cloning function (for example, the cloner function object) throws, all the constructed elements are disposed using the disposer function object.

Here is an example of `clone_from`:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
#include <boost/intrusive/list.hpp>
#include <iostream>
#include <vector>

using namespace boost::intrusive;

//A class that can be inserted in an intrusive list
class my_class : public list_base_hook<>
{
   public:
   friend bool operator==(const my_class &a, const my_class &b)
   {  return a.int_ == b.int_;   }

   int int_;

   //...
};

//Definition of the intrusive list
typedef list<my_class> my_class_list;

//Cloner object function
struct new_cloner
{
   my_class *operator()(const my_class &clone_this)
   {  return new my_class(clone_this);  }
};

//The disposer object function
struct delete_disposer
{
   void operator()(my_class *delete_this)
   {  delete delete_this;  }
};

int main()
{
   const int MaxElem = 100;
   std::vector<my_class> nodes(MaxElem);

   //Fill all the nodes and insert them in the list
   my_class_list list;

   for(int i = 0; i < MaxElem; ++i) nodes[i].int_ = i;

   list.insert(list.end(), nodes.begin(), nodes.end());

   //Now clone "list" using "new" and "delete" object functions
   my_class_list cloned_list;
   cloned_list.clone_from(list, new_cloner(), delete_disposer());

   //Test that both are equal
   if(cloned_list != list)
      std::cout << "Both lists are different" << std::endl;
   else
      std::cout << "Both lists are equal" << std::endl;

   //Don't forget to free the memory from the second list
   cloned_list.clear_and_dispose(delete_disposer());
   return 0;
}
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Using function hooks

A programmer might find that base or member hooks are not flexible enough in some situations. In some applications it would be optimal to put a hook deep inside a member of a class or just outside the class. **Boost.Intrusive** has an easy option to allow such cases: `function_hook`.

This option is similar to `member_hook` or `base_hook`, but the programmer can specify a function object that tells the container how to obtain a hook from a value and vice versa. The programmer just needs to define the following function object:

```
//This functor converts between value_type and a hook_type
struct Functor
{
   //Required types
   typedef /*impl-defined*/      hook_type;
   typedef /*impl-defined*/      hook_ptr;
   typedef /*impl-defined*/      const_hook_ptr;
   typedef /*impl-defined*/      value_type;
   typedef /*impl-defined*/      pointer;
   typedef /*impl-defined*/      const_pointer;
   //Required static functions
   static hook_ptr to_hook_ptr (value_type &value);
   static const_hook_ptr to_hook_ptr(const value_type &value);
   static pointer to_value_ptr(hook_ptr n);
   static const_pointer to_value_ptr(const_hook_ptr n);
};
```

Converting from values to hooks is generally easy, since most hooks are in practice members or base classes of class data members. The inverse operation is a bit more complicated, but **Boost.Intrusive** offers a bit of help with the function `get_parent_from_member`, which allows easy conversions from the address of a data member to the address of the parent holding that member. Let's see a little example of `function_hook`:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
#include <boost/intrusive/list.hpp>
#include <boost/intrusive/parent_from_member.hpp>

using namespace boost::intrusive;

struct MyClass
{
   int dummy;
   //This internal type has a member hook
   struct InnerNode
   {
      int dummy;
      list_member_hook<> hook;
   } inner;
};

//This functor converts between MyClass and InnerNode's member hook
struct Functor
{
   //Required types
   typedef list_member_hook<>    hook_type;
   typedef hook_type*            hook_ptr;
   typedef const hook_type*      const_hook_ptr;
   typedef MyClass               value_type;
   typedef value_type*           pointer;
   typedef const value_type*     const_pointer;

   //Required static functions
   static hook_ptr to_hook_ptr (value_type &value)
      {  return &value.inner.hook; }
   static const_hook_ptr to_hook_ptr(const value_type &value)
      {  return &value.inner.hook; }
   static pointer to_value_ptr(hook_ptr n)
   {
      return get_parent_from_member<MyClass>
         (get_parent_from_member<MyClass::InnerNode>(n, &MyClass::InnerNode::hook)
         ,&MyClass::inner
      );
   }
   static const_pointer to_value_ptr(const_hook_ptr n)
   {
      return get_parent_from_member<MyClass>
         (get_parent_from_member<MyClass::InnerNode>(n, &MyClass::InnerNode::hook)
         ,&MyClass::inner
      );
   }
};

//Define a list that will use the hook accessed through the function object
typedef list< MyClass, function_hook< Functor> >  List;

int main()
{
   MyClass n;
   List l;
   //Insert the node in both lists
   l.insert(l.begin(), n);
   assert(l.size() == 1);
   return 0;
}
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Recursive Boost.Intrusive containers

Boost.Intrusive containers can be used to define recursive structures very easily, allowing complex data structures with very low overhead. Let's see an example:

```
#include <boost/intrusive/list.hpp>
#include <cassert>

using namespace boost::intrusive;

typedef list_base_hook<> BaseHook;

//A recursive class
class Recursive : public BaseHook
{
   private:
   Recursive(const Recursive&);
   Recursive & operator=(const Recursive&);

   public:
   Recursive() : BaseHook(), children(){}
   list< Recursive, base_hook<BaseHook> > children;
};

int main()
{
   Recursive f, f2;
   //A recursive list of Recursive
   list< Recursive, base_hook<BaseHook> > l;

   //Insert a node in parent list
   l.insert(l.begin(), f);

   //Insert a node in child list
   l.begin()->children.insert(l.begin()->children.begin(), f2);

   //Objects properly inserted
   assert(l.size() == l.begin()->children.size());
   assert(l.size() == 1);

   //Clear both lists
   l.begin()->children.clear();
   l.clear();
   return 0;
}
```

Recursive data structures using **Boost.Intrusive** containers must avoid using hook deduction to avoid early type instantiation:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
//This leads to compilation error (Recursive is instantiated by
//'list' to deduce hook properties (pointer type, tag, safe-mode...)
class Recursive
{  //...

   list< Recursive > l;
   //...
};

//Ok, programmer must specify the hook type to avoid early Recursive instantiation
class Recursive
{  //...
   list< Recursive, base_hook<BaseHook> > l;
   //...
};
```

Member hooks are not suitable for recursive structures:

```
class Recursive
{
   private:
   Recursive(const Recursive&);
   Recursive & operator=(const Recursive&);

   public:
   list_member_hook<> memhook;
   list< Recursive, member_hook<Recursive, list_member_hook<>, &Recursive::memhook> > children;
};
```

Specifying `&Recursive::memhook` (that is, the offset between memhook and Recursive) provokes an early instantiation of `Recursive`. To define recursive structures using member hooks, a programmer should use `function_hook`:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
#include <boost/intrusive/list.hpp>
#include <boost/intrusive/parent_from_member.hpp>

using namespace boost::intrusive;

class Recursive;

//Declaration of the functor that converts betwen the Recursive
//class and the hook
struct Functor
{
   //Required types
   typedef list_member_hook<>    hook_type;
   typedef hook_type*            hook_ptr;
   typedef const hook_type*      const_hook_ptr;
   typedef Recursive             value_type;
   typedef value_type*           pointer;
   typedef const value_type*     const_pointer;

   //Required static functions
   static hook_ptr to_hook_ptr (value_type &value);
   static const_hook_ptr to_hook_ptr(const value_type &value);
   static pointer to_value_ptr(hook_ptr n);
   static const_pointer to_value_ptr(const_hook_ptr n);
};

//Define the recursive class
class Recursive
{
   private:
   Recursive(const Recursive&);
   Recursive & operator=(const Recursive&);

   public:
   Recursive() : hook(), children() {}
   list_member_hook<> hook;
   list< Recursive, function_hook< Functor> > children;
};

//Definition of Functor functions
inline Functor::hook_ptr Functor::to_hook_ptr (Functor::value_type &value)
   {  return &value.hook; }
inline Functor::const_hook_ptr Functor::to_hook_ptr(const Functor::value_type &value)
   {  return &value.hook; }
inline Functor::pointer Functor::to_value_ptr(Functor::hook_ptr n)
   {  return get_parent_from_member<Recursive>(n, &Recursive::hook);  }
inline Functor::const_pointer Functor::to_value_ptr(Functor::const_hook_ptr n)
   {  return get_parent_from_member<Recursive>(n, &Recursive::hook);  }

int main()
{
   Recursive f, f2;
   //A recursive list of Recursive
   list< Recursive, function_hook< Functor> > l;

   //Insert a node in parent list
   l.insert(l.begin(), f);

   //Insert a node in child list
   l.begin()->children.insert(l.begin()->children.begin(), f2);

   //Objects properly inserted
   assert(l.size() == l.begin()->children.size());
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
   assert(l.size() == 1);

   //Clear both lists
   l.begin()->children.clear();
   l.clear();
   return 0;
}
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Using smart pointers with Boost.Intrusive containers

Boost.Intrusive hooks can be configured to use other pointers than raw pointers. When a **Boost.Intrusive** hook is configured with a smart pointer as an argument, this pointer configuration is passed to the containers. For example, if the following hook is configured with a smart pointer (for example, an offset pointer from **Boost.Interprocess**):

```
#include <boost/intrusive/list.hpp>
#include <boost/interprocess/offset_ptr.hpp>

using namespace boost::intrusive;
namespace ip = boost::interprocess;

class shared_memory_data
   //Declare the hook with an offset_ptr from Boost.Interprocess
   //to make this class compatible with shared memory
   :  public list_base_hook< void_pointer< ip::offset_ptr<void> > >
{
   int data_id_;
   public:

   int get() const    {  return data_id_;  }
   void set(int id)  {  data_id_ = id;     }
};
```

Any intrusive list constructed using this hook will be ready for shared memory, because the intrusive list will also use offset pointers internally. For example, we can create an intrusive list in shared memory combining **Boost.Interprocess** and **Boost.Intrusive**:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
#include <boost/interprocess/managed_shared_memory.hpp>
#include <boost/interprocess/containers/vector.hpp>
#include <boost/interprocess/containers/list.hpp>
#include <boost/interprocess/allocators/allocator.hpp>

//Definition of the shared memory friendly intrusive list
typedef list<shared_memory_data> intrusive_list_t;

int main()
{
   //Now create an intrusive list in shared memory:
   //nodes and the container itself must be created in shared memory
   const int MaxElem    = 100;
   const int ShmSize    = 50000;
   const char *ShmName  = get_shared_memory_name();
   {
      //Erase all old shared memory
      ip::shared_memory_object::remove(ShmName);
      ip::managed_shared_memory shm(ip::create_only, ShmName, ShmSize);

      //Create all nodes in shared memory using a shared memory vector
      //See Boost.Interprocess documentation for more information on this
      typedef ip::allocator
         < shared_memory_data, ip::managed_shared_memory::segment_manager>
            shm_allocator_t;
      typedef ip::vector<shared_memory_data, shm_allocator_t> shm_vector_t;
      shm_allocator_t shm_alloc(shm.get_segment_manager());
      shm_vector_t *pshm_vect =
         shm.construct<shm_vector_t>(ip::anonymous_instance)(shm_alloc);
      pshm_vect->resize(MaxElem);

      //Initialize all the nodes
      for(int i = 0; i < MaxElem; ++i)    (*pshm_vect)[i].set(i);

      //Now create the shared memory intrusive list
      intrusive_list_t *plist = shm.construct<intrusive_list_t>(ip::anonymous_instance)();

      //Insert objects stored in shared memory vector in the intrusive list
      plist->insert(plist->end(), pshm_vect->begin(), pshm_vect->end());

      //Check all the inserted nodes
      int checker = 0;
      for( intrusive_list_t::const_iterator it = plist->begin(), itend(plist->end())
         ; it != itend; ++it, ++checker){
         if(it->get() != checker)   return false;
      }

      //Now delete the list and after that, the nodes
      shm.destroy_ptr(plist);
      shm.destroy_ptr(pshm_vect);
   }
   ip::shared_memory_object::remove(ShmName);
   return 0;
}
```

Requirements for smart pointers compatible with Boost.Intrusive

Not every smart pointer is compatible with **Boost.Intrusive**:

- It must be compatible with C++11 `std::pointer_traits` requirements. **Boost.Intrusive** uses its own `pointer_traits` class to implement those features in both C++11 and C++03 compilers.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

- It must have the same ownership semantics as a raw pointer. This means that resource management smart pointers (like `boost::shared_ptr`) can't be used.

The conversion from the smart pointer to a raw pointer will be implemented as a recursive call to `operator->()` until the function returns a raw pointer.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Obtaining iterators from values

Boost.Intrusive offers another useful feature that's not present in STL containers: it's possible to obtain an iterator to a value from the value itself. This feature is implemented in **Boost.Intrusive** containers by a function called `iterator_to`:

```
iterator iterator_to(reference value);
const_iterator iterator_to(const_reference value);
```

For **Boost.Intrusive** containers that have local iterators, like unordered associative containers, we can also obtain local iterators:

```
local_iterator local_iterator_to(reference value);
const_local_iterator local_iterator_to(const_reference value) const;
```

For most **Boost.Intrusive** containers (`list`, `slist`, `set`, `multiset`) we have an alternative static `s_iterator_to` function.

For unordered associative containers (`unordered_set`, `multiset`), `iterator_to` has no static alternative function. On the other hand, `local_iterator_to` functions have their `s_local_iterator_to` static alternatives.

Alternative static functions are available under certain circumstances explained in the Stateful value traits section; if the programmer uses hooks provided by **Boost.Intrusive**, those functions will be available.

Let's see a small function that shows the use of `iterator_to` and `local_iterator_to`:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
#include <boost/intrusive/list.hpp>
#include <boost/intrusive/unordered_set.hpp>
#include <boost/functional/hash.hpp>
#include <vector>

using namespace boost::intrusive;

class intrusive_data
{
   int data_id_;
   public:

   void set(int id)  {  data_id_ = id;    }

   //This class can be inserted in an intrusive list
   list_member_hook<>   list_hook_;

   //This class can be inserted in an intrusive unordered_set
   unordered_set_member_hook<>   unordered_set_hook_;

   //Comparison operators
   friend bool operator==(const intrusive_data &a, const intrusive_data &b)
   {  return a.data_id_ == b.data_id_; }

   friend bool operator!=(const intrusive_data &a, const intrusive_data &b)
   {  return a.data_id_ != b.data_id_; }

   //The hash function
   friend std::size_t hash_value(const intrusive_data &i)
   {  return boost::hash<int>()(i.data_id_);  }
};

//Definition of the intrusive list that will hold intrusive_data
typedef member_hook<intrusive_data, list_member_hook<>
        , &intrusive_data::list_hook_> MemberListOption;
typedef list<intrusive_data, MemberListOption> list_t;

//Definition of the intrusive unordered_set that will hold intrusive_data
typedef member_hook
      < intrusive_data, unordered_set_member_hook<>
      , &intrusive_data::unordered_set_hook_> MemberUsetOption;
typedef boost::intrusive::unordered_set
   < intrusive_data, MemberUsetOption> unordered_set_t;

int main()
{
   //Create MaxElem objects
   const int MaxElem = 100;
   std::vector<intrusive_data> nodes(MaxElem);

   //Declare the intrusive containers
   list_t     list;
   unordered_set_t::bucket_type buckets[MaxElem];
   unordered_set_t  unordered_set
      (unordered_set_t::bucket_traits(buckets, MaxElem));

   //Initialize all the nodes
   for(int i = 0; i < MaxElem; ++i) nodes[i].set(i);

   //Now insert them in both intrusive containers
   list.insert(list.end(), nodes.begin(), nodes.end());
   unordered_set.insert(nodes.begin(), nodes.end());
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
   //Now check the iterator_to function
   list_t::iterator list_it(list.begin());
   for(int i = 0; i < MaxElem; ++i, ++list_it)
      if(list.iterator_to(nodes[i])       != list_it ||
         list_t::s_iterator_to(nodes[i])  != list_it)
         return 1;

   //Now check unordered_set::s_iterator_to (which is a member function)
   //and unordered_set::s_local_iterator_to (which is an static member function)
   unordered_set_t::iterator unordered_set_it(unordered_set.begin());
   for(int i = 0; i < MaxElem; ++i){
      unordered_set_it = unordered_set.find(nodes[i]);
      if(unordered_set.iterator_to(nodes[i]) != unordered_set_it)
         return 1;
      if(*unordered_set.local_iterator_to(nodes[i])      != *unordered_set_it ||
         *unordered_set_t::s_local_iterator_to(nodes[i]) != *unordered_set_it )
         return 1;
   }

   return 0;
}
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Any Hooks: A single hook for any Intrusive container

Sometimes, a class programmer wants to place a class in several intrusive containers but no at the same time. In this case, the programmer might decide to insert two hooks in the same class.

```
class MyClass
   : public list_base_hook<>, public slist_base_hook<> //...
{};
```

However, there is a more size-efficient alternative in **Boost.Intrusive**: "any" hooks (`any_base_hook` and `any_member_hook`). These hooks can be used to store a type in several containers offered by **Boost.Intrusive** minimizing the size of the class.

These hooks support these options:

- **`tag<class Tag>`** (for base hooks only): This argument serves as a tag, so you can derive from more than one slist hook. Default: `tag<default_tag>`.
- **`link_mode<link_mode_type LinkMode>`**: The linking policy. `link_mode<auto_unlink>` is **not** supported and `link_mode<safe_mode>` might offer weaker error detection in any hooks than in other hooks. Default: `link_mode<safe_link>`.
- **`void_pointer<class VoidPointer>`**: The pointer type to be used internally in the hook and propagated to the container. Default: `void_pointer<void*>`.

`auto_unlink` can't be supported because the hook does not know in which type of container might be currently inserted. Additionally, these hooks don't support `unlink()` and `swap_nodes()` operations for the same reason.

Here is an example that creates a class with two any hooks, and uses one to insert the class in a `slist` and the other one in a `list`.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
#include <vector>
#include <boost/intrusive/any_hook.hpp>
#include <boost/intrusive/slist.hpp>
#include <boost/intrusive/list.hpp>

using namespace boost::intrusive;

class MyClass : public any_base_hook<> //Base hook
{
   int int_;

   public:
   any_member_hook<> member_hook_;  //Member hook

   MyClass(int i = 0) : int_(i)
   {}
};

int main()
{
   //Define a base hook option that converts any_base_hook to a slist hook
   typedef any_to_slist_hook < base_hook< any_base_hook<> > >     BaseSlistOption;
   typedef slist<MyClass, BaseSlistOption>                        BaseSList;

   //Define a member hook option that converts any_member_hook to a list hook
   typedef any_to_list_hook< member_hook
         < MyClass, any_member_hook<>, &MyClass::member_hook_> >  MemberListOption;
   typedef list<MyClass, MemberListOption>                        MemberList;

   //Create several MyClass objects, each one with a different value
   std::vector<MyClass> values;
   for(int i = 0; i < 100; ++i){ values.push_back(MyClass(i)); }

   BaseSList base_slist;   MemberList member_list;

   //Now insert them in reverse order in the slist and in order in the list
   for(std::vector<MyClass>::iterator it(values.begin()), itend(values.end()); it != itend; ++it)
      base_slist.push_front(*it), member_list.push_back(*it);

   //Now test lists
   BaseSList::iterator bit(base_slist.begin());
   MemberList::reverse_iterator mrit(member_list.rbegin());
   std::vector<MyClass>::reverse_iterator rit(values.rbegin()), ritend(values.rend());

   //Test the objects inserted in the base hook list
   for(; rit != ritend; ++rit, ++bit, ++mrit)
      if(&*bit != &*rit || &*mrit != &*rit)  return 1;
   return 0;
}
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Concepts explained

This section will expand the explanation of previously presented basic concepts before explaining the customization options of **Boost.Intrusive**.

- **Node Algorithms**: A set of static functions that implement basic operations on a group of nodes: initialize a node, link_mode_type a node to a group of nodes, unlink a node from another group of nodes, etc. For example, a circular singly linked list is a group of nodes, where each node has a pointer to the next node. **Node Algorithms** just require a **NodeTraits** template parameter and they can work with any **NodeTraits** class that fulfills the needed interface. As an example, here is a class that implements operations to manage a group of nodes forming a circular singly linked list:

```
template<class NodeTraits>
struct my_slist_algorithms
{
   typedef typename NodeTraits::node_ptr       node_ptr;
   typedef typename NodeTraits::const_node_ptr const_node_ptr;

   //Get the previous node of "this_node"
   static node_ptr get_prev_node(node_ptr this_node)
   {
      node_ptr p = this_node;
      while (this_node != NodeTraits::get_next(p))
         p = NodeTraits::get_next(p);
      return p;
   }

   // number of elements in the group of nodes containing "this_node"
   static std::size_t count(const_node_ptr this_node)
   {
      std::size_t result = 0;
      const_node_ptr p = this_node;
      do{
         p = NodeTraits::get_next(p);
         ++result;
      } while (p != this_node);
      return result;
   }

   // More operations
   // ...
};
```

- **Node Traits**: A class that encapsulates the basic information and operations on a node within a group of nodes: the type of the node, a function to obtain the pointer to the next node, etc. **Node Traits** specify the configuration information **Node Algorithms** need. Each type of **Node Algorithm** expects an interface that compatible **Node Traits** classes must implement. As an example, this is the definition of a **Node Traits** class that is compatible with the previously presented `my_slist_algorithms`:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
struct my_slist_node_traits
{
   //The type of the node
   struct node
   {
      node *next_;
   };

   typedef node *       node_ptr;
   typedef const node * const_node_ptr;

   //A function to obtain a pointer to the next node
   static node_ptr get_next(const_node_ptr n)
   {  return n->next_;  }

   //A function to set the pointer to the next node
   static void set_next(node_ptr n, node_ptr next)
   {  n->next_ = next;  }
};
```

- **Hook**: A class that the user must add as a base class or as a member to his own class to make that class insertable in an intrusive container. Usually the hook contains a node object that will be used to form the group of nodes: For example, the following class is a **Hook** that the user can add as a base class, to make the user class compatible with a singly linked list container:

```
class my_slist_base_hook
      //This hook contains a node, that will be used
      //to link the user object in the group of nodes
   : private my_slist_node_traits::node
{
   typedef my_slist_node_traits::node_ptr       node_ptr;
   typedef my_slist_node_traits::const_node_ptr const_node_ptr;

   //Converts the generic node to the hook
   static my_slist_base_hook *to_hook_ptr(node_ptr p)
   {  return static_cast<my_slist_base_hook*>(p); }

   //Returns the generic node stored by this hook
   node_ptr to_node_ptr()
   { return static_cast<node *const>(this); }

   // More operations
   // ...
};

//To make MyClass compatible with an intrusive singly linked list
//derive our class from the hook.
class MyClass
   :  public my_slist_base_hook
{
   void set(int value);
   int get() const;

   private:
   int value_;
};
```

- **Intrusive Container**: A container that offers a STL-like interface to store user objects. An intrusive container can be templatized to store different value types that use different hooks. An intrusive container is also more elaborate than a group of nodes: it can store the number of elements to achieve constant-time size information, it can offer debugging facilities, etc. For example, an `slist` container (intrusive singly linked list) should be able to hold `MyClass` objects that might have decided to store the hook as a base class or as a member. Internally, the container will use **Node Algorithms** to implement its operations, and an intrusive

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

container is configured using a template parameter called **ValueTraits**. **ValueTraits** will contain the information to convert user classes in nodes compatible with **Node Algorithms**. For example, this a possible `slist` implementation:

```
template<class ValueTraits, ...>
class slist
{
   public:
   typedef typename ValueTraits::value_type value_type;

   //More typedefs and functions
   // ...

   //Insert the value as the first element of the list
   void push_front (reference value)
   {
      node_ptr to_insert(ValueTraits::to_node_ptr(value));
      circular_list_algorithms::link_after(to_insert, get_root_node());
   }

   // More operations
   // ...
};
```

- **Semi-Intrusive Container**: A semi-intrusive container is similar to an intrusive container, but apart from the values to be inserted in the container, it needs additional memory (for example, auxiliary arrays or indexes).
- **Value Traits**: As we can see, to make our classes intrusive-friendly we add a simple hook as a member or base class. The hook contains a generic node that will be inserted in a group of nodes. **Node Algorithms** just work with nodes and don't know anything about user classes. On the other hand, an intrusive container needs to know how to obtain a node from a user class, and also the inverse operation. So we can define **ValueTraits** as the glue between user classes and nodes required by **Node Algorithms**. Let's see a possible implementation of a value traits class that glues MyClass and the node stored in the hook:

```
struct my_slist_derivation_value_traits
{
   public:
   typedef slist_node_traits           node_traits;
   typedef MyClass                     value_type;
   typedef node_traits::node_ptr       node_ptr;
   typedef value_type*                 pointer;
   //...

   //Converts user's value to a generic node
   static node_ptr to_node_ptr(reference value)
   { return static_cast<slist_base_hook &>(value).to_node_ptr(); }

   //Converts a generic node into user's value
   static value_type *to_value_ptr(node_traits::node *n)
   { static_cast<value_type*>(slist_base_hook::to_hook_ptr(n)); }

   // More operations
   // ...
};
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Node algorithms with custom NodeTraits

As explained in the Concepts section, **Boost.Intrusive** containers are implemented using node algorithms that work on generic nodes.

Sometimes, the use of intrusive containers is expensive for some environments and the programmer might want to avoid all the template instantiations related to **Boost.Intrusive** containers. However, the user can still benefit from **Boost.Intrusive** using the node algorithms, because some of those algorithms, like red-black tree algorithms, are not trivial to write.

All node algorithm classes are templatized by a `NodeTraits` class. This class encapsulates the needed internal type declarations and operations to make a node compatible with node algorithms. Each type of node algorithms has its own requirements:

Intrusive singly linked list algorithms

These algorithms are static members of the `circular_slist_algorithms` class:

```
template<class NodeTraits>
struct circular_slist_algorithms;
```

An empty list is formed by a node whose pointer to the next node points to itself. `circular_slist_algorithms` is configured with a NodeTraits class, which encapsulates the information about the node to be manipulated. NodeTraits must support the following interface:

Typedefs:

- `node`: The type of the node that forms the circular list
- `node_ptr`: The type of a pointer to a node (usually node*)
- `const_node_ptr`: The type of a pointer to a const node (usually const node*)

Static functions:

- `static node_ptr get_next(const_node_ptr n);`: Returns a pointer to the next node stored in "n".
- `static void set_next(node_ptr n, node_ptr next);`: Sets the pointer to the next node stored in "n" to "next".

Once we have a node traits configuration we can use **Boost.Intrusive** algorithms with our nodes:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
#include <boost/intrusive/circular_slist_algorithms.hpp>
#include <cassert>

struct my_node
{
   my_node *next_;
   //other members...
};

//Define our own slist_node_traits
struct my_slist_node_traits
{
   typedef my_node                                  node;
   typedef my_node *                                node_ptr;
   typedef const my_node *                          const_node_ptr;
   static node_ptr get_next(const_node_ptr n)      {  return n->next_;  }
   static void set_next(node_ptr n, node_ptr next) {  n->next_ = next;  }
};

int main()
{
   typedef boost::intrusive::circular_slist_algorithms<my_slist_node_traits> algo;
   my_node one, two, three;

   //Create an empty singly linked list container:
   //"one" will be the first node of the container
   algo::init_header(&one);
   assert(algo::count(&one) == 1);

   //Now add a new node
   algo::link_after(&one, &two);
   assert(algo::count(&one) == 2);

   //Now add a new node after "one"
   algo::link_after(&one, &three);
   assert(algo::count(&one) == 3);

   //Now unlink the node after one
   algo::unlink_after(&one);
   assert(algo::count(&one) == 2);

   //Now unlink two
   algo::unlink(&two);
   assert(algo::count(&one) == 1);

   return 0;
}
```

For a complete list of functions see `circular_slist_algorithms reference`.

Intrusive doubly linked list algorithms

These algorithms are static members of the `circular_list_algorithms` class:

```
template<class NodeTraits>
struct circular_list_algorithms;
```

An empty list is formed by a node whose pointer to the next node points to itself. `circular_list_algorithms` is configured with a NodeTraits class, which encapsulates the information about the node to be manipulated. NodeTraits must support the following interface:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Typedefs:

- `node`: The type of the node that forms the circular list
- `node_ptr`: The type of a pointer to a node (usually node*)
- `const_node_ptr`: The type of a pointer to a const node (usually const node*)

Static functions:

- `static node_ptr get_next(const_node_ptr n);`: Returns a pointer to the next node stored in "n".
- `static void set_next(node_ptr n, node_ptr next);`: Sets the pointer to the next node stored in "n" to "next".
- `static node_ptr get_previous(const_node_ptr n);`: Returns a pointer to the previous node stored in "n".
- `static void set_previous(node_ptr n, node_ptr prev);`: Sets the pointer to the previous node stored in "n" to "prev".

Once we have a node traits configuration we can use **Boost.Intrusive** algorithms with our nodes:

```
#include <boost/intrusive/circular_list_algorithms.hpp>
#include <cassert>

struct my_node
{
   my_node *next_, *prev_;
   //other members...
};

//Define our own list_node_traits
struct my_list_node_traits
{
   typedef my_node                                    node;
   typedef my_node *                                  node_ptr;
   typedef const my_node *                            const_node_ptr;
   static node_ptr get_next(const_node_ptr n)         {  return n->next_;  }
   static void set_next(node_ptr n, node_ptr next)    {  n->next_ = next;  }
   static node *get_previous(const_node_ptr n)        {  return n->prev_;  }
   static void set_previous(node_ptr n, node_ptr prev){  n->prev_ = prev;  }
};

int main()
{
   typedef boost::intrusive::circular_list_algorithms<my_list_node_traits> algo;
   my_node one, two, three;

   //Create an empty doubly linked list container:
   //"one" will be the first node of the container
   algo::init_header(&one);
   assert(algo::count(&one) == 1);

   //Now add a new node before "one"
   algo::link_before(&one, &two);
   assert(algo::count(&one) == 2);

   //Now add a new node after "two"
   algo::link_after(&two, &three);
   assert(algo::count(&one) == 3);

   //Now unlink the node after one
   algo::unlink(&three);
   assert(algo::count(&one) == 2);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
   //Now unlink two
   algo::unlink(&two);
   assert(algo::count(&one) == 1);

   //Now unlink one
   algo::unlink(&one);
   assert(algo::count(&one) == 1);

   return 0;
}
```

For a complete list of functions see `circular_list_algorithms reference`.

Intrusive red-black tree algorithms

These algorithms are static members of the `rbtree_algorithms` class:

```
template<class NodeTraits>
struct rbtree_algorithms;
```

An empty tree is formed by a node whose pointer to the parent node is null, the left and right node pointers point to itself, and whose color is red. `rbtree_algorithms` is configured with a NodeTraits class, which encapsulates the information about the node to be manipulated. NodeTraits must support the following interface:

Typedefs:

- `node`: The type of the node that forms the circular rbtree
- `node_ptr`: The type of a pointer to a node (usually node*)
- `const_node_ptr`: The type of a pointer to a const node (usually const node*)
- `color`: The type that can store the color of a node

Static functions:

- `static node_ptr get_parent(const_node_ptr n);`: Returns a pointer to the parent node stored in "n".
- `static void set_parent(node_ptr n, node_ptr p);`: Sets the pointer to the parent node stored in "n" to "p".
- `static node_ptr get_left(const_node_ptr n);`: Returns a pointer to the left node stored in "n".
- `static void set_left(node_ptr n, node_ptr l);`: Sets the pointer to the left node stored in "n" to "l".
- `static node_ptr get_right(const_node_ptr n);`: Returns a pointer to the right node stored in "n".
- `static void set_right(node_ptr n, node_ptr r);`: Sets the pointer to the right node stored in "n" to "r".
- `static color get_color(const_node_ptr n);`: Returns the color stored in "n".
- `static void set_color(node_ptr n, color c);`: Sets the color stored in "n" to "c".
- `static color black();`: Returns a value representing the black color.
- `static color red();`: Returns a value representing the red color.

Once we have a node traits configuration we can use **Boost.Intrusive** algorithms with our nodes:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
#include <boost/intrusive/rbtree_algorithms.hpp>
#include <cassert>

struct my_node
{
   my_node(int i = 0)
      :  int_(i)
   {}
   my_node *parent_, *left_, *right_;
   int      color_;
   //other members
   int      int_;
};

//Define our own rbtree_node_traits
struct my_rbtree_node_traits
{
   typedef my_node                                    node;
   typedef my_node *                                  node_ptr;
   typedef const my_node *                            const_node_ptr;
   typedef int                                        color;
   static node_ptr get_parent(const_node_ptr n)       {  return n->parent_;   }
   static void set_parent(node_ptr n, node_ptr parent){  n->parent_ = parent; }
   static node_ptr get_left(const_node_ptr n)         {  return n->left_;     }
   static void set_left(node_ptr n, node_ptr left)    {  n->left_ = left;     }
   static node_ptr get_right(const_node_ptr n)        {  return n->right_;    }
   static void set_right(node_ptr n, node_ptr right)  {  n->right_ = right;   }
   static color get_color(const_node_ptr n)           {  return n->color_;    }
   static void set_color(node_ptr n, color c)         {  n->color_ = c;       }
   static color black()                               {  return color(0);     }
   static color red()                                 {  return color(1);     }
};

struct node_ptr_compare
{
   bool operator()(const my_node *a, const my_node *b)
   {  return a->int_ < b->int_;  }
};

int main()
{
   typedef boost::intrusive::rbtree_algorithms<my_rbtree_node_traits> algo;
   my_node header, two(2), three(3);

   //Create an empty rbtree container:
   //"header" will be the header node of the tree
   algo::init_header(&header);

   //Now insert node "two" in the tree using the sorting functor
   algo::insert_equal_upper_bound(&header, &two, node_ptr_compare());

   //Now insert node "three" in the tree using the sorting functor
   algo::insert_equal_lower_bound(&header, &three, node_ptr_compare());

   //Now take the first node (the left node of the header)
   my_node *n = header.left_;
   assert(n == &two);

   //Now go to the next node
   n = algo::next_node(n);
   assert(n == &three);

   //Erase a node just using a pointer to it
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
   algo::unlink(&two);

   //Erase a node using also the header (faster)
   algo::erase(&header, &three);
   return 0;
}
```

For a complete list of functions see `rbtree_algorithms reference`.

Intrusive splay tree algorithms

These algorithms are static members of the `splaytree_algorithms` class:

```
template<class NodeTraits>
struct splaytree_algorithms;
```

An empty tree is formed by a node whose pointer to the parent node is null, and whose left and right nodes pointers point to itself. `splaytree_algorithms` is configured with a NodeTraits class, which encapsulates the information about the node to be manipulated. NodeTraits must support the following interface:

Typedefs:

- `node`: The type of the node that forms the circular splaytree
- `node_ptr`: The type of a pointer to a node (usually node*)
- `const_node_ptr`: The type of a pointer to a const node (usually const node*)

Static functions:

- `static node_ptr get_parent(const_node_ptr n);`: Returns a pointer to the parent node stored in "n".
- `static void set_parent(node_ptr n, node_ptr p);`: Sets the pointer to the parent node stored in "n" to "p".
- `static node_ptr get_left(const_node_ptr n);`: Returns a pointer to the left node stored in "n".
- `static void set_left(node_ptr n, node_ptr l);`: Sets the pointer to the left node stored in "n" to "l".
- `static node_ptr get_right(const_node_ptr n);`: Returns a pointer to the right node stored in "n".
- `static void set_right(node_ptr n, node_ptr r);`: Sets the pointer to the right node stored in "n" to "r".

Once we have a node traits configuration we can use **Boost.Intrusive** algorithms with our nodes:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
#include <boost/intrusive/splaytree_algorithms.hpp>
#include <cassert>

struct my_node
{
   my_node(int i = 0)
      :  int_(i)
   {}
   my_node *parent_, *left_, *right_;
   //other members
   int      int_;
};

//Define our own splaytree_node_traits
struct my_splaytree_node_traits
{
   typedef my_node                                        node;
   typedef my_node *                                      node_ptr;
   typedef const my_node *                                const_node_ptr;

   static node_ptr get_parent(const_node_ptr n)       {  return n->parent_;   }
   static void set_parent(node_ptr n, node_ptr parent){  n->parent_ = parent; }
   static node_ptr get_left(const_node_ptr n)         {  return n->left_;     }
   static void set_left(node_ptr n, node_ptr left)    {  n->left_ = left;     }
   static node_ptr get_right(const_node_ptr n)        {  return n->right_;    }
   static void set_right(node_ptr n, node_ptr right)  {  n->right_ = right;   }
};

struct node_ptr_compare
{
   bool operator()(const my_node *a, const my_node *b)
   {  return a->int_ < b->int_;  }
};

int main()
{
   typedef boost::intrusive::splaytree_algorithms<my_splaytree_node_traits> algo;
   my_node header, two(2), three(3);

   //Create an empty splaytree container:
   //"header" will be the header node of the tree
   algo::init_header(&header);

   //Now insert node "two" in the tree using the sorting functor
   algo::insert_equal_upper_bound(&header, &two, node_ptr_compare());

   //Now insert node "three" in the tree using the sorting functor
   algo::insert_equal_lower_bound(&header, &three, node_ptr_compare());

   //Now take the first node (the left node of the header)
   my_node *n = header.left_;
   assert(n == &two);

   //Now go to the next node
   n = algo::next_node(n);
   assert(n == &three);

   //Erase a node just using a pointer to it
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
   algo::unlink(&two);

   //Erase a node using also the header (faster)
   algo::erase(&header, &three);
   return 0;
}
```

For a complete list of functions see `splaytree_algorithms reference`.

Intrusive avl tree algorithms

`avltree_algorithms` have the same interface as `rbtree_algorithms`.

```
template<class NodeTraits>
struct avltree_algorithms;
```

`avltree_algorithms` is configured with a NodeTraits class, which encapsulates the information about the node to be manipulated. NodeTraits must support the following interface:

Typedefs:

- `node`: The type of the node that forms the circular avltree
- `node_ptr`: The type of a pointer to a node (usually node*)
- `const_node_ptr`: The type of a pointer to a const node (usually const node*)
- `balance`: A type that can represent 3 balance types (usually an integer)

Static functions:

- `static node_ptr get_parent(const_node_ptr n);`: Returns a pointer to the parent node stored in "n".
- `static void set_parent(node_ptr n, node_ptr p);`: Sets the pointer to the parent node stored in "n" to "p".
- `static node_ptr get_left(const_node_ptr n);`: Returns a pointer to the left node stored in "n".
- `static void set_left(node_ptr n, node_ptr l);`: Sets the pointer to the left node stored in "n" to "l".
- `static node_ptr get_right(const_node_ptr n);`: Returns a pointer to the right node stored in "n".
- `static void set_right(node_ptr n, node_ptr r);`: Sets the pointer to the right node stored in "n" to "r".
- `static balance get_balance(const_node_ptr n);`: Returns the balance factor stored in "n".
- `static void set_balance(node_ptr n, balance b);`: Sets the balance factor stored in "n" to "b".
- `static balance negative();`: Returns a value representing a negative balance factor.
- `static balance zero();`: Returns a value representing a zero balance factor.
- `static balance positive();`: Returns a value representing a positive balance factor.

Once we have a node traits configuration we can use **Boost.Intrusive** algorithms with our nodes:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
#include <boost/intrusive/avltree_algorithms.hpp>
#include <cassert>

struct my_node
{
   my_node(int i = 0)
      :  int_(i)
   {}
   my_node *parent_, *left_, *right_;
   int balance_;
   //other members
   int      int_;
};

//Define our own avltree_node_traits
struct my_avltree_node_traits
{
   typedef my_node                                    node;
   typedef my_node *                                  node_ptr;
   typedef const my_node *                            const_node_ptr;
   typedef int                                        balance;

   static node_ptr get_parent(const_node_ptr n)       {  return n->parent_;   }
   static void set_parent(node_ptr n, node_ptr parent){  n->parent_ = parent; }
   static node_ptr get_left(const_node_ptr n)         {  return n->left_;     }
   static void set_left(node_ptr n, node_ptr left)    {  n->left_ = left;     }
   static node_ptr get_right(const_node_ptr n)        {  return n->right_;    }
   static void set_right(node_ptr n, node_ptr right)  {  n->right_ = right;   }
   static balance get_balance(const_node_ptr n)       {  return n->balance_;  }
   static void set_balance(node_ptr n, balance b)     {  n->balance_ = b;     }
   static balance negative()                          {  return -1; }
   static balance zero()                              {  return 0;  }
   static balance positive()                          {  return 1;  }
};

struct node_ptr_compare
{
   bool operator()(const my_node *a, const my_node *b)
   {  return a->int_ < b->int_;  }
};

int main()
{
   typedef boost::intrusive::avltree_algorithms<my_avltree_node_traits> algo;
   my_node header, two(2), three(3);

   //Create an empty avltree container:
   //"header" will be the header node of the tree
   algo::init_header(&header);

   //Now insert node "two" in the tree using the sorting functor
   algo::insert_equal_upper_bound(&header, &two, node_ptr_compare());

   //Now insert node "three" in the tree using the sorting functor
   algo::insert_equal_lower_bound(&header, &three, node_ptr_compare());

   //Now take the first node (the left node of the header)
   my_node *n = header.left_;
   assert(n == &two);

   //Now go to the next node
   n = algo::next_node(n);
   assert(n == &three);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
   //Erase a node just using a pointer to it
   algo::unlink(&two);

   //Erase a node using also the header (faster)
   algo::erase(&header, &three);
   return 0;
}
```

For a complete list of functions see `avltree_algorithms reference`.

Intrusive treap algorithms

`treap_algorithms` have the same interface as `rbtree_algorithms`.

```
template<class NodeTraits>
struct treap_algorithms;
```

`treap_algorithms` is configured with a NodeTraits class, which encapsulates the information about the node to be manipulated. NodeTraits must support the following interface:

Typedefs:

- `node`: The type of the node that forms the circular treap
- `node_ptr`: The type of a pointer to a node (usually node*)
- `const_node_ptr`: The type of a pointer to a const node (usually const node*)

Static functions:

- `static node_ptr get_parent(const_node_ptr n);`: Returns a pointer to the parent node stored in "n".
- `static void set_parent(node_ptr n, node_ptr p);`: Sets the pointer to the parent node stored in "n" to "p".
- `static node_ptr get_left(const_node_ptr n);`: Returns a pointer to the left node stored in "n".
- `static void set_left(node_ptr n, node_ptr l);`: Sets the pointer to the left node stored in "n" to "l".
- `static node_ptr get_right(const_node_ptr n);`: Returns a pointer to the right node stored in "n".
- `static void set_right(node_ptr n, node_ptr r);`: Sets the pointer to the right node stored in "n" to "r".

Once we have a node traits configuration we can use **Boost.Intrusive** algorithms with our nodes:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
#include <boost/intrusive/treap_algorithms.hpp>
#include <cassert>

struct my_node
{
   my_node(int i = 0, unsigned int priority = 0)
      :  prio_(priority), int_(i)
   {}
   my_node *parent_, *left_, *right_;
   int prio_;
   //other members
   int      int_;
};

//Define our own treap_node_traits
struct my_treap_node_traits
{
   typedef my_node                                    node;
   typedef my_node *                                  node_ptr;
   typedef const my_node *                            const_node_ptr;

   static node_ptr get_parent(const_node_ptr n)       {  return n->parent_;   }
   static void set_parent(node_ptr n, node_ptr parent){  n->parent_ = parent; }
   static node_ptr get_left(const_node_ptr n)         {  return n->left_;     }
   static void set_left(node_ptr n, node_ptr left)    {  n->left_ = left;     }
   static node_ptr get_right(const_node_ptr n)        {  return n->right_;    }
   static void set_right(node_ptr n, node_ptr right)  {  n->right_ = right;   }
};

struct node_ptr_compare
{  bool operator()(const my_node *a, const my_node *b) {  return a->int_ < b->int_;  }  };

struct node_ptr_priority
{  bool operator()(const my_node *a, const my_node *b) {  return a->prio_ < b->prio_;}  };

int main()
{
   typedef boost::intrusive::treap_algorithms<my_treap_node_traits> algo;
   my_node header, two(2, 5), three(3, 1);

   //Create an empty treap container:
   //"header" will be the header node of the tree
   algo::init_header(&header);

   //Now insert node "two" in the tree using the sorting functor
   algo::insert_equal_upper_bound(&header, &two, node_ptr_compare(), node_ptr_priority());

   //Now insert node "three" in the tree using the sorting functor
   algo::insert_equal_lower_bound(&header, &three, node_ptr_compare(), node_ptr_priority());

   //Now take the first node (the left node of the header)
   my_node *n = header.left_;
   assert(n == &two);

   //Now go to the next node
   n = algo::next_node(n);
   assert(n == &three);

   //Erase a node just using a pointer to it
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
   algo::unlink(&two, node_ptr_priority());

   //Erase a node using also the header (faster)
   algo::erase(&header, &three, node_ptr_priority());
   return 0;
}
```

For a complete list of functions see `treap_algorithms reference`.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Containers with custom ValueTraits

As explained in the Concepts section, **Boost.Intrusive** containers need a `ValueTraits` class to perform transformations between nodes and user values. `ValueTraits` can be explicitly configured (using the `value_traits<>` option) or implicitly configured (using hooks and their `base_hook<>`/`member_hook<>` options). `ValueTraits` contains all the information to glue the `value_type` of the containers and the node to be used in node algorithms, since these types can be different. Apart from this, `ValueTraits` also stores information about the link policy of the values to be inserted.

Instead of using **Boost.Intrusive** predefined hooks a user might want to develop customized containers, for example, using nodes that are optimized for a specific application or that are compatible with a legacy ABI. A user might want to have only two additional pointers in his class and insert the class in a doubly linked list sometimes and in a singly linked list in other situations. You can't achieve this using **Boost.Intrusive** predefined hooks. Now, instead of using `base_hook<...>` or `member_hook<...>` options the user will specify the `value_traits<...>` options. Let's see how we can do this:

ValueTraits interface

`ValueTraits` has the following interface:

```
#include <boost/intrusive/pointer_traits.hpp>
#include <boost/intrusive/link_mode.hpp>

struct my_value_traits
{
   typedef implementation_defined                                    node_traits;
   typedef implementation_defined                                    value_type;
   typedef node_traits::node_ptr                                     node_ptr;
   typedef node_traits::const_node_ptr                               const_node_ptr;
   typedef boost::intrusive::pointer_traits<node_ptr>::rebind_traits
      <value_type>::type::pointer                                    pointer;
   typedef boost::intrusive::pointer_traits<node_ptr>::rebind_traits
      <const value_type>::type::pointer                              const_pointer;

   static const link_mode_type link_mode = some_linking_policy;

   static node_ptr       to_node_ptr    (value_type &value);
   static const_node_ptr to_node_ptr    (const value_type &value);
   static pointer        to_value_ptr   (node_ptr n);
   static const_pointer  to_value_ptr   (const_node_ptr n);
};
```

Let's explain each type and function:

- **node_traits**: The node configuration that is needed by node algorithms. These node traits and algorithms are described in the previous chapter: Node Algorithms.
  - If my_value_traits is meant to be used with `slist`, `node_traits` should follow the interface needed by `circular_slist_algorithms`.
  - If my_value_traits is meant to be used with `list`, `node_traits` should follow the interface needed by `circular_list_algorithms`.
  - If my_value_traits is meant to be used with `set`/`multiset`, `node_traits` should follow the interface needed by `rbtree_algorithms`.
  - If my_value_traits is meant to be used with `unordered_set`/ `unordered_multiset`, `node_traits` should follow the interface needed by `circular_slist_algorithms`.
- **node_ptr**: A typedef for `node_traits::node_ptr`.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

- **const_node_ptr**: A typedef for `node_traits::const_node_ptr`.
- **value_type**: The type that the user wants to insert in the container. This type can be the same as `node_traits::node` but it can be different (for example, `node_traits::node` can be a member type of `value_type`). If `value_type` and `node_traits::node` are the same type, the `to_node_ptr` and `to_value_ptr` functions are trivial.
- **pointer**: The type of a pointer to a `value_type`. It must be the same pointer type as `node_ptr`: If `node_ptr` is `node*`, `pointer` must be `value_type*`. If `node_ptr` is `smart_ptr<node_traits::node>`, `pointer` must be `smart_ptr<value_type>`. This can be generically achieved using `boost::intrusive::pointer_traits` (portable implementation of C++11 `std::pointer_traits`) or `boost::pointer_to_other` utility from **Boost SmartPointers** defined in `<boost/pointer_to_other.hpp>`.
- **const_pointer**: The type of a pointer to a `const value_type`. It must be the same pointer type as `node_ptr`: If `node_ptr` is `node*`, `const_pointer` must be `const value_type*`. If `node_ptr` is `smart_ptr<node_traits::node>`, `const_pointer` must be `smart_ptr<const value_type>`.
- **link_mode**: Indicates that `value_traits` needs some additional work or checks from the container. The types are enumerations defined in the `link_mode.hpp` header. These are the possible types:
  - **`normal_link`**: If this linking policy is specified in a `ValueTraits` class as the link mode, containers configured with such `ValueTraits` won't set the hooks of the erased values to a default state. Containers also won't check that the hooks of the new values are default initialized.
  - **`safe_link`**: If this linking policy is specified as the link mode in a `ValueTraits` class, containers configured with this `ValueTraits` will set the hooks of the erased values to a default state. Containers also will check that the hooks of the new values are default initialized.
  - **`auto_unlink`**: Same as "safe_link" but containers with constant-time size features won't be compatible with `ValueTraits` configured with this policy. Containers also know that a value can be silently erased from the container without using any function provided by the containers.
- **static node_ptr to_node_ptr (value_type &value)** and **static const_node_ptr to_node_ptr (const value_type &value)**: These functions take a reference to a value_type and return a pointer to the node to be used with node algorithms.
- **static pointer to_value_ptr (node_ptr n)** and **static const_pointer to_value_ptr (const_node_ptr n)**: These functions take a pointer to a node and return a pointer to the value that contains the node.

Custom ValueTraits example

Let's define our own `value_traits` class to be able to use **Boost.Intrusive** containers with an old C structure whose definition can't be changed. That legacy type has two pointers that can be used to build singly and doubly linked lists: in singly linked lists we only need a pointer, whereas in doubly linked lists, we need two pointers. Since we only have two pointers, we can't insert the object in both a singly and a doubly linked list at the same time. This is the definition of the old node:

```
#include <boost/intrusive/link_mode.hpp>
#include <boost/intrusive/list.hpp>
#include <boost/intrusive/slist.hpp>
#include <vector>

//This node is the legacy type we can't modify and we want to insert in
//intrusive list and slist containers using only two pointers, since
//we know the object will never be at the same time in both lists.
struct legacy_value
{
   legacy_value *prev_;
   legacy_value *next_;
   int id_;
};
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Now we have to define a NodeTraits class that will implement the functions/typedefs that will make the legacy node compatible with **Boost.Intrusive** algorithms. After that, we'll define a ValueTraits class that will configure **Boost.Intrusive** containers:

```
//Define our own NodeTraits that will configure singly and doubly linked
//list algorithms. Note that this node traits is compatible with
//circular_slist_algorithms and circular_list_algorithms.

namespace bi = boost::intrusive;

struct legacy_node_traits
{
   typedef legacy_value                            node;
   typedef legacy_value *                          node_ptr;
   typedef const legacy_value *                    const_node_ptr;

   static node *get_next(const node *n)            {  return n->next_;  }
   static void set_next(node *n, node *next)       {  n->next_ = next;  }
   static node *get_previous(const node *n)        {  return n->prev_;  }
   static void set_previous(node *n, node *prev)   {  n->prev_ = prev;  }
};

//This ValueTraits will configure list and slist. In this case,
//legacy_node_traits::node is the same as the
//legacy_value_traits::value_type so to_node_ptr/to_value_ptr
//functions are trivial.
struct legacy_value_traits
{
   typedef legacy_node_traits                                  node_traits;
   typedef node_traits::node_ptr                               node_ptr;
   typedef node_traits::const_node_ptr                         const_node_ptr;
   typedef legacy_value                                        value_type;
   typedef legacy_value *                                      pointer;
   typedef const legacy_value *                                const_pointer;
   static const bi::link_mode_type link_mode = bi::normal_link;
   static node_ptr to_node_ptr (value_type &value)             {  return node_ptr(&value); }
  static const_node_ptr to_node_ptr (const value_type &value) {  return const_node_ptr(&value); }
   static pointer to_value_ptr(node_ptr n)                     {  return pointer(n); }
   static const_pointer to_value_ptr(const_node_ptr n)         {  return const_pointer(n); }
};
```

Defining a value traits class that simply defines `value_type` as `legacy_node_traits::node` is a common approach when defining customized intrusive containers, so **Boost.Intrusive** offers a templatized `trivial_value_traits` class that does exactly what we want:

```
#include <boost/intrusive/trivial_value_traits.hpp>

//Now we can define legacy_value_traits just with a single line
using namespace boost::intrusive;
typedef trivial_value_traits<legacy_node_traits, normal_link> legacy_value_traits;
```

Now we can just define the containers that will store the legacy abi objects and write a little test:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
//Now define an intrusive list and slist that will store legacy_value objects
typedef bi::value_traits<legacy_value_traits>      ValueTraitsOption;
typedef bi::list<legacy_value, ValueTraitsOption>  LegacyAbiList;
typedef bi::slist<legacy_value, ValueTraitsOption> LegacyAbiSlist;

template<class List>
bool test_list()
{
   typedef std::vector<legacy_value> Vect;

   //Create legacy_value objects, with a different internal number
   Vect legacy_vector;
   for(int i = 0; i < 100; ++i){
      legacy_value value;     value.id_ = i;    legacy_vector.push_back(value);
   }

   //Create the list with the objects
   List mylist(legacy_vector.begin(), legacy_vector.end());

   //Now test both lists
   typename List::const_iterator bit(mylist.begin()), bitend(mylist.end());
   typename Vect::const_iterator it(legacy_vector.begin()), itend(legacy_vector.end());

   //Test the objects inserted in our list
   for(; it != itend; ++it, ++bit)
      if(&*bit != &*it) return false;
   return true;
}

int main()
{
   return test_list<LegacyAbiList>() && test_list<LegacyAbiSlist>() ? 0 : 1;
}
```

As seen, several key elements of **Boost.Intrusive** can be reused with custom user types, if the user does not want to use the provided **Boost.Intrusive** facilities.

Reusing node algorithms for different values

In the previous example, `legacy_node_traits::node` type and `legacy_value_traits::value_type` are the same type, but this is not necessary. It's possible to have several `ValueTraits` defining the same `node_traits` type (and thus, the same `node_traits::node`). This reduces the number of node algorithm instantiations, but now `ValueTraits::to_node_ptr` and `ValueTraits::to_value_ptr` functions need to offer conversions between both types. Let's see a small example:

First, we'll define the node to be used in the algorithms. For a linked list, we just need a node that stores two pointers:

```
#include <boost/intrusive/link_mode.hpp>
#include <boost/intrusive/list.hpp>
#include <vector>

//This is the node that will be used with algorithms.
struct simple_node
{
   simple_node *prev_;
   simple_node *next_;
};
```

Now we'll define two different types that will be inserted in intrusive lists and a templatized `ValueTraits` that will work for both types:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
class base_1{};
class base_2{};

struct value_1 :  public base_1, public simple_node
{  int   id_;  };

struct value_2 :  public base_1, public base_2, public simple_node
{  float id_;  };

//Define the node traits. A single node_traits will be enough.
struct simple_node_traits
{
   typedef simple_node                             node;
   typedef node *                                  node_ptr;
   typedef const node *                            const_node_ptr;
   static node *get_next(const node *n)            {  return n->next_;  }
   static void set_next(node *n, node *next)       {  n->next_ = next;  }
   static node *get_previous(const node *n)        {  return n->prev_;  }
   static void set_previous(node *n, node *prev)   {  n->prev_ = prev;  }
};

//A templatized value traits for value_1 and value_2
template<class ValueType>
struct simple_value_traits
{
   typedef simple_node_traits                                        node_traits;
   typedef node_traits::node_ptr                                     node_ptr;
   typedef node_traits::const_node_ptr                               const_node_ptr;
   typedef ValueType                                                 value_type;
   typedef ValueType *                                               pointer;
   typedef const ValueType *                                         const_pointer;
   static const boost::intrusive::link_mode_type link_mode = boost::intrusive::normal_link;
   static node_ptr to_node_ptr (value_type &value)                   {  return node_ptr(&value); }
  static const_node_ptr to_node_ptr (const value_type &value) {  return const_node_ptr(&value); }
   static pointer to_value_ptr(node_ptr n)                           {  return stat↵
ic_cast<value_type*>(n); }
   static const_pointer to_value_ptr(const_node_ptr n)               {  return stat↵
ic_cast<const value_type*>(n); }
};
```

Now define two containers. Both containers will instantiate the same list algorithms (`circular_list_algorithms<simple_node_traits>`), due to the fact that the value traits used to define the containers provide the same `node_traits` type:

```
//Now define two intrusive lists. Both lists will use the same algorithms:
// circular_list_algorithms<simple_node_traits>

using namespace boost::intrusive;
typedef list <value_1, value_traits<simple_value_traits<value_1> > > Value1List;
typedef list <value_2, value_traits<simple_value_traits<value_2> > > Value2List;
```

All **Boost.Intrusive** containers using predefined hooks use this technique to minimize code size: all possible `list` containers created with predefined hooks that define the same `VoidPointer` type share the same list algorithms.

Simplifying value traits definition

The previous example can be further simplified using the `derivation_value_traits` class to define a value traits class with a value that stores the `simple_node` as a base class:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
#include <boost/intrusive/derivation_value_traits.hpp>

//value_1, value_2, simple_node and simple_node_traits are defined
//as in the previous example...
//...

using namespace boost::intrusive;

//Now define the needed value traits using
typedef derivation_value_traits<value_1, simple_node_traits, normal_link> ValueTraits1;
typedef derivation_value_traits<value_2, simple_node_traits, normal_link> ValueTraits2;

//Now define the containers
typedef list <value1, value_traits<ValueTraits1> > Value1List;
typedef list <value2, value_traits<ValueTraits2> > Value2List;
```

We can even choose to store `simple_node` as a member of `value_1` and `value_2` classes and use `member_value_traits` to define the needed value traits classes:

```
class base_1{};
class base_2{};

struct value_1 :  public base_1, public simple_node
{
   int   id_;
   simple_node node_;
};

struct value_2 :  public base_1, public base_2, public simple_node
{
   simple_node node_;
   float id_;
};

using namespace boost::intrusive;

typedef member_value_traits
   <value_1, simple_node_traits, &value_1::node_, normal_link> ValueTraits1;
typedef member_value_traits
<value_2, simple_node_traits, &value_2::node_, normal_link> ValueTraits2;

//Now define two intrusive lists. Both lists will use the same algorithms:
// circular_list_algorithms<simple_node_traits>
typedef list <value_1, value_traits<ValueTraits1> > Value1List;
typedef list <value_2, value_traits<ValueTraits2> > Value2List;
```

Stateful value traits

Until now all shown custom value traits are stateless, that is, **the transformation between nodes and values is implemented in terms of static functions**. It's possible to use **stateful** value traits so that we can separate nodes and values and **avoid modifying types to insert nodes**. **Boost.Intrusive** differentiates between stateful and stateless value traits by checking if all Node <-> Value transformation functions are static or not (except for Visual 7.1, since overloaded static function detection is not possible, in this case the implementation checks if the class is empty):

- If all Node <-> Value transformation functions are static , a **stateless** value traits is assumed. transformations must be static functions.
- Otherwise a **stateful** value traits is assumed.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Using stateful value traits it's possible to create containers of non-copyable/movable objects **without modifying** the definition of the class to be inserted. This interesting property is achieved without using global variables (stateless value traits could use global variables to achieve the same goal), so:

- **Thread-safety guarantees**: Better thread-safety guarantees can be achieved with stateful value traits, since accessing global resources might require synchronization primitives that can be avoided when using internal state.
- **Flexibility**: A stateful value traits type can be configured at run-time.
- **Run-time polymorphism**: A value traits might implement node <-> value transformations as virtual functions. A single container type could be configured at run-time to use different node <-> value relationships.

Stateful value traits have many advantages but also some downsides:

- **Performance**: Value traits operations should be very efficient since they are basic operations used by containers. **A heavy node <-> value transformation will hurt intrusive containers' performance**.
- **Exception guarantees**: The stateful ValueTraits must maintain no-throw guarantees, otherwise, the container invariants won't be preserved.
- **Static functions**: Some static functions offered by intrusive containers are not available because node <-> value transformations are not static.
- **Bigger iterators**: The size of some iterators is increased because the iterator needs to store a pointer to the stateful value traits to implement node to value transformations (e.g. `operator*()` and `operator->()`).

An easy and useful example of stateful value traits is when an array of values can be indirectly introduced in a list guaranteeing no additional allocation apart from the initial resource reservation:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
#include <boost/intrusive/list.hpp>

using namespace boost::intrusive;

//This type is not modifiable so we can't store hooks or custom nodes
typedef int identifier_t;

//This value traits will associate elements from an array of identifiers with
//elements of an array of nodes. The element i of the value array will use the
//node i of the node array:
struct stateful_value_traits
{
   typedef list_node_traits<void*>          node_traits;
   typedef node_traits::node                node;
   typedef node *                           node_ptr;
   typedef const node *                     const_node_ptr;
   typedef identifier_t                     value_type;
   typedef identifier_t *                   pointer;
   typedef const identifier_t *             const_pointer;
   static const link_mode_type link_mode =   normal_link;

   stateful_value_traits(pointer ids, node_ptr node_array)
      :  ids_(ids),  nodes_(node_array)
   {}

   ///Note: non static functions!
   node_ptr to_node_ptr (value_type &value)
      {  return this->nodes_ + (&value - this->ids_); }
   const_node_ptr to_node_ptr (const value_type &value) const
      {  return this->nodes_ + (&value - this->ids_); }
   pointer to_value_ptr(node_ptr n)
      {  return this->ids_ + (n - this->nodes_); }
   const_pointer to_value_ptr(const_node_ptr n) const
      {  return this->ids_ + (n - this->nodes_); }

   private:
   pointer  ids_;
   node_ptr nodes_;
};

int main()
{
   const int NumElements = 100;

   //This is an array of ids that we want to "store"
   identifier_t                  ids   [NumElements];

   //This is an array of nodes that is necessary to form the linked list
   list_node_traits<void*>::node nodes [NumElements];

   //Initialize id objects, each one with a different number
   for(int i = 0; i != NumElements; ++i)   ids[i] = i;

   //Define a list that will "link" identifiers using external nodes
   typedef list<identifier_t, value_traits<stateful_value_traits> > List;

   //This list will store ids without modifying identifier_t instances
   //Stateful value traits must be explicitly passed in the constructor.
   List  my_list (stateful_value_traits (ids, nodes));

   //Insert ids in reverse order in the list
   for(identifier_t * it(&ids[0]), *itend(&ids[NumElements]); it != itend; ++it)
      my_list.push_front(*it);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
   //Now test lists
   List::const_iterator   list_it (my_list.cbegin());
   identifier_t *it_val(&ids[NumElements-1]), *it_rbeg_val(&ids[0]-1);

   //Test the objects inserted in the base hook list
   for(; it_val != it_rbeg_val; --it_val, ++list_it)
      if(&*list_it  != &*it_val)   return 1;

   return 0;
}
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Thread safety guarantees

Intrusive containers have thread safety guarantees similar to STL containers.

- Several threads having read or write access to different instances is safe as long as inserted objects are different.
- Concurrent read-only access to the same container is safe.

Some Intrusive hooks (auto-unlink hooks, for example) modify containers without having a reference to them: this is considered a write access to the container.

Other functions, like checking if an object is already inserted in a container using the `is_linked()` member of safe hooks, constitute read access on the container without having a reference to it, so no other thread should have write access (direct or indirect) to that container.

Since the same object can be inserted in several containers at the same time using different hooks, the thread safety of **Boost.Intrusive** is related to the containers and also to the object whose lifetime is manually managed by the user.

As we can see, the analysis of the thread-safety of a program using **Boost.Intrusive** is harder than with non-intrusive containers.

To analyze the thread safety, consider the following points:

- The auto-unlink hook's destructor and `unlink()` functions modify the container indirectly.
- The safe mode and auto-unlink hooks' `is_linked()` functions are a read access to the container.
- Inserting an object in containers that will be modified by different threads has no thread safety guarantee, although in most platforms it will be thread-safe without locking.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Stability and insertion with hint in ordered associative containers with equivalent keys

Boost.Intrusive ordered associative containers with equivalent keys offer stability guarantees, following C++ standard library's defect #233 resolution, explained in document Comments on LWG issue 233: Insertion hints in associative containers. This means that:

- A *Insert without hint* member function always insert at the upper bound of an equal range.
- A *Insert with hint* member function inserts the new value **before the hint** if hint's and new node's keys are equivalent.
- Implements Andrew Koenig *as close as possible to hint* proposal. A new element is always be inserted as close to the hint as possible. So, for example, if there is a subsequence of equivalent values, `a.begin()` as the hint means that the new element should be inserted before the subsequence even if `a.begin()` is far away. This allows code to always append (or prepend) an equal range with something as simple as: `m.insert(m.end(), new_node);` or `m.insert(m.begin(), new_node);`

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Obtaining the same types and reducing symbol length

The flexible option specification mechanism used by **Boost.Intrusive** for hooks and containers has a couple of downsides:

- If a user specifies the same options in different order or specifies some options and leaves the rest as defaults, the type of the created container/hook will be different. Sometimes this is annoying, because two programmers specifying the same options might end up with incompatible types. For example, the following two lists, although using the same options, do not have the same type:

```
#include <boost/intrusive/list.hpp>

using namespace boost::intrusive;

//Explicitly specify constant-time size and size type
typedef list<T, constant_time_size<true>, size_type<std::size_t> List1;

//Implicitly specify constant-time size and size type
typedef list<T> List2;
```

- Option specifiers lead to long template symbols for classes and functions. Option specifiers themselves are verbose and without variadic templates, several default template parameters are assigned for non-specified options. Object and debugging information files can grow and compilation times may suffer if long names are produced.

To solve these issues **Boost.Intrusive** offers some helper metafunctions that reduce symbol lengths and create the same type if the same options (either explicitly or implicitly) are used. These also improve compilation times. All containers and hooks have their respective `make_xxx` versions. The previously shown example can be rewritten like this to obtain the same list type:

```
#include <boost/intrusive/list.hpp>

 using namespace boost::intrusive;

 #include <boost/intrusive/list.hpp>

 using namespace boost::intrusive;

 //Explicitly specify constant-time size and size type
 typedef make_list<T, constant_time_size<true>, size_type<std::size_t>::type List1;

 //Implicitly specify constant-time size and size type
 typedef make_list<T>::type List2;
```

Produced symbol lengths and compilation times will usually be shorter and object/debug files smaller. If you are concerned with file sizes and compilation times, this option is your best choice.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Design Notes

When designing **Boost.Intrusive** the following guidelines have been taken into account:

Boost.Intrusive in performance sensitive environments

Boost.Intrusive should be a valuable tool in performance sensitive environments, and following this guideline, **Boost.Intrusive** has been designed to offer well known complexity guarantees. Apart from that, some options, like optional constant-time, have been designed to offer faster complexity guarantees in some functions, like `slist::splice`.

The advanced lookup and insertion functions for associative containers, taking an arbitrary key type and predicates, were designed to avoid unnecessary object constructions.

Boost.Intrusive in space constrained environments

Boost.Intrusive should be useful in space constrained environments, and following this guideline **Boost.Intrusive** separates node algorithms and intrusive containers to avoid instantiating node algorithms for each user type. For example, a single class of red-black algorithms will be instantiated to implement all set and multiset containers using raw pointers. This way, **Boost.Intrusive** seeks to avoid any code size overhead associated with templates.

Apart from that, **Boost.Intrusive** implements some size improvements: for example, red-black trees embed the color bit in the parent pointer lower bit, if nodes are two-byte aligned. The option to forgo constant-time size operations can reduce container size, and this extra size optimization is noticeable when the container is empty or contains few values.

Boost.Intrusive as a basic building block

Boost.Intrusive can be a basic building block to build more complex containers and this potential has motivated many design decisions. For example, the ability to have more than one hook per user type opens the opportunity to implement multi-index containers on top of **Boost.Intrusive**.

Boost.Intrusive containers implement advanced functions taking function objects as arguments (`clone_from`, `erase_and_dispose`, `insert_check`, etc.). These functions come in handy when implementing non-intrusive containers (for example, STL-like containers) on top of intrusive containers.

Extending Boost.Intrusive

Boost.Intrusive offers a wide range of containers but also allows the construction of custom containers reusing **Boost.Intrusive** elements. The programmer might want to use node algorithms directly or build special hooks that take advantage of an application environment.

For example, the programmer can customize parts of **Boost.Intrusive** to manage old data structures whose definition can't be changed.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Performance

Boost.Intrusive containers offer speed improvements compared to non-intrusive containers primarily because:

- They minimize memory allocation/deallocation calls.
- They obtain better memory locality.

This section will show performance tests comparing some operations on `boost::intrusive::list` and `std::list`:

- Insertions using `push_back` and container destruction will show the overhead associated with memory allocation/deallocation.
- The `reverse` member function will show the advantages of the compact memory representation that can be achieved with intrusive containers.
- The `sort` and `write access` tests will show the advantage of intrusive containers minimizing memory accesses compared to containers of pointers.

Given an object of type `T`, `boost::intrusive::list<T>` can replace `std::list<T>` to avoid memory allocation overhead, or it can replace `std::list<T*>` when the user wants containers with polymorphic values or wants to share values between several containers. Because of this versatility, the performance tests will be executed for 6 different list types:

- 3 intrusive lists holding a class named `itest_class`, each one with a different linking policy (`normal_link`, `safe_link`, `auto_unlink`). The `itest_class` objects will be tightly packed in a `std::vector<itest_class>` object.
- `std::list<test_class>`, where `test_class` is exactly the same as `itest_class`, but without the intrusive hook.
- `std::list<test_class*>`. The list will contain pointers to `test_class` objects tightly packed in a `std::vector<test_class>` object. We'll call this configuration *compact pointer list*
- `std::list<test_class*>`. The list will contain pointers to `test_class` objects owned by a `std::list<test_class>` object. We'll call this configuration *disperse pointer list*.

Both `test_class` and `itest_class` are templatized classes that can be configured with a boolean to increase the size of the object. This way, the tests can be executed with small and big objects. Here is the first part of the testing code, which shows the definitions of `test_class` and `itest_class` classes, and some other utilities:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
//Iteration and element count defines
const int NumIter = 100;
const int NumElements   = 50000;

using namespace boost::intrusive;

template<bool BigSize>  struct filler        {  int dummy[10];   };
template <>             struct filler<false> {};

template<bool BigSize> //The object for non-intrusive containers
struct test_class :  private filler<BigSize>
{
   int i_;
   test_class()               {}
   test_class(int i) :  i_(i) {}
   friend bool operator <(const test_class &l, const test_class &r)  {  return l.i_ < r.i_;  }
   friend bool operator >(const test_class &l, const test_class &r)  {  return l.i_ > r.i_;  }
};

template <bool BigSize, link_mode_type LinkMode>
struct itest_class   //The object for intrusive containers
   :  public list_base_hook<link_mode<LinkMode> >,  public test_class<BigSize>
{
   itest_class()                                {}
   itest_class(int i) : test_class<BigSize>(i)  {}
};

template<class FuncObj> //Adapts functors taking values to functors taking pointers
struct func_ptr_adaptor  :  public FuncObj
{
   typedef typename FuncObj::first_argument_type*  first_argument_type;
   typedef typename FuncObj::second_argument_type* second_argument_type;
   typedef typename FuncObj::result_type           result_type;
   result_type operator()(first_argument_type a,  second_argument_type b) const
      {  return FuncObj::operator()(*a, *b); }
};
```

As we can see, `test_class` is a very simple class holding an `int`. `itest_class` is just a class that has a base hook (`list_base_hook`) and also derives from `test_class`.

`func_ptr_adaptor` is just a functor adaptor to convert function objects taking `test_list` objects to function objects taking pointers to them.

You can find the full test code code in the perf_list.cpp source file.

Back insertion and destruction

The first test will measure the benefits we can obtain with intrusive containers avoiding memory allocations and deallocations. All the objects to be inserted in intrusive containers are allocated in a single allocation call, whereas `std::list` will need to allocate memory for each object and deallocate it for every erasure (or container destruction).

Let's compare the code to be executed for each container type for different insertion tests:

```
std::vector<typename ilist::value_type> objects(NumElements);
ilist l;
for(int i = 0; i < NumElements; ++i)
   l.push_back(objects[i]);
//Elements are unlinked in ilist's destructor
//Elements are destroyed in vector's destructor
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

For intrusive containers, all the values are created in a vector and after that inserted in the intrusive list.

```
stdlist l;
for(int i = 0; i < NumElements; ++i)
   l.push_back(typename stdlist::value_type(i));
//Elements unlinked and destroyed in stdlist's destructor
```

For a standard list, elements are pushed back using push_back().

```
std::vector<typename stdlist::value_type> objects(NumElements);
stdptrlist l;
for(int i = 0; i < NumElements; ++i)
   l.push_back(&objects[i]);
//Pointers to elements unlinked and destroyed in stdptrlist's destructor
//Elements destroyed in vector's destructor
```

For a standard compact pointer list, elements are created in a vector and pushed back in the pointer list using push_back().

```
stdlist objects;  stdptrlist l;
for(int i = 0; i < NumElements; ++i){
   objects.push_back(typename stdlist::value_type(i));
   l.push_back(&objects.back());
}
//Pointers to elements unlinked and destroyed in stdptrlist's destructor
//Elements unlinked and destroyed in stdlist's destructor
```

For a *disperse pointer list*, elements are created in a list and pushed back in the pointer list using push_back().

These are the times in microseconds for each case, and the normalized time:

Table 2. Back insertion + destruction times for Visual C++ 7.1 / Windows XP

Container	Time in us/iteration (small object / big object)	Normalized time (small object / big object)
`normal_link` intrusive list	5000 / 22500	1 / 1
`safe_link` intrusive list	7812 / 32187	1.56 / 1.43
`auto_unlink` intrusive list	10156 / 41562	2.03 / 1.84
Standard list	26875 / 97500	5.37 / 4.33
Standard compact pointer list	76406 / 86718	15.28 / 3.85
Standard disperse pointer list	146562 / 175625	29.31 / 7.80

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Table 3. Back insertion + destruction times for GCC 4.1.1 / MinGW over Windows XP

Container	Time in us/iteration (small object / big object)	Normalized time (small object / big object)
`normal_link` intrusive list	4375 / 22187	1 / 1
`safe_link` intrusive list	7812 / 32812	1.78 / 1.47
`auto_unlink` intrusive list	10468 / 42031	2.39 / 1.89
Standard list	81250 / 98125	18.57 / 4.42
Standard compact pointer list	83750 / 94218	19.14 / 4.24
Standard disperse pointer list	155625 / 175625	35.57 / 7.91

Table 4. Back insertion + destruction times for GCC 4.1.2 / Linux Kernel 2.6.18 (OpenSuse 10.2)

Container	Time in us/iteration (small object / big object)	Normalized time (small object / big object)
`normal_link` intrusive list	4792 / 20495	1 / 1
`safe_link` intrusive list	7709 / 30803	1.60 / 1.5
`auto_unlink` intrusive list	10180 / 41183	2.12 / 2.0
Standard list	17031 / 32586	3.55 / 1.58
Standard compact pointer list	27221 / 34823	5.68 / 1.69
Standard disperse pointer list	102272 / 60056	21.34 / 2.93

The results are logical: intrusive lists just need one allocation. The destruction time of the `normal_link` intrusive container is trivial (complexity: `O(1)`), whereas `safe_link` and `auto_unlink` intrusive containers need to put the hooks of erased values in the default state (complexity: `O(NumElements)`). That's why `normal_link` intrusive list shines in this test.

Non-intrusive containers need to make many more allocations and that's why they lag behind. The `disperse pointer list` needs to make `NumElements*2` allocations, so the result is not surprising.

The Linux test shows that standard containers perform very well against intrusive containers with big objects. Nearly the same GCC version in MinGW performs worse, so maybe a good memory allocator is the reason for these excellent results.

Reversing

The next test measures the time needed to complete calls to the member function `reverse()`. Values (`test_class` and `itest_class`) and lists are created as explained in the previous section.

Note that for pointer lists, `reverse` **does not need to access `test_class` values stored in another list or vector**, since this function just needs to adjust internal pointers, so in theory all tested lists need to perform the same operations.

These are the results:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Table 5. Reverse times for Visual C++ 7.1 / Windows XP

Container	Time in us/iteration (small object / big object)	Normalized time (small object / big object)
`normal_link` intrusive list	2656 / 10625	1 / 1.83
`safe_link` intrusive list	2812 / 10937	1.05 / 1.89
`auto_unlink` intrusive list	2710 / 10781	1.02 / 1.86
Standard list	5781 / 14531	2.17 / 2.51
Standard compact pointer list	5781 / 5781	2.17 / 1
Standard disperse pointer list	10781 / 15781	4.05 / 2.72

Table 6. Reverse times for GCC 4.1.1 / MinGW over Windows XP

Container	Time in us/iteration (small object / big object)	Normalized time (small object / big object)
`normal_link` intrusive list	2656 / 10781	1 / 2.22
`safe_link` intrusive list	2656 / 10781	1 / 2.22
`auto_unlink` intrusive list	2812 / 10781	1.02 / 2.22
Standard list	4843 / 12500	1.82 / 2.58
Standard compact pointer list	4843 / 4843	1.82 / 1
Standard disperse pointer list	9218 / 12968	3.47 / 2.67

Table 7. Reverse times for GCC 4.1.2 / Linux Kernel 2.6.18 (OpenSuse 10.2)

Container	Time in us/iteration (small object / big object)	Normalized time (small object / big object)
`normal_link` intrusive list	2742 / 10847	1 / 3.41
`safe_link` intrusive list	2742 / 10847	1 / 3.41
`auto_unlink` intrusive list	2742 / 11027	1 / 3.47
Standard list	3184 / 10942	1.16 / 3.44
Standard compact pointer list	3207 / 3176	1.16 / 1
Standard disperse pointer list	5814 / 13381	2.12 / 4.21

For small objects the results show that the compact storage of values in intrusive containers improve locality and reversing is faster than with standard containers, whose values might be dispersed in memory because each value is independently allocated. Note that the dispersed pointer list (a list of pointers to values allocated in another list) suffers more because nodes of the pointer list might be more dispersed, since allocations from both lists are interleaved in the code:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
//Object list (holding `test_class`)
stdlist objects;

//Pointer list (holding `test_class` pointers)
stdptrlist l;

for(int i = 0; i < NumElements; ++i){
   //Allocation from the object list
   objects.push_back(stdlist::value_type(i));
   //Allocation from the pointer list
   l.push_back(&objects.back());
}
```

For big objects the compact pointer list wins because the reversal test doesn't need access to values stored in another container. Since all the allocations for nodes of this pointer list are likely to be close (since there is no other allocation in the process until the pointer list is created) locality is better than with intrusive containers. The dispersed pointer list, as with small values, has poor locality.

Sorting

The next test measures the time needed to complete calls to the member function `sort(Pred pred)`. Values (`test_class` and `itest_class`) and lists are created as explained in the first section. The values will be sorted in ascending and descending order each iteration. For example, if *l* is a list:

```
for(int i = 0; i < NumIter; ++i){
   if(!(i % 2))
      l.sort(std::greater<stdlist::value_type>());
   else
      l.sort(std::less<stdlist::value_type>());
}
```

For a pointer list, the function object will be adapted using `func_ptr_adaptor`:

```
for(int i = 0; i < NumIter; ++i){
   if(!(i % 2))
      l.sort(func_ptr_adaptor<std::greater<stdlist::value_type> >());
   else
      l.sort(func_ptr_adaptor<std::less<stdlist::value_type> >());
}
```

Note that for pointer lists, `sort` will take a function object that **will access `test_class` values stored in another list or vector**, so pointer lists will suffer an extra indirection: they will need to access the `test_class` values stored in another container to compare two elements.

These are the results:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Table 8. Sort times for Visual C++ 7.1 / Windows XP

Container	Time in us/iteration (small object / big object)	Normalized time (small object / big object)
`normal_link` intrusive list	16093 / 38906	1 / 1
`safe_link` intrusive list	16093 / 39062	1 / 1
`auto_unlink` intrusive list	16093 / 38906	1 / 1
Standard list	32343 / 56406	2.0 / 1.44
Standard compact pointer list	33593 / 46093	2.08 / 1.18
Standard disperse pointer list	46875 / 68593	2.91 / 1.76

Table 9. Sort times for GCC 4.1.1 / MinGW over Windows XP

Container	Time in us/iteration (small object / big object)	Normalized time (small object / big object)
`normal_link` intrusive list	15000 / 39218	1 / 1
`safe_link` intrusive list	15156 / 39531	1.01 / 1.01
`auto_unlink` intrusive list	15156 / 39531	1.01 / 1.01
Standard list	34218 / 56875	2.28 / 1.45
Standard compact pointer list	35468 / 49218	2.36 / 1.25
Standard disperse pointer list	47656 / 70156	3.17 / 1.78

Table 10. Sort times for GCC 4.1.2 / Linux Kernel 2.6.18 (OpenSuse 10.2)

Container	Time in us/iteration (small object / big object)	Normalized time (small object / big object)
`normal_link` intrusive list	18003 / 40795	1 / 1
`safe_link` intrusive list	18003 / 41017	1 / 1
`auto_unlink` intrusive list	18230 / 40941	1.01 / 1
Standard list	26273 / 49643	1.45 / 1.21
Standard compact pointer list	28540 / 43172	1.58 / 1.05
Standard disperse pointer list	35077 / 57638	1.94 / 1.41

The results show that intrusive containers are faster than standard containers. We can see that the pointer list holding pointers to values stored in a vector is quite fast, so the extra indirection that is needed to access the value is minimized because all the values are tightly stored, improving caching. The disperse list, on the other hand, is slower because the indirection to access values stored in the object list is more expensive than accessing values stored in a vector.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Write access

The next test measures the time needed to iterate through all the elements of a list, and increment the value of the internal `i_` member:

```
stdlist::iterator it(l.begin()), end(l.end());
for(; it != end; ++it)
   ++(it->i_);
```

Values (`test_class` and `itest_class`) and lists are created as explained in the first section. Note that for pointer lists, the iteration will suffer an extra indirection: they will need to access the `test_class` values stored in another container:

```
stdptrlist::iterator it(l.begin()), end(l.end());
for(; it != end; ++it)
   ++((*it)->i_);
```

These are the results:

Table 11. Write access times for Visual C++ 7.1 / Windows XP

Container	Time in us/iteration (small object / big object)	Normalized time (small object / big object)
`normal_link` intrusive list	2031 / 8125	1 / 1
`safe_link` intrusive list	2031 / 8281	1 / 1.01
`auto_unlink` intrusive list	2031 / 8281	1 / 1.01
Standard list	4218 / 10000	2.07 / 1.23
Standard compact pointer list	4062 / 8437	2.0 / 1.03
Standard disperse pointer list	8593 / 13125	4.23 / 1.61

Table 12. Write access times for GCC 4.1.1 / MinGW over Windows XP

Container	Time in us/iteration (small object / big object)	Normalized time (small object / big object)
`normal_link` intrusive list	2343 / 8281	1 / 1
`safe_link` intrusive list	2500 / 8281	1.06 / 1
`auto_unlink` intrusive list	2500 / 8281	1.06 / 1
Standard list	4218 / 10781	1.8 / 1.3
Standard compact pointer list	3906 / 8281	1.66 / 1
Standard disperse pointer list	8281 / 13750	3.53 / 1.66

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Table 13. Write access times for GCC 4.1.2 / Linux Kernel 2.6.18 (OpenSuse 10.2)

Container	Time in us/iteration (small object / big object)	Normalized time (small object / big object)
`normal_link` intrusive list	2286 / 8468	1 / 1.1
`safe_link` intrusive list	2381 / 8412	1.04 / 1.09
`auto_unlink` intrusive list	2301 / 8437	1.01 / 1.1
Standard list	3044 / 9061	1.33 / 1.18
Standard compact pointer list	2755 / 7660	1.20 / 1
Standard disperse pointer list	6118 / 12453	2.67 / 1.62

As with the read access test, the results show that intrusive containers outperform all other containers if the values are tightly packed in a vector. The disperse list is again the slowest.

Conclusions

Intrusive containers can offer performance benefits that cannot be achieved with equivalent non-intrusive containers. Memory locality improvements are noticeable when the objects to be inserted are small. Minimizing memory allocation/deallocation calls is also an important factor and intrusive containers make this simple if all objects to be inserted in intrusive containers are allocated using `std::vector` or `std::deque`.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Release Notes

Boost 1.54 Release

- Added `BOOST_NO_EXCEPTIONS` support (bug #7849).

Boost 1.53 Release

- Fixed bugs #7174, #7529, #7815.
- Fixed GCC -Wshadow warnings.
- Added missing `explicit` keyword in several intrusive container constructors.
- Replaced deprecated BOOST_NO_XXXX with newer BOOST_NO_CXX11_XXX macros.
- Small documentation fixes.

Boost 1.51 Release

- Fixed bugs #6841, #6907, #6922, #7033,
- Added `bounded_range` function to trees.

Boost 1.49 Release

- Fixed bugs #6347, #6223, #6153.

Boost 1.48 Release

- Fixed bugs #4797, #5165, #5183, #5191.

Boost 1.46 Release

- Fixed bug #4980,

Boost 1.45 Release

- Added `function_hook` option.
- Fixed bugs #2611, #3288, #3304, #3489, #3668, #3688, #3698, #3706, #3721. #3729, #3746, #3781, #3840, #3849, #3339, #3419, #3431, #4021.

Boost 1.40 Release

- Code cleanup in tree_algorithms.hpp and avl_tree_algorithms.hpp
- Fixed bug #3164.

Boost 1.39 Release

- Optimized `list::merge` and `slist::merge`
- `list::sort` and `slist::sort` are now stable.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

- Fixed bugs #2689, #2755, #2786, #2807, #2810, #2862.

Boost 1.38 Release

- New treap-based containers: treap, treap_set, treap_multiset.
- Corrected compilation bug for Windows-based 64 bit compilers.
- Corrected exception-safety bugs in container constructors.
- Updated documentation to show rvalue-references functions instead of emulation functions.

Boost 1.37 Release

- Intrusive now takes advantage of compilers with variadic templates.
- `clone_from` functions now copy predicates and hash functions of associative containers.
- Added incremental hashing to unordered containers via `incremental<>` option.
- Update some function parameters from `iterator` to `const_iterator` in containers to keep up with the draft of the next standard.
- Added an option to specify include files for intrusive configurable assertion macros.

Boost 1.36 Release

- Added `linear<>` and `cache_last<>` options to singly linked lists.
- Added `optimize_multikey<>` option to unordered container hooks.
- Optimized unordered containers when `store_hash` option is used in the hook.
- Implementation changed to be exception agnostic so that it can be used in environments without exceptions.
- Added `container_from_iterator` function to tree-based containers.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Tested compilers

Boost.Intrusive has been tested on the following compilers/platforms:

- Visual >= 7.1
- GCC >= 4.1
- Intel 11

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

References

- SGI's STL Programmer's Guide. **Boost.Intrusive** is based on STL concepts and interfaces.
- Dr. Dobb's, September 1, 2005: *Implementing Splay Trees in C++* . **Boost.Intrusive** splay containers code is based on this article.
- Olaf's original intrusive container library: *STL-like intrusive containers* .

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at **http://www.renderx.com/**

Acknowledgements

Olaf Krzikalla would like to thank:

- **Markus Schaaf** for pointing out the possibility and the advantages of the derivation approach.
- **Udo Steinbach** for encouragements to present this work for boost, a lot of fixes and helpful discussions.
- **Jaap Suter** for the initial hint, which eventually lead to the member value_traits.

Ion Gaztanaga would like to thank:

- **Olaf Krzikalla** for the permission to continue his great work.
- **Joaquin M. Lopez Munoz** for his thorough review, help, and ideas.
- **Cory Nelson**, **Daniel James**, **Dave Harris**, **Guillaume Melquiond**, **Henri Bavestrello**, **Hervé Bronnimann**, **Kai Bruning**, **Kevin Sopp**, **Paul Rose**, **Pavel Vozelinek**, **Howard Hinnant**, **Olaf Krzikalla**, **Samuel Debionne**, **Stjepan Rajko**, **Thorsten Ottosen**, **Tobias Schwinger**, **Tom Brinkman** and **Steven Watanabe** for their comments and reviews in the Boost.Intrusive formal review.
- Thanks to **Julienne Walker** and **The EC Team** (http://eternallyconfuzzled.com) for their great algorithms.
- Thanks to **Daniel K. O.** for his AVL tree rebalancing code.
- Thanks to **Ralf Mattethat** for his splay tree article and code.
- Special thanks to **Steven Watanabe** and **Tobias Schwinger** for their invaluable suggestions and improvements.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Indexes

Class Index

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Typedef Index

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Function Index

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Macro Index

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Reference

Header <boost/intrusive/any_hook.hpp>

```
namespace boost {
  namespace intrusive {
    template<class... Options> struct make_any_base_hook;

    template<class... Options> class any_base_hook;

    template<class... Options> struct make_any_member_hook;

    template<class... Options> class any_member_hook;

    template<typename BaseHook> struct any_to_slist_hook;
    template<typename BaseHook> struct any_to_list_hook;
    template<typename BaseHook> struct any_to_set_hook;
    template<typename BaseHook> struct any_to_avl_set_hook;
    template<typename BaseHook> struct any_to_bs_set_hook;
    template<typename BaseHook> struct any_to_unordered_set_hook;
  }
}
```

Struct template make_any_base_hook

boost::intrusive::make_any_base_hook

Synopsis

```
// In header: <boost/intrusive/any_hook.hpp>

template<class... Options>
struct make_any_base_hook {
  // types
  typedef implementation_defined type;
};
```

Description

Helper metafunction to define a `any_base_hook` `that` yields to the same type when the same options (either explicitly or implicitly) are used.

Class template any_base_hook

boost::intrusive::any_base_hook

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/intrusive/any_hook.hpp>

template<class... Options>
class any_base_hook : public make_any_base_hook::type< O1, O2, O3 > {
public:
  // construct/copy/destruct
  any_base_hook();
  any_base_hook(const any_base_hook &);
  any_base_hook& operator=(const any_base_hook &);
  ~any_base_hook();

  // public member functions
  bool is_linked() const;
};
```

Description

Derive a class from this hook in order to store objects of that class in an intrusive container.

The hook admits the following options: `tag<>`, `void_pointer<>` and `link_mode<>`.

`tag<>` defines a tag to identify the node. The same tag value can be used in different classes, but if a class is derived from more than one `any_base_hook`, then each `any_base_hook` needs its unique tag.

`link_mode<>` will specify the linking mode of the hook (`normal_link`, `safe_link`).

`void_pointer<>` is the pointer type that will be used internally in the hook and the the container configured to use this hook.

any_base_hook public construct/copy/destruct

1.
```
any_base_hook();
```

Effects: If `link_mode` is or `safe_link` initializes the node to an unlinked state.

Throws: Nothing.

2.
```
any_base_hook(const any_base_hook &);
```

Effects: If `link_mode` is or `safe_link` initializes the node to an unlinked state. The argument is ignored.

Throws: Nothing.

Rationale: Providing a copy-constructor makes classes using the hook STL-compliant without forcing the user to do some additional work. `swap` can be used to emulate move-semantics.

3.
```
any_base_hook& operator=(const any_base_hook &);
```

Effects: Empty function. The argument is ignored.

Throws: Nothing.

Rationale: Providing an assignment operator makes classes using the hook STL-compliant without forcing the user to do some additional work. `swap` can be used to emulate move-semantics.

4.
```
~any_base_hook();
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: If `link_mode` is `normal_link`, the destructor does nothing (ie. no code is generated). If `link_mode` is `safe_link` and the object is stored in a container an assertion is raised.

Throws: Nothing.

any_base_hook public member functions

1.
   ```
bool is_linked() const;
```

   Precondition: `link_mode` must be `safe_link`.

   Returns: true, if the node belongs to a container, false otherwise. This function can be used to test whether `container::iterator_to` will return a valid iterator.

   Complexity: Constant

Struct template make_any_member_hook

boost::intrusive::make_any_member_hook

Synopsis

```
// In header: <boost/intrusive/any_hook.hpp>

template<class... Options>
struct make_any_member_hook {
  // types
  typedef implementation_defined type;
};
```

Description

Helper metafunction to define a `any_member_hook that` yields to the same type when the same options (either explicitly or implicitly) are used.

Class template any_member_hook

boost::intrusive::any_member_hook

Synopsis

```
// In header: <boost/intrusive/any_hook.hpp>

template<class... Options>
class any_member_hook : public make_any_member_hook::type< O1, O2, O3 > {
public:
  // construct/copy/destruct
  any_member_hook();
  any_member_hook(const any_member_hook &);
  any_member_hook& operator=(const any_member_hook &);
  ~any_member_hook();

  // public member functions
  bool is_linked() const;
};
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Description

Store this hook in a class to be inserted in an intrusive container.

The hook admits the following options: `void_pointer<>` and `link_mode<>`.

`link_mode<>` will specify the linking mode of the hook (`normal_link` or `safe_link`).

`void_pointer<>` is the pointer type that will be used internally in the hook and the the container configured to use this hook.

`any_member_hook` public construct/copy/destruct

1.
```
any_member_hook();
```

   Effects: If `link_mode` is or `safe_link` initializes the node to an unlinked state.

   Throws: Nothing.

2.
```
any_member_hook(const any_member_hook &);
```

   Effects: If `link_mode` is or `safe_link` initializes the node to an unlinked state. The argument is ignored.

   Throws: Nothing.

   Rationale: Providing a copy-constructor makes classes using the hook STL-compliant without forcing the user to do some additional work. `swap` can be used to emulate move-semantics.

3.
```
any_member_hook& operator=(const any_member_hook &);
```

   Effects: Empty function. The argument is ignored.

   Throws: Nothing.

   Rationale: Providing an assignment operator makes classes using the hook STL-compliant without forcing the user to do some additional work. `swap` can be used to emulate move-semantics.

4.
```
~any_member_hook();
```

   Effects: If `link_mode` is `normal_link`, the destructor does nothing (ie. no code is generated). If `link_mode` is `safe_link` and the object is stored in a container an assertion is raised.

   Throws: Nothing.

`any_member_hook` public member functions

1.
```
bool is_linked() const;
```

   Precondition: `link_mode` must be `safe_link`.

   Returns: true, if the node belongs to a container, false otherwise. This function can be used to test whether `container::iterator_to` will return a valid iterator.

   Complexity: Constant

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Struct template any_to_slist_hook

boost::intrusive::any_to_slist_hook

Synopsis

```
// In header: <boost/intrusive/any_hook.hpp>

template<typename BaseHook> 
struct any_to_slist_hook {
};
```

Description

This option setter specifies that any hook should behave as an slist hook

Struct template any_to_list_hook

boost::intrusive::any_to_list_hook

Synopsis

```
// In header: <boost/intrusive/any_hook.hpp>

template<typename BaseHook> 
struct any_to_list_hook {
};
```

Description

This option setter specifies that any hook should behave as an list hook

Struct template any_to_set_hook

boost::intrusive::any_to_set_hook

Synopsis

```
// In header: <boost/intrusive/any_hook.hpp>

template<typename BaseHook> 
struct any_to_set_hook {
};
```

Description

This option setter specifies that any hook should behave as a set hook

Struct template any_to_avl_set_hook

boost::intrusive::any_to_avl_set_hook

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/intrusive/any_hook.hpp>

template<typename BaseHook>
struct any_to_avl_set_hook {
};
```

Description

This option setter specifies that any hook should behave as an avl_set hook

Struct template any_to_bs_set_hook

boost::intrusive::any_to_bs_set_hook

Synopsis

```
// In header: <boost/intrusive/any_hook.hpp>

template<typename BaseHook>
struct any_to_bs_set_hook {
};
```

Description

This option setter specifies that any hook should behave as a bs_set hook

Struct template any_to_unordered_set_hook

boost::intrusive::any_to_unordered_set_hook

Synopsis

```
// In header: <boost/intrusive/any_hook.hpp>

template<typename BaseHook>
struct any_to_unordered_set_hook {
};
```

Description

This option setter specifies that any hook should behave as an unordered set hook

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Header <boost/intrusive/avl_set.hpp>

```
namespace boost {
  namespace intrusive {
    template<typename T, class... Options> class avl_set;

    template<typename T, class... Options> struct make_avl_set;

    template<typename T, class... Options> class avl_multiset;

    template<typename T, class... Options> struct make_avl_multiset;
    template<typename T, class... Options>
      bool operator!=(const avl_set< T, Options...> & x,
                      const avl_set< T, Options...> & y);
    template<typename T, class... Options>
      bool operator>(const avl_set< T, Options...> & x,
                     const avl_set< T, Options...> & y);
    template<typename T, class... Options>
      bool operator<=(const avl_set< T, Options...> & x,
                      const avl_set< T, Options...> & y);
    template<typename T, class... Options>
      bool operator>=(const avl_set< T, Options...> & x,
                      const avl_set< T, Options...> & y);
    template<typename T, class... Options>
      void swap(avl_set< T, Options...> & x, avl_set< T, Options...> & y);
    template<typename T, class... Options>
      bool operator!=(const avl_multiset< T, Options...> & x,
                      const avl_multiset< T, Options...> & y);
    template<typename T, class... Options>
      bool operator>(const avl_multiset< T, Options...> & x,
                     const avl_multiset< T, Options...> & y);
    template<typename T, class... Options>
      bool operator<=(const avl_multiset< T, Options...> & x,
                      const avl_multiset< T, Options...> & y);
    template<typename T, class... Options>
      bool operator>=(const avl_multiset< T, Options...> & x,
                      const avl_multiset< T, Options...> & y);
    template<typename T, class... Options>
      void swap(avl_multiset< T, Options...> & x,
                avl_multiset< T, Options...> & y);
  }
}
```

Class template avl_set

boost::intrusive::avl_set

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/intrusive/avl_set.hpp>

template<typename T, class... Options>
class avl_set {
public:
  // types
  typedef implementation_defined::value_type             value_type;
  typedef implementation_defined::value_traits           value_traits;
  typedef implementation_defined::pointer                pointer;
  typedef implementation_defined::const_pointer          const_pointer;
  typedef implementation_defined::reference              reference;
  typedef implementation_defined::const_reference        const_reference;
  typedef implementation_defined::difference_type        difference_type;
  typedef implementation_defined::size_type              size_type;
  typedef implementation_defined::value_compare          value_compare;
  typedef implementation_defined::key_compare            key_compare;
  typedef implementation_defined::iterator               iterator;
  typedef implementation_defined::const_iterator         const_iterator;
  typedef implementation_defined::reverse_iterator       reverse_iterator;
  typedef implementation_defined::const_reverse_iterator const_reverse_iterator;
  typedef implementation_defined::insert_commit_data     insert_commit_data;
  typedef implementation_defined::node_traits            node_traits;
  typedef implementation_defined::node                   node;
  typedef implementation_defined::node_ptr               node_ptr;
  typedef implementation_defined::const_node_ptr         const_node_ptr;
  typedef implementation_defined::node_algorithms        node_algorithms;

  // construct/copy/destruct
  explicit avl_set(const value_compare & = value_compare(),
                   const value_traits & = value_traits());
  template<typename Iterator>
    avl_set(Iterator, Iterator, const value_compare & = value_compare(),
            const value_traits & = value_traits());
  avl_set(BOOST_RV_REF(avl_set));
  avl_set& operator=(BOOST_RV_REF(avl_set));
  ~avl_set();

  // public member functions
  iterator begin();
  const_iterator begin() const;
  const_iterator cbegin() const;
  iterator end();
  const_iterator end() const;
  const_iterator cend() const;
  reverse_iterator rbegin();
  const_reverse_iterator rbegin() const;
  const_reverse_iterator crbegin() const;
  reverse_iterator rend();
  const_reverse_iterator rend() const;
  const_reverse_iterator crend() const;
  key_compare key_comp() const;
  value_compare value_comp() const;
  bool empty() const;
  size_type size() const;
  void swap(avl_set &);
  template<typename Cloner, typename Disposer>
    void clone_from(const avl_set &, Cloner, Disposer);
  std::pair< iterator, bool > insert(reference);
  iterator insert(const_iterator, reference);
  template<typename KeyType, typename KeyValueCompare>
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
  std::pair< iterator, bool >
  insert_check(const KeyType &, KeyValueCompare, insert_commit_data &);
template<typename KeyType, typename KeyValueCompare>
  std::pair< iterator, bool >
  insert_check(const_iterator, const KeyType &, KeyValueCompare,
               insert_commit_data &);
iterator insert_commit(reference, const insert_commit_data &);
template<typename Iterator> void insert(Iterator, Iterator);
iterator insert_before(const_iterator, reference);
void push_back(reference);
void push_front(reference);
iterator erase(const_iterator);
iterator erase(const_iterator, const_iterator);
size_type erase(const_reference);
template<typename KeyType, typename KeyValueCompare>
  size_type erase(const KeyType &, KeyValueCompare);
template<typename Disposer>
  iterator erase_and_dispose(const_iterator, Disposer);
template<typename Disposer>
  iterator erase_and_dispose(const_iterator, const_iterator, Disposer);
template<typename Disposer>
  size_type erase_and_dispose(const_reference, Disposer);
template<typename KeyType, typename KeyValueCompare, typename Disposer>
  size_type erase_and_dispose(const KeyType &, KeyValueCompare, Disposer);
void clear();
template<typename Disposer> void clear_and_dispose(Disposer);
size_type count(const_reference) const;
template<typename KeyType, typename KeyValueCompare>
  size_type count(const KeyType &, KeyValueCompare) const;
iterator lower_bound(const_reference);
template<typename KeyType, typename KeyValueCompare>
  iterator lower_bound(const KeyType &, KeyValueCompare);
const_iterator lower_bound(const_reference) const;
template<typename KeyType, typename KeyValueCompare>
  const_iterator lower_bound(const KeyType &, KeyValueCompare) const;
iterator upper_bound(const_reference);
template<typename KeyType, typename KeyValueCompare>
  iterator upper_bound(const KeyType &, KeyValueCompare);
const_iterator upper_bound(const_reference) const;
template<typename KeyType, typename KeyValueCompare>
  const_iterator upper_bound(const KeyType &, KeyValueCompare) const;
iterator find(const_reference);
template<typename KeyType, typename KeyValueCompare>
  iterator find(const KeyType &, KeyValueCompare);
const_iterator find(const_reference) const;
template<typename KeyType, typename KeyValueCompare>
  const_iterator find(const KeyType &, KeyValueCompare) const;
std::pair< iterator, iterator > equal_range(const_reference);
template<typename KeyType, typename KeyValueCompare>
  std::pair< iterator, iterator >
  equal_range(const KeyType &, KeyValueCompare);
std::pair< const_iterator, const_iterator >
equal_range(const_reference) const;
template<typename KeyType, typename KeyValueCompare>
  std::pair< const_iterator, const_iterator >
  equal_range(const KeyType &, KeyValueCompare) const;
std::pair< iterator, iterator >
bounded_range(const_reference, const_reference, bool, bool);
template<typename KeyType, typename KeyValueCompare>
  std::pair< iterator, iterator >
  bounded_range(const KeyType &, const KeyType &, KeyValueCompare, bool,
                bool);
std::pair< const_iterator, const_iterator >
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
  bounded_range(const_reference, const_reference, bool, bool) const;
  template<typename KeyType, typename KeyValueCompare>
    std::pair< const_iterator, const_iterator >
    bounded_range(const KeyType &, const KeyType &, KeyValueCompare, bool,
                  bool) const;
  iterator iterator_to(reference);
  const_iterator iterator_to(const_reference) const;
  pointer unlink_leftmost_without_rebalance();
  void replace_node(iterator, reference);

  // public static functions
  static avl_set & container_from_end_iterator(iterator);
  static const avl_set & container_from_end_iterator(const_iterator);
  static avl_set & container_from_iterator(iterator);
  static const avl_set & container_from_iterator(const_iterator);
  static iterator s_iterator_to(reference);
  static const_iterator s_iterator_to(const_reference);
  static void init_node(reference);

  // public data members
  static const bool constant_time_size;
};
```

Description

The class template avl_set is an intrusive container, that mimics most of the interface of std::set as described in the C++ standard.

The template parameter `T` is the type to be managed by the container. The user can specify additional options and if no options are provided default options are used.

The container supports the following options: `base_hook<>/member_hook<>/value_traits<>`, `constant_time_size<>`, `size_type<>` and `compare<>`.

`avl_set` public construct/copy/destruct

1.
```
explicit avl_set(const value_compare & cmp = value_compare(),
                 const value_traits & v_traits = value_traits());
```

   Effects: Constructs an empty `avl_set`.

   Complexity: Constant.

   Throws: If value_traits::node_traits::node constructor throws (this does not happen with predefined Boost.Intrusive hooks) or the copy constructor of the value_compare object throws.

2.
```
template<typename Iterator>
  avl_set(Iterator b, Iterator e, const value_compare & cmp = value_compare(),
          const value_traits & v_traits = value_traits());
```

   Requires: Dereferencing iterator must yield an lvalue of type value_type. cmp must be a comparison function that induces a strict weak ordering.

   Effects: Constructs an empty `avl_set` and inserts elements from [b, e).

   Complexity: Linear in N if [b, e) is already sorted using comp and otherwise N * log N, where N is std::distance(last, first).

   Throws: If value_traits::node_traits::node constructor throws (this does not happen with predefined Boost.Intrusive hooks) or the copy constructor/operator() of the value_compare object throws.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

3.
```
avl_set(BOOST_RV_REF(avl_set) x);
```

Effects: to-do

4.
```
avl_set& operator=(BOOST_RV_REF(avl_set) x);
```

Effects: to-do

5.
```
~avl_set();
```

Effects: Detaches all elements from this. The objects in the `avl_set` are not deleted (i.e. no destructors are called).

Complexity: Linear to the number of elements on the container. if it's a safe-mode or auto-unlink value_type. Constant time otherwise.

Throws: Nothing.

`avl_set` public member functions

1.
```
iterator begin();
```

Effects: Returns an iterator pointing to the beginning of the `avl_set`.

Complexity: Constant.

Throws: Nothing.

2.
```
const_iterator begin() const;
```

Effects: Returns a const_iterator pointing to the beginning of the `avl_set`.

Complexity: Constant.

Throws: Nothing.

3.
```
const_iterator cbegin() const;
```

Effects: Returns a const_iterator pointing to the beginning of the `avl_set`.

Complexity: Constant.

Throws: Nothing.

4.
```
iterator end();
```

Effects: Returns an iterator pointing to the end of the `avl_set`.

Complexity: Constant.

Throws: Nothing.

5.
```
const_iterator end() const;
```

Effects: Returns a const_iterator pointing to the end of the `avl_set`.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Constant.

Throws: Nothing.

6.
```
const_iterator cend() const;
```

Effects: Returns a const_iterator pointing to the end of the `avl_set`.

Complexity: Constant.

Throws: Nothing.

7.
```
reverse_iterator rbegin();
```

Effects: Returns a reverse_iterator pointing to the beginning of the reversed `avl_set`.

Complexity: Constant.

Throws: Nothing.

8.
```
const_reverse_iterator rbegin() const;
```

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed `avl_set`.

Complexity: Constant.

Throws: Nothing.

9.
```
const_reverse_iterator crbegin() const;
```

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed `avl_set`.

Complexity: Constant.

Throws: Nothing.

10.
```
reverse_iterator rend();
```

Effects: Returns a reverse_iterator pointing to the end of the reversed `avl_set`.

Complexity: Constant.

Throws: Nothing.

11.
```
const_reverse_iterator rend() const;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed `avl_set`.

Complexity: Constant.

Throws: Nothing.

12.
```
const_reverse_iterator crend() const;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed `avl_set`.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Constant.

Throws: Nothing.

13.
```
key_compare key_comp() const;
```

Effects: Returns the key_compare object used by the `avl_set`.

Complexity: Constant.

Throws: If key_compare copy-constructor throws.

14.
```
value_compare value_comp() const;
```

Effects: Returns the value_compare object used by the `avl_set`.

Complexity: Constant.

Throws: If value_compare copy-constructor throws.

15.
```
bool empty() const;
```

Effects: Returns true is the container is empty.

Complexity: Constant.

Throws: Nothing.

16.
```
size_type size() const;
```

Effects: Returns the number of elements stored in the `avl_set`.

Complexity: Linear to elements contained in *this if, constant-time size option is enabled. Constant-time otherwise.

Throws: Nothing.

17.
```
void swap(avl_set & other);
```

Effects: Swaps the contents of two sets.

Complexity: Constant.

Throws: If the swap() call for the comparison functor found using ADL throws. Strong guarantee.

18.
```
template<typename Cloner, typename Disposer> 
  void clone_from(const avl_set & src, Cloner cloner, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw. Cloner should yield to nodes equivalent to the original nodes.

Effects: Erases all the elements from *this calling Disposer::operator()(pointer), clones all the elements from src calling Cloner::operator()(const_reference) and inserts them on *this. Copies the predicate from the source container.

If cloner throws, all cloned elements are unlinked and disposed calling Disposer::operator()(pointer).

Complexity: Linear to erased plus inserted elements.

Throws: If cloner throws or predicate copy assignment throws. Basic guarantee.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

19.
```
std::pair< iterator, bool > insert(reference value);
```

Requires: value must be an lvalue

Effects: Treaps to inserts value into the `avl_set`.

Returns: If the value is not already present inserts it and returns a pair containing the iterator to the new value and true. If there is an equivalent value returns a pair containing an iterator to the already present value and false.

Complexity: Average complexity for insert element is at most logarithmic.

Throws: If the internal value_compare ordering function throws. Strong guarantee.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

20.
```
iterator insert(const_iterator hint, reference value);
```

Requires: value must be an lvalue

Effects: Treaps to to insert x into the `avl_set`, using "hint" as a hint to where it will be inserted.

Returns: An iterator that points to the position where the new element was inserted into the `avl_set`.

Complexity: Logarithmic in general, but it's amortized constant time if t is inserted immediately before hint.

Throws: If the internal value_compare ordering function throws. Strong guarantee.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

21.
```
template<typename KeyType, typename KeyValueCompare> 
  std::pair< iterator, bool > 
  insert_check(const KeyType & key, KeyValueCompare key_value_comp, 
               insert_commit_data & commit_data);
```

Requires: key_value_comp must be a comparison function that induces the same strict weak ordering as value_compare. The difference is that key_value_comp compares an arbitrary key with the contained values.

Effects: Checks if a value can be inserted in the `avl_set`, using a user provided key instead of the value itself.

Returns: If there is an equivalent value returns a pair containing an iterator to the already present value and false. If the value can be inserted returns true in the returned pair boolean and fills "commit_data" that is meant to be used with the "insert_commit" function.

Complexity: Average complexity is at most logarithmic.

Throws: If the key_value_comp ordering function throws. Strong guarantee.

Notes: This function is used to improve performance when constructing a value_type is expensive: if there is an equivalent value the constructed object must be discarded. Many times, the part of the node that is used to impose the order is much cheaper to construct than the value_type and this function offers the possibility to use that part to check if the insertion will be successful.

If the check is successful, the user can construct the value_type and use "insert_commit" to insert the object in constant-time. This gives a total logarithmic complexity to the insertion: check(O(log(N)) + commit(O(1)).

"commit_data" remains valid for a subsequent "insert_commit" only if no more objects are inserted or erased from the `avl_set`.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

22.
```
template<typename KeyType, typename KeyValueCompare>
  std::pair< iterator, bool >
  insert_check(const_iterator hint, const KeyType & key,
               KeyValueCompare key_value_comp,
               insert_commit_data & commit_data);
```

Requires: key_value_comp must be a comparison function that induces the same strict weak ordering as value_compare. The difference is that key_value_comp compares an arbitrary key with the contained values.

Effects: Checks if a value can be inserted in the `avl_set`, using a user provided key instead of the value itself, using "hint" as a hint to where it will be inserted.

Returns: If there is an equivalent value returns a pair containing an iterator to the already present value and false. If the value can be inserted returns true in the returned pair boolean and fills "commit_data" that is meant to be used with the "insert_commit" function.

Complexity: Logarithmic in general, but it's amortized constant time if t is inserted immediately before hint.

Throws: If the key_value_comp ordering function throws. Strong guarantee.

Notes: This function is used to improve performance when constructing a value_type is expensive: if there is an equivalent value the constructed object must be discarded. Many times, the part of the constructing that is used to impose the order is much cheaper to construct than the value_type and this function offers the possibility to use that key to check if the insertion will be successful.

If the check is successful, the user can construct the value_type and use "insert_commit" to insert the object in constant-time. This can give a total constant-time complexity to the insertion: check(O(1)) + commit(O(1)).

"commit_data" remains valid for a subsequent "insert_commit" only if no more objects are inserted or erased from the `avl_set`.

23.
```
iterator insert_commit(reference value,
                       const insert_commit_data & commit_data);
```

Requires: value must be an lvalue of type value_type. commit_data must have been obtained from a previous call to "insert_check". No objects should have been inserted or erased from the `avl_set` between the "insert_check" that filled "commit_data" and the call to "insert_commit".

Effects: Inserts the value in the `avl_set` using the information obtained from the "commit_data" that a previous "insert_check" filled.

Returns: An iterator to the newly inserted object.

Complexity: Constant time.

Throws: Nothing.

Notes: This function has only sense if a "insert_check" has been previously executed to fill "commit_data". No value should be inserted or erased between the "insert_check" and "insert_commit" calls.

24.
```
template<typename Iterator> void insert(Iterator b, Iterator e);
```

Requires: Dereferencing iterator must yield an lvalue of type value_type.

Effects: Inserts a range into the `avl_set`.

Complexity: Insert range is in general O(N * log(N)), where N is the size of the range. However, it is linear in N if the range is already sorted by value_comp().

Throws: If the internal value_compare ordering function throws. Basic guarantee.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

25.
```
iterator insert_before(const_iterator pos, reference value);
```

Requires: value must be an lvalue, "pos" must be a valid iterator (or end) and must be the succesor of value once inserted according to the predicate. "value" must not be equal to any inserted key according to the predicate.

Effects: Inserts x into the tree before "pos".

Complexity: Constant time.

Throws: Nothing.

Note: This function does not check preconditions so if "pos" is not the successor of "value" or "value" is not unique tree ordering and uniqueness invariants will be broken respectively. This is a low-level function to be used only for performance reasons by advanced users.

26.
```
void push_back(reference value);
```

Requires: value must be an lvalue, and it must be greater than any inserted key according to the predicate.

Effects: Inserts x into the tree in the last position.

Complexity: Constant time.

Throws: Nothing.

Note: This function does not check preconditions so if value is less than or equal to the greatest inserted key tree ordering invariant will be broken. This function is slightly more efficient than using "insert_before". This is a low-level function to be used only for performance reasons by advanced users.

27.
```
void push_front(reference value);
```

Requires: value must be an lvalue, and it must be less than any inserted key according to the predicate.

Effects: Inserts x into the tree in the first position.

Complexity: Constant time.

Throws: Nothing.

Note: This function does not check preconditions so if value is greater than or equal to the the mimum inserted key tree ordering or uniqueness invariants will be broken. This function is slightly more efficient than using "insert_before". This is a low-level function to be used only for performance reasons by advanced users.

28.
```
iterator erase(const_iterator i);
```

Effects: Erases the element pointed to by pos.

Complexity: Average complexity is constant time.

Returns: An iterator to the element after the erased element.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

29.
```
iterator erase(const_iterator b, const_iterator e);
```

Effects: Erases the range pointed to by b end e.

Complexity: Average complexity for erase range is at most O(log(size() + N)), where N is the number of elements in the range.

Returns: An iterator to the element after the erased elements.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

30.
```
size_type erase(const_reference value);
```

Effects: Erases all the elements with the given value.

Returns: The number of erased elements.

Complexity: O(log(size()) + this->count(value)).

Throws: If the internal value_compare ordering function throws. Basic guarantee.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

31.
```
template<typename KeyType, typename KeyValueCompare> 
  size_type erase(const KeyType & key, KeyValueCompare comp);
```

Effects: Erases all the elements that compare equal with the given key and the given comparison functor.

Returns: The number of erased elements.

Complexity: O(log(size() + this->count(key, comp)).

Throws: If the comp ordering function throws. Basic guarantee.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

32.
```
template<typename Disposer> 
  iterator erase_and_dispose(const_iterator i, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases the element pointed to by pos. Disposer::operator()(pointer) is called for the removed element.

Complexity: Average complexity for erase element is constant time.

Returns: An iterator to the element after the erased element.

Throws: Nothing.

Note: Invalidates the iterators to the erased elements.

33.
```
template<typename Disposer> 
  iterator erase_and_dispose(const_iterator b, const_iterator e, 
                             Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Erases the range pointed to by b end e. Disposer::operator()(pointer) is called for the removed elements.

Complexity: Average complexity for erase range is at most O(log(size() + N)), where N is the number of elements in the range.

Returns: An iterator to the element after the erased elements.

Throws: Nothing.

Note: Invalidates the iterators to the erased elements.

34.
```
template<typename Disposer> 
  size_type erase_and_dispose(const_reference value, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases all the elements with the given value. Disposer::operator()(pointer) is called for the removed elements.

Throws: If the internal value_compare ordering function throws.

Complexity: O(log(size() + this->count(value)). Basic guarantee.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

35.
```
template<typename KeyType, typename KeyValueCompare, typename Disposer> 
  size_type erase_and_dispose(const KeyType & key, KeyValueCompare comp, 
                              Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases all the elements with the given key. according to the comparison functor "comp". Disposer::operator()(pointer) is called for the removed elements.

Returns: The number of erased elements.

Complexity: O(log(size() + this->count(key, comp)).

Throws: If comp ordering function throws. Basic guarantee.

Note: Invalidates the iterators to the erased elements.

36.
```
void clear();
```

Effects: Erases all the elements of the container.

Complexity: Linear to the number of elements on the container. if it's a safe-mode or auto-unlink value_type. Constant time otherwise.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

37.
```
template<typename Disposer> void clear_and_dispose(Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases all the elements of the container.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Linear to the number of elements on the container. Disposer::operator()(pointer) is called for the removed elements.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

38.
```
size_type count(const_reference value) const;
```

Effects: Returns the number of contained elements with the given key

Complexity: Logarithmic to the number of elements contained plus lineal to number of objects with the given key.

Throws: If the internal value_compare ordering function throws.

39.
```
template<typename KeyType, typename KeyValueCompare> 
  size_type count(const KeyType & key, KeyValueCompare comp) const;
```

Effects: Returns the number of contained elements with the same key compared with the given comparison functor.

Complexity: Logarithmic to the number of elements contained plus lineal to number of objects with the given key.

Throws: If comp ordering function throws.

40.
```
iterator lower_bound(const_reference value);
```

Effects: Returns an iterator to the first element whose key is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

41.
```
template<typename KeyType, typename KeyValueCompare> 
  iterator lower_bound(const KeyType & key, KeyValueCompare comp);
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Returns an iterator to the first element whose key according to the comparison functor is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

42.
```
const_iterator lower_bound(const_reference value) const;
```

Effects: Returns a const iterator to the first element whose key is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

43.
```
template<typename KeyType, typename KeyValueCompare>
  const_iterator lower_bound(const KeyType & key, KeyValueCompare comp) const;
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Returns a const_iterator to the first element whose key according to the comparison functor is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

44.
```
iterator upper_bound(const_reference value);
```

Effects: Returns an iterator to the first element whose key is greater than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

45.
```
template<typename KeyType, typename KeyValueCompare>
  iterator upper_bound(const KeyType & key, KeyValueCompare comp);
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Returns an iterator to the first element whose key according to the comparison functor is greater than key or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

46.
```
const_iterator upper_bound(const_reference value) const;
```

Effects: Returns an iterator to the first element whose key is greater than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

47.
```
template<typename KeyType, typename KeyValueCompare>
  const_iterator upper_bound(const KeyType & key, KeyValueCompare comp) const;
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Returns a const_iterator to the first element whose key according to the comparison functor is greater than key or end() if that element does not exist.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

48.
```
iterator find(const_reference value);
```

Effects: Finds an iterator to the first element whose value is "value" or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

49.
```
template<typename KeyType, typename KeyValueCompare>
  iterator find(const KeyType & key, KeyValueCompare comp);
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Finds an iterator to the first element whose key is "key" according to the comparison functor or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

50.
```
const_iterator find(const_reference value) const;
```

Effects: Finds a const_iterator to the first element whose value is "value" or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

51.
```
template<typename KeyType, typename KeyValueCompare>
  const_iterator find(const KeyType & key, KeyValueCompare comp) const;
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Finds a const_iterator to the first element whose key is "key" according to the comparison functor or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

52.
```
std::pair< iterator, iterator > equal_range(const_reference value);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Finds a range containing all elements whose key is k or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

53.
```
template<typename KeyType, typename KeyValueCompare>
  std::pair< iterator, iterator >
  equal_range(const KeyType & key, KeyValueCompare comp);
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Finds a range containing all elements whose key is k according to the comparison functor or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

54.
```
std::pair< const_iterator, const_iterator >
equal_range(const_reference value) const;
```

Effects: Finds a range containing all elements whose key is k or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

55.
```
template<typename KeyType, typename KeyValueCompare>
  std::pair< const_iterator, const_iterator >
  equal_range(const KeyType & key, KeyValueCompare comp) const;
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Finds a range containing all elements whose key is k according to the comparison functor or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

56.
```
std::pair< iterator, iterator >
bounded_range(const_reference lower_value, const_reference upper_value,
              bool left_closed, bool right_closed);
```

Requires: 'lower_value' must not be greater than 'upper_value'. If 'lower_value' == 'upper_value', ('left_closed' || 'right_closed') must be false.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key) if left_closed, upper_bound(lower_key) otherwise

second = upper_bound(upper_key) if right_closed, lower_bound(upper_key) otherwise

Complexity: Logarithmic.

Throws: If the predicate throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_value and upper_value.

57.
```
template<typename KeyType, typename KeyValueCompare>
  std::pair< iterator, iterator >
  bounded_range(const KeyType & lower_key, const KeyType & upper_key,
                KeyValueCompare comp, bool left_closed, bool right_closed);
```

Requires: KeyValueCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. 'lower_key' must not be greater than 'upper_key' according to 'comp'. If 'lower_key' == 'upper_key', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key, comp) if left_closed, upper_bound(lower_key, comp) otherwise

second = upper_bound(upper_key, comp) if right_closed, lower_bound(upper_key, comp) otherwise

Complexity: Logarithmic.

Throws: If "comp" throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_key and upper_key.

58.
```
std::pair< const_iterator, const_iterator >
bounded_range(const_reference lower_value, const_reference upper_value,
              bool left_closed, bool right_closed) const;
```

Requires: 'lower_value' must not be greater than 'upper_value'. If 'lower_value' == 'upper_value', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key) if left_closed, upper_bound(lower_key) otherwise

second = upper_bound(upper_key) if right_closed, lower_bound(upper_key) otherwise

Complexity: Logarithmic.

Throws: If the predicate throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_value and upper_value.

59.
```
template<typename KeyType, typename KeyValueCompare>
  std::pair< const_iterator, const_iterator >
  bounded_range(const KeyType & lower_key, const KeyType & upper_key,
                KeyValueCompare comp, bool left_closed, bool right_closed) const;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Requires: KeyValueCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. 'lower_key' must not be greater than 'upper_key' according to 'comp'. If 'lower_key' == 'upper_key', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key, comp) if left_closed, upper_bound(lower_key, comp) otherwise

second = upper_bound(upper_key, comp) if right_closed, lower_bound(upper_key, comp) otherwise

Complexity: Logarithmic.

Throws: If "comp" throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_key and upper_key.

60.
```
iterator iterator_to(reference value);
```

Requires: value must be an lvalue and shall be in a `avl_set` of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid iterator i belonging to the `avl_set` that points to the value

Complexity: Constant.

Throws: Nothing.

61.
```
const_iterator iterator_to(const_reference value) const;
```

Requires: value must be an lvalue and shall be in a `avl_set` of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid const_iterator i belonging to the `avl_set` that points to the value

Complexity: Constant.

Throws: Nothing.

62.
```
pointer unlink_leftmost_without_rebalance();
```

Effects: Unlinks the leftmost node from the tree.

Complexity: Average complexity is constant time.

Throws: Nothing.

Notes: This function breaks the tree and the tree can only be used for more unlink_leftmost_without_rebalance calls. This function is normally used to achieve a step by step controlled destruction of the tree.

63.
```
void replace_node(iterator replace_this, reference with_this);
```

Requires: replace_this must be a valid iterator of *this and with_this must not be inserted in any tree.

Effects: Replaces replace_this in its position in the tree with with_this. The tree does not need to be rebalanced.

Complexity: Constant.

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Note: This function will break container ordering invariants if with_this is not equivalent to *replace_this according to the ordering rules. This function is faster than erasing and inserting the node, since no rebalancing or comparison is needed.

`avl_set` public static functions

1.
```
static avl_set & container_from_end_iterator(iterator end_iterator);
```

Precondition: end_iterator must be a valid end iterator of `avl_set`.

Effects: Returns a const reference to the `avl_set` associated to the end iterator

Throws: Nothing.

Complexity: Constant.

2.
```
static const avl_set &
container_from_end_iterator(const_iterator end_iterator);
```

Precondition: end_iterator must be a valid end const_iterator of `avl_set`.

Effects: Returns a const reference to the set associated to the end iterator

Throws: Nothing.

Complexity: Constant.

3.
```
static avl_set & container_from_iterator(iterator it);
```

Precondition: it must be a valid iterator of set.

Effects: Returns a reference to the set associated to the iterator

Throws: Nothing.

Complexity: Logarithmic.

4.
```
static const avl_set & container_from_iterator(const_iterator it);
```

Precondition: it must be a valid const_iterator of set.

Effects: Returns a const reference to the set associated to the iterator

Throws: Nothing.

Complexity: Logarithmic.

5.
```
static iterator s_iterator_to(reference value);
```

Requires: value must be an lvalue and shall be in a `avl_set` of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid iterator i belonging to the `avl_set` that points to the value

Complexity: Constant.

Throws: Nothing.

Note: This static function is available only if the *value traits* is stateless.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

6.
```
static const_iterator s_iterator_to(const_reference value);
```

Requires: value must be an lvalue and shall be in a `avl_set` of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid const_iterator i belonging to the `avl_set` that points to the value

Complexity: Constant.

Throws: Nothing.

Note: This static function is available only if the *value traits* is stateless.

7.
```
static void init_node(reference value);
```

Requires: value shall not be in a avl_set/avl_multiset.

Effects: init_node puts the hook of a value in a well-known default state.

Throws: Nothing.

Complexity: Constant time.

Note: This function puts the hook in the well-known default state used by auto_unlink and safe hooks.

Struct template make_avl_set

boost::intrusive::make_avl_set

Synopsis

```
// In header: <boost/intrusive/avl_set.hpp>

template<typename T, class... Options>
struct make_avl_set {
  // types
  typedef implementation_defined type;
};
```

Description

Helper metafunction to define a `avl_set` that yields to the same type when the same options (either explicitly or implicitly) are used.

Class template avl_multiset

boost::intrusive::avl_multiset

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/intrusive/avl_set.hpp>

template<typename T, class... Options>
class avl_multiset {
public:
  // types
  typedef implementation_defined::value_type             value_type;
  typedef implementation_defined::value_traits           value_traits;
  typedef implementation_defined::pointer                pointer;
  typedef implementation_defined::const_pointer          const_pointer;
  typedef implementation_defined::reference              reference;
  typedef implementation_defined::const_reference        const_reference;
  typedef implementation_defined::difference_type        difference_type;
  typedef implementation_defined::size_type              size_type;
  typedef implementation_defined::value_compare          value_compare;
  typedef implementation_defined::key_compare            key_compare;
  typedef implementation_defined::iterator               iterator;
  typedef implementation_defined::const_iterator         const_iterator;
  typedef implementation_defined::reverse_iterator       reverse_iterator;
  typedef implementation_defined::const_reverse_iterator const_reverse_iterator;
  typedef implementation_defined::insert_commit_data     insert_commit_data;
  typedef implementation_defined::node_traits            node_traits;
  typedef implementation_defined::node                   node;
  typedef implementation_defined::node_ptr               node_ptr;
  typedef implementation_defined::const_node_ptr         const_node_ptr;
  typedef implementation_defined::node_algorithms        node_algorithms;

  // construct/copy/destruct
  explicit avl_multiset(const value_compare & = value_compare(),
                        const value_traits & = value_traits());
  template<typename Iterator>
    avl_multiset(Iterator, Iterator, const value_compare & = value_compare(),
                 const value_traits & = value_traits());
  avl_multiset(BOOST_RV_REF(avl_multiset));
  avl_multiset& operator=(BOOST_RV_REF(avl_multiset));
  ~avl_multiset();

  // public member functions
  iterator begin();
  const_iterator begin() const;
  const_iterator cbegin() const;
  iterator end();
  const_iterator end() const;
  const_iterator cend() const;
  reverse_iterator rbegin();
  const_reverse_iterator rbegin() const;
  const_reverse_iterator crbegin() const;
  reverse_iterator rend();
  const_reverse_iterator rend() const;
  const_reverse_iterator crend() const;
  key_compare key_comp() const;
  value_compare value_comp() const;
  bool empty() const;
  size_type size() const;
  void swap(avl_multiset &);
  template<typename Cloner, typename Disposer>
    void clone_from(const avl_multiset &, Cloner, Disposer);
  iterator insert(reference);
  iterator insert(const_iterator, reference);
  template<typename Iterator> void insert(Iterator, Iterator);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
iterator insert_before(const_iterator, reference);
void push_back(reference);
void push_front(reference);
iterator erase(const_iterator);
iterator erase(const_iterator, const_iterator);
size_type erase(const_reference);
template<typename KeyType, typename KeyValueCompare> 
  size_type erase(const KeyType &, KeyValueCompare);
template<typename Disposer> 
  iterator erase_and_dispose(const_iterator, Disposer);
template<typename Disposer> 
  iterator erase_and_dispose(const_iterator, const_iterator, Disposer);
template<typename Disposer> 
  size_type erase_and_dispose(const_reference, Disposer);
template<typename KeyType, typename KeyValueCompare, typename Disposer> 
  size_type erase_and_dispose(const KeyType &, KeyValueCompare, Disposer);
void clear();
template<typename Disposer> void clear_and_dispose(Disposer);
size_type count(const_reference) const;
template<typename KeyType, typename KeyValueCompare> 
  size_type count(const KeyType &, KeyValueCompare) const;
iterator lower_bound(const_reference);
template<typename KeyType, typename KeyValueCompare> 
  iterator lower_bound(const KeyType &, KeyValueCompare);
const_iterator lower_bound(const_reference) const;
template<typename KeyType, typename KeyValueCompare> 
  const_iterator lower_bound(const KeyType &, KeyValueCompare) const;
iterator upper_bound(const_reference);
template<typename KeyType, typename KeyValueCompare> 
  iterator upper_bound(const KeyType &, KeyValueCompare);
const_iterator upper_bound(const_reference) const;
template<typename KeyType, typename KeyValueCompare> 
  const_iterator upper_bound(const KeyType &, KeyValueCompare) const;
iterator find(const_reference);
template<typename KeyType, typename KeyValueCompare> 
  iterator find(const KeyType &, KeyValueCompare);
const_iterator find(const_reference) const;
template<typename KeyType, typename KeyValueCompare> 
  const_iterator find(const KeyType &, KeyValueCompare) const;
std::pair< iterator, iterator > equal_range(const_reference);
template<typename KeyType, typename KeyValueCompare> 
  std::pair< iterator, iterator > 
  equal_range(const KeyType &, KeyValueCompare);
std::pair< const_iterator, const_iterator > 
equal_range(const_reference) const;
template<typename KeyType, typename KeyValueCompare> 
  std::pair< const_iterator, const_iterator > 
  equal_range(const KeyType &, KeyValueCompare) const;
std::pair< iterator, iterator > 
bounded_range(const_reference, const_reference, bool, bool);
template<typename KeyType, typename KeyValueCompare> 
  std::pair< iterator, iterator > 
  bounded_range(const KeyType &, const KeyType &, KeyValueCompare, bool, 
                bool);
std::pair< const_iterator, const_iterator > 
bounded_range(const_reference, const_reference, bool, bool) const;
template<typename KeyType, typename KeyValueCompare> 
  std::pair< const_iterator, const_iterator > 
  bounded_range(const KeyType &, const KeyType &, KeyValueCompare, bool, 
                bool) const;
iterator iterator_to(reference);
const_iterator iterator_to(const_reference) const;
pointer unlink_leftmost_without_rebalance();
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
  void replace_node(iterator, reference);

  // public static functions
  static avl_multiset & container_from_end_iterator(iterator);
  static const avl_multiset & container_from_end_iterator(const_iterator);
  static avl_multiset & container_from_iterator(iterator);
  static const avl_multiset & container_from_iterator(const_iterator);
  static iterator s_iterator_to(reference);
  static const_iterator s_iterator_to(const_reference);
  static void init_node(reference);

  // public data members
  static const bool constant_time_size;
};
```

Description

The class template avl_multiset is an intrusive container, that mimics most of the interface of std::avl_multiset as described in the C++ standard.

The template parameter `T` is the type to be managed by the container. The user can specify additional options and if no options are provided default options are used.

The container supports the following options: `base_hook<>/member_hook<>/value_traits<>`, `constant_time_size<>`, `size_type<>` and `compare<>`.

`avl_multiset` public construct/copy/destruct

1.
```
explicit avl_multiset(const value_compare & cmp = value_compare(),
                      const value_traits & v_traits = value_traits());
```

   Effects: Constructs an empty `avl_multiset`.

   Complexity: Constant.

   Throws: If value_traits::node_traits::node constructor throws (this does not happen with predefined Boost.Intrusive hooks) or the copy constructor/operator() of the value_compare object throws.

2.
```
template<typename Iterator>
  avl_multiset(Iterator b, Iterator e,
               const value_compare & cmp = value_compare(),
               const value_traits & v_traits = value_traits());
```

   Requires: Dereferencing iterator must yield an lvalue of type value_type. cmp must be a comparison function that induces a strict weak ordering.

   Effects: Constructs an empty `avl_multiset` and inserts elements from [b, e).

   Complexity: Linear in N if [b, e) is already sorted using comp and otherwise N * log N, where N is the distance between first and last

   Throws: If value_traits::node_traits::node constructor throws (this does not happen with predefined Boost.Intrusive hooks) or the copy constructor/operator() of the value_compare object throws.

3.
```
avl_multiset(BOOST_RV_REF(avl_multiset) x);
```

   Effects: to-do

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

4.
```
avl_multiset& operator=(BOOST_RV_REF(avl_multiset) x);
```

Effects: to-do

5.
```
~avl_multiset();
```

Effects: Detaches all elements from this. The objects in the `avl_multiset` are not deleted (i.e. no destructors are called).

Complexity: Linear to the number of elements on the container. if it's a safe-mode or auto-unlink value_type. Constant time otherwise.

Throws: Nothing.

`avl_multiset` public member functions

1.
```
iterator begin();
```

Effects: Returns an iterator pointing to the beginning of the `avl_multiset`.

Complexity: Constant.

Throws: Nothing.

2.
```
const_iterator begin() const;
```

Effects: Returns a const_iterator pointing to the beginning of the `avl_multiset`.

Complexity: Constant.

Throws: Nothing.

3.
```
const_iterator cbegin() const;
```

Effects: Returns a const_iterator pointing to the beginning of the `avl_multiset`.

Complexity: Constant.

Throws: Nothing.

4.
```
iterator end();
```

Effects: Returns an iterator pointing to the end of the `avl_multiset`.

Complexity: Constant.

Throws: Nothing.

5.
```
const_iterator end() const;
```

Effects: Returns a const_iterator pointing to the end of the `avl_multiset`.

Complexity: Constant.

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

6.
```
const_iterator cend() const;
```

   Effects: Returns a const_iterator pointing to the end of the `avl_multiset`.

   Complexity: Constant.

   Throws: Nothing.

7.
```
reverse_iterator rbegin();
```

   Effects: Returns a reverse_iterator pointing to the beginning of the reversed `avl_multiset`.

   Complexity: Constant.

   Throws: Nothing.

8.
```
const_reverse_iterator rbegin() const;
```

   Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed `avl_multiset`.

   Complexity: Constant.

   Throws: Nothing.

9.
```
const_reverse_iterator crbegin() const;
```

   Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed `avl_multiset`.

   Complexity: Constant.

   Throws: Nothing.

10.
```
reverse_iterator rend();
```

   Effects: Returns a reverse_iterator pointing to the end of the reversed `avl_multiset`.

   Complexity: Constant.

   Throws: Nothing.

11.
```
const_reverse_iterator rend() const;
```

   Effects: Returns a const_reverse_iterator pointing to the end of the reversed `avl_multiset`.

   Complexity: Constant.

   Throws: Nothing.

12.
```
const_reverse_iterator crend() const;
```

   Effects: Returns a const_reverse_iterator pointing to the end of the reversed `avl_multiset`.

   Complexity: Constant.

   Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

13.
```
key_compare key_comp() const;
```

Effects: Returns the key_compare object used by the `avl_multiset`.

Complexity: Constant.

Throws: If key_compare copy-constructor throws.

14.
```
value_compare value_comp() const;
```

Effects: Returns the value_compare object used by the `avl_multiset`.

Complexity: Constant.

Throws: If value_compare copy-constructor throws.

15.
```
bool empty() const;
```

Effects: Returns true is the container is empty.

Complexity: Constant.

Throws: Nothing.

16.
```
size_type size() const;
```

Effects: Returns the number of elements stored in the `avl_multiset`.

Complexity: Linear to elements contained in *this if, constant-time size option is enabled. Constant-time otherwise.

Throws: Nothing.

17.
```
void swap(avl_multiset & other);
```

Effects: Swaps the contents of two avl_multisets.

Complexity: Constant.

Throws: If the swap() call for the comparison functor found using ADL throws. Strong guarantee.

18.
```
template<typename Cloner, typename Disposer> 
  void clone_from(const avl_multiset & src, Cloner cloner, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw. Cloner should yield to nodes equivalent to the original nodes.

Effects: Erases all the elements from *this calling Disposer::operator()(pointer), clones all the elements from src calling Cloner::operator()(const_reference) and inserts them on *this. Copies the predicate from the source container.

If cloner throws, all cloned elements are unlinked and disposed calling Disposer::operator()(pointer).

Complexity: Linear to erased plus inserted elements.

Throws: If cloner throws or predicate copy assignment throws. Basic guarantee.

19.
```
iterator insert(reference value);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Requires: value must be an lvalue

Effects: Inserts value into the `avl_multiset`.

Returns: An iterator that points to the position where the new element was inserted.

Complexity: Average complexity for insert element is at most logarithmic.

Throws: If the internal value_compare ordering function throws. Strong guarantee.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

20.
```
iterator insert(const_iterator hint, reference value);
```

Requires: value must be an lvalue

Effects: Inserts x into the `avl_multiset`, using pos as a hint to where it will be inserted.

Returns: An iterator that points to the position where the new element was inserted.

Complexity: Logarithmic in general, but it is amortized constant time if t is inserted immediately before hint.

Throws: If the internal value_compare ordering function throws. Strong guarantee.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

21.
```
template<typename Iterator> void insert(Iterator b, Iterator e);
```

Requires: Dereferencing iterator must yield an lvalue of type value_type.

Effects: Inserts a range into the `avl_multiset`.

Returns: An iterator that points to the position where the new element was inserted.

Complexity: Insert range is in general O(N * log(N)), where N is the size of the range. However, it is linear in N if the range is already sorted by value_comp().

Throws: If the internal value_compare ordering function throws. Basic guarantee.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

22.
```
iterator insert_before(const_iterator pos, reference value);
```

Requires: value must be an lvalue, "pos" must be a valid iterator (or end) and must be the succesor of value once inserted according to the predicate. "value" must not be equal to any inserted key according to the predicate.

Effects: Inserts x into the tree before "pos".

Complexity: Constant time.

Throws: Nothing.

Note: This function does not check preconditions so if "pos" is not the successor of "value" or "value" is not unique tree ordering and uniqueness invariants will be broken respectively. This is a low-level function to be used only for performance reasons by advanced users.

23.
```
void push_back(reference value);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Requires: value must be an lvalue, and it must be greater than any inserted key according to the predicate.

Effects: Inserts x into the tree in the last position.

Complexity: Constant time.

Throws: Nothing.

Note: This function does not check preconditions so if value is less than or equal to the greatest inserted key tree ordering invariant will be broken. This function is slightly more efficient than using "insert_before". This is a low-level function to be used only for performance reasons by advanced users.

24.
```
void push_front(reference value);
```

Requires: value must be an lvalue, and it must be less than any inserted key according to the predicate.

Effects: Inserts x into the tree in the first position.

Complexity: Constant time.

Throws: Nothing.

Note: This function does not check preconditions so if value is greater than or equal to the the mimum inserted key tree ordering or uniqueness invariants will be broken. This function is slightly more efficient than using "insert_before". This is a low-level function to be used only for performance reasons by advanced users.

25.
```
iterator erase(const_iterator i);
```

Effects: Erases the element pointed to by pos.

Complexity: Average complexity is constant time.

Returns: An iterator to the element after the erased element.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

26.
```
iterator erase(const_iterator b, const_iterator e);
```

Effects: Erases the range pointed to by b end e.

Returns: An iterator to the element after the erased elements.

Complexity: Average complexity for erase range is at most O(log(size() + N)), where N is the number of elements in the range.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

27.
```
size_type erase(const_reference value);
```

Effects: Erases all the elements with the given value.

Returns: The number of erased elements.

Complexity: O(log(size() + this->count(value)).

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: If the internal value_compare ordering function throws. Basic guarantee.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

28.
```
template<typename KeyType, typename KeyValueCompare>
  size_type erase(const KeyType & key, KeyValueCompare comp);
```

Effects: Erases all the elements that compare equal with the given key and the given comparison functor.

Returns: The number of erased elements.

Complexity: O(log(size() + this->count(key, comp)).

Throws: If comp ordering function throws. Basic guarantee.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

29.
```
template<typename Disposer>
  iterator erase_and_dispose(const_iterator i, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Returns: An iterator to the element after the erased element.

Effects: Erases the element pointed to by pos. Disposer::operator()(pointer) is called for the removed element.

Complexity: Average complexity for erase element is constant time.

Throws: Nothing.

Note: Invalidates the iterators to the erased elements.

30.
```
template<typename Disposer>
  iterator erase_and_dispose(const_iterator b, const_iterator e,
                             Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Returns: An iterator to the element after the erased elements.

Effects: Erases the range pointed to by b end e. Disposer::operator()(pointer) is called for the removed elements.

Complexity: Average complexity for erase range is at most O(log(size() + N)), where N is the number of elements in the range.

Throws: Nothing.

Note: Invalidates the iterators to the erased elements.

31.
```
template<typename Disposer>
  size_type erase_and_dispose(const_reference value, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases all the elements with the given value. Disposer::operator()(pointer) is called for the removed elements.

Returns: The number of erased elements.

Complexity: O(log(size() + this->count(value)).

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: If the internal value_compare ordering function throws. Basic guarantee.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

32.
```
template<typename KeyType, typename KeyValueCompare, typename Disposer> 
  size_type erase_and_dispose(const KeyType & key, KeyValueCompare comp, 
                              Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases all the elements with the given key. according to the comparison functor "comp". Disposer::operator()(pointer) is called for the removed elements.

Returns: The number of erased elements.

Complexity: O(log(size() + this->count(key, comp)).

Throws: If comp ordering function throws. Basic guarantee.

Note: Invalidates the iterators to the erased elements.

33.
```
void clear();
```

Effects: Erases all the elements of the container.

Complexity: Linear to the number of elements on the container. if it's a safe-mode or auto-unlink value_type. Constant time otherwise.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

34.
```
template<typename Disposer> void clear_and_dispose(Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases all the elements of the container.

Complexity: Linear to the number of elements on the container. Disposer::operator()(pointer) is called for the removed elements.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

35.
```
size_type count(const_reference value) const;
```

Effects: Returns the number of contained elements with the given key

Complexity: Logarithmic to the number of elements contained plus lineal to number of objects with the given key.

Throws: If the internal value_compare ordering function throws.

36.
```
template<typename KeyType, typename KeyValueCompare> 
  size_type count(const KeyType & key, KeyValueCompare comp) const;
```

Effects: Returns the number of contained elements with the same key compared with the given comparison functor.

Complexity: Logarithmic to the number of elements contained plus lineal to number of objects with the given key.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: If comp ordering function throws.

37.
```
iterator lower_bound(const_reference value);
```

Effects: Returns an iterator to the first element whose key is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

38.
```
template<typename KeyType, typename KeyValueCompare>
  iterator lower_bound(const KeyType & key, KeyValueCompare comp);
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Returns an iterator to the first element whose key according to the comparison functor is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

39.
```
const_iterator lower_bound(const_reference value) const;
```

Effects: Returns a const iterator to the first element whose key is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

40.
```
template<typename KeyType, typename KeyValueCompare>
  const_iterator lower_bound(const KeyType & key, KeyValueCompare comp) const;
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Returns a const_iterator to the first element whose key according to the comparison functor is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

41.
```
iterator upper_bound(const_reference value);
```

Effects: Returns an iterator to the first element whose key is greater than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

42.
```
template<typename KeyType, typename KeyValueCompare> 
  iterator upper_bound(const KeyType & key, KeyValueCompare comp);
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Returns an iterator to the first element whose key according to the comparison functor is greater than key or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

43.
```
const_iterator upper_bound(const_reference value) const;
```

Effects: Returns an iterator to the first element whose key is greater than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

44.
```
template<typename KeyType, typename KeyValueCompare> 
  const_iterator upper_bound(const KeyType & key, KeyValueCompare comp) const;
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Returns a const_iterator to the first element whose key according to the comparison functor is greater than key or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

45.
```
iterator find(const_reference value);
```

Effects: Finds an iterator to the first element whose value is "value" or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

46.
```
template<typename KeyType, typename KeyValueCompare> 
  iterator find(const KeyType & key, KeyValueCompare comp);
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Finds an iterator to the first element whose key is "key" according to the comparison functor or end() if that element does not exist.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

47.
```
const_iterator find(const_reference value) const;
```

Effects: Finds a const_iterator to the first element whose value is "value" or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

48.
```
template<typename KeyType, typename KeyValueCompare>
  const_iterator find(const KeyType & key, KeyValueCompare comp) const;
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Finds a const_iterator to the first element whose key is "key" according to the comparison functor or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

49.
```
std::pair< iterator, iterator > equal_range(const_reference value);
```

Effects: Finds a range containing all elements whose key is k or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

50.
```
template<typename KeyType, typename KeyValueCompare>
  std::pair< iterator, iterator >
  equal_range(const KeyType & key, KeyValueCompare comp);
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Finds a range containing all elements whose key is k according to the comparison functor or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

51.
```
std::pair< const_iterator, const_iterator >
equal_range(const_reference value) const;
```

Effects: Finds a range containing all elements whose key is k or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

52.
```
template<typename KeyType, typename KeyValueCompare>
  std::pair< const_iterator, const_iterator >
  equal_range(const KeyType & key, KeyValueCompare comp) const;
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Finds a range containing all elements whose key is k according to the comparison functor or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

53.
```
std::pair< iterator, iterator >
bounded_range(const_reference lower_value, const_reference upper_value,
              bool left_closed, bool right_closed);
```

Requires: 'lower_value' must not be greater than 'upper_value'. If 'lower_value' == 'upper_value', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key) if left_closed, upper_bound(lower_key) otherwise

second = upper_bound(upper_key) if right_closed, lower_bound(upper_key) otherwise

Complexity: Logarithmic.

Throws: If the predicate throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_value and upper_value.

54.
```
template<typename KeyType, typename KeyValueCompare>
  std::pair< iterator, iterator >
  bounded_range(const KeyType & lower_key, const KeyType & upper_key,
                KeyValueCompare comp, bool left_closed, bool right_closed);
```

Requires: KeyValueCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. 'lower_key' must not be greater than 'upper_key' according to 'comp'. If 'lower_key' == 'upper_key', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key, comp) if left_closed, upper_bound(lower_key, comp) otherwise

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

second = upper_bound(upper_key, comp) if right_closed, lower_bound(upper_key, comp) otherwise

Complexity: Logarithmic.

Throws: If "comp" throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_key and upper_key.

55.
```
std::pair< const_iterator, const_iterator >
bounded_range(const_reference lower_value, const_reference upper_value,
              bool left_closed, bool right_closed) const;
```

Requires: 'lower_value' must not be greater than 'upper_value'. If 'lower_value' == 'upper_value', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key) if left_closed, upper_bound(lower_key) otherwise

second = upper_bound(upper_key) if right_closed, lower_bound(upper_key) otherwise

Complexity: Logarithmic.

Throws: If the predicate throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_value and upper_value.

56.
```
template<typename KeyType, typename KeyValueCompare>
  std::pair< const_iterator, const_iterator >
  bounded_range(const KeyType & lower_key, const KeyType & upper_key,
                KeyValueCompare comp, bool left_closed, bool right_closed) const;
```

Requires: KeyValueCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. 'lower_key' must not be greater than 'upper_key' according to 'comp'. If 'lower_key' == 'upper_key', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key, comp) if left_closed, upper_bound(lower_key, comp) otherwise

second = upper_bound(upper_key, comp) if right_closed, lower_bound(upper_key, comp) otherwise

Complexity: Logarithmic.

Throws: If "comp" throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_key and upper_key.

57.
```
iterator iterator_to(reference value);
```

Requires: value must be an lvalue and shall be in a `avl_multiset` of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid iterator i belonging to the `avl_multiset` that points to the value

Complexity: Constant.

Throws: Nothing.

58.
```
const_iterator iterator_to(const_reference value) const;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Requires: value must be an lvalue and shall be in a `avl_multiset` of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid const_iterator i belonging to the `avl_multiset` that points to the value

Complexity: Constant.

Throws: Nothing.

59.
```
pointer unlink_leftmost_without_rebalance();
```

Effects: Unlinks the leftmost node from the tree.

Complexity: Average complexity is constant time.

Throws: Nothing.

Notes: This function breaks the tree and the tree can only be used for more unlink_leftmost_without_rebalance calls. This function is normally used to achieve a step by step controlled destruction of the tree.

60.
```
void replace_node(iterator replace_this, reference with_this);
```

Requires: replace_this must be a valid iterator of *this and with_this must not be inserted in any tree.

Effects: Replaces replace_this in its position in the tree with with_this. The tree does not need to be rebalanced.

Complexity: Constant.

Throws: Nothing.

Note: This function will break container ordering invariants if with_this is not equivalent to *replace_this according to the ordering rules. This function is faster than erasing and inserting the node, since no rebalancing or comparison is needed.

`avl_multiset` public static functions

1.
```
static avl_multiset & container_from_end_iterator(iterator end_iterator);
```

Precondition: end_iterator must be a valid end iterator of `avl_multiset`.

Effects: Returns a const reference to the `avl_multiset` associated to the end iterator

Throws: Nothing.

Complexity: Constant.

2.
```
static const avl_multiset &
container_from_end_iterator(const_iterator end_iterator);
```

Precondition: end_iterator must be a valid end const_iterator of `avl_multiset`.

Effects: Returns a const reference to the `avl_multiset` associated to the end iterator

Throws: Nothing.

Complexity: Constant.

3.
```
static avl_multiset & container_from_iterator(iterator it);
```

Precondition: it must be a valid iterator of multiset.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Returns a const reference to the multiset associated to the iterator

Throws: Nothing.

Complexity: Logarithmic.

4.
```
static const avl_multiset & container_from_iterator(const_iterator it);
```

Precondition: it must be a valid const_iterator of multiset.

Effects: Returns a const reference to the multiset associated to the iterator

Throws: Nothing.

Complexity: Logarithmic.

5.
```
static iterator s_iterator_to(reference value);
```

Requires: value must be an lvalue and shall be in a `avl_multiset` of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid iterator i belonging to the `avl_multiset` that points to the value

Complexity: Constant.

Throws: Nothing.

Note: This static function is available only if the *value traits* is stateless.

6.
```
static const_iterator s_iterator_to(const_reference value);
```

Requires: value must be an lvalue and shall be in a `avl_multiset` of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid const_iterator i belonging to the `avl_multiset` that points to the value

Complexity: Constant.

Throws: Nothing.

Note: This static function is available only if the *value traits* is stateless.

7.
```
static void init_node(reference value);
```

Requires: value shall not be in a avl_multiset/avl_multiset.

Effects: init_node puts the hook of a value in a well-known default state.

Throws: Nothing.

Complexity: Constant time.

Note: This function puts the hook in the well-known default state used by auto_unlink and safe hooks.

Struct template make_avl_multiset

boost::intrusive::make_avl_multiset

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/intrusive/avl_set.hpp>

template<typename T, class... Options>
struct make_avl_multiset {
  // types
  typedef implementation_defined type;
};
```

Description

Helper metafunction to define a `avl_multiset` that yields to the same type when the same options (either explicitly or implicitly) are used.

Header <boost/intrusive/avl_set_hook.hpp>

```
namespace boost {
  namespace intrusive {
    template<class... Options> struct make_avl_set_base_hook;

    template<class... Options> class avl_set_base_hook;

    template<class... Options> struct make_avl_set_member_hook;

    template<class... Options> class avl_set_member_hook;
  }
}
```

Struct template make_avl_set_base_hook

boost::intrusive::make_avl_set_base_hook

Synopsis

```
// In header: <boost/intrusive/avl_set_hook.hpp>

template<class... Options>
struct make_avl_set_base_hook {
  // types
  typedef implementation_defined type;
};
```

Description

Helper metafunction to define a `avl_set_base_hook` that yields to the same type when the same options (either explicitly or implicitly) are used.

Class template avl_set_base_hook

boost::intrusive::avl_set_base_hook

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/intrusive/avl_set_hook.hpp>

template<class... Options>
class avl_set_base_hook :
  public make_avl_set_base_hook::type< O1, O2, O3, O4 >
{
public:
  // construct/copy/destruct
  avl_set_base_hook();
  avl_set_base_hook(const avl_set_base_hook &);
  avl_set_base_hook& operator=(const avl_set_base_hook &);
  ~avl_set_base_hook();

  // public member functions
  void swap_nodes(avl_set_base_hook &);
  bool is_linked() const;
  void unlink();
};
```

Description

Derive a class from avl_set_base_hook in order to store objects in in an avl_set/avl_multiset. avl_set_base_hook holds the data necessary to maintain the avl_set/avl_multiset and provides an appropriate value_traits class for avl_set/avl_multiset.

The hook admits the following options: `tag<>`, `void_pointer<>`, `link_mode<>` and `optimize_size<>`.

`tag<>` defines a tag to identify the node. The same tag value can be used in different classes, but if a class is derived from more than one `list_base_hook`, then each `list_base_hook` needs its unique tag.

`void_pointer<>` is the pointer type that will be used internally in the hook and the the container configured to use this hook.

`link_mode<>` will specify the linking mode of the hook (`normal_link`, `auto_unlink` or `safe_link`).

`optimize_size<>` will tell the hook to optimize the hook for size instead of speed.

`avl_set_base_hook` public construct/copy/destruct

1.
   ```
   avl_set_base_hook();
   ```

   Effects: If `link_mode` is `auto_unlink` or `safe_link` initializes the node to an unlinked state.

   Throws: Nothing.

2.
   ```
   avl_set_base_hook(const avl_set_base_hook &);
   ```

   Effects: If `link_mode` is `auto_unlink` or `safe_link` initializes the node to an unlinked state. The argument is ignored.

   Throws: Nothing.

   Rationale: Providing a copy-constructor makes classes using the hook STL-compliant without forcing the user to do some additional work. `swap` can be used to emulate move-semantics.

3.
   ```
   avl_set_base_hook& operator=(const avl_set_base_hook &);
   ```

   Effects: Empty function. The argument is ignored.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: Nothing.

Rationale: Providing an assignment operator makes classes using the hook STL-compliant without forcing the user to do some additional work. `swap` can be used to emulate move-semantics.

4.
   ```
   ~avl_set_base_hook();
   ```

   Effects: If `link_mode` is `normal_link`, the destructor does nothing (ie. no code is generated). If `link_mode` is `safe_link` and the object is stored in a set an assertion is raised. If `link_mode` is `auto_unlink` and `is_linked()` is true, the node is unlinked.

   Throws: Nothing.

`avl_set_base_hook` public member functions

1.
   ```
   void swap_nodes(avl_set_base_hook & other);
   ```

   Effects: Swapping two nodes swaps the position of the elements related to those nodes in one or two containers. That is, if the node this is part of the element e1, the node x is part of the element e2 and both elements are included in the containers s1 and s2, then after the swap-operation e1 is in s2 at the position of e2 and e2 is in s1 at the position of e1. If one element is not in a container, then after the swap-operation the other element is not in a container. Iterators to e1 and e2 related to those nodes are invalidated.

   Complexity: Constant

   Throws: Nothing.

2.
   ```
   bool is_linked() const;
   ```

   Precondition: `link_mode` must be `safe_link` or `auto_unlink`.

   Returns: true, if the node belongs to a container, false otherwise. This function can be used to test whether `set::iterator_to` will return a valid iterator.

   Complexity: Constant

3.
   ```
   void unlink();
   ```

   Effects: Removes the node if it's inserted in a container. This function is only allowed if `link_mode` is `auto_unlink`.

   Throws: Nothing.

Struct template make_avl_set_member_hook

boost::intrusive::make_avl_set_member_hook

Synopsis

```
// In header: <boost/intrusive/avl_set_hook.hpp>

template<class... Options>
struct make_avl_set_member_hook {
  // types
  typedef implementation_defined type;
};
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Description

Helper metafunction to define a `avl_set_member_hook` that yields to the same type when the same options (either explicitly or implicitly) are used.

Class template avl_set_member_hook

boost::intrusive::avl_set_member_hook

Synopsis

```
// In header: <boost/intrusive/avl_set_hook.hpp>

template<class... Options> 
class avl_set_member_hook : 
  public make_avl_set_member_hook::type< O1, O2, O3, O4 > 
{
public:
  // construct/copy/destruct
  avl_set_member_hook();
  avl_set_member_hook(const avl_set_member_hook &);
  avl_set_member_hook& operator=(const avl_set_member_hook &);
  ~avl_set_member_hook();

  // public member functions
  void swap_nodes(avl_set_member_hook &);
  bool is_linked() const;
  void unlink();
};
```

Description

Put a public data member avl_set_member_hook in order to store objects of this class in an avl_set/avl_multiset. avl_set_member_hook holds the data necessary for maintaining the avl_set/avl_multiset and provides an appropriate value_traits class for avl_set/avl_multiset.

The hook admits the following options: `void_pointer<>`, `link_mode<>` and `optimize_size<>`.

`void_pointer<>` is the pointer type that will be used internally in the hook and the the container configured to use this hook.

`link_mode<>` will specify the linking mode of the hook (`normal_link`, `auto_unlink` or `safe_link`).

`optimize_size<>` will tell the hook to optimize the hook for size instead of speed.

`avl_set_member_hook` public construct/copy/destruct

1.
```
avl_set_member_hook();
```

   Effects: If `link_mode` is `auto_unlink` or `safe_link` initializes the node to an unlinked state.

   Throws: Nothing.

2.
```
avl_set_member_hook(const avl_set_member_hook &);
```

   Effects: If `link_mode` is `auto_unlink` or `safe_link` initializes the node to an unlinked state. The argument is ignored.

   Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Rationale: Providing a copy-constructor makes classes using the hook STL-compliant without forcing the user to do some additional work. `swap` can be used to emulate move-semantics.

3.
```
avl_set_member_hook& operator=(const avl_set_member_hook &);
```

Effects: Empty function. The argument is ignored.

Throws: Nothing.

Rationale: Providing an assignment operator makes classes using the hook STL-compliant without forcing the user to do some additional work. `swap` can be used to emulate move-semantics.

4.
```
~avl_set_member_hook();
```

Effects: If `link_mode` is `normal_link`, the destructor does nothing (ie. no code is generated). If `link_mode` is `safe_link` and the object is stored in a set an assertion is raised. If `link_mode` is `auto_unlink` and `is_linked()` is true, the node is unlinked.

Throws: Nothing.

`avl_set_member_hook` public member functions

1.
```
void swap_nodes(avl_set_member_hook & other);
```

Effects: Swapping two nodes swaps the position of the elements related to those nodes in one or two containers. That is, if the node this is part of the element e1, the node x is part of the element e2 and both elements are included in the containers s1 and s2, then after the swap-operation e1 is in s2 at the position of e2 and e2 is in s1 at the position of e1. If one element is not in a container, then after the swap-operation the other element is not in a container. Iterators to e1 and e2 related to those nodes are invalidated.

Complexity: Constant

Throws: Nothing.

2.
```
bool is_linked() const;
```

Precondition: `link_mode` must be `safe_link` or `auto_unlink`.

Returns: true, if the node belongs to a container, false otherwise. This function can be used to test whether `set::iterator_to` will return a valid iterator.

Complexity: Constant

3.
```
void unlink();
```

Effects: Removes the node if it's inserted in a container. This function is only allowed if `link_mode` is `auto_unlink`.

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Header <boost/intrusive/avltree.hpp>

```
namespace boost {
  namespace intrusive {
    template<typename T, class... Options> class avltree;

    template<typename T, class... Options> struct make_avltree;
    template<typename T, class... Options> 
      bool operator<(const avltree< T, Options...> & x, 
                     const avltree< T, Options...> & y);
    template<typename T, class... Options> 
      bool operator==(const avltree< T, Options...> & x, 
                      const avltree< T, Options...> & y);
    template<typename T, class... Options> 
      bool operator!=(const avltree< T, Options...> & x, 
                      const avltree< T, Options...> & y);
    template<typename T, class... Options> 
      bool operator>(const avltree< T, Options...> & x, 
                     const avltree< T, Options...> & y);
    template<typename T, class... Options> 
      bool operator<=(const avltree< T, Options...> & x, 
                      const avltree< T, Options...> & y);
    template<typename T, class... Options> 
      bool operator>=(const avltree< T, Options...> & x, 
                      const avltree< T, Options...> & y);
    template<typename T, class... Options> 
      void swap(avltree< T, Options...> & x, avltree< T, Options...> & y);
  }
}
```

Class template avltree

boost::intrusive::avltree

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/intrusive/avltree.hpp>

template<typename T, class... Options>
class avltree {
public:
  // types
  typedef Config::value_traits                                              value_traits; ↵

  typedef real_value_traits::pointer                                        pointer;      ↵

  typedef real_value_traits::const_pointer                                  const_pointer; ↵

  typedef boost::intrusive::pointer_traits< pointer >::element_type         value_type;   ↵

  typedef value_type                                                        key_type;     ↵

  typedef boost::intrusive::pointer_traits< pointer >::reference            reference;    ↵

  typedef boost::intrusive::pointer_traits< const_pointer >::reference        const_refer↵
ence;
  typedef boost::intrusive::pointer_traits< pointer >::difference_type      difference_type; ↵

  typedef Config::size_type                                                 size_type;    ↵

  typedef Config::compare                                                   value_compare; ↵

  typedef value_compare                                                     key_compare;  ↵

  typedef tree_iterator< avltree, false >                                   iterator;     ↵

  typedef tree_iterator< avltree, true >                                      const_iterat↵
or;
  typedef unspecified                                                       reverse_iterat↵
or;
  typedef unspecified                                                       const_reverse_iter↵
ator;
  typedef real_value_traits::node_traits                                    node_traits;  ↵

  typedef node_traits::node                                                 node;         ↵

  typedef pointer_traits< pointer >::template rebind_pointer< node >::type  node_ptr;     ↵

  typedef pointer_traits< pointer >::template rebind_pointer< const node >::type const_node_ptr; ↵

  typedef avltree_algorithms< node_traits >                                 node_algorithms; ↵

  typedef node_algorithms::insert_commit_data                                 insert_com↵
mit_data;

  // construct/copy/destruct
  explicit avltree(const value_compare & = value_compare(),
                   const value_traits & = value_traits());
  template<typename Iterator>
    avltree(bool, Iterator, Iterator, const value_compare & = value_compare(),
            const value_traits & = value_traits());
  avltree(BOOST_RV_REF(avltree));
  avltree& operator=(BOOST_RV_REF(avltree));
  ~avltree();

  // public member functions
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
const real_value_traits & get_real_value_traits() const;
real_value_traits & get_real_value_traits();
iterator begin();
const_iterator begin() const;
const_iterator cbegin() const;
iterator end();
const_iterator end() const;
const_iterator cend() const;
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
const_reverse_iterator crbegin() const;
reverse_iterator rend();
const_reverse_iterator rend() const;
const_reverse_iterator crend() const;
value_compare value_comp() const;
bool empty() const;
size_type size() const;
void swap(avltree &);
iterator insert_equal(reference);
iterator insert_equal(const_iterator, reference);
template<typename Iterator> void insert_equal(Iterator, Iterator);
std::pair< iterator, bool > insert_unique(reference);
iterator insert_unique(const_iterator, reference);
template<typename Iterator> void insert_unique(Iterator, Iterator);
template<typename KeyType, typename KeyValueCompare>
  std::pair< iterator, bool >
  insert_unique_check(const KeyType &, KeyValueCompare,
                      insert_commit_data &);
template<typename KeyType, typename KeyValueCompare>
  std::pair< iterator, bool >
  insert_unique_check(const_iterator, const KeyType &, KeyValueCompare,
                      insert_commit_data &);
iterator insert_unique_commit(reference, const insert_commit_data &);
iterator insert_before(const_iterator, reference);
void push_back(reference);
void push_front(reference);
iterator erase(const_iterator);
iterator erase(const_iterator, const_iterator);
size_type erase(const_reference);
template<typename KeyType, typename KeyValueCompare>
  size_type erase(const KeyType &, KeyValueCompare);
template<typename Disposer>
  iterator erase_and_dispose(const_iterator, Disposer);
template<typename Disposer>
  iterator erase_and_dispose(const_iterator, const_iterator, Disposer);
template<typename Disposer>
  size_type erase_and_dispose(const_reference, Disposer);
template<typename KeyType, typename KeyValueCompare, typename Disposer>
  size_type erase_and_dispose(const KeyType &, KeyValueCompare, Disposer);
void clear();
template<typename Disposer> void clear_and_dispose(Disposer);
size_type count(const_reference) const;
template<typename KeyType, typename KeyValueCompare>
  size_type count(const KeyType &, KeyValueCompare) const;
iterator lower_bound(const_reference);
const_iterator lower_bound(const_reference) const;
template<typename KeyType, typename KeyValueCompare>
  iterator lower_bound(const KeyType &, KeyValueCompare);
template<typename KeyType, typename KeyValueCompare>
  const_iterator lower_bound(const KeyType &, KeyValueCompare) const;
iterator upper_bound(const_reference);
template<typename KeyType, typename KeyValueCompare>
  iterator upper_bound(const KeyType &, KeyValueCompare);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
  const_iterator upper_bound(const_reference) const;
  template<typename KeyType, typename KeyValueCompare>
    const_iterator upper_bound(const KeyType &, KeyValueCompare) const;
  iterator find(const_reference);
  template<typename KeyType, typename KeyValueCompare>
    iterator find(const KeyType &, KeyValueCompare);
  const_iterator find(const_reference) const;
  template<typename KeyType, typename KeyValueCompare>
    const_iterator find(const KeyType &, KeyValueCompare) const;
  std::pair< iterator, iterator > equal_range(const_reference);
  template<typename KeyType, typename KeyValueCompare>
    std::pair< iterator, iterator >
    equal_range(const KeyType &, KeyValueCompare);
  std::pair< const_iterator, const_iterator >
  equal_range(const_reference) const;
  template<typename KeyType, typename KeyValueCompare>
    std::pair< const_iterator, const_iterator >
    equal_range(const KeyType &, KeyValueCompare) const;
  std::pair< iterator, iterator >
  bounded_range(const_reference, const_reference, bool, bool);
  template<typename KeyType, typename KeyValueCompare>
    std::pair< iterator, iterator >
    bounded_range(const KeyType &, const KeyType &, KeyValueCompare, bool,
                  bool);
  std::pair< const_iterator, const_iterator >
  bounded_range(const_reference, const_reference, bool, bool) const;
  template<typename KeyType, typename KeyValueCompare>
    std::pair< const_iterator, const_iterator >
    bounded_range(const KeyType &, const KeyType &, KeyValueCompare, bool,
                  bool) const;
  template<typename Cloner, typename Disposer>
    void clone_from(const avltree &, Cloner, Disposer);
  pointer unlink_leftmost_without_rebalance();
  void replace_node(iterator, reference);
  iterator iterator_to(reference);
  const_iterator iterator_to(const_reference) const;

  // public static functions
  static avltree & container_from_end_iterator(iterator);
  static const avltree & container_from_end_iterator(const_iterator);
  static avltree & container_from_iterator(iterator);
  static const avltree & container_from_iterator(const_iterator);
  static iterator s_iterator_to(reference);
  static const_iterator s_iterator_to(const_reference);
  static void init_node(reference);

  // private static functions
  static avltree & priv_container_from_end_iterator(const const_iterator &);
  static avltree & priv_container_from_iterator(const const_iterator &);

  // public data members
  static const bool constant_time_size;
  static const bool stateful_value_traits;
};
```

Description

The class template avltree is an intrusive AVL tree container, that is used to construct intrusive avl_set and avl_multiset containers. The no-throw guarantee holds only, if the value_compare object doesn't throw.

The template parameter `T` is the type to be managed by the container. The user can specify additional options and if no options are provided default options are used.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

The container supports the following options: `base_hook<>/member_hook<>/value_traits<>`, `constant_time_size<>`, `size_type<>` and `compare<>`.

`avltree` public construct/copy/destruct

1.
```
explicit avltree(const value_compare & cmp = value_compare(),
                 const value_traits & v_traits = value_traits());
```

   Effects: Constructs an empty tree.

   Complexity: Constant.

   Throws: If value_traits::node_traits::node constructor throws (this does not happen with predefined Boost.Intrusive hooks) or the copy constructor of the value_compare object throws. Basic guarantee.

2.
```
template<typename Iterator>
  avltree(bool unique, Iterator b, Iterator e,
          const value_compare & cmp = value_compare(),
          const value_traits & v_traits = value_traits());
```

   Requires: Dereferencing iterator must yield an lvalue of type value_type. cmp must be a comparison function that induces a strict weak ordering.

   Effects: Constructs an empty tree and inserts elements from [b, e).

   Complexity: Linear in N if [b, e) is already sorted using comp and otherwise N * log N, where N is the distance between first and last.

   Throws: If value_traits::node_traits::node constructor throws (this does not happen with predefined Boost.Intrusive hooks) or the copy constructor/operator() of the value_compare object throws. Basic guarantee.

3.
```
avltree(BOOST_RV_REF(avltree) x);
```

   Effects: to-do

4.
```
avltree& operator=(BOOST_RV_REF(avltree) x);
```

   Effects: to-do

5.
```
~avltree();
```

   Effects: Detaches all elements from this. The objects in the set are not deleted (i.e. no destructors are called), but the nodes according to the `value_traits` template parameter are reinitialized and thus can be reused.

   Complexity: Linear to elements contained in *this.

   Throws: Nothing.

`avltree` public member functions

1.
```
const real_value_traits & get_real_value_traits() const;
```

2.
```
real_value_traits & get_real_value_traits();
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

3.
```
iterator begin();
```

Effects: Returns an iterator pointing to the beginning of the tree.

Complexity: Constant.

Throws: Nothing.

4.
```
const_iterator begin() const;
```

Effects: Returns a const_iterator pointing to the beginning of the tree.

Complexity: Constant.

Throws: Nothing.

5.
```
const_iterator cbegin() const;
```

Effects: Returns a const_iterator pointing to the beginning of the tree.

Complexity: Constant.

Throws: Nothing.

6.
```
iterator end();
```

Effects: Returns an iterator pointing to the end of the tree.

Complexity: Constant.

Throws: Nothing.

7.
```
const_iterator end() const;
```

Effects: Returns a const_iterator pointing to the end of the tree.

Complexity: Constant.

Throws: Nothing.

8.
```
const_iterator cend() const;
```

Effects: Returns a const_iterator pointing to the end of the tree.

Complexity: Constant.

Throws: Nothing.

9.
```
reverse_iterator rbegin();
```

Effects: Returns a reverse_iterator pointing to the beginning of the reversed tree.

Complexity: Constant.

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

10.
```
const_reverse_iterator rbegin() const;
```

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed tree.

Complexity: Constant.

Throws: Nothing.

11.
```
const_reverse_iterator crbegin() const;
```

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed tree.

Complexity: Constant.

Throws: Nothing.

12.
```
reverse_iterator rend();
```

Effects: Returns a reverse_iterator pointing to the end of the reversed tree.

Complexity: Constant.

Throws: Nothing.

13.
```
const_reverse_iterator rend() const;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed tree.

Complexity: Constant.

Throws: Nothing.

14.
```
const_reverse_iterator crend() const;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed tree.

Complexity: Constant.

Throws: Nothing.

15.
```
value_compare value_comp() const;
```

Effects: Returns the value_compare object used by the tree.

Complexity: Constant.

Throws: If value_compare copy-constructor throws.

16.
```
bool empty() const;
```

Effects: Returns true if the container is empty.

Complexity: Constant.

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

17.
```
size_type size() const;
```

Effects: Returns the number of elements stored in the tree.

Complexity: Linear to elements contained in *this if constant-time size option is disabled. Constant time otherwise.

Throws: Nothing.

18.
```
void swap(avltree & other);
```

Effects: Swaps the contents of two avltrees.

Complexity: Constant.

Throws: If the comparison functor's swap call throws.

19.
```
iterator insert_equal(reference value);
```

Requires: value must be an lvalue

Effects: Inserts value into the tree before the upper bound.

Complexity: Average complexity for insert element is at most logarithmic.

Throws: If the internal value_compare ordering function throws. Strong guarantee.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

20.
```
iterator insert_equal(const_iterator hint, reference value);
```

Requires: value must be an lvalue, and "hint" must be a valid iterator.

Effects: Inserts x into the tree, using "hint" as a hint to where it will be inserted. If "hint" is the upper_bound the insertion takes constant time (two comparisons in the worst case)

Complexity: Logarithmic in general, but it is amortized constant time if t is inserted immediately before hint.

Throws: If the internal value_compare ordering function throws. Strong guarantee.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

21.
```
template<typename Iterator> void insert_equal(Iterator b, Iterator e);
```

Requires: Dereferencing iterator must yield an lvalue of type value_type.

Effects: Inserts a each element of a range into the tree before the upper bound of the key of each element.

Complexity: Insert range is in general O(N * log(N)), where N is the size of the range. However, it is linear in N if the range is already sorted by value_comp().

Throws: Nothing.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

22.
```
std::pair< iterator, bool > insert_unique(reference value);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Requires: value must be an lvalue

Effects: Inserts value into the tree if the value is not already present.

Complexity: Average complexity for insert element is at most logarithmic.

Throws: Nothing.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

23.
```
iterator insert_unique(const_iterator hint, reference value);
```

Requires: value must be an lvalue, and "hint" must be a valid iterator

Effects: Tries to insert x into the tree, using "hint" as a hint to where it will be inserted.

Complexity: Logarithmic in general, but it is amortized constant time (two comparisons in the worst case) if t is inserted immediately before hint.

Throws: Nothing.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

24.
```
template<typename Iterator> void insert_unique(Iterator b, Iterator e);
```

Requires: Dereferencing iterator must yield an lvalue of type value_type.

Effects: Tries to insert each element of a range into the tree.

Complexity: Insert range is in general O(N * log(N)), where N is the size of the range. However, it is linear in N if the range is already sorted by value_comp().

Throws: Nothing.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

25.
```
template<typename KeyType, typename KeyValueCompare> 
  std::pair< iterator, bool > 
  insert_unique_check(const KeyType & key, KeyValueCompare key_value_comp, 
                      insert_commit_data & commit_data);
```

Requires: key_value_comp must be a comparison function that induces the same strict weak ordering as value_compare. The difference is that key_value_comp compares an arbitrary key with the contained values.

Effects: Checks if a value can be inserted in the container, using a user provided key instead of the value itself.

Returns: If there is an equivalent value returns a pair containing an iterator to the already present value and false. If the value can be inserted returns true in the returned pair boolean and fills "commit_data" that is meant to be used with the "insert_commit" function.

Complexity: Average complexity is at most logarithmic.

Throws: If the key_value_comp ordering function throws. Strong guarantee.

Notes: This function is used to improve performance when constructing a value_type is expensive: if there is an equivalent value the constructed object must be discarded. Many times, the part of the node that is used to impose the order is much cheaper to construct than the value_type and this function offers the possibility to use that part to check if the insertion will be successful.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

If the check is successful, the user can construct the value_type and use "insert_commit" to insert the object in constant-time. This gives a total logarithmic complexity to the insertion: check(O(log(N)) + commit(O(1)).

"commit_data" remains valid for a subsequent "insert_commit" only if no more objects are inserted or erased from the container.

26.
```
template<typename KeyType, typename KeyValueCompare>
  std::pair< iterator, bool >
  insert_unique_check(const_iterator hint, const KeyType & key,
                      KeyValueCompare key_value_comp,
                      insert_commit_data & commit_data);
```

Requires: key_value_comp must be a comparison function that induces the same strict weak ordering as value_compare. The difference is that key_value_comp compares an arbitrary key with the contained values.

Effects: Checks if a value can be inserted in the container, using a user provided key instead of the value itself, using "hint" as a hint to where it will be inserted.

Returns: If there is an equivalent value returns a pair containing an iterator to the already present value and false. If the value can be inserted returns true in the returned pair boolean and fills "commit_data" that is meant to be used with the "insert_commit" function.

Complexity: Logarithmic in general, but it's amortized constant time if t is inserted immediately before hint.

Throws: If the key_value_comp ordering function throws. Strong guarantee.

Notes: This function is used to improve performance when constructing a value_type is expensive: if there is an equivalent value the constructed object must be discarded. Many times, the part of the constructing that is used to impose the order is much cheaper to construct than the value_type and this function offers the possibility to use that key to check if the insertion will be successful.

If the check is successful, the user can construct the value_type and use "insert_commit" to insert the object in constant-time. This can give a total constant-time complexity to the insertion: check(O(1)) + commit(O(1)).

"commit_data" remains valid for a subsequent "insert_commit" only if no more objects are inserted or erased from the container.

27.
```
iterator insert_unique_commit(reference value,
                              const insert_commit_data & commit_data);
```

Requires: value must be an lvalue of type value_type. commit_data must have been obtained from a previous call to "insert_check". No objects should have been inserted or erased from the container between the "insert_check" that filled "commit_data" and the call to "insert_commit".

Effects: Inserts the value in the `avl_set` using the information obtained from the "commit_data" that a previous "insert_check" filled.

Returns: An iterator to the newly inserted object.

Complexity: Constant time.

Throws: Nothing.

Notes: This function has only sense if a "insert_check" has been previously executed to fill "commit_data". No value should be inserted or erased between the "insert_check" and "insert_commit" calls.

28.
```
iterator insert_before(const_iterator pos, reference value);
```

Requires: value must be an lvalue, "pos" must be a valid iterator (or end) and must be the succesor of value once inserted according to the predicate

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Inserts x into the tree before "pos".

Complexity: Constant time.

Throws: Nothing.

Note: This function does not check preconditions so if "pos" is not the successor of "value" tree ordering invariant will be broken. This is a low-level function to be used only for performance reasons by advanced users.

29.
```
void push_back(reference value);
```

Requires: value must be an lvalue, and it must be no less than the greatest inserted key

Effects: Inserts x into the tree in the last position.

Complexity: Constant time.

Throws: Nothing.

Note: This function does not check preconditions so if value is less than the greatest inserted key tree ordering invariant will be broken. This function is slightly more efficient than using "insert_before". This is a low-level function to be used only for performance reasons by advanced users.

30.
```
void push_front(reference value);
```

Requires: value must be an lvalue, and it must be no greater than the minimum inserted key

Effects: Inserts x into the tree in the first position.

Complexity: Constant time.

Throws: Nothing.

Note: This function does not check preconditions so if value is greater than the minimum inserted key tree ordering invariant will be broken. This function is slightly more efficient than using "insert_before". This is a low-level function to be used only for performance reasons by advanced users.

31.
```
iterator erase(const_iterator i);
```

Effects: Erases the element pointed to by pos.

Complexity: Average complexity for erase element is constant time.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

32.
```
iterator erase(const_iterator b, const_iterator e);
```

Effects: Erases the range pointed to by b end e.

Complexity: Average complexity for erase range is at most O(log(size() + N)), where N is the number of elements in the range.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

33.
```
size_type erase(const_reference value);
```

Effects: Erases all the elements with the given value.

Returns: The number of erased elements.

Complexity: O(log(size() + N).

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

34.
```
template<typename KeyType, typename KeyValueCompare>
  size_type erase(const KeyType & key, KeyValueCompare comp);
```

Effects: Erases all the elements with the given key. according to the comparison functor "comp".

Returns: The number of erased elements.

Complexity: O(log(size() + N).

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

35.
```
template<typename Disposer>
  iterator erase_and_dispose(const_iterator i, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases the element pointed to by pos. Disposer::operator()(pointer) is called for the removed element.

Complexity: Average complexity for erase element is constant time.

Throws: Nothing.

Note: Invalidates the iterators to the erased elements.

36.
```
template<typename Disposer>
  iterator erase_and_dispose(const_iterator b, const_iterator e,
                             Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases the range pointed to by b end e. Disposer::operator()(pointer) is called for the removed elements.

Complexity: Average complexity for erase range is at most O(log(size() + N)), where N is the number of elements in the range.

Throws: Nothing.

Note: Invalidates the iterators to the erased elements.

37.
```
template<typename Disposer>
  size_type erase_and_dispose(const_reference value, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases all the elements with the given value. Disposer::operator()(pointer) is called for the removed elements.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Returns: The number of erased elements.

Complexity: O(log(size() + N).

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

38.
```
template<typename KeyType, typename KeyValueCompare, typename Disposer>
  size_type erase_and_dispose(const KeyType & key, KeyValueCompare comp,
                              Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases all the elements with the given key. according to the comparison functor "comp". Disposer::operator()(pointer) is called for the removed elements.

Returns: The number of erased elements.

Complexity: O(log(size() + N).

Throws: Nothing.

Note: Invalidates the iterators to the erased elements.

39.
```
void clear();
```

Effects: Erases all of the elements.

Complexity: Linear to the number of elements on the container. if it's a safe-mode or auto-unlink value_type. Constant time otherwise.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

40.
```
template<typename Disposer> void clear_and_dispose(Disposer disposer);
```

Effects: Erases all of the elements calling disposer(p) for each node to be erased. **Complexity**: Average complexity for is at most O(log(size() + N)), where N is the number of elements in the container.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. Calls N times to disposer functor.

41.
```
size_type count(const_reference value) const;
```

Effects: Returns the number of contained elements with the given value

Complexity: Logarithmic to the number of elements contained plus lineal to number of objects with the given value.

Throws: Nothing.

42.
```
template<typename KeyType, typename KeyValueCompare>
  size_type count(const KeyType & key, KeyValueCompare comp) const;
```

Effects: Returns the number of contained elements with the given key

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Logarithmic to the number of elements contained plus lineal to number of objects with the given key.

Throws: Nothing.

43.
```
iterator lower_bound(const_reference value);
```

Effects: Returns an iterator to the first element whose key is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

44.
```
const_iterator lower_bound(const_reference value) const;
```

Effects: Returns an iterator to the first element whose key is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

45.
```
template<typename KeyType, typename KeyValueCompare> 
  iterator lower_bound(const KeyType & key, KeyValueCompare comp);
```

Effects: Returns an iterator to the first element whose key is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

46.
```
template<typename KeyType, typename KeyValueCompare> 
  const_iterator lower_bound(const KeyType & key, KeyValueCompare comp) const;
```

Effects: Returns a const iterator to the first element whose key is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

47.
```
iterator upper_bound(const_reference value);
```

Effects: Returns an iterator to the first element whose key is greater than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

48.
```
template<typename KeyType, typename KeyValueCompare> 
  iterator upper_bound(const KeyType & key, KeyValueCompare comp);
```

Effects: Returns an iterator to the first element whose key is greater than k according to comp or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

49.
```
const_iterator upper_bound(const_reference value) const;
```

Effects: Returns an iterator to the first element whose key is greater than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

50.
```
template<typename KeyType, typename KeyValueCompare>
  const_iterator upper_bound(const KeyType & key, KeyValueCompare comp) const;
```

Effects: Returns an iterator to the first element whose key is greater than k according to comp or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

51.
```
iterator find(const_reference value);
```

Effects: Finds an iterator to the first element whose key is k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

52.
```
template<typename KeyType, typename KeyValueCompare>
  iterator find(const KeyType & key, KeyValueCompare comp);
```

Effects: Finds an iterator to the first element whose key is k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

53.
```
const_iterator find(const_reference value) const;
```

Effects: Finds a const_iterator to the first element whose key is k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

54.
```
template<typename KeyType, typename KeyValueCompare>
  const_iterator find(const KeyType & key, KeyValueCompare comp) const;
```

Effects: Finds a const_iterator to the first element whose key is k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

55.
```
std::pair< iterator, iterator > equal_range(const_reference value);
```

Effects: Finds a range containing all elements whose key is k or an empty range that indicates the position where those elements would be if they there is no elements with key k.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Logarithmic.

Throws: Nothing.

56.
```
template<typename KeyType, typename KeyValueCompare>
  std::pair< iterator, iterator >
  equal_range(const KeyType & key, KeyValueCompare comp);
```

Effects: Finds a range containing all elements whose key is k or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: Nothing.

57.
```
std::pair< const_iterator, const_iterator >
equal_range(const_reference value) const;
```

Effects: Finds a range containing all elements whose key is k or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: Nothing.

58.
```
template<typename KeyType, typename KeyValueCompare>
  std::pair< const_iterator, const_iterator >
  equal_range(const KeyType & key, KeyValueCompare comp) const;
```

Effects: Finds a range containing all elements whose key is k or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: Nothing.

59.
```
std::pair< iterator, iterator >
bounded_range(const_reference lower_value, const_reference upper_value,
              bool left_closed, bool right_closed);
```

Requires: 'lower_value' must not be greater than 'upper_value'. If 'lower_value' == 'upper_value', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key) if left_closed, upper_bound(lower_key) otherwise

second = upper_bound(upper_key) if right_closed, lower_bound(upper_key) otherwise

Complexity: Logarithmic.

Throws: If the predicate throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_value and upper_value.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

60.
```
template<typename KeyType, typename KeyValueCompare>
  std::pair< iterator, iterator >
  bounded_range(const KeyType & lower_key, const KeyType & upper_key,
                KeyValueCompare comp, bool left_closed, bool right_closed);
```

Requires: KeyValueCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. 'lower_key' must not be greater than 'upper_key' according to 'comp'. If 'lower_key' == 'upper_key', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key, comp) if left_closed, upper_bound(lower_key, comp) otherwise

second = upper_bound(upper_key, comp) if right_closed, lower_bound(upper_key, comp) otherwise

Complexity: Logarithmic.

Throws: If "comp" throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_key and upper_key.

61.
```
std::pair< const_iterator, const_iterator >
bounded_range(const_reference lower_value, const_reference upper_value,
              bool left_closed, bool right_closed) const;
```

Requires: 'lower_value' must not be greater than 'upper_value'. If 'lower_value' == 'upper_value', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key) if left_closed, upper_bound(lower_key) otherwise

second = upper_bound(upper_key) if right_closed, lower_bound(upper_key) otherwise

Complexity: Logarithmic.

Throws: If the predicate throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_value and upper_value.

62.
```
template<typename KeyType, typename KeyValueCompare>
  std::pair< const_iterator, const_iterator >
  bounded_range(const KeyType & lower_key, const KeyType & upper_key,
                KeyValueCompare comp, bool left_closed, bool right_closed) const;
```

Requires: KeyValueCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. 'lower_key' must not be greater than 'upper_key' according to 'comp'. If 'lower_key' == 'upper_key', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key, comp) if left_closed, upper_bound(lower_key, comp) otherwise

second = upper_bound(upper_key, comp) if right_closed, lower_bound(upper_key, comp) otherwise

Complexity: Logarithmic.

Throws: If "comp" throws.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_key and upper_key.

63.
```
template<typename Cloner, typename Disposer>
  void clone_from(const avltree & src, Cloner cloner, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw. Cloner should yield to nodes equivalent to the original nodes.

Effects: Erases all the elements from *this calling Disposer::operator()(pointer), clones all the elements from src calling Cloner::operator()(const_reference) and inserts them on *this. Copies the predicate from the source container.

If cloner throws, all cloned elements are unlinked and disposed calling Disposer::operator()(pointer).

Complexity: Linear to erased plus inserted elements.

Throws: If cloner throws or predicate copy assignment throws. Basic guarantee.

64.
```
pointer unlink_leftmost_without_rebalance();
```

Effects: Unlinks the leftmost node from the tree.

Complexity: Average complexity is constant time.

Throws: Nothing.

Notes: This function breaks the tree and the tree can only be used for more unlink_leftmost_without_rebalance calls. This function is normally used to achieve a step by step controlled destruction of the tree.

65.
```
void replace_node(iterator replace_this, reference with_this);
```

Requires: replace_this must be a valid iterator of *this and with_this must not be inserted in any tree.

Effects: Replaces replace_this in its position in the tree with with_this. The tree does not need to be rebalanced.

Complexity: Constant.

Throws: Nothing.

Note: This function will break container ordering invariants if with_this is not equivalent to *replace_this according to the ordering rules. This function is faster than erasing and inserting the node, since no rebalancing or comparison is needed.

66.
```
iterator iterator_to(reference value);
```

Requires: value must be an lvalue and shall be in a set of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid iterator i belonging to the set that points to the value

Complexity: Constant.

Throws: Nothing.

67.
```
const_iterator iterator_to(const_reference value) const;
```

Requires: value must be an lvalue and shall be in a set of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid const_iterator i belonging to the set that points to the value

Complexity: Constant.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: Nothing.

`avltree` public static functions

1.
```
static avltree & container_from_end_iterator(iterator end_iterator);
```

   Precondition: end_iterator must be a valid end iterator of avltree.

   Effects: Returns a const reference to the avltree associated to the end iterator

   Throws: Nothing.

   Complexity: Constant.

2.
```
static const avltree &
container_from_end_iterator(const_iterator end_iterator);
```

   Precondition: end_iterator must be a valid end const_iterator of avltree.

   Effects: Returns a const reference to the avltree associated to the end iterator

   Throws: Nothing.

   Complexity: Constant.

3.
```
static avltree & container_from_iterator(iterator it);
```

   Precondition: it must be a valid iterator of rbtree.

   Effects: Returns a const reference to the tree associated to the iterator

   Throws: Nothing.

   Complexity: Logarithmic.

4.
```
static const avltree & container_from_iterator(const_iterator it);
```

   Precondition: it must be a valid end const_iterator of rbtree.

   Effects: Returns a const reference to the tree associated to the iterator

   Throws: Nothing.

   Complexity: Logarithmic.

5.
```
static iterator s_iterator_to(reference value);
```

   Requires: value must be an lvalue and shall be in a set of appropriate type. Otherwise the behavior is undefined.

   Effects: Returns: a valid iterator i belonging to the set that points to the value

   Complexity: Constant.

   Throws: Nothing.

   Note: This static function is available only if the *value traits* is stateless.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

6.
```
static const_iterator s_iterator_to(const_reference value);
```

   Requires: value must be an lvalue and shall be in a set of appropriate type. Otherwise the behavior is undefined.

   Effects: Returns: a valid const_iterator i belonging to the set that points to the value

   Complexity: Constant.

   Throws: Nothing.

   Note: This static function is available only if the *value traits* is stateless.

7.
```
static void init_node(reference value);
```

   Requires: value shall not be in a tree.

   Effects: init_node puts the hook of a value in a well-known default state.

   Throws: Nothing.

   Complexity: Constant time.

   Note: This function puts the hook in the well-known default state used by auto_unlink and safe hooks.

`avltree` private static functions

1.
```
static avltree &
priv_container_from_end_iterator(const const_iterator & end_iterator);
```

2.
```
static avltree & priv_container_from_iterator(const const_iterator & it);
```

Struct template make_avltree

boost::intrusive::make_avltree

Synopsis

```
// In header: <boost/intrusive/avltree.hpp>

template<typename T, class... Options>
struct make_avltree {
  // types
  typedef implementation_defined type;
};
```

Description

Helper metafunction to define a `avltree` that yields to the same type when the same options (either explicitly or implicitly) are used.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Header <boost/intrusive/avltree_algorithms.hpp>

```
namespace boost {
  namespace intrusive {
    template<typename NodeTraits> class avltree_algorithms;
  }
}
```

Class template avltree_algorithms

boost::intrusive::avltree_algorithms

Synopsis

```
// In header: <boost/intrusive/avltree_algorithms.hpp>

template<typename NodeTraits>
class avltree_algorithms {
public:
  // types
  typedef NodeTraits::node                        node;
  typedef NodeTraits                              node_traits;
  typedef NodeTraits::node_ptr                    node_ptr;
  typedef NodeTraits::const_node_ptr              const_node_ptr;
  typedef NodeTraits::balance                     balance;
  typedef tree_algorithms::insert_commit_data insert_commit_data;

  // public static functions
  static node_ptr begin_node(const const_node_ptr &);
  static node_ptr end_node(const const_node_ptr &);
  static void swap_tree(const node_ptr &, const node_ptr &);
  static void swap_nodes(const node_ptr &, const node_ptr &);
  static void swap_nodes(const node_ptr &, const node_ptr &, const node_ptr &,
                         const node_ptr &);
  static void replace_node(const node_ptr &, const node_ptr &);
  static void replace_node(const node_ptr &, const node_ptr &,
                           const node_ptr &);
  static void unlink(const node_ptr &);
  static node_ptr unlink_leftmost_without_rebalance(const node_ptr &);
  static bool unique(const const_node_ptr &);
  static std::size_t count(const const_node_ptr &);
  static std::size_t size(const const_node_ptr &);
  static node_ptr next_node(const node_ptr &);
  static node_ptr prev_node(const node_ptr &);
  static void init(const node_ptr &);
  static void init_header(const node_ptr &);
  static node_ptr erase(const node_ptr &, const node_ptr &);
  template<typename Cloner, typename Disposer>
    static void clone(const const_node_ptr &, const node_ptr &, Cloner,
                      Disposer);
  template<typename Disposer>
    static void clear_and_dispose(const node_ptr &, Disposer);
  template<typename KeyType, typename KeyNodePtrCompare>
    static node_ptr
    lower_bound(const const_node_ptr &, const KeyType &, KeyNodePtrCompare);
  template<typename KeyType, typename KeyNodePtrCompare>
    static node_ptr
    upper_bound(const const_node_ptr &, const KeyType &, KeyNodePtrCompare);
  template<typename KeyType, typename KeyNodePtrCompare>
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
    static node_ptr
    find(const const_node_ptr &, const KeyType &, KeyNodePtrCompare);
  template<typename KeyType, typename KeyNodePtrCompare>
    static std::pair< node_ptr, node_ptr >
    equal_range(const const_node_ptr &, const KeyType &, KeyNodePtrCompare);
  template<typename KeyType, typename KeyNodePtrCompare>
    static std::pair< node_ptr, node_ptr >
    bounded_range(const const_node_ptr &, const KeyType &, const KeyType &,
                  KeyNodePtrCompare, bool, bool);
  template<typename NodePtrCompare>
    static node_ptr
    insert_equal_upper_bound(const node_ptr &, const node_ptr &,
                             NodePtrCompare);
  template<typename NodePtrCompare>
    static node_ptr
    insert_equal_lower_bound(const node_ptr &, const node_ptr &,
                             NodePtrCompare);
  template<typename NodePtrCompare>
    static node_ptr
    insert_equal(const node_ptr &, const node_ptr &, const node_ptr &,
                 NodePtrCompare);
  static node_ptr
  insert_before(const node_ptr &, const node_ptr &, const node_ptr &);
  static void push_back(const node_ptr &, const node_ptr &);
  static void push_front(const node_ptr &, const node_ptr &);
  template<typename KeyType, typename KeyNodePtrCompare>
    static std::pair< node_ptr, bool >
    insert_unique_check(const const_node_ptr &, const KeyType &,
                        KeyNodePtrCompare, insert_commit_data &);
  template<typename KeyType, typename KeyNodePtrCompare>
    static std::pair< node_ptr, bool >
    insert_unique_check(const const_node_ptr &, const node_ptr &,
                        const KeyType &, KeyNodePtrCompare,
                        insert_commit_data &);
  static void insert_unique_commit(const node_ptr &, const node_ptr &,
                                   const insert_commit_data &);
  static node_ptr get_header(const node_ptr &);
};
```

Description

avltree_algorithms is configured with a NodeTraits class, which encapsulates the information about the node to be manipulated. NodeTraits must support the following interface:

Typedefs:

`node`: The type of the node that forms the circular list

`node_ptr`: A pointer to a node

`const_node_ptr`: A pointer to a const node

`balance`: The type of the balance factor

Static functions:

```
static node_ptr get_parent(const_node_ptr n);
```

```
static void set_parent(node_ptr n, node_ptr parent);
```

```
static node_ptr get_left(const_node_ptr n);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
static void set_left(node_ptr n, node_ptr left);

static node_ptr get_right(const_node_ptr n);

static void set_right(node_ptr n, node_ptr right);

static balance get_balance(const_node_ptr n);

static void set_balance(node_ptr n, balance b);

static balance negative();

static balance zero();

static balance positive();
```

`avltree_algorithms` public types

1. typedef tree_algorithms::insert_commit_data insert_commit_data;

   This type is the information that will be filled by insert_unique_check

`avltree_algorithms` public static functions

1.
```
static node_ptr begin_node(const const_node_ptr & header);
```

2.
```
static node_ptr end_node(const const_node_ptr & header);
```

3.
```
static void swap_tree(const node_ptr & header1, const node_ptr & header2);
```

   Requires: header1 and header2 must be the header nodes of two trees.

   Effects: Swaps two trees. After the function header1 will contain links to the second tree and header2 will have links to the first tree.

   Complexity: Constant.

   Throws: Nothing.

4.
```
static void swap_nodes(const node_ptr & node1, const node_ptr & node2);
```

   Requires: node1 and node2 can't be header nodes of two trees.

   Effects: Swaps two nodes. After the function node1 will be inserted in the position node2 before the function. node2 will be inserted in the position node1 had before the function.

   Complexity: Logarithmic.

   Throws: Nothing.

   Note: This function will break container ordering invariants if node1 and node2 are not equivalent according to the ordering rules.

   Experimental function

5.
```
static void swap_nodes(const node_ptr & node1, const node_ptr & header1, 
                       const node_ptr & node2, const node_ptr & header2);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Requires: node1 and node2 can't be header nodes of two trees with header header1 and header2.

Effects: Swaps two nodes. After the function node1 will be inserted in the position node2 before the function. node2 will be inserted in the position node1 had before the function.

Complexity: Constant.

Throws: Nothing.

Note: This function will break container ordering invariants if node1 and node2 are not equivalent according to the ordering rules.

Experimental function

6.
```
static void replace_node(const node_ptr & node_to_be_replaced,
                         const node_ptr & new_node);
```

Requires: node_to_be_replaced must be inserted in a tree and new_node must not be inserted in a tree.

Effects: Replaces node_to_be_replaced in its position in the tree with new_node. The tree does not need to be rebalanced

Complexity: Logarithmic.

Throws: Nothing.

Note: This function will break container ordering invariants if new_node is not equivalent to node_to_be_replaced according to the ordering rules. This function is faster than erasing and inserting the node, since no rebalancing and comparison is needed.

Experimental function

7.
```
static void replace_node(const node_ptr & node_to_be_replaced,
                         const node_ptr & header, const node_ptr & new_node);
```

Requires: node_to_be_replaced must be inserted in a tree with header "header" and new_node must not be inserted in a tree.

Effects: Replaces node_to_be_replaced in its position in the tree with new_node. The tree does not need to be rebalanced

Complexity: Constant.

Throws: Nothing.

Note: This function will break container ordering invariants if new_node is not equivalent to node_to_be_replaced according to the ordering rules. This function is faster than erasing and inserting the node, since no rebalancing or comparison is needed.

Experimental function

8.
```
static void unlink(const node_ptr & node);
```

Requires: node is a tree node but not the header.

Effects: Unlinks the node and rebalances the tree.

Complexity: Average complexity is constant time.

Throws: Nothing.

9.
```
static node_ptr unlink_leftmost_without_rebalance(const node_ptr & header);
```

Requires: header is the header of a tree.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Unlinks the leftmost node from the tree, and updates the header link to the new leftmost node.

Complexity: Average complexity is constant time.

Throws: Nothing.

Notes: This function breaks the tree and the tree can only be used for more unlink_leftmost_without_rebalance calls. This function is normally used to achieve a step by step controlled destruction of the tree.

10.
```
static bool unique(const const_node_ptr & node);
```

Requires: node is a node of the tree or an node initialized by init(...).

Effects: Returns true if the node is initialized by init().

Complexity: Constant time.

Throws: Nothing.

11.
```
static std::size_t count(const const_node_ptr & node);
```

Requires: node is a node of the tree but it's not the header.

Effects: Returns the number of nodes of the subtree.

Complexity: Linear time.

Throws: Nothing.

12.
```
static std::size_t size(const const_node_ptr & header);
```

Requires: header is the header node of the tree.

Effects: Returns the number of nodes above the header.

Complexity: Linear time.

Throws: Nothing.

13.
```
static node_ptr next_node(const node_ptr & p);
```

Requires: p is a node from the tree except the header.

Effects: Returns the next node of the tree.

Complexity: Average constant time.

Throws: Nothing.

14.
```
static node_ptr prev_node(const node_ptr & p);
```

Requires: p is a node from the tree except the leftmost node.

Effects: Returns the previous node of the tree.

Complexity: Average constant time.

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

15.
```
static void init(const node_ptr & node);
```

Requires: node must not be part of any tree.

Effects: After the function unique(node) == true.

Complexity: Constant.

Throws: Nothing.

Nodes: If node is inserted in a tree, this function corrupts the tree.

16.
```
static void init_header(const node_ptr & header);
```

Requires: node must not be part of any tree.

Effects: Initializes the header to represent an empty tree. unique(header) == true.

Complexity: Constant.

Throws: Nothing.

Nodes: If node is inserted in a tree, this function corrupts the tree.

17.
```
static node_ptr erase(const node_ptr & header, const node_ptr & z);
```

Requires: header must be the header of a tree, z a node of that tree and z != header.

Effects: Erases node "z" from the tree with header "header".

Complexity: Amortized constant time.

Throws: Nothing.

18.
```
template<typename Cloner, typename Disposer>
  static void clone(const const_node_ptr & source_header,
                    const node_ptr & target_header, Cloner cloner,
                    Disposer disposer);
```

Requires: "cloner" must be a function object taking a node_ptr and returning a new cloned node of it. "disposer" must take a node_ptr and shouldn't throw.

Effects: First empties target tree calling `void disposer::operator()(const node_ptr &)` for every node of the tree except the header.

Then, duplicates the entire tree pointed by "source_header" cloning each source node with `node_ptr Cloner::operator()(const node_ptr &)` to obtain the nodes of the target tree. If "cloner" throws, the cloned target nodes are disposed using `void disposer(const node_ptr &)`.

Complexity: Linear to the number of element of the source tree plus the. number of elements of tree target tree when calling this function.

Throws: If cloner functor throws. If this happens target nodes are disposed.

19.
```
template<typename Disposer>
  static void clear_and_dispose(const node_ptr & header, Disposer disposer);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Requires: "disposer" must be an object function taking a node_ptr parameter and shouldn't throw.

Effects: Empties the target tree calling `void disposer::operator()(const node_ptr &)` for every node of the tree except the header.

Complexity: Linear to the number of element of the source tree plus the. number of elements of tree target tree when calling this function.

Throws: If cloner functor throws. If this happens target nodes are disposed.

20.
```
template<typename KeyType, typename KeyNodePtrCompare> 
  static node_ptr 
  lower_bound(const const_node_ptr & header, const KeyType & key, 
              KeyNodePtrCompare comp);
```

Requires: "header" must be the header node of a tree. KeyNodePtrCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. KeyNodePtrCompare can compare KeyType with tree's node_ptrs.

Effects: Returns an node_ptr to the first element that is not less than "key" according to "comp" or "header" if that element does not exist.

Complexity: Logarithmic.

Throws: If "comp" throws.

21.
```
template<typename KeyType, typename KeyNodePtrCompare> 
  static node_ptr 
  upper_bound(const const_node_ptr & header, const KeyType & key, 
              KeyNodePtrCompare comp);
```

Requires: "header" must be the header node of a tree. KeyNodePtrCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. KeyNodePtrCompare can compare KeyType with tree's node_ptrs.

Effects: Returns an node_ptr to the first element that is greater than "key" according to "comp" or "header" if that element does not exist.

Complexity: Logarithmic.

Throws: If "comp" throws.

22.
```
template<typename KeyType, typename KeyNodePtrCompare> 
  static node_ptr 
  find(const const_node_ptr & header, const KeyType & key, 
       KeyNodePtrCompare comp);
```

Requires: "header" must be the header node of a tree. KeyNodePtrCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. KeyNodePtrCompare can compare KeyType with tree's node_ptrs.

Effects: Returns an node_ptr to the element that is equivalent to "key" according to "comp" or "header" if that element does not exist.

Complexity: Logarithmic.

Throws: If "comp" throws.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

23.
```
template<typename KeyType, typename KeyNodePtrCompare> 
  static std::pair< node_ptr, node_ptr > 
  equal_range(const const_node_ptr & header, const KeyType & key, 
              KeyNodePtrCompare comp);
```

Requires: "header" must be the header node of a tree. KeyNodePtrCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. KeyNodePtrCompare can compare KeyType with tree's node_ptrs.

Effects: Returns an a pair of node_ptr delimiting a range containing all elements that are equivalent to "key" according to "comp" or an empty range that indicates the position where those elements would be if they there are no equivalent elements.

Complexity: Logarithmic.

Throws: If "comp" throws.

24.
```
template<typename KeyType, typename KeyNodePtrCompare> 
  static std::pair< node_ptr, node_ptr > 
  bounded_range(const const_node_ptr & header, const KeyType & lower_key, 
                const KeyType & upper_key, KeyNodePtrCompare comp, 
                bool left_closed, bool right_closed);
```

Requires: "header" must be the header node of a tree. KeyNodePtrCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. KeyNodePtrCompare can compare KeyType with tree's node_ptrs. 'lower_key' must not be greater than 'upper_key' according to 'comp'. If 'lower_key' == 'upper_key', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key) if left_closed, upper_bound(lower_key) otherwise

second = upper_bound(upper_key) if right_closed, lower_bound(upper_key) otherwise

Complexity: Logarithmic.

Throws: If "comp" throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_key and upper_key.

25.
```
template<typename NodePtrCompare> 
  static node_ptr 
  insert_equal_upper_bound(const node_ptr & h, const node_ptr & new_node, 
                           NodePtrCompare comp);
```

Requires: "h" must be the header node of a tree. NodePtrCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. NodePtrCompare compares two node_ptrs.

Effects: Inserts new_node into the tree before the upper bound according to "comp".

Complexity: Average complexity for insert element is at most logarithmic.

Throws: If "comp" throws.

26.
```
template<typename NodePtrCompare> 
  static node_ptr 
  insert_equal_lower_bound(const node_ptr & h, const node_ptr & new_node, 
                           NodePtrCompare comp);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Requires: "h" must be the header node of a tree. NodePtrCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. NodePtrCompare compares two node_ptrs.

Effects: Inserts new_node into the tree before the lower bound according to "comp".

Complexity: Average complexity for insert element is at most logarithmic.

Throws: If "comp" throws.

27.
```
template<typename NodePtrCompare>
  static node_ptr
  insert_equal(const node_ptr & header, const node_ptr & hint,
               const node_ptr & new_node, NodePtrCompare comp);
```

Requires: "header" must be the header node of a tree. NodePtrCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. NodePtrCompare compares two node_ptrs. "hint" is node from the "header"'s tree.

Effects: Inserts new_node into the tree, using "hint" as a hint to where it will be inserted. If "hint" is the upper_bound the insertion takes constant time (two comparisons in the worst case).

Complexity: Logarithmic in general, but it is amortized constant time if new_node is inserted immediately before "hint".

Throws: If "comp" throws.

28.
```
static node_ptr
insert_before(const node_ptr & header, const node_ptr & pos,
              const node_ptr & new_node);
```

Requires: "header" must be the header node of a tree. "pos" must be a valid iterator or header (end) node. "pos" must be an iterator pointing to the successor to "new_node" once inserted according to the order of already inserted nodes. This function does not check "pos" and this precondition must be guaranteed by the caller.

Effects: Inserts new_node into the tree before "pos".

Complexity: Constant-time.

Throws: Nothing.

Note: If "pos" is not the successor of the newly inserted "new_node" tree invariants might be broken.

29.
```
static void push_back(const node_ptr & header, const node_ptr & new_node);
```

Requires: "header" must be the header node of a tree. "new_node" must be, according to the used ordering no less than the greatest inserted key.

Effects: Inserts new_node into the tree before "pos".

Complexity: Constant-time.

Throws: Nothing.

Note: If "new_node" is less than the greatest inserted key tree invariants are broken. This function is slightly faster than using "insert_before".

30.
```
static void push_front(const node_ptr & header, const node_ptr & new_node);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Requires: "header" must be the header node of a tree. "new_node" must be, according to the used ordering, no greater than the lowest inserted key.

Effects: Inserts new_node into the tree before "pos".

Complexity: Constant-time.

Throws: Nothing.

Note: If "new_node" is greater than the lowest inserted key tree invariants are broken. This function is slightly faster than using "insert_before".

31.
```
template<typename KeyType, typename KeyNodePtrCompare> 
  static std::pair< node_ptr, bool > 
  insert_unique_check(const_node_ptr & header, const KeyType & key, 
                      KeyNodePtrCompare comp, 
                      insert_commit_data & commit_data);
```

Requires: "header" must be the header node of a tree. KeyNodePtrCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. NodePtrCompare compares KeyType with a node_ptr.

Effects: Checks if there is an equivalent node to "key" in the tree according to "comp" and obtains the needed information to realize a constant-time node insertion if there is no equivalent node.

Returns: If there is an equivalent value returns a pair containing a node_ptr to the already present node and false. If there is not equivalent key can be inserted returns true in the returned pair's boolean and fills "commit_data" that is meant to be used with the "insert_commit" function to achieve a constant-time insertion function.

Complexity: Average complexity is at most logarithmic.

Throws: If "comp" throws.

Notes: This function is used to improve performance when constructing a node is expensive and the user does not want to have two equivalent nodes in the tree: if there is an equivalent value the constructed object must be discarded. Many times, the part of the node that is used to impose the order is much cheaper to construct than the node and this function offers the possibility to use that part to check if the insertion will be successful.

If the check is successful, the user can construct the node and use "insert_commit" to insert the node in constant-time. This gives a total logarithmic complexity to the insertion: check(O(log(N)) + commit(O(1)).

"commit_data" remains valid for a subsequent "insert_unique_commit" only if no more objects are inserted or erased from the set.

32.
```
template<typename KeyType, typename KeyNodePtrCompare> 
  static std::pair< node_ptr, bool > 
  insert_unique_check(const_node_ptr & header, const node_ptr & hint, 
                      const KeyType & key, KeyNodePtrCompare comp, 
                      insert_commit_data & commit_data);
```

Requires: "header" must be the header node of a tree. KeyNodePtrCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. NodePtrCompare compares KeyType with a node_ptr. "hint" is node from the "header"'s tree.

Effects: Checks if there is an equivalent node to "key" in the tree according to "comp" using "hint" as a hint to where it should be inserted and obtains the needed information to realize a constant-time node insertion if there is no equivalent node. If "hint" is the upper_bound the function has constant time complexity (two comparisons in the worst case).

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Returns: If there is an equivalent value returns a pair containing a node_ptr to the already present node and false. If there is not equivalent key can be inserted returns true in the returned pair's boolean and fills "commit_data" that is meant to be used with the "insert_commit" function to achieve a constant-time insertion function.

Complexity: Average complexity is at most logarithmic, but it is amortized constant time if new_node should be inserted immediately before "hint".

Throws: If "comp" throws.

Notes: This function is used to improve performance when constructing a node is expensive and the user does not want to have two equivalent nodes in the tree: if there is an equivalent value the constructed object must be discarded. Many times, the part of the node that is used to impose the order is much cheaper to construct than the node and this function offers the possibility to use that part to check if the insertion will be successful.

If the check is successful, the user can construct the node and use "insert_commit" to insert the node in constant-time. This gives a total logarithmic complexity to the insertion: check(O(log(N)) + commit(O(1)).

"commit_data" remains valid for a subsequent "insert_unique_commit" only if no more objects are inserted or erased from the set.

33.
```
static void insert_unique_commit(const node_ptr & header,
                                 const node_ptr & new_value,
                                 const insert_commit_data & commit_data);
```

Requires: "header" must be the header node of a tree. "commit_data" must have been obtained from a previous call to "insert_unique_check". No objects should have been inserted or erased from the set between the "insert_unique_check" that filled "commit_data" and the call to "insert_commit".

Effects: Inserts new_node in the set using the information obtained from the "commit_data" that a previous "insert_check" filled.

Complexity: Constant time.

Throws: Nothing.

Notes: This function has only sense if a "insert_unique_check" has been previously executed to fill "commit_data". No value should be inserted or erased between the "insert_check" and "insert_commit" calls.

34.
```
static node_ptr get_header(const node_ptr & n);
```

Requires: "n" must be a node inserted in a tree.

Effects: Returns a pointer to the header node of the tree.

Complexity: Logarithmic.

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Header <boost/intrusive/bs_set_hook.hpp>

```
namespace boost {
  namespace intrusive {
    template<class... Options> struct make_bs_set_base_hook;

    template<class... Options> class bs_set_base_hook;

    template<class... Options> struct make_bs_set_member_hook;

    template<class... Options> class bs_set_member_hook;
  }
}
```

Struct template make_bs_set_base_hook

boost::intrusive::make_bs_set_base_hook

Synopsis

```
// In header: <boost/intrusive/bs_set_hook.hpp>

template<class... Options>
struct make_bs_set_base_hook {
  // types
  typedef implementation_defined type;
};
```

Description

Helper metafunction to define a `bs_set_base_hook` that yields to the same type when the same options (either explicitly or implicitly) are used.

Class template bs_set_base_hook

boost::intrusive::bs_set_base_hook

Synopsis

```
// In header: <boost/intrusive/bs_set_hook.hpp>

template<class... Options>
class bs_set_base_hook : public make_bs_set_base_hook::type< O1, O2, O3 > {
public:
  // construct/copy/destruct
  bs_set_base_hook();
  bs_set_base_hook(const bs_set_base_hook &);
  bs_set_base_hook& operator=(const bs_set_base_hook &);
  ~bs_set_base_hook();

  // public member functions
  void swap_nodes(bs_set_base_hook &);
  bool is_linked() const;
  void unlink();
};
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Description

Derive a class from bs_set_base_hook in order to store objects in in a bs_set/bs_multiset. bs_set_base_hook holds the data necessary to maintain the bs_set/bs_multiset and provides an appropriate value_traits class for bs_set/bs_multiset.

The hook admits the following options: `tag<>`, `void_pointer<>`, `link_mode<>`.

`tag<>` defines a tag to identify the node. The same tag value can be used in different classes, but if a class is derived from more than one `list_base_hook`, then each `list_base_hook` needs its unique tag.

`void_pointer<>` is the pointer type that will be used internally in the hook and the the container configured to use this hook.

`link_mode<>` will specify the linking mode of the hook (`normal_link`, `auto_unlink` or `safe_link`).

`bs_set_base_hook` public construct/copy/destruct

1.
```
bs_set_base_hook();
```

   Effects: If `link_mode` is `auto_unlink` or `safe_link` initializes the node to an unlinked state.

   Throws: Nothing.

2.
```
bs_set_base_hook(const bs_set_base_hook &);
```

   Effects: If `link_mode` is `auto_unlink` or `safe_link` initializes the node to an unlinked state. The argument is ignored.

   Throws: Nothing.

   Rationale: Providing a copy-constructor makes classes using the hook STL-compliant without forcing the user to do some additional work. `swap` can be used to emulate move-semantics.

3.
```
bs_set_base_hook& operator=(const bs_set_base_hook &);
```

   Effects: Empty function. The argument is ignored.

   Throws: Nothing.

   Rationale: Providing an assignment operator makes classes using the hook STL-compliant without forcing the user to do some additional work. `swap` can be used to emulate move-semantics.

4.
```
~bs_set_base_hook();
```

   Effects: If `link_mode` is `normal_link`, the destructor does nothing (ie. no code is generated). If `link_mode` is `safe_link` and the object is stored in a set an assertion is raised. If `link_mode` is `auto_unlink` and `is_linked()` is true, the node is unlinked.

   Throws: Nothing.

`bs_set_base_hook` public member functions

1.
```
void swap_nodes(bs_set_base_hook & other);
```

   Effects: Swapping two nodes swaps the position of the elements related to those nodes in one or two containers. That is, if the node this is part of the element e1, the node x is part of the element e2 and both elements are included in the containers s1 and s2, then after the swap-operation e1 is in s2 at the position of e2 and e2 is in s1 at the position of e1. If one element is not in a container, then after the swap-operation the other element is not in a container. Iterators to e1 and e2 related to those nodes are invalidated.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Constant

Throws: Nothing.

2.

```
bool is_linked() const;
```

Precondition: `link_mode` must be `safe_link` or `auto_unlink`.

Returns: true, if the node belongs to a container, false otherwise. This function can be used to test whether `set::iterator_to` will return a valid iterator.

Complexity: Constant

3.

```
void unlink();
```

Effects: Removes the node if it's inserted in a container. This function is only allowed if `link_mode` is `auto_unlink`.

Throws: Nothing.

Struct template make_bs_set_member_hook

boost::intrusive::make_bs_set_member_hook

Synopsis

```
// In header: <boost/intrusive/bs_set_hook.hpp>

template<class... Options>
struct make_bs_set_member_hook {
  // types
  typedef implementation_defined type;
};
```

Description

Helper metafunction to define a `bs_set_member_hook` that yields to the same type when the same options (either explicitly or implicitly) are used.

Class template bs_set_member_hook

boost::intrusive::bs_set_member_hook

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/intrusive/bs_set_hook.hpp>

template<class... Options>
class bs_set_member_hook : public make_bs_set_member_hook::type< O1, O2, O3 > {
public:
  // construct/copy/destruct
  bs_set_member_hook();
  bs_set_member_hook(const bs_set_member_hook &);
  bs_set_member_hook& operator=(const bs_set_member_hook &);
  ~bs_set_member_hook();

  // public member functions
  void swap_nodes(bs_set_member_hook &);
  bool is_linked() const;
  void unlink();
};
```

Description

Put a public data member bs_set_member_hook in order to store objects of this class in a bs_set/bs_multiset. bs_set_member_hook holds the data necessary for maintaining the bs_set/bs_multiset and provides an appropriate value_traits class for bs_set/bs_multiset.

The hook admits the following options: `void_pointer<>`, `link_mode<>`.

`void_pointer<>` is the pointer type that will be used internally in the hook and the the container configured to use this hook.

`link_mode<>` will specify the linking mode of the hook (`normal_link`, `auto_unlink` or `safe_link`).

`bs_set_member_hook` public construct/copy/destruct

1.
   ```
   bs_set_member_hook();
   ```

   Effects: If `link_mode` is `auto_unlink` or `safe_link` initializes the node to an unlinked state.

   Throws: Nothing.

2.
   ```
   bs_set_member_hook(const bs_set_member_hook &);
   ```

   Effects: If `link_mode` is `auto_unlink` or `safe_link` initializes the node to an unlinked state. The argument is ignored.

   Throws: Nothing.

   Rationale: Providing a copy-constructor makes classes using the hook STL-compliant without forcing the user to do some additional work. `swap` can be used to emulate move-semantics.

3.
   ```
   bs_set_member_hook& operator=(const bs_set_member_hook &);
   ```

   Effects: Empty function. The argument is ignored.

   Throws: Nothing.

   Rationale: Providing an assignment operator makes classes using the hook STL-compliant without forcing the user to do some additional work. `swap` can be used to emulate move-semantics.

4.
   ```
   ~bs_set_member_hook();
   ```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: If `link_mode` is `normal_link`, the destructor does nothing (ie. no code is generated). If `link_mode` is `safe_link` and the object is stored in a set an assertion is raised. If `link_mode` is `auto_unlink` and `is_linked()` is true, the node is unlinked.

Throws: Nothing.

`bs_set_member_hook` public member functions

1.
```
void swap_nodes(bs_set_member_hook & other);
```

   Effects: Swapping two nodes swaps the position of the elements related to those nodes in one or two containers. That is, if the node this is part of the element e1, the node x is part of the element e2 and both elements are included in the containers s1 and s2, then after the swap-operation e1 is in s2 at the position of e2 and e2 is in s1 at the position of e1. If one element is not in a container, then after the swap-operation the other element is not in a container. Iterators to e1 and e2 related to those nodes are invalidated.

   Complexity: Constant

   Throws: Nothing.

2.
```
bool is_linked() const;
```

   Precondition: `link_mode` must be `safe_link` or `auto_unlink`.

   Returns: true, if the node belongs to a container, false otherwise. This function can be used to test whether `set::iterator_to` will return a valid iterator.

   Complexity: Constant

3.
```
void unlink();
```

   Effects: Removes the node if it's inserted in a container. This function is only allowed if `link_mode` is `auto_unlink`.

   Throws: Nothing.

Header <boost/intrusive/circular_list_algorithms.hpp>

```
namespace boost {
  namespace intrusive {
    template<typename NodeTraits> class circular_list_algorithms;
  }
}
```

Class template circular_list_algorithms

boost::intrusive::circular_list_algorithms

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/intrusive/circular_list_algorithms.hpp>

template<typename NodeTraits> 
class circular_list_algorithms {
public:
  // types
  typedef NodeTraits::node           node;          
  typedef NodeTraits::node_ptr       node_ptr;      
  typedef NodeTraits::const_node_ptr const_node_ptr;
  typedef NodeTraits                 node_traits;   

  // public static functions
  static void init(const node_ptr &);
  static bool inited(const const_node_ptr &);
  static void init_header(const node_ptr &);
  static bool unique(const const_node_ptr &);
  static std::size_t count(const const_node_ptr &);
  static node_ptr unlink(const node_ptr &);
  static void unlink(const node_ptr &, const node_ptr &);
  static void link_before(const node_ptr &, const node_ptr &);
  static void link_after(const node_ptr &, const node_ptr &);
  static void swap_nodes(const node_ptr &, const node_ptr &);
  static void transfer(const node_ptr &, const node_ptr &, const node_ptr &);
  static void transfer(const node_ptr &, const node_ptr &);
  static void reverse(const node_ptr &);
  static void move_backwards(const node_ptr &, std::size_t);
  static void move_forward(const node_ptr &, std::size_t);

  // private static functions
  static void swap_prev(const node_ptr &, const node_ptr &);
  static void swap_next(const node_ptr &, const node_ptr &);
};
```

Description

circular_list_algorithms provides basic algorithms to manipulate nodes forming a circular doubly linked list. An empty circular list is formed by a node whose pointers point to itself.

circular_list_algorithms is configured with a NodeTraits class, which encapsulates the information about the node to be manipulated. NodeTraits must support the following interface:

Typedefs:

`node`: The type of the node that forms the circular list

`node_ptr`: A pointer to a node

`const_node_ptr`: A pointer to a const node

Static functions:

```
static node_ptr get_previous(const_node_ptr n);

static void set_previous(node_ptr n, node_ptr prev);

static node_ptr get_next(const_node_ptr n);

static void set_next(node_ptr n, node_ptr next);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

`circular_list_algorithms` public static functions

1.
   ```
   static void init(const node_ptr & this_node);
   ```

   Effects: Constructs an non-used list element, so that inited(this_node) == true

   Complexity: Constant

   Throws: Nothing.

2.
   ```
   static bool inited(const const_node_ptr & this_node);
   ```

   Effects: Returns true is "this_node" is in a non-used state as if it was initialized by the "init" function.

   Complexity: Constant

   Throws: Nothing.

3.
   ```
   static void init_header(const node_ptr & this_node);
   ```

   Effects: Constructs an empty list, making this_node the only node of the circular list: `NodeTraits::get_next(this_node) == NodeTraits::get_previous(this_node) == this_node`.

   Complexity: Constant

   Throws: Nothing.

4.
   ```
   static bool unique(const const_node_ptr & this_node);
   ```

   Requires: this_node must be in a circular list or be an empty circular list.

   Effects: Returns true is "this_node" is the only node of a circular list: `return NodeTraits::get_next(this_node) == this_node`

   Complexity: Constant

   Throws: Nothing.

5.
   ```
   static std::size_t count(const const_node_ptr & this_node);
   ```

   Requires: this_node must be in a circular list or be an empty circular list.

   Effects: Returns the number of nodes in a circular list. If the circular list is empty, returns 1.

   Complexity: Linear

   Throws: Nothing.

6.
   ```
   static node_ptr unlink(const node_ptr & this_node);
   ```

   Requires: this_node must be in a circular list or be an empty circular list.

   Effects: Unlinks the node from the circular list.

   Complexity: Constant

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: Nothing.

7.
```
static void unlink(const node_ptr & b, const node_ptr & e);
```

Requires: b and e must be nodes of the same circular list or an empty range.

Effects: Unlinks the node [b, e) from the circular list.

Complexity: Constant

Throws: Nothing.

8.
```
static void link_before(const node_ptr & nxt_node, const node_ptr & this_node);
```

Requires: nxt_node must be a node of a circular list.

Effects: Links this_node before nxt_node in the circular list.

Complexity: Constant

Throws: Nothing.

9.
```
static void link_after(const node_ptr & prev_node, const node_ptr & this_node);
```

Requires: prev_node must be a node of a circular list.

Effects: Links this_node after prev_node in the circular list.

Complexity: Constant

Throws: Nothing.

10.
```
static void swap_nodes(const node_ptr & this_node,
                       const node_ptr & other_node);
```

11.
```
static void transfer(const node_ptr & p, const node_ptr & b,
                     const node_ptr & e);
```

Requires: b and e must be nodes of the same circular list or an empty range. and p must be a node of a different circular list or may not be an iterator in **Effects**: Removes the nodes from [b, e) range from their circular list and inserts them before p in p's circular list.

Complexity: Constant

Throws: Nothing.

12.
```
static void transfer(const node_ptr & p, const node_ptr & i);
```

Requires: i must a node of a circular list and p must be a node of a different circular list.

Effects: Removes the node i from its circular list and inserts it before p in p's circular list. If p == i or p == NodeTraits::get_next(i), this function is a null operation.

Complexity: Constant

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

13.
```
static void reverse(const node_ptr & p);
```

Effects: Reverses the order of elements in the list.

Throws: Nothing.

Complexity: This function is linear time.

14.
```
static void move_backwards(const node_ptr & p, std::size_t n);
```

Effects: Moves the node p n positions towards the end of the list.

Throws: Nothing.

Complexity: Linear to the number of moved positions.

15.
```
static void move_forward(const node_ptr & p, std::size_t n);
```

Effects: Moves the node p n positions towards the beginning of the list.

Throws: Nothing.

Complexity: Linear to the number of moved positions.

`circular_list_algorithms` private static functions

1.
```
static void swap_prev(const node_ptr & this_node, const node_ptr & other_node);
```

Requires: this_node and other_node must be nodes inserted in circular lists or be empty circular lists.

Effects: Swaps the position of the nodes: this_node is inserted in other_nodes position in the second circular list and the other_node is inserted in this_node's position in the first circular list.

Complexity: Constant

Throws: Nothing.

2.
```
static void swap_next(const node_ptr & this_node, const node_ptr & other_node);
```

Header <boost/intrusive/circular_slist_algorithms.hpp>

```
namespace boost {
  namespace intrusive {
    template<typename NodeTraits> class circular_slist_algorithms;
  }
}
```

Class template circular_slist_algorithms

boost::intrusive::circular_slist_algorithms

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/intrusive/circular_slist_algorithms.hpp>

template<typename NodeTraits>
class circular_slist_algorithms {
public:
  // types
  typedef NodeTraits::node           node;
  typedef NodeTraits::node_ptr       node_ptr;
  typedef NodeTraits::const_node_ptr const_node_ptr;
  typedef NodeTraits                 node_traits;

  // public static functions
  static void init(node_ptr);
  static bool unique(const_node_ptr);
  static bool inited(const_node_ptr);
  static void unlink_after(node_ptr);
  static void unlink_after(node_ptr, node_ptr);
  static void link_after(node_ptr, node_ptr);
  static void transfer_after(node_ptr, node_ptr, node_ptr);
  static void init_header(const node_ptr &);
  static node_ptr get_previous_node(const node_ptr &, const node_ptr &);
  static node_ptr get_previous_node(const node_ptr &);
  static node_ptr get_previous_previous_node(const node_ptr &);
  static node_ptr get_previous_previous_node(node_ptr, const node_ptr &);
  static std::size_t count(const const_node_ptr &);
  static void unlink(const node_ptr &);
  static void link_before(const node_ptr &, const node_ptr &);
  static void swap_nodes(const node_ptr &, const node_ptr &);
  static void reverse(const node_ptr &);
  static node_ptr move_backwards(const node_ptr &, std::size_t);
  static node_ptr move_forward(const node_ptr &, std::size_t);
};
```

Description

circular_slist_algorithms provides basic algorithms to manipulate nodes forming a circular singly linked list. An empty circular list is formed by a node whose pointer to the next node points to itself.

circular_slist_algorithms is configured with a NodeTraits class, which encapsulates the information about the node to be manipulated. NodeTraits must support the following interface:

Typedefs:

`node`: The type of the node that forms the circular list

`node_ptr`: A pointer to a node

`const_node_ptr`: A pointer to a const node

Static functions:

`static node_ptr get_next(const_node_ptr n);`

`static void set_next(node_ptr n, node_ptr next);`

`circular_slist_algorithms` public static functions

1.
```
static void init(node_ptr this_node);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Constructs an non-used list element, putting the next pointer to null: `NodeTraits::get_next(this_node) == node_ptr()`

Complexity: Constant

Throws: Nothing.

2.
```
static bool unique(const_node_ptr this_node);
```

Requires: this_node must be in a circular list or be an empty circular list.

Effects: Returns true is "this_node" is the only node of a circular list: or it's a not inserted node: `return node_ptr() == NodeTraits::get_next(this_node) || NodeTraits::get_next(this_node) == this_node`

Complexity: Constant

Throws: Nothing.

3.
```
static bool inited(const_node_ptr this_node);
```

Effects: Returns true is "this_node" has the same state as if it was inited using "init(node_ptr)"

Complexity: Constant

Throws: Nothing.

4.
```
static void unlink_after(node_ptr prev_node);
```

Requires: prev_node must be in a circular list or be an empty circular list.

Effects: Unlinks the next node of prev_node from the circular list.

Complexity: Constant

Throws: Nothing.

5.
```
static void unlink_after(node_ptr prev_node, node_ptr last_node);
```

Requires: prev_node and last_node must be in a circular list or be an empty circular list.

Effects: Unlinks the range (prev_node, last_node) from the circular list.

Complexity: Constant

Throws: Nothing.

6.
```
static void link_after(node_ptr prev_node, node_ptr this_node);
```

Requires: prev_node must be a node of a circular list.

Effects: Links this_node after prev_node in the circular list.

Complexity: Constant

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

7.
```
static void transfer_after(node_ptr p, node_ptr b, node_ptr e);
```

Requires: b and e must be nodes of the same circular list or an empty range. and p must be a node of a different circular list.

Effects: Removes the nodes from (b, e] range from their circular list and inserts them after p in p's circular list.

Complexity: Constant

Throws: Nothing.

8.
```
static void init_header(const node_ptr & this_node);
```

Effects: Constructs an empty list, making this_node the only node of the circular list: `NodeTraits::get_next(this_node) == this_node`.

Complexity: Constant

Throws: Nothing.

9.
```
static node_ptr
get_previous_node(const node_ptr & prev_init_node, const node_ptr & this_node);
```

Requires: this_node and prev_init_node must be in the same circular list.

Effects: Returns the previous node of this_node in the circular list starting. the search from prev_init_node. The first node checked for equality is NodeTraits::get_next(prev_init_node).

Complexity: Linear to the number of elements between prev_init_node and this_node.

Throws: Nothing.

10.
```
static node_ptr get_previous_node(const node_ptr & this_node);
```

Requires: this_node must be in a circular list or be an empty circular list.

Effects: Returns the previous node of this_node in the circular list.

Complexity: Linear to the number of elements in the circular list.

Throws: Nothing.

11.
```
static node_ptr get_previous_previous_node(const node_ptr & this_node);
```

Requires: this_node must be in a circular list or be an empty circular list.

Effects: Returns the previous node of the previous node of this_node in the circular list.

Complexity: Linear to the number of elements in the circular list.

Throws: Nothing.

12.
```
static node_ptr
get_previous_previous_node(node_ptr p, const node_ptr & this_node);
```

Requires: this_node and p must be in the same circular list.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Returns the previous node of the previous node of this_node in the circular list starting. the search from p. The first node checked for equality is NodeTraits::get_next((NodeTraits::get_next(p)).

Complexity: Linear to the number of elements in the circular list.

Throws: Nothing.

13.
```
static std::size_t count(const const_node_ptr & this_node);
```

Requires: this_node must be in a circular list or be an empty circular list.

Effects: Returns the number of nodes in a circular list. If the circular list is empty, returns 1.

Complexity: Linear

Throws: Nothing.

14.
```
static void unlink(const node_ptr & this_node);
```

Requires: this_node must be in a circular list, be an empty circular list or be inited.

Effects: Unlinks the node from the circular list.

Complexity: Linear to the number of elements in the circular list

Throws: Nothing.

15.
```
static void link_before(const node_ptr & nxt_node, const node_ptr & this_node);
```

Requires: nxt_node must be a node of a circular list.

Effects: Links this_node before nxt_node in the circular list.

Complexity: Linear to the number of elements in the circular list.

Throws: Nothing.

16.
```
static void swap_nodes(const node_ptr & this_node, 
                       const node_ptr & other_node);
```

Requires: this_node and other_node must be nodes inserted in circular lists or be empty circular lists.

Effects: Swaps the position of the nodes: this_node is inserted in other_nodes position in the second circular list and the other_node is inserted in this_node's position in the first circular list.

Complexity: Linear to number of elements of both lists

Throws: Nothing.

17.
```
static void reverse(const node_ptr & p);
```

Effects: Reverses the order of elements in the list.

Throws: Nothing.

Complexity: This function is linear to the contained elements.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

18.
```
static node_ptr move_backwards(const node_ptr & p, std::size_t n);
```

Effects: Moves the node p n positions towards the end of the list.

Returns: The previous node of p after the function if there has been any movement, Null if n leads to no movement.

Throws: Nothing.

Complexity: Linear to the number of elements plus the number moved positions.

19.
```
static node_ptr move_forward(const node_ptr & p, std::size_t n);
```

Effects: Moves the node p n positions towards the beginning of the list.

Returns: The previous node of p after the function if there has been any movement, Null if n leads equals to no movement.

Throws: Nothing.

Complexity: Linear to the number of elements plus the number moved positions.

Header <boost/intrusive/derivation_value_traits.hpp>

```
namespace boost {
  namespace intrusive {
    template<typename T, typename NodeTraits, 
             link_mode_type LinkMode = safe_link> 
      struct derivation_value_traits;
  }
}
```

Struct template derivation_value_traits

boost::intrusive::derivation_value_traits

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/intrusive/derivation_value_traits.hpp>

template<typename T, typename NodeTraits, link_mode_type LinkMode = safe_link> 
struct derivation_value_traits {
  // types
  typedef NodeTraits                                                 node_traits;      
  typedef T                                                          value_type;       
  typedef node_traits::node                                          node;             
  typedef node_traits::node_ptr                                      node_ptr;         
  typedef node_traits::const_node_ptr                                const_node_ptr;   
  typedef boost::pointer_to_other< node_ptr, T >::type               pointer;          
  typedef boost::pointer_to_other< node_ptr, const T >::type         const_pointer;    
  typedef boost::intrusive::pointer_traits< pointer >::reference       reference;        
  typedef boost::intrusive::pointer_traits< const_pointer >::reference const_reference;

  // public static functions
  static node_ptr to_node_ptr(reference);
  static const_node_ptr to_node_ptr(const_reference);
  static pointer to_value_ptr(const node_ptr &);
  static const_pointer to_value_ptr(const const_node_ptr &);

  // public data members
  static const link_mode_type link_mode;
};
```

Description

This value traits template is used to create value traits from user defined node traits where value_traits::value_type will derive from node_traits::node

`derivation_value_traits` public static functions

1.
```
static node_ptr to_node_ptr(reference value);
```

2.
```
static const_node_ptr to_node_ptr(const_reference value);
```

3.
```
static pointer to_value_ptr(const node_ptr & n);
```

4.
```
static const_pointer to_value_ptr(const const_node_ptr & n);
```

Header <boost/intrusive/hashtable.hpp>

```
namespace boost {
  namespace intrusive {
    template<typename T, class... Options> class hashtable;

    template<typename T, class... Options> struct make_hashtable;
  }
}
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Class template hashtable

boost::intrusive::hashtable

Synopsis

```
// In header: <boost/intrusive/hashtable.hpp>

template<typename T, class... Options> 
class hashtable {
public:
  // types
  typedef Config::value_traits                                                             value_traits; ↵

  typedef real_value_traits::pointer                                                       pointer;      ↵

  typedef real_value_traits::const_pointer                                                 const_point↵
er;
  typedef real_value_traits::value_type                                                    value_type;   ↵

  typedef pointer_traits< pointer >::reference                                             reference;    ↵

  typedef pointer_traits< const_pointer >::reference                                       const_refer↵
ence;
  typedef pointer_traits< pointer >::difference_type                                  difference_type; ↵

  typedef Config::size_type                                                                size_type;    ↵

  typedef value_type                                                                       key_type;     ↵

  typedef Config::equal                                                                    key_equal;    ↵

  typedef Config::hash                                                                     hasher;       ↵

  typedef unspecified                                                                      bucket_type;  ↵

  typedef pointer_traits< pointer >::template rebind_pointer< bucket_type >::type bucket_ptr;  ↵

  typedef slist::iterator                                                                  siterator;    ↵

  typedef slist::const_iterator                                                            const_siterat↵
or;
  typedef unspecified                                                                      iterator;     ↵

  typedef unspecified                                                                      const_iterat↵
or;
  typedef real_value_traits::node_traits                                                   node_traits;  ↵

  typedef node_traits::node                                                                node;         ↵

  typedef pointer_traits< pointer >::template rebind_pointer< node >::type       node_ptr;     ↵

  typedef pointer_traits< pointer >::template rebind_pointer< const node >::type  const_node_ptr; ↵

  typedef slist::node_algorithms                                                      node_algorithms; ↵

  typedef unspecified                                                                      insert_com↵
mit_data;
  typedef unspecified                                                                      local_iterat↵
or;
  typedef unspecified                                                                  const_local_iter↵
ator;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
// construct/copy/destruct
explicit hashtable(const bucket_traits &, const hasher & = hasher(),
                   const key_equal & = key_equal(),
                   const value_traits & = value_traits());
hashtable(BOOST_RV_REF(hashtable));
hashtable& operator=(BOOST_RV_REF(hashtable));
~hashtable();

// public member functions
const real_value_traits & get_real_value_traits() const;
real_value_traits & get_real_value_traits();
iterator begin();
const_iterator begin() const;
const_iterator cbegin() const;
iterator end();
const_iterator end() const;
const_iterator cend() const;
hasher hash_function() const;
key_equal key_eq() const;
bool empty() const;
size_type size() const;
void swap(hashtable &);
template<typename Cloner, typename Disposer>
  void clone_from(const hashtable &, Cloner, Disposer);
iterator insert_equal(reference);
template<typename Iterator> void insert_equal(Iterator, Iterator);
std::pair< iterator, bool > insert_unique(reference);
template<typename Iterator> void insert_unique(Iterator, Iterator);
template<typename KeyType, typename KeyHasher, typename KeyValueEqual>
  std::pair< iterator, bool >
  insert_unique_check(const KeyType &, KeyHasher, KeyValueEqual,
                      insert_commit_data &);
iterator insert_unique_commit(reference, const insert_commit_data &);
void erase(const_iterator);
void erase(const_iterator, const_iterator);
size_type erase(const_reference);
template<typename KeyType, typename KeyHasher, typename KeyValueEqual>
  size_type erase(const KeyType &, KeyHasher, KeyValueEqual);
template<typename Disposer> void erase_and_dispose(const_iterator, Disposer);
template<typename Disposer>
  void erase_and_dispose(const_iterator, const_iterator, Disposer);
template<typename Disposer>
  size_type erase_and_dispose(const_reference, Disposer);
template<typename KeyType, typename KeyHasher, typename KeyValueEqual,
         typename Disposer>
  size_type erase_and_dispose(const KeyType &, KeyHasher, KeyValueEqual,
                              Disposer);
void clear();
template<typename Disposer> void clear_and_dispose(Disposer);
size_type count(const_reference) const;
template<typename KeyType, typename KeyHasher, typename KeyValueEqual>
  size_type count(const KeyType &, const KeyHasher &, const KeyValueEqual &) const;
iterator find(const_reference);
template<typename KeyType, typename KeyHasher, typename KeyValueEqual>
  iterator find(const KeyType &, KeyHasher, KeyValueEqual);
const_iterator find(const_reference) const;
template<typename KeyType, typename KeyHasher, typename KeyValueEqual>
  const_iterator find(const KeyType &, KeyHasher, KeyValueEqual) const;
std::pair< iterator, iterator > equal_range(const_reference);
template<typename KeyType, typename KeyHasher, typename KeyValueEqual>
  std::pair< iterator, iterator >
  equal_range(const KeyType &, KeyHasher, KeyValueEqual);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
  std::pair< const_iterator, const_iterator >
  equal_range(const_reference) const;
  template<typename KeyType, typename KeyHasher, typename KeyValueEqual> 
    std::pair< const_iterator, const_iterator > 
    equal_range(const KeyType &, KeyHasher, KeyValueEqual) const;
  iterator iterator_to(reference);
  const_iterator iterator_to(const_reference) const;
  local_iterator local_iterator_to(reference);
  const_local_iterator local_iterator_to(const_reference) const;
  size_type bucket_count() const;
  size_type bucket_size(size_type) const;
  size_type bucket(const key_type &) const;
  template<typename KeyType, typename KeyHasher> 
    size_type bucket(const KeyType &, const KeyHasher &) const;
  bucket_ptr bucket_pointer() const;
  local_iterator begin(size_type);
  const_local_iterator begin(size_type) const;
  const_local_iterator cbegin(size_type) const;
  local_iterator end(size_type);
  const_local_iterator end(size_type) const;
  const_local_iterator cend(size_type) const;
  void rehash(const bucket_traits &);
  bool incremental_rehash(bool = true);
  bool incremental_rehash(const bucket_traits &);
  size_type split_count() const;

  // public static functions
  static local_iterator s_local_iterator_to(reference);
  static const_local_iterator s_local_iterator_to(const_reference);
  static size_type suggested_upper_bucket_count(size_type);
  static size_type suggested_lower_bucket_count(size_type);

  // public data members
  static const bool stateful_value_traits;
  static const bool store_hash;
  static const bool unique_keys;
  static const bool constant_time_size;
  static const bool cache_begin;
  static const bool compare_hash;
  static const bool incremental;
  static const bool power_2_buckets;
  static const bool optimize_multikey;
};
```

Description

The class template hashtable is an intrusive hash table container, that is used to construct intrusive unordered_set and unordered_multiset containers. The no-throw guarantee holds only, if the Equal object and Hasher don't throw.

hashtable is a semi-intrusive container: each object to be stored in the container must contain a proper hook, but the container also needs additional auxiliary memory to work: hashtable needs a pointer to an array of type `bucket_type` to be passed in the constructor. This bucket array must have at least the same lifetime as the container. This makes the use of hashtable more complicated than purely intrusive containers. `bucket_type` is default-constructible, copyable and assignable

The template parameter `T` is the type to be managed by the container. The user can specify additional options and if no options are provided default options are used.

The container supports the following options: `base_hook<>/member_hook<>/value_traits<>`, `constant_time_size<>`, `size_type<>`, `hash<>` and `equal<>` `bucket_traits<>`, power_2_buckets<>, cache_begin<> and incremental<>.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

hashtable only provides forward iterators but it provides 4 iterator types: iterator and const_iterator to navigate through the whole container and local_iterator and const_local_iterator to navigate through the values stored in a single bucket. Local iterators are faster and smaller.

It's not recommended to use non constant-time size hashtables because several key functions, like "empty()", become non-constant time functions. Non constant_time size hashtables are mainly provided to support auto-unlink hooks.

hashtables, does not make automatic rehashings nor offers functions related to a load factor. Rehashing can be explicitly requested and the user must provide a new bucket array that will be used from that moment.

Since no automatic rehashing is done, iterators are never invalidated when inserting or erasing elements. Iterators are only invalidated when rehashing.

`hashtable` public construct/copy/destruct

1.
```
explicit hashtable(const bucket_traits & b_traits, 
                   const hasher & hash_func = hasher(), 
                   const key_equal & equal_func = key_equal(), 
                   const value_traits & v_traits = value_traits());
```

   Requires: buckets must not be being used by any other resource.

   Effects: Constructs an empty `unordered_set`, storing a reference to the bucket array and copies of the key_hasher and equal_func functors.

   Complexity: Constant.

   Throws: If value_traits::node_traits::node constructor throws (this does not happen with predefined Boost.Intrusive hooks) or the copy constructor or invocation of hash_func or equal_func throws.

   Notes: buckets array must be disposed only after *this is disposed.

2.
```
hashtable(BOOST_RV_REF(hashtable) x);
```

   Effects: to-do

3.
```
hashtable& operator=(BOOST_RV_REF(hashtable) x);
```

   Effects: to-do

4.
```
~hashtable();
```

   Effects: Detaches all elements from this. The objects in the `unordered_set` are not deleted (i.e. no destructors are called).

   Complexity: Linear to the number of elements in the `unordered_set`, if it's a safe-mode or auto-unlink value. Otherwise constant.

   Throws: Nothing.

`hashtable` public member functions

1.
```
const real_value_traits & get_real_value_traits() const;
```

2.
```
real_value_traits & get_real_value_traits();
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

3.
```
iterator begin();
```

Effects: Returns an iterator pointing to the beginning of the `unordered_set`.

Complexity: Amortized constant time. Worst case (empty `unordered_set`): O(this->bucket_count())

Throws: Nothing.

4.
```
const_iterator begin() const;
```

Effects: Returns a const_iterator pointing to the beginning of the `unordered_set`.

Complexity: Amortized constant time. Worst case (empty `unordered_set`): O(this->bucket_count())

Throws: Nothing.

5.
```
const_iterator cbegin() const;
```

Effects: Returns a const_iterator pointing to the beginning of the `unordered_set`.

Complexity: Amortized constant time. Worst case (empty `unordered_set`): O(this->bucket_count())

Throws: Nothing.

6.
```
iterator end();
```

Effects: Returns an iterator pointing to the end of the `unordered_set`.

Complexity: Constant.

Throws: Nothing.

7.
```
const_iterator end() const;
```

Effects: Returns a const_iterator pointing to the end of the `unordered_set`.

Complexity: Constant.

Throws: Nothing.

8.
```
const_iterator cend() const;
```

Effects: Returns a const_iterator pointing to the end of the `unordered_set`.

Complexity: Constant.

Throws: Nothing.

9.
```
hasher hash_function() const;
```

Effects: Returns the hasher object used by the `unordered_set`.

Complexity: Constant.

Throws: If hasher copy-constructor throws.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

10.
```
key_equal key_eq() const;
```

Effects: Returns the key_equal object used by the `unordered_set`.

Complexity: Constant.

Throws: If key_equal copy-constructor throws.

11.
```
bool empty() const;
```

Effects: Returns true if the container is empty.

Complexity: if constant-time size and `cache_begin` options are disabled, average constant time (worst case, with empty() == true: O(this->bucket_count()). Otherwise constant.

Throws: Nothing.

12.
```
size_type size() const;
```

Effects: Returns the number of elements stored in the `unordered_set`.

Complexity: Linear to elements contained in *this if `constant_time_size` is false. Constant-time otherwise.

Throws: Nothing.

13.
```
void swap(hashtable & other);
```

Requires: the hasher and the equality function unqualified swap call should not throw.

Effects: Swaps the contents of two unordered_sets. Swaps also the contained bucket array and equality and hasher functors.

Complexity: Constant.

Throws: If the swap() call for the comparison or hash functors found using ADL throw. Basic guarantee.

14.
```
template<typename Cloner, typename Disposer> 
  void clone_from(const hashtable & src, Cloner cloner, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw Cloner should yield to nodes that compare equal and produce the same hash than the original node.

Effects: Erases all the elements from *this calling Disposer::operator()(pointer), clones all the elements from src calling Cloner::operator()(const_reference) and inserts them on *this. The hash function and the equality predicate are copied from the source.

If `store_hash` option is true, this method does not use the hash function.

If any operation throws, all cloned elements are unlinked and disposed calling Disposer::operator()(pointer).

Complexity: Linear to erased plus inserted elements.

Throws: If cloner or hasher throw or hash or equality predicate copying throws. Basic guarantee.

15.
```
iterator insert_equal(reference value);
```

Requires: value must be an lvalue

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Inserts the value into the `unordered_set`.

Returns: An iterator to the inserted value.

Complexity: Average case O(1), worst case O(this->size()).

Throws: If the internal hasher or the equality functor throws. Strong guarantee.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

16.
```
template<typename Iterator> void insert_equal(Iterator b, Iterator e);
```

Requires: Dereferencing iterator must yield an lvalue of type value_type.

Effects: Equivalent to this->insert_equal(t) for each element in [b, e).

Complexity: Average case O(N), where N is std::distance(b, e). Worst case O(N*this->size()).

Throws: If the internal hasher or the equality functor throws. Basic guarantee.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

17.
```
std::pair< iterator, bool > insert_unique(reference value);
```

Requires: value must be an lvalue

Effects: Tries to inserts value into the `unordered_set`.

Returns: If the value is not already present inserts it and returns a pair containing the iterator to the new value and true. If there is an equivalent value returns a pair containing an iterator to the already present value and false.

Complexity: Average case O(1), worst case O(this->size()).

Throws: If the internal hasher or the equality functor throws. Strong guarantee.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

18.
```
template<typename Iterator> void insert_unique(Iterator b, Iterator e);
```

Requires: Dereferencing iterator must yield an lvalue of type value_type.

Effects: Equivalent to this->insert_unique(t) for each element in [b, e).

Complexity: Average case O(N), where N is std::distance(b, e). Worst case O(N*this->size()).

Throws: If the internal hasher or the equality functor throws. Basic guarantee.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

19.
```
template<typename KeyType, typename KeyHasher, typename KeyValueEqual> 
  std::pair< iterator, bool > 
  insert_unique_check(const KeyType & key, KeyHasher hash_func, 
                      KeyValueEqual equal_func, 
                      insert_commit_data & commit_data);
```

Requires: "hash_func" must be a hash function that induces the same hash values as the stored hasher. The difference is that "hash_func" hashes the given key instead of the value_type.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

"equal_func" must be a equality function that induces the same equality as key_equal. The difference is that "equal_func" compares an arbitrary key with the contained values.

Effects: Checks if a value can be inserted in the `unordered_set`, using a user provided key instead of the value itself.

Returns: If there is an equivalent value returns a pair containing an iterator to the already present value and false. If the value can be inserted returns true in the returned pair boolean and fills "commit_data" that is meant to be used with the "insert_commit" function.

Complexity: Average case O(1), worst case O(this->size()).

Throws: If hash_func or equal_func throw. Strong guarantee.

Notes: This function is used to improve performance when constructing a value_type is expensive: if there is an equivalent value the constructed object must be discarded. Many times, the part of the node that is used to impose the hash or the equality is much cheaper to construct than the value_type and this function offers the possibility to use that the part to check if the insertion will be successful.

If the check is successful, the user can construct the value_type and use "insert_commit" to insert the object in constant-time.

"commit_data" remains valid for a subsequent "insert_commit" only if no more objects are inserted or erased from the `unordered_set`.

After a successful rehashing insert_commit_data remains valid.

20.
```
iterator insert_unique_commit(reference value,
                              const insert_commit_data & commit_data);
```

Requires: value must be an lvalue of type value_type. commit_data must have been obtained from a previous call to "insert_check". No objects should have been inserted or erased from the `unordered_set` between the "insert_check" that filled "commit_data" and the call to "insert_commit".

Effects: Inserts the value in the `unordered_set` using the information obtained from the "commit_data" that a previous "insert_check" filled.

Returns: An iterator to the newly inserted object.

Complexity: Constant time.

Throws: Nothing.

Notes: This function has only sense if a "insert_check" has been previously executed to fill "commit_data". No value should be inserted or erased between the "insert_check" and "insert_commit" calls.

After a successful rehashing insert_commit_data remains valid.

21.
```
void erase(const_iterator i);
```

Effects: Erases the element pointed to by i.

Complexity: Average case O(1), worst case O(this->size()).

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased element. No destructors are called.

22.
```
void erase(const_iterator b, const_iterator e);
```

Effects: Erases the range pointed to by b end e.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Average case O(std::distance(b, e)), worst case O(this->size()).

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

23.
```
size_type erase(const_reference value);
```

Effects: Erases all the elements with the given value.

Returns: The number of erased elements.

Complexity: Average case O(this->count(value)). Worst case O(this->size()).

Throws: If the internal hasher or the equality functor throws. Basic guarantee.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

24.
```
template<typename KeyType, typename KeyHasher, typename KeyValueEqual>
  size_type erase(const KeyType & key, KeyHasher hash_func,
                  KeyValueEqual equal_func);
```

Requires: "hash_func" must be a hash function that induces the same hash values as the stored hasher. The difference is that "hash_func" hashes the given key instead of the value_type.

"equal_func" must be a equality function that induces the same equality as key_equal. The difference is that "equal_func" compares an arbitrary key with the contained values.

Effects: Erases all the elements that have the same hash and compare equal with the given key.

Returns: The number of erased elements.

Complexity: Average case O(this->count(value)). Worst case O(this->size()).

Throws: If hash_func or equal_func throw. Basic guarantee.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

25.
```
template<typename Disposer>
  void erase_and_dispose(const_iterator i, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases the element pointed to by i. Disposer::operator()(pointer) is called for the removed element.

Complexity: Average case O(1), worst case O(this->size()).

Throws: Nothing.

Note: Invalidates the iterators to the erased elements.

26.
```
template<typename Disposer>
  void erase_and_dispose(const_iterator b, const_iterator e,
                         Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases the range pointed to by b end e. Disposer::operator()(pointer) is called for the removed elements.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Average case O(std::distance(b, e)), worst case O(this->size()).

Throws: Nothing.

Note: Invalidates the iterators to the erased elements.

27.
```
template<typename Disposer> 
  size_type erase_and_dispose(const_reference value, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases all the elements with the given value. Disposer::operator()(pointer) is called for the removed elements.

Returns: The number of erased elements.

Complexity: Average case O(this->count(value)). Worst case O(this->size()).

Throws: If the internal hasher or the equality functor throws. Basic guarantee.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

28.
```
template<typename KeyType, typename KeyHasher, typename KeyValueEqual, 
         typename Disposer> 
  size_type erase_and_dispose(const KeyType & key, KeyHasher hash_func, 
                              KeyValueEqual equal_func, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases all the elements with the given key. according to the comparison functor "equal_func". Disposer::operator()(pointer) is called for the removed elements.

Returns: The number of erased elements.

Complexity: Average case O(this->count(value)). Worst case O(this->size()).

Throws: If hash_func or equal_func throw. Basic guarantee.

Note: Invalidates the iterators to the erased elements.

29.
```
void clear();
```

Effects: Erases all of the elements.

Complexity: Linear to the number of elements on the container. if it's a safe-mode or auto-unlink value_type. Constant time otherwise.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

30.
```
template<typename Disposer> void clear_and_dispose(Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases all of the elements.

Complexity: Linear to the number of elements on the container. Disposer::operator()(pointer) is called for the removed elements.

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

31.
```
size_type count(const_reference value) const;
```

Effects: Returns the number of contained elements with the given value

Complexity: Average case O(1), worst case O(this->size()).

Throws: If the internal hasher or the equality functor throws.

32.
```
template<typename KeyType, typename KeyHasher, typename KeyValueEqual>
  size_type count(const KeyType & key, const KeyHasher & hash_func,
                  const KeyValueEqual & equal_func) const;
```

Requires: "hash_func" must be a hash function that induces the same hash values as the stored hasher. The difference is that "hash_func" hashes the given key instead of the value_type.

"equal_func" must be a equality function that induces the same equality as key_equal. The difference is that "equal_func" compares an arbitrary key with the contained values.

Effects: Returns the number of contained elements with the given key

Complexity: Average case O(1), worst case O(this->size()).

Throws: If hash_func or equal throw.

33.
```
iterator find(const_reference value);
```

Effects: Finds an iterator to the first element is equal to "value" or end() if that element does not exist.

Complexity: Average case O(1), worst case O(this->size()).

Throws: If the internal hasher or the equality functor throws.

34.
```
template<typename KeyType, typename KeyHasher, typename KeyValueEqual>
  iterator find(const KeyType & key, KeyHasher hash_func,
                KeyValueEqual equal_func);
```

Requires: "hash_func" must be a hash function that induces the same hash values as the stored hasher. The difference is that "hash_func" hashes the given key instead of the value_type.

"equal_func" must be a equality function that induces the same equality as key_equal. The difference is that "equal_func" compares an arbitrary key with the contained values.

Effects: Finds an iterator to the first element whose key is "key" according to the given hash and equality functor or end() if that element does not exist.

Complexity: Average case O(1), worst case O(this->size()).

Throws: If hash_func or equal_func throw.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

35.
```
const_iterator find(const_reference value) const;
```

Effects: Finds a const_iterator to the first element whose key is "key" or end() if that element does not exist.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Average case O(1), worst case O(this->size()).

Throws: If the internal hasher or the equality functor throws.

36.
```
template<typename KeyType, typename KeyHasher, typename KeyValueEqual>
  const_iterator
  find(const KeyType & key, KeyHasher hash_func, KeyValueEqual equal_func) const;
```

Requires: "hash_func" must be a hash function that induces the same hash values as the stored hasher. The difference is that "hash_func" hashes the given key instead of the value_type.

"equal_func" must be a equality function that induces the same equality as key_equal. The difference is that "equal_func" compares an arbitrary key with the contained values.

Effects: Finds an iterator to the first element whose key is "key" according to the given hasher and equality functor or end() if that element does not exist.

Complexity: Average case O(1), worst case O(this->size()).

Throws: If hash_func or equal_func throw.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

37.
```
std::pair< iterator, iterator > equal_range(const_reference value);
```

Effects: Returns a range containing all elements with values equivalent to value. Returns std::make_pair(this->end(), this->end()) if no such elements exist.

Complexity: Average case O(this->count(value)). Worst case O(this->size()).

Throws: If the internal hasher or the equality functor throws.

38.
```
template<typename KeyType, typename KeyHasher, typename KeyValueEqual>
  std::pair< iterator, iterator >
  equal_range(const KeyType & key, KeyHasher hash_func,
              KeyValueEqual equal_func);
```

Requires: "hash_func" must be a hash function that induces the same hash values as the stored hasher. The difference is that "hash_func" hashes the given key instead of the value_type.

"equal_func" must be a equality function that induces the same equality as key_equal. The difference is that "equal_func" compares an arbitrary key with the contained values.

Effects: Returns a range containing all elements with equivalent keys. Returns std::make_pair(this->end(), this->end()) if no such elements exist.

Complexity: Average case O(this->count(key, hash_func, equal_func)). Worst case O(this->size()).

Throws: If hash_func or the equal_func throw.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

39.
```
std::pair< const_iterator, const_iterator >
equal_range(const_reference value) const;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Returns a range containing all elements with values equivalent to value. Returns std::make_pair(this->end(), this->end()) if no such elements exist.

Complexity: Average case O(this->count(value)). Worst case O(this->size()).

Throws: If the internal hasher or the equality functor throws.

40.
```
template<typename KeyType, typename KeyHasher, typename KeyValueEqual>
  std::pair< const_iterator, const_iterator >
  equal_range(const KeyType & key, KeyHasher hash_func,
              KeyValueEqual equal_func) const;
```

Requires: "hash_func" must be a hash function that induces the same hash values as the stored hasher. The difference is that "hash_func" hashes the given key instead of the value_type.

"equal_func" must be a equality function that induces the same equality as key_equal. The difference is that "equal_func" compares an arbitrary key with the contained values.

Effects: Returns a range containing all elements with equivalent keys. Returns std::make_pair(this->end(), this->end()) if no such elements exist.

Complexity: Average case O(this->count(key, hash_func, equal_func)). Worst case O(this->size()).

Throws: If the hasher or equal_func throw.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

41.
```
iterator iterator_to(reference value);
```

Requires: value must be an lvalue and shall be in a `unordered_set` of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid iterator belonging to the `unordered_set` that points to the value

Complexity: Constant.

Throws: If the internal hash function throws.

42.
```
const_iterator iterator_to(const_reference value) const;
```

Requires: value must be an lvalue and shall be in a `unordered_set` of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid const_iterator belonging to the `unordered_set` that points to the value

Complexity: Constant.

Throws: If the internal hash function throws.

43.
```
local_iterator local_iterator_to(reference value);
```

Requires: value must be an lvalue and shall be in a `unordered_set` of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid local_iterator belonging to the `unordered_set` that points to the value

Complexity: Constant.

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

44.
```
const_local_iterator local_iterator_to(const_reference value) const;
```

Requires: value must be an lvalue and shall be in a `unordered_set` of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid const_local_iterator belonging to the `unordered_set` that points to the value

Complexity: Constant.

Throws: Nothing.

45.
```
size_type bucket_count() const;
```

Effects: Returns the number of buckets passed in the constructor or the last rehash function.

Complexity: Constant.

Throws: Nothing.

46.
```
size_type bucket_size(size_type n) const;
```

Requires: n is in the range [0, this->bucket_count()).

Effects: Returns the number of elements in the nth bucket.

Complexity: Constant.

Throws: Nothing.

47.
```
size_type bucket(const key_type & k) const;
```

Effects: Returns the index of the bucket in which elements with keys equivalent to k would be found, if any such element existed.

Complexity: Constant.

Throws: If the hash functor throws.

Note: the return value is in the range [0, this->bucket_count()).

48.
```
template<typename KeyType, typename KeyHasher> 
  size_type bucket(const KeyType & k, const KeyHasher & hash_func) const;
```

Requires: "hash_func" must be a hash function that induces the same hash values as the stored hasher. The difference is that "hash_func" hashes the given key instead of the value_type.

Effects: Returns the index of the bucket in which elements with keys equivalent to k would be found, if any such element existed.

Complexity: Constant.

Throws: If hash_func throws.

Note: the return value is in the range [0, this->bucket_count()).

49.
```
bucket_ptr bucket_pointer() const;
```

Effects: Returns the bucket array pointer passed in the constructor or the last rehash function.

Complexity: Constant.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: Nothing.

50.
```
local_iterator begin(size_type n);
```

Requires: n is in the range [0, this->bucket_count()).

Effects: Returns a local_iterator pointing to the beginning of the sequence stored in the bucket n.

Complexity: Constant.

Throws: Nothing.

Note: [this->begin(n), this->end(n)) is a valid range containing all of the elements in the nth bucket.

51.
```
const_local_iterator begin(size_type n) const;
```

Requires: n is in the range [0, this->bucket_count()).

Effects: Returns a const_local_iterator pointing to the beginning of the sequence stored in the bucket n.

Complexity: Constant.

Throws: Nothing.

Note: [this->begin(n), this->end(n)) is a valid range containing all of the elements in the nth bucket.

52.
```
const_local_iterator cbegin(size_type n) const;
```

Requires: n is in the range [0, this->bucket_count()).

Effects: Returns a const_local_iterator pointing to the beginning of the sequence stored in the bucket n.

Complexity: Constant.

Throws: Nothing.

Note: [this->begin(n), this->end(n)) is a valid range containing all of the elements in the nth bucket.

53.
```
local_iterator end(size_type n);
```

Requires: n is in the range [0, this->bucket_count()).

Effects: Returns a local_iterator pointing to the end of the sequence stored in the bucket n.

Complexity: Constant.

Throws: Nothing.

Note: [this->begin(n), this->end(n)) is a valid range containing all of the elements in the nth bucket.

54.
```
const_local_iterator end(size_type n) const;
```

Requires: n is in the range [0, this->bucket_count()).

Effects: Returns a const_local_iterator pointing to the end of the sequence stored in the bucket n.

Complexity: Constant.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: Nothing.

Note: [this->begin(n), this->end(n)) is a valid range containing all of the elements in the nth bucket.

55.
```
const_local_iterator cend(size_type n) const;
```

Requires: n is in the range [0, this->bucket_count()).

Effects: Returns a const_local_iterator pointing to the end of the sequence stored in the bucket n.

Complexity: Constant.

Throws: Nothing.

Note: [this->begin(n), this->end(n)) is a valid range containing all of the elements in the nth bucket.

56.
```
void rehash(const bucket_traits & new_bucket_traits);
```

Requires: new_bucket_traits can hold a pointer to a new bucket array or the same as the old bucket array with a different length. new_size is the length of the the array pointed by new_buckets. If new_bucket_traits.bucket_begin() == this->bucket_pointer() new_bucket_traits.bucket_count() can be bigger or smaller than this->bucket_count(). 'new_bucket_traits' copy constructor should not throw.

Effects: Updates the internal reference with the new bucket, erases the values from the old bucket and inserts then in the new one. Bucket traits hold by *this is assigned from new_bucket_traits. If the container is configured as incremental<>, the split bucket is set to the new bucket_count().

If `store_hash` option is true, this method does not use the hash function.

Complexity: Average case linear in this->size(), worst case quadratic.

Throws: If the hasher functor throws. Basic guarantee.

57.
```
bool incremental_rehash(bool grow = true);
```

Requires:

Effects:

Complexity:

Throws:

Note: this method is only available if incremental<true> option is activated.

58.
```
bool incremental_rehash(const bucket_traits & new_bucket_traits);
```

Effects: If new_bucket_traits.bucket_count() is not this->bucket_count()/2 or this->bucket_count()*2, or this->split_bucket() != new_bucket_traits.bucket_count() returns false and does nothing.

Otherwise, copy assigns new_bucket_traits to the internal `bucket_traits` and transfers all the objects from old buckets to the new ones.

Complexity: Linear to size().

Throws: Nothing

Note: this method is only available if incremental<true> option is activated.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

59.
```
size_type split_count() const;
```

Requires:

Effects:

Complexity:

Throws:

`hashtable` public static functions

1.
```
static local_iterator s_local_iterator_to(reference value);
```

Requires: value must be an lvalue and shall be in a `unordered_set` of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid local_iterator belonging to the `unordered_set` that points to the value

Complexity: Constant.

Throws: Nothing.

Note: This static function is available only if the *value traits* is stateless.

2.
```
static const_local_iterator s_local_iterator_to(const_reference value);
```

Requires: value must be an lvalue and shall be in a `unordered_set` of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid const_local_iterator belonging to the `unordered_set` that points to the value

Complexity: Constant.

Throws: Nothing.

Note: This static function is available only if the *value traits* is stateless.

3.
```
static size_type suggested_upper_bucket_count(size_type n);
```

Effects: Returns the nearest new bucket count optimized for the container that is bigger or equal than n. This suggestion can be used to create bucket arrays with a size that will usually improve container's performance. If such value does not exist, the higher possible value is returned.

Complexity: Amortized constant time.

Throws: Nothing.

4.
```
static size_type suggested_lower_bucket_count(size_type n);
```

Effects: Returns the nearest new bucket count optimized for the container that is smaller or equal than n. This suggestion can be used to create bucket arrays with a size that will usually improve container's performance. If such value does not exist, the lowest possible value is returned.

Complexity: Amortized constant time.

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Struct template make_hashtable

boost::intrusive::make_hashtable

Synopsis

```
// In header: <boost/intrusive/hashtable.hpp>

template<typename T, class... Options>
struct make_hashtable {
  // types
  typedef implementation_defined type;
};
```

Description

Helper metafunction to define a `hashtable` that yields to the same type when the same options (either explicitly or implicitly) are used.

Header <boost/intrusive/linear_slist_algorithms.hpp>

```
namespace boost {
  namespace intrusive {
    template<typename NodeTraits> class linear_slist_algorithms;
  }
}
```

Class template linear_slist_algorithms

boost::intrusive::linear_slist_algorithms

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/intrusive/linear_slist_algorithms.hpp>

template<typename NodeTraits>
class linear_slist_algorithms {
public:
  // types
  typedef NodeTraits::node           node;
  typedef NodeTraits::node_ptr       node_ptr;
  typedef NodeTraits::const_node_ptr const_node_ptr;
  typedef NodeTraits                 node_traits;

  // public static functions
  static void init(const node_ptr &);
  static bool unique(const_node_ptr);
  static bool inited(const_node_ptr);
  static void unlink_after(const node_ptr &);
  static void unlink_after(const node_ptr &, const node_ptr &);
  static void link_after(const node_ptr &, const node_ptr &);
  static void transfer_after(const node_ptr &, const node_ptr &,
                             const node_ptr &);
  static void init_header(const node_ptr &);
  static node_ptr get_previous_node(const node_ptr &, const node_ptr &);
  static std::size_t count(const const_node_ptr &);
  static void swap_trailing_nodes(const node_ptr &, const node_ptr &);
  static node_ptr reverse(const node_ptr &);
  static std::pair< node_ptr, node_ptr >
  move_first_n_backwards(const node_ptr &, std::size_t);
  static std::pair< node_ptr, node_ptr >
  move_first_n_forward(const node_ptr &, std::size_t);
};
```

Description

linear_slist_algorithms provides basic algorithms to manipulate nodes forming a linear singly linked list.

linear_slist_algorithms is configured with a NodeTraits class, which encapsulates the information about the node to be manipulated. NodeTraits must support the following interface:

Typedefs:

`node`: The type of the node that forms the linear list

`node_ptr`: A pointer to a node

`const_node_ptr`: A pointer to a const node

Static functions:

`static node_ptr get_next(const_node_ptr n);`

`static void set_next(node_ptr n, node_ptr next);`

`linear_slist_algorithms` public static functions

1.
```
static void init(const node_ptr & this_node);
```

   Effects: Constructs an non-used list element, putting the next pointer to null: `NodeTraits::get_next(this_node) == node_ptr()`

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Constant

Throws: Nothing.

2.
```
static bool unique(const_node_ptr this_node);
```

Requires: this_node must be in a circular list or be an empty circular list.

Effects: Returns true is "this_node" is the only node of a circular list: or it's a not inserted node: `return node_ptr() == NodeTraits::get_next(this_node) || NodeTraits::get_next(this_node) == this_node`

Complexity: Constant

Throws: Nothing.

3.
```
static bool inited(const_node_ptr this_node);
```

Effects: Returns true is "this_node" has the same state as if it was inited using "init(node_ptr)"

Complexity: Constant

Throws: Nothing.

4.
```
static void unlink_after(const node_ptr & prev_node);
```

Requires: prev_node must be in a circular list or be an empty circular list.

Effects: Unlinks the next node of prev_node from the circular list.

Complexity: Constant

Throws: Nothing.

5.
```
static void unlink_after(const node_ptr & prev_node, 
                         const node_ptr & last_node);
```

Requires: prev_node and last_node must be in a circular list or be an empty circular list.

Effects: Unlinks the range (prev_node, last_node) from the linear list.

Complexity: Constant

Throws: Nothing.

6.
```
static void link_after(const node_ptr & prev_node, const node_ptr & this_node);
```

Requires: prev_node must be a node of a linear list.

Effects: Links this_node after prev_node in the linear list.

Complexity: Constant

Throws: Nothing.

7.
```
static void transfer_after(const node_ptr & p, const node_ptr & b, 
                           const node_ptr & e);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Requires: b and e must be nodes of the same linear list or an empty range. and p must be a node of a different linear list.

Effects: Removes the nodes from (b, e] range from their linear list and inserts them after p in p's linear list.

Complexity: Constant

Throws: Nothing.

8.
```
static void init_header(const node_ptr & this_node);
```

Effects: Constructs an empty list, making this_node the only node of the circular list: `NodeTraits::get_next(this_node) == this_node`.

Complexity: Constant

Throws: Nothing.

9.
```
static node_ptr
get_previous_node(const node_ptr & prev_init_node, const node_ptr & this_node);
```

Requires: this_node and prev_init_node must be in the same linear list.

Effects: Returns the previous node of this_node in the linear list starting. the search from prev_init_node. The first node checked for equality is NodeTraits::get_next(prev_init_node).

Complexity: Linear to the number of elements between prev_init_node and this_node.

Throws: Nothing.

10.
```
static std::size_t count(const const_node_ptr & this_node);
```

Requires: this_node must be in a linear list or be an empty linear list.

Effects: Returns the number of nodes in a linear list. If the linear list is empty, returns 1.

Complexity: Linear

Throws: Nothing.

11.
```
static void swap_trailing_nodes(const node_ptr & this_node,
                                const node_ptr & other_node);
```

Requires: this_node and other_node must be nodes inserted in linear lists or be empty linear lists.

Effects: Moves all the nodes previously chained after this_node after other_node and vice-versa.

Complexity: Constant

Throws: Nothing.

12.
```
static node_ptr reverse(const node_ptr & p);
```

Effects: Reverses the order of elements in the list.

Returns: The new first node of the list.

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: This function is linear to the contained elements.

13.
```
static std::pair< node_ptr, node_ptr >
move_first_n_backwards(const node_ptr & p, std::size_t n);
```

Effects: Moves the first n nodes starting at p to the end of the list.

Returns: A pair containing the new first and last node of the list or if there has been any movement, a null pair if n leads to no movement.

Throws: Nothing.

Complexity: Linear to the number of elements plus the number moved positions.

14.
```
static std::pair< node_ptr, node_ptr >
move_first_n_forward(const node_ptr & p, std::size_t n);
```

Effects: Moves the first n nodes starting at p to the beginning of the list.

Returns: A pair containing the new first and last node of the list or if there has been any movement, a null pair if n leads to no movement.

Throws: Nothing.

Complexity: Linear to the number of elements plus the number moved positions.

Header <boost/intrusive/link_mode.hpp>

```
namespace boost {
  namespace intrusive {
    enum link_mode_type;
  }
}
```

Type link_mode_type

boost::intrusive::link_mode_type

Synopsis

```
// In header: <boost/intrusive/link_mode.hpp>


enum link_mode_type { normal_link, safe_link, auto_unlink };
```

Description

This enumeration defines the type of value_traits that can be defined for Boost.Intrusive containers

`normal_link`	If this linking policy is specified in a value_traits class as the link_mode, containers configured with such value_traits won't set the hooks of the erased values to a default state. Containers also won't check that the hooks of the new values are default initialized.
`safe_link`	If this linking policy is specified in a value_traits class as the link_mode, containers configured with such value_traits will set the hooks of the erased values to a default state. Containers also will check that the hooks of the new values are default initialized.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

`auto_unlink`	Same as "safe_link" but the user type is an auto-unlink type, so the containers with constant-time size features won't be compatible with value_traits configured with this policy. Containers also know that the a value can be silently erased from the container without using any function provided by the containers.

Header <boost/intrusive/list.hpp>

```
namespace boost {
  namespace intrusive {
    template<typename T, class... Options> class list;

    template<typename T, class... Options> struct make_list;
    template<typename T, class... Options>
      bool operator<(const list< T, Options...> & x,
                     const list< T, Options...> & y);
    template<typename T, class... Options>
      bool operator==(const list< T, Options...> & x,
                      const list< T, Options...> & y);
    template<typename T, class... Options>
      bool operator!=(const list< T, Options...> & x,
                      const list< T, Options...> & y);
    template<typename T, class... Options>
      bool operator>(const list< T, Options...> & x,
                     const list< T, Options...> & y);
    template<typename T, class... Options>
      bool operator<=(const list< T, Options...> & x,
                      const list< T, Options...> & y);
    template<typename T, class... Options>
      bool operator>=(const list< T, Options...> & x,
                      const list< T, Options...> & y);
    template<typename T, class... Options>
      void swap(list< T, Options...> & x, list< T, Options...> & y);
  }
}
```

Class template list

boost::intrusive::list

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/intrusive/list.hpp>

template<typename T, class... Options>
class list {
public:
  // types
  typedef Config::value_traits                            value_traits;
  typedef real_value_traits::pointer                      pointer;
  typedef real_value_traits::const_pointer                const_pointer;
  typedef pointer_traits< pointer >::element_type    value_type;
  typedef pointer_traits< pointer >::reference       reference;
  typedef pointer_traits< const_pointer >::reference const_reference;
  typedef pointer_traits< pointer >::difference_type difference_type;
  typedef Config::size_type                               size_type;
  typedef list_iterator< list, false >                   iterator;
  typedef list_iterator< list, true >                    const_iterator;
  typedef unspecified                                     reverse_iterator;
  typedef unspecified                                     const_reverse_iterator;
  typedef real_value_traits::node_traits                  node_traits;
  typedef node_traits::node                               node;
  typedef node_traits::node_ptr                           node_ptr;
  typedef node_traits::const_node_ptr                     const_node_ptr;
  typedef circular_list_algorithms< node_traits >        node_algorithms;

  // construct/copy/destruct
  explicit list(const value_traits & = value_traits());
  template<typename Iterator>
    list(Iterator, Iterator, const value_traits & = value_traits());
  list(BOOST_RV_REF(list));
  list& operator=(BOOST_RV_REF(list));
  ~list();

  // public member functions
  const real_value_traits & get_real_value_traits() const;
  real_value_traits & get_real_value_traits();
  void push_back(reference);
  void push_front(reference);
  void pop_back();
  template<typename Disposer> void pop_back_and_dispose(Disposer);
  void pop_front();
  template<typename Disposer> void pop_front_and_dispose(Disposer);
  reference front();
  const_reference front() const;
  reference back();
  const_reference back() const;
  iterator begin();
  const_iterator begin() const;
  const_iterator cbegin() const;
  iterator end();
  const_iterator end() const;
  const_iterator cend() const;
  reverse_iterator rbegin();
  const_reverse_iterator rbegin() const;
  const_reverse_iterator crbegin() const;
  reverse_iterator rend();
  const_reverse_iterator rend() const;
  const_reverse_iterator crend() const;
  size_type size() const;
  bool empty() const;
  void swap(list &);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
  void shift_backwards(size_type = 1);
  void shift_forward(size_type = 1);
  iterator erase(const_iterator);
  iterator erase(const_iterator, const_iterator);
  iterator erase(const_iterator, const_iterator, difference_type);
  template<typename Disposer> 
    iterator erase_and_dispose(const_iterator, Disposer);
  template<typename Disposer> 
    iterator erase_and_dispose(const_iterator, const_iterator, Disposer);
  void clear();
  template<typename Disposer> void clear_and_dispose(Disposer);
  template<typename Cloner, typename Disposer> 
    void clone_from(const list &, Cloner, Disposer);
  iterator insert(const_iterator, reference);
  template<typename Iterator> void insert(const_iterator, Iterator, Iterator);
  template<typename Iterator> void assign(Iterator, Iterator);
  template<typename Iterator, typename Disposer> 
    void dispose_and_assign(Disposer, Iterator, Iterator);
  void splice(const_iterator, list &);
  void splice(const_iterator, list &, const_iterator);
  void splice(const_iterator, list &, const_iterator, const_iterator);
  void splice(const_iterator, list &, const_iterator, const_iterator, 
              difference_type);
  void sort();
  template<typename Predicate> void sort(Predicate);
  void merge(list &);
  template<typename Predicate> void merge(list &, Predicate);
  void reverse();
  void remove(const_reference);
  template<typename Disposer> 
    void remove_and_dispose(const_reference, Disposer);
  template<typename Pred> void remove_if(Pred);
  template<typename Pred, typename Disposer> 
    void remove_and_dispose_if(Pred, Disposer);
  void unique();
  template<typename BinaryPredicate> void unique(BinaryPredicate);
  template<typename Disposer> void unique_and_dispose(Disposer);
  template<typename BinaryPredicate, typename Disposer> 
    void unique_and_dispose(BinaryPredicate, Disposer);
  iterator iterator_to(reference);
  const_iterator iterator_to(const_reference) const;

  // public static functions
  static list & container_from_end_iterator(iterator);
  static const list & container_from_end_iterator(const_iterator);
  static iterator s_iterator_to(reference);
  static const_iterator s_iterator_to(const_reference);

  // public data members
  static const bool constant_time_size;
  static const bool stateful_value_traits;
};
```

Description

The class template list is an intrusive container that mimics most of the interface of std::list as described in the C++ standard.

The template parameter `T` is the type to be managed by the container. The user can specify additional options and if no options are provided default options are used.

The container supports the following options: `base_hook<>/member_hook<>/value_traits<>`, `constant_time_size<>` and `size_type<>`.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

`list` public construct/copy/destruct

1.
```
explicit list(const value_traits & v_traits = value_traits());
```

   Effects: constructs an empty list.

   Complexity: Constant

   Throws: If real_value_traits::node_traits::node constructor throws (this does not happen with predefined Boost.Intrusive hooks).

2.
```
template<typename Iterator>
  list(Iterator b, Iterator e, const value_traits & v_traits = value_traits());
```

   Requires: Dereferencing iterator must yield an lvalue of type value_type.

   Effects: Constructs a list equal to the range [first,last).

   Complexity: Linear in std::distance(b, e). No copy constructors are called.

   Throws: If real_value_traits::node_traits::node constructor throws (this does not happen with predefined Boost.Intrusive hooks).

3.
```
list(BOOST_RV_REF(list) x);
```

   Effects: to-do

4.
```
list& operator=(BOOST_RV_REF(list) x);
```

   Effects: to-do

5.
```
~list();
```

   Effects: If it's not a safe-mode or an auto-unlink value_type the destructor does nothing (ie. no code is generated). Otherwise it detaches all elements from this. In this case the objects in the list are not deleted (i.e. no destructors are called), but the hooks according to the ValueTraits template parameter are set to their default value.

   Complexity: Linear to the number of elements in the list, if it's a safe-mode or auto-unlink value . Otherwise constant.

`list` public member functions

1.
```
const real_value_traits & get_real_value_traits() const;
```

2.
```
real_value_traits & get_real_value_traits();
```

3.
```
void push_back(reference value);
```

   Requires: value must be an lvalue.

   Effects: Inserts the value in the back of the list. No copy constructors are called.

   Throws: Nothing.

   Complexity: Constant.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Note: Does not affect the validity of iterators and references.

4.
```
void push_front(reference value);
```

Requires: value must be an lvalue.

Effects: Inserts the value in the front of the list. No copy constructors are called.

Throws: Nothing.

Complexity: Constant.

Note: Does not affect the validity of iterators and references.

5.
```
void pop_back();
```

Effects: Erases the last element of the list. No destructors are called.

Throws: Nothing.

Complexity: Constant.

Note: Invalidates the iterators (but not the references) to the erased element.

6.
```
template<typename Disposer> void pop_back_and_dispose(Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases the last element of the list. No destructors are called. Disposer::operator()(pointer) is called for the removed element.

Throws: Nothing.

Complexity: Constant.

Note: Invalidates the iterators to the erased element.

7.
```
void pop_front();
```

Effects: Erases the first element of the list. No destructors are called.

Throws: Nothing.

Complexity: Constant.

Note: Invalidates the iterators (but not the references) to the erased element.

8.
```
template<typename Disposer> void pop_front_and_dispose(Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases the first element of the list. No destructors are called. Disposer::operator()(pointer) is called for the removed element.

Throws: Nothing.

Complexity: Constant.

Note: Invalidates the iterators to the erased element.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

9.
```
reference front();
```

Effects: Returns a reference to the first element of the list.

Throws: Nothing.

Complexity: Constant.

10.
```
const_reference front() const;
```

Effects: Returns a const_reference to the first element of the list.

Throws: Nothing.

Complexity: Constant.

11.
```
reference back();
```

Effects: Returns a reference to the last element of the list.

Throws: Nothing.

Complexity: Constant.

12.
```
const_reference back() const;
```

Effects: Returns a const_reference to the last element of the list.

Throws: Nothing.

Complexity: Constant.

13.
```
iterator begin();
```

Effects: Returns an iterator to the first element contained in the list.

Throws: Nothing.

Complexity: Constant.

14.
```
const_iterator begin() const;
```

Effects: Returns a const_iterator to the first element contained in the list.

Throws: Nothing.

Complexity: Constant.

15.
```
const_iterator cbegin() const;
```

Effects: Returns a const_iterator to the first element contained in the list.

Throws: Nothing.

Complexity: Constant.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

16.
```
iterator end();
```

Effects: Returns an iterator to the end of the list.

Throws: Nothing.

Complexity: Constant.

17.
```
const_iterator end() const;
```

Effects: Returns a const_iterator to the end of the list.

Throws: Nothing.

Complexity: Constant.

18.
```
const_iterator cend() const;
```

Effects: Returns a constant iterator to the end of the list.

Throws: Nothing.

Complexity: Constant.

19.
```
reverse_iterator rbegin();
```

Effects: Returns a reverse_iterator pointing to the beginning of the reversed list.

Throws: Nothing.

Complexity: Constant.

20.
```
const_reverse_iterator rbegin() const;
```

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed list.

Throws: Nothing.

Complexity: Constant.

21.
```
const_reverse_iterator crbegin() const;
```

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed list.

Throws: Nothing.

Complexity: Constant.

22.
```
reverse_iterator rend();
```

Effects: Returns a reverse_iterator pointing to the end of the reversed list.

Throws: Nothing.

Complexity: Constant.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

23.
```
const_reverse_iterator rend() const;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed list.

Throws: Nothing.

Complexity: Constant.

24.
```
const_reverse_iterator crend() const;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed list.

Throws: Nothing.

Complexity: Constant.

25.
```
size_type size() const;
```

Effects: Returns the number of the elements contained in the list.

Throws: Nothing.

Complexity: Linear to the number of elements contained in the list. if constant-time size option is disabled. Constant time otherwise.

Note: Does not affect the validity of iterators and references.

26.
```
bool empty() const;
```

Effects: Returns true if the list contains no elements.

Throws: Nothing.

Complexity: Constant.

Note: Does not affect the validity of iterators and references.

27.
```
void swap(list & other);
```

Effects: Swaps the elements of x and *this.

Throws: Nothing.

Complexity: Constant.

Note: Does not affect the validity of iterators and references.

28.
```
void shift_backwards(size_type n = 1);
```

Effects: Moves backwards all the elements, so that the first element becomes the second, the second becomes the third... the last element becomes the first one.

Throws: Nothing.

Complexity: Linear to the number of shifts.

Note: Does not affect the validity of iterators and references.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

29.
```
void shift_forward(size_type n = 1);
```

Effects: Moves forward all the elements, so that the second element becomes the first, the third becomes the second... the first element becomes the last one.

Throws: Nothing.

Complexity: Linear to the number of shifts.

Note: Does not affect the validity of iterators and references.

30.
```
iterator erase(const_iterator i);
```

Effects: Erases the element pointed by i of the list. No destructors are called.

Returns: the first element remaining beyond the removed element, or end() if no such element exists.

Throws: Nothing.

Complexity: Constant.

Note: Invalidates the iterators (but not the references) to the erased element.

31.
```
iterator erase(const_iterator b, const_iterator e);
```

Requires: b and e must be valid iterators to elements in *this.

Effects: Erases the element range pointed by b and e No destructors are called.

Returns: the first element remaining beyond the removed elements, or end() if no such element exists.

Throws: Nothing.

Complexity: Linear to the number of erased elements if it's a safe-mode or auto-unlink value, or constant-time size is enabled. Constant-time otherwise.

Note: Invalidates the iterators (but not the references) to the erased elements.

32.
```
iterator erase(const_iterator b, const_iterator e, difference_type n);
```

Requires: b and e must be valid iterators to elements in *this. n must be std::distance(b, e).

Effects: Erases the element range pointed by b and e No destructors are called.

Returns: the first element remaining beyond the removed elements, or end() if no such element exists.

Throws: Nothing.

Complexity: Linear to the number of erased elements if it's a safe-mode or auto-unlink value is enabled. Constant-time otherwise.

Note: Invalidates the iterators (but not the references) to the erased elements.

33.
```
template<typename Disposer>
  iterator erase_and_dispose(const_iterator i, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Erases the element pointed by i of the list. No destructors are called. Disposer::operator()(pointer) is called for the removed element.

Returns: the first element remaining beyond the removed element, or end() if no such element exists.

Throws: Nothing.

Complexity: Constant.

Note: Invalidates the iterators to the erased element.

34.
```
template<typename Disposer> 
  iterator erase_and_dispose(const_iterator b, const_iterator e, 
                             Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases the element range pointed by b and e No destructors are called. Disposer::operator()(pointer) is called for the removed elements.

Returns: the first element remaining beyond the removed elements, or end() if no such element exists.

Throws: Nothing.

Complexity: Linear to the number of elements erased.

Note: Invalidates the iterators to the erased elements.

35.
```
void clear();
```

Effects: Erases all the elements of the container. No destructors are called.

Throws: Nothing.

Complexity: Linear to the number of elements of the list. if it's a safe-mode or auto-unlink value_type. Constant time otherwise.

Note: Invalidates the iterators (but not the references) to the erased elements.

36.
```
template<typename Disposer> void clear_and_dispose(Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases all the elements of the container. No destructors are called. Disposer::operator()(pointer) is called for the removed elements.

Throws: Nothing.

Complexity: Linear to the number of elements of the list.

Note: Invalidates the iterators to the erased elements.

37.
```
template<typename Cloner, typename Disposer> 
  void clone_from(const list & src, Cloner cloner, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw. Cloner should yield to nodes equivalent to the original nodes.

Effects: Erases all the elements from *this calling Disposer::operator()(pointer), clones all the elements from src calling Cloner::operator()(const_reference) and inserts them on *this.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

If cloner throws, all cloned elements are unlinked and disposed calling Disposer::operator()(pointer).

Complexity: Linear to erased plus inserted elements.

Throws: If cloner throws. Basic guarantee.

38.
```
iterator insert(const_iterator p, reference value);
```

Requires: value must be an lvalue and p must be a valid iterator of *this.

Effects: Inserts the value before the position pointed by p.

Returns: An iterator to the inserted element.

Throws: Nothing.

Complexity: Constant time. No copy constructors are called.

Note: Does not affect the validity of iterators and references.

39.
```
template<typename Iterator> 
  void insert(const_iterator p, Iterator b, Iterator e);
```

Requires: Dereferencing iterator must yield an lvalue of type value_type and p must be a valid iterator of *this.

Effects: Inserts the range pointed by b and e before the position p. No copy constructors are called.

Throws: Nothing.

Complexity: Linear to the number of elements inserted.

Note: Does not affect the validity of iterators and references.

40.
```
template<typename Iterator> void assign(Iterator b, Iterator e);
```

Requires: Dereferencing iterator must yield an lvalue of type value_type.

Effects: Clears the list and inserts the range pointed by b and e. No destructors or copy constructors are called.

Throws: Nothing.

Complexity: Linear to the number of elements inserted plus linear to the elements contained in the list if it's a safe-mode or auto-unlink value. Linear to the number of elements inserted in the list otherwise.

Note: Invalidates the iterators (but not the references) to the erased elements.

41.
```
template<typename Iterator, typename Disposer> 
  void dispose_and_assign(Disposer disposer, Iterator b, Iterator e);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Requires: Dereferencing iterator must yield an lvalue of type value_type.

Effects: Clears the list and inserts the range pointed by b and e. No destructors or copy constructors are called. Disposer::operator()(pointer) is called for the removed elements.

Throws: Nothing.

Complexity: Linear to the number of elements inserted plus linear to the elements contained in the list.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Note: Invalidates the iterators (but not the references) to the erased elements.

42.
```
void splice(const_iterator p, list & x);
```

Requires: p must be a valid iterator of *this.

Effects: Transfers all the elements of list x to this list, before the the element pointed by p. No destructors or copy constructors are called.

Throws: Nothing.

Complexity: Constant.

Note: Iterators of values obtained from list x now point to elements of this list. Iterators of this list and all the references are not invalidated.

43.
```
void splice(const_iterator p, list & x, const_iterator new_ele);
```

Requires: p must be a valid iterator of *this. new_ele must point to an element contained in list x.

Effects: Transfers the value pointed by new_ele, from list x to this list, before the the element pointed by p. No destructors or copy constructors are called. If p == new_ele or p == ++new_ele, this function is a null operation.

Throws: Nothing.

Complexity: Constant.

Note: Iterators of values obtained from list x now point to elements of this list. Iterators of this list and all the references are not invalidated.

44.
```
void splice(const_iterator p, list & x, const_iterator f, const_iterator e);
```

Requires: p must be a valid iterator of *this. f and e must point to elements contained in list x.

Effects: Transfers the range pointed by f and e from list x to this list, before the the element pointed by p. No destructors or copy constructors are called.

Throws: Nothing.

Complexity: Linear to the number of elements transferred if constant-time size option is enabled. Constant-time otherwise.

Note: Iterators of values obtained from list x now point to elements of this list. Iterators of this list and all the references are not invalidated.

45.
```
void splice(const_iterator p, list & x, const_iterator f, const_iterator e,
            difference_type n);
```

Requires: p must be a valid iterator of *this. f and e must point to elements contained in list x. n == std::distance(f, e)

Effects: Transfers the range pointed by f and e from list x to this list, before the the element pointed by p. No destructors or copy constructors are called.

Throws: Nothing.

Complexity: Constant.

Note: Iterators of values obtained from list x now point to elements of this list. Iterators of this list and all the references are not invalidated.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

46.
```
void sort();
```

Effects: This function sorts the list *this according to std::less<value_type>. The sort is stable, that is, the relative order of equivalent elements is preserved.

Throws: If real_value_traits::node_traits::node constructor throws (this does not happen with predefined Boost.Intrusive hooks) or std::less<value_type> throws. Basic guarantee.

Notes: Iterators and references are not invalidated.

Complexity: The number of comparisons is approximately N log N, where N is the list's size.

47.
```
template<typename Predicate> void sort(Predicate p);
```

Requires: p must be a comparison function that induces a strict weak ordering

Effects: This function sorts the list *this according to p. The sort is stable, that is, the relative order of equivalent elements is preserved.

Throws: If real_value_traits::node_traits::node constructor throws (this does not happen with predefined Boost.Intrusive hooks) or the predicate throws. Basic guarantee.

Notes: This won't throw if list_base_hook<> or `list_member_hook` are used. Iterators and references are not invalidated.

Complexity: The number of comparisons is approximately N log N, where N is the list's size.

48.
```
void merge(list & x);
```

Effects: This function removes all of x's elements and inserts them in order into *this according to std::less<value_type>. The merge is stable; that is, if an element from *this is equivalent to one from x, then the element from *this will precede the one from x.

Throws: If std::less<value_type> throws. Basic guarantee.

Complexity: This function is linear time: it performs at most size() + x.size() - 1 comparisons.

Note: Iterators and references are not invalidated

49.
```
template<typename Predicate> void merge(list & x, Predicate p);
```

Requires: p must be a comparison function that induces a strict weak ordering and both *this and x must be sorted according to that ordering The lists x and *this must be distinct.

Effects: This function removes all of x's elements and inserts them in order into *this. The merge is stable; that is, if an element from *this is equivalent to one from x, then the element from *this will precede the one from x.

Throws: If the predicate throws. Basic guarantee.

Complexity: This function is linear time: it performs at most size() + x.size() - 1 comparisons.

Note: Iterators and references are not invalidated.

50.
```
void reverse();
```

Effects: Reverses the order of elements in the list.

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: This function is linear time.

Note: Iterators and references are not invalidated

51.
```
void remove(const_reference value);
```

Effects: Removes all the elements that compare equal to value. No destructors are called.

Throws: If std::equal_to<value_type> throws. Basic guarantee.

Complexity: Linear time. It performs exactly size() comparisons for equality.

Note: The relative order of elements that are not removed is unchanged, and iterators to elements that are not removed remain valid.

52.
```
template<typename Disposer> 
  void remove_and_dispose(const_reference value, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Removes all the elements that compare equal to value. Disposer::operator()(pointer) is called for every removed element.

Throws: If std::equal_to<value_type> throws. Basic guarantee.

Complexity: Linear time. It performs exactly size() comparisons for equality.

Note: The relative order of elements that are not removed is unchanged, and iterators to elements that are not removed remain valid.

53.
```
template<typename Pred> void remove_if(Pred pred);
```

Effects: Removes all the elements for which a specified predicate is satisfied. No destructors are called.

Throws: If pred throws. Basic guarantee.

Complexity: Linear time. It performs exactly size() calls to the predicate.

Note: The relative order of elements that are not removed is unchanged, and iterators to elements that are not removed remain valid.

54.
```
template<typename Pred, typename Disposer> 
  void remove_and_dispose_if(Pred pred, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Removes all the elements for which a specified predicate is satisfied. Disposer::operator()(pointer) is called for every removed element.

Throws: If pred throws. Basic guarantee.

Complexity: Linear time. It performs exactly size() comparisons for equality.

Note: The relative order of elements that are not removed is unchanged, and iterators to elements that are not removed remain valid.

55.
```
void unique();
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Removes adjacent duplicate elements or adjacent elements that are equal from the list. No destructors are called.

Throws: If std::equal_to<value_type throws. Basic guarantee.

Complexity: Linear time (size()-1 comparisons calls to pred()).

Note: The relative order of elements that are not removed is unchanged, and iterators to elements that are not removed remain valid.

56.
```
template<typename BinaryPredicate> void unique(BinaryPredicate pred);
```

Effects: Removes adjacent duplicate elements or adjacent elements that satisfy some binary predicate from the list. No destructors are called.

Throws: If pred throws. Basic guarantee.

Complexity: Linear time (size()-1 comparisons equality comparisons).

Note: The relative order of elements that are not removed is unchanged, and iterators to elements that are not removed remain valid.

57.
```
template<typename Disposer> void unique_and_dispose(Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Removes adjacent duplicate elements or adjacent elements that are equal from the list. Disposer::operator()(pointer) is called for every removed element.

Throws: If std::equal_to<value_type throws. Basic guarantee.

Complexity: Linear time (size()-1) comparisons equality comparisons.

Note: The relative order of elements that are not removed is unchanged, and iterators to elements that are not removed remain valid.

58.
```
template<typename BinaryPredicate, typename Disposer> 
  void unique_and_dispose(BinaryPredicate pred, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Removes adjacent duplicate elements or adjacent elements that satisfy some binary predicate from the list. Disposer::operator()(pointer) is called for every removed element.

Throws: If pred throws. Basic guarantee.

Complexity: Linear time (size()-1) comparisons equality comparisons.

Note: The relative order of elements that are not removed is unchanged, and iterators to elements that are not removed remain valid.

59.
```
iterator iterator_to(reference value);
```

Requires: value must be a reference to a value inserted in a list.

Effects: This function returns a const_iterator pointing to the element

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Constant time.

Note: Iterators and references are not invalidated.

60.
```
const_iterator iterator_to(const_reference value) const;
```

Requires: value must be a const reference to a value inserted in a list.

Effects: This function returns an iterator pointing to the element.

Throws: Nothing.

Complexity: Constant time.

Note: Iterators and references are not invalidated.

`list` public static functions

1.
```
static list & container_from_end_iterator(iterator end_iterator);
```

Precondition: end_iterator must be a valid end iterator of list.

Effects: Returns a const reference to the list associated to the end iterator

Throws: Nothing.

Complexity: Constant.

2.
```
static const list & container_from_end_iterator(const_iterator end_iterator);
```

Precondition: end_iterator must be a valid end const_iterator of list.

Effects: Returns a const reference to the list associated to the end iterator

Throws: Nothing.

Complexity: Constant.

3.
```
static iterator s_iterator_to(reference value);
```

Requires: value must be a reference to a value inserted in a list.

Effects: This function returns a const_iterator pointing to the element

Throws: Nothing.

Complexity: Constant time.

Note: Iterators and references are not invalidated. This static function is available only if the *value traits* is stateless.

4.
```
static const_iterator s_iterator_to(const_reference value);
```

Requires: value must be a const reference to a value inserted in a list.

Effects: This function returns an iterator pointing to the element.

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Constant time.

Note: Iterators and references are not invalidated. This static function is available only if the *value traits* is stateless.

Struct template make_list

boost::intrusive::make_list

Synopsis

```
// In header: <boost/intrusive/list.hpp>

template<typename T, class... Options>
struct make_list {
  // types
  typedef implementation_defined type;
};
```

Description

Helper metafunction to define a `list` that yields to the same type when the same options (either explicitly or implicitly) are used.

Header <boost/intrusive/list_hook.hpp>

```
namespace boost {
  namespace intrusive {
    template<class... Options> struct make_list_base_hook;

    template<class... Options> class list_base_hook;

    template<class... Options> struct make_list_member_hook;

    template<class... Options> class list_member_hook;
  }
}
```

Struct template make_list_base_hook

boost::intrusive::make_list_base_hook

Synopsis

```
// In header: <boost/intrusive/list_hook.hpp>

template<class... Options>
struct make_list_base_hook {
  // types
  typedef implementation_defined type;
};
```

Description

Helper metafunction to define a `list_base_hook that` yields to the same type when the same options (either explicitly or implicitly) are used.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Class template list_base_hook

boost::intrusive::list_base_hook

Synopsis

```
// In header: <boost/intrusive/list_hook.hpp>

template<class... Options> 
class list_base_hook : public make_list_base_hook::type< O1, O2, O3 > {
public:
  // construct/copy/destruct
  list_base_hook();
  list_base_hook(const list_base_hook &);
  list_base_hook& operator=(const list_base_hook &);
  ~list_base_hook();

  // public member functions
  void swap_nodes(list_base_hook &);
  bool is_linked() const;
  void unlink();
};
```

Description

Derive a class from this hook in order to store objects of that class in an list.

The hook admits the following options: `tag<>`, `void_pointer<>` and `link_mode<>`.

`tag<>` defines a tag to identify the node. The same tag value can be used in different classes, but if a class is derived from more than one `list_base_hook`, then each `list_base_hook` needs its unique tag.

`link_mode<>` will specify the linking mode of the hook (`normal_link`, `auto_unlink` or `safe_link`).

`void_pointer<>` is the pointer type that will be used internally in the hook and the the container configured to use this hook.

`list_base_hook` public construct/copy/destruct

1.
```
list_base_hook();
```

   Effects: If `link_mode` is `auto_unlink` or `safe_link` initializes the node to an unlinked state.

   Throws: Nothing.

2.
```
list_base_hook(const list_base_hook &);
```

   Effects: If `link_mode` is `auto_unlink` or `safe_link` initializes the node to an unlinked state. The argument is ignored.

   Throws: Nothing.

   Rationale: Providing a copy-constructor makes classes using the hook STL-compliant without forcing the user to do some additional work. `swap` can be used to emulate move-semantics.

3.
```
list_base_hook& operator=(const list_base_hook &);
```

   Effects: Empty function. The argument is ignored.

   Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Rationale: Providing an assignment operator makes classes using the hook STL-compliant without forcing the user to do some additional work. `swap` can be used to emulate move-semantics.

4.
```
~list_base_hook();
```

Effects: If `link_mode` is `normal_link`, the destructor does nothing (ie. no code is generated). If `link_mode` is `safe_link` and the object is stored in an list an assertion is raised. If `link_mode` is `auto_unlink` and `is_linked()` is true, the node is unlinked.

Throws: Nothing.

`list_base_hook` public member functions

1.
```
void swap_nodes(list_base_hook & other);
```

Effects: Swapping two nodes swaps the position of the elements related to those nodes in one or two containers. That is, if the node this is part of the element e1, the node x is part of the element e2 and both elements are included in the containers s1 and s2, then after the swap-operation e1 is in s2 at the position of e2 and e2 is in s1 at the position of e1. If one element is not in a container, then after the swap-operation the other element is not in a container. Iterators to e1 and e2 related to those nodes are invalidated.

Complexity: Constant

Throws: Nothing.

2.
```
bool is_linked() const;
```

Precondition: `link_mode` must be `safe_link` or `auto_unlink`.

Returns: true, if the node belongs to a container, false otherwise. This function can be used to test whether `list::iterator_to` will return a valid iterator.

Complexity: Constant

3.
```
void unlink();
```

Effects: Removes the node if it's inserted in a container. This function is only allowed if `link_mode` is `auto_unlink`.

Throws: Nothing.

Struct template make_list_member_hook

boost::intrusive::make_list_member_hook

Synopsis

```
// In header: <boost/intrusive/list_hook.hpp>

template<class... Options>
struct make_list_member_hook {
  // types
  typedef implementation_defined type;
};
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Description

Helper metafunction to define a `list_member_hook` `that` yields to the same type when the same options (either explicitly or implicitly) are used.

Class template list_member_hook

boost::intrusive::list_member_hook

Synopsis

```
// In header: <boost/intrusive/list_hook.hpp>

template<class... Options>
class list_member_hook : public make_list_member_hook::type< O1, O2, O3 > {
public:
  // construct/copy/destruct
  list_member_hook();
  list_member_hook(const list_member_hook &);
  list_member_hook& operator=(const list_member_hook &);
  ~list_member_hook();

  // public member functions
  void swap_nodes(list_member_hook &);
  bool is_linked() const;
  void unlink();
};
```

Description

Store this hook in a class to be inserted in an list.

The hook admits the following options: `void_pointer<>` and `link_mode<>`.

`link_mode<>` will specify the linking mode of the hook (`normal_link`, `auto_unlink` or `safe_link`).

`void_pointer<>` is the pointer type that will be used internally in the hook and the the container configured to use this hook.

`list_member_hook` public construct/copy/destruct

1.
```
list_member_hook();
```

   Effects: If `link_mode` is `auto_unlink` or `safe_link` initializes the node to an unlinked state.

   Throws: Nothing.

2.
```
list_member_hook(const list_member_hook &);
```

   Effects: If `link_mode` is `auto_unlink` or `safe_link` initializes the node to an unlinked state. The argument is ignored.

   Throws: Nothing.

   Rationale: Providing a copy-constructor makes classes using the hook STL-compliant without forcing the user to do some additional work. `swap` can be used to emulate move-semantics.

3.
```
list_member_hook& operator=(const list_member_hook &);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Empty function. The argument is ignored.

Throws: Nothing.

Rationale: Providing an assignment operator makes classes using the hook STL-compliant without forcing the user to do some additional work. `swap` can be used to emulate move-semantics.

4.
```
~list_member_hook();
```

Effects: If `link_mode` is `normal_link`, the destructor does nothing (ie. no code is generated). If `link_mode` is `safe_link` and the object is stored in an list an assertion is raised. If `link_mode` is `auto_unlink` and `is_linked()` is true, the node is unlinked.

Throws: Nothing.

`list_member_hook` public member functions

1.
```
void swap_nodes(list_member_hook & other);
```

Effects: Swapping two nodes swaps the position of the elements related to those nodes in one or two containers. That is, if the node this is part of the element e1, the node x is part of the element e2 and both elements are included in the containers s1 and s2, then after the swap-operation e1 is in s2 at the position of e2 and e2 is in s1 at the position of e1. If one element is not in a container, then after the swap-operation the other element is not in a container. Iterators to e1 and e2 related to those nodes are invalidated.

Complexity: Constant

Throws: Nothing.

2.
```
bool is_linked() const;
```

Precondition: `link_mode` must be `safe_link` or `auto_unlink`.

Returns: true, if the node belongs to a container, false otherwise. This function can be used to test whether `list::iterator_to` will return a valid iterator.

Complexity: Constant

3.
```
void unlink();
```

Effects: Removes the node if it's inserted in a container. This function is only allowed if `link_mode` is `auto_unlink`.

Throws: Nothing.

Header <boost/intrusive/member_value_traits.hpp>

```
namespace boost {
  namespace intrusive {
    template<typename T, typename NodeTraits,
             typename NodeTraits::node T::* PtrToMember,
             link_mode_type LinkMode = safe_link>
      struct member_value_traits;
  }
}
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Struct template member_value_traits

boost::intrusive::member_value_traits

Synopsis

```
// In header: <boost/intrusive/member_value_traits.hpp>

template<typename T, typename NodeTraits,
         typename NodeTraits::node T::* PtrToMember,
         link_mode_type LinkMode = safe_link>
struct member_value_traits {
  // types
  typedef NodeTraits                                                   node_traits;
  typedef T                                                            value_type;
  typedef node_traits::node                                            node;
  typedef node_traits::node_ptr                                        node_ptr;
  typedef node_traits::const_node_ptr                                  const_node_ptr;
  typedef pointer_traits< node_ptr >::template rebind_pointer< T >::type       pointer;
  typedef pointer_traits< node_ptr >::template rebind_pointer< const T >::type const_pointer;
  typedef value_type &                                                 reference;
  typedef const value_type &                                           const_reference;

  // public static functions
  static node_ptr to_node_ptr(reference);
  static const_node_ptr to_node_ptr(const_reference);
  static pointer to_value_ptr(const node_ptr &);
  static const_pointer to_value_ptr(const const_node_ptr &);

  // public data members
  static const link_mode_type link_mode;
};
```

Description

This value traits template is used to create value traits from user defined node traits where value_traits::value_type will store a node_traits::node

`member_value_traits` public static functions

1.
```
static node_ptr to_node_ptr(reference value);
```

2.
```
static const_node_ptr to_node_ptr(const_reference value);
```

3.
```
static pointer to_value_ptr(const node_ptr & n);
```

4.
```
static const_pointer to_value_ptr(const const_node_ptr & n);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Header <boost/intrusive/options.hpp>

```
namespace boost {
  namespace intrusive {
    template<bool Enabled> struct constant_time_size;
    template<typename SizeType> struct size_type;
    template<typename Compare> struct compare;
    template<bool Enabled> struct floating_point;
    template<typename Equal> struct equal;
    template<typename Priority> struct priority;
    template<typename Hash> struct hash;
    template<typename ValueTraits> struct value_traits;
    template<typename Parent, typename MemberHook,
             MemberHook Parent::* PtrToMember>
      struct member_hook;
    template<typename Functor> struct function_hook;
    template<typename BaseHook> struct base_hook;
    template<typename VoidPointer> struct void_pointer;
    template<typename Tag> struct tag;
    template<link_mode_type LinkType> struct link_mode;
    template<bool Enabled> struct optimize_size;
    template<bool Enabled> struct linear;
    template<bool Enabled> struct cache_last;
    template<typename BucketTraits> struct bucket_traits;
    template<bool Enabled> struct store_hash;
    template<bool Enabled> struct optimize_multikey;
    template<bool Enabled> struct power_2_buckets;
    template<bool Enabled> struct cache_begin;
    template<bool Enabled> struct compare_hash;
    template<bool Enabled> struct incremental;
  }
}
```

Struct template constant_time_size

boost::intrusive::constant_time_size

Synopsis

```
// In header: <boost/intrusive/options.hpp>

template<bool Enabled>
struct constant_time_size {
};
```

Description

This option setter specifies if the intrusive container stores its size as a member to obtain constant-time size() member.

Struct template size_type

boost::intrusive::size_type

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/intrusive/options.hpp>

template<typename SizeType>
struct size_type {
};
```

Description

This option setter specifies the type that the container will use to store its size.

Struct template compare

boost::intrusive::compare

Synopsis

```
// In header: <boost/intrusive/options.hpp>

template<typename Compare>
struct compare {
};
```

Description

This option setter specifies the strict weak ordering comparison functor for the value type

Struct template floating_point

boost::intrusive::floating_point

Synopsis

```
// In header: <boost/intrusive/options.hpp>

template<bool Enabled>
struct floating_point {
};
```

Description

This option setter for scapegoat containers specifies if the intrusive scapegoat container should use a non-variable alpha value that does not need floating-point operations.

If activated, the fixed alpha value is 1/sqrt(2). This option also saves some space in the container since the alpha value and some additional data does not need to be stored in the container.

If the user only needs an alpha value near 1/sqrt(2), this option also improves performance since avoids logarithm and division operations when rebalancing the tree.

Struct template equal

boost::intrusive::equal

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/intrusive/options.hpp>

template<typename Equal>
struct equal {
};
```

Description

This option setter specifies the equality functor for the value type

Struct template priority

boost::intrusive::priority

Synopsis

```
// In header: <boost/intrusive/options.hpp>

template<typename Priority>
struct priority {
};
```

Description

This option setter specifies the equality functor for the value type

Struct template hash

boost::intrusive::hash

Synopsis

```
// In header: <boost/intrusive/options.hpp>

template<typename Hash>
struct hash {
};
```

Description

This option setter specifies the hash functor for the value type

Struct template value_traits

boost::intrusive::value_traits

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/intrusive/options.hpp>

template<typename ValueTraits> 
struct value_traits {
};
```

Description

This option setter specifies the relationship between the type to be managed by the container (the value type) and the node to be used in the node algorithms. It also specifies the linking policy.

Struct template member_hook

boost::intrusive::member_hook

Synopsis

```
// In header: <boost/intrusive/options.hpp>

template<typename Parent, typename MemberHook, 
         MemberHook Parent::* PtrToMember> 
struct member_hook {
};
```

Description

This option setter specifies the member hook the container must use.

Struct template function_hook

boost::intrusive::function_hook

Synopsis

```
// In header: <boost/intrusive/options.hpp>

template<typename Functor> 
struct function_hook {
};
```

Description

This option setter specifies the function object that will be used to convert between values to be inserted in a container and the hook to be used for that purpose.

Struct template base_hook

boost::intrusive::base_hook

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/intrusive/options.hpp>

template<typename BaseHook>
struct base_hook {
};
```

Description

This option setter specifies that the container must use the specified base hook

Struct template void_pointer

boost::intrusive::void_pointer

Synopsis

```
// In header: <boost/intrusive/options.hpp>

template<typename VoidPointer>
struct void_pointer {
};
```

Description

This option setter specifies the type of a void pointer. This will instruct the hook to use this type of pointer instead of the default one

Struct template tag

boost::intrusive::tag

Synopsis

```
// In header: <boost/intrusive/options.hpp>

template<typename Tag>
struct tag {
};
```

Description

This option setter specifies the type of the tag of a base hook. A type cannot have two base hooks of the same type, so a tag can be used to differentiate two base hooks with otherwise same type

Struct template link_mode

boost::intrusive::link_mode

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/intrusive/options.hpp>

template<link_mode_type LinkType>
struct link_mode {
};
```

Description

This option setter specifies the link mode (normal_link, safe_link or auto_unlink)

Struct template optimize_size

boost::intrusive::optimize_size

Synopsis

```
// In header: <boost/intrusive/options.hpp>

template<bool Enabled>
struct optimize_size {
};
```

Description

This option setter specifies if the hook should be optimized for size instead of for speed.

Struct template linear

boost::intrusive::linear

Synopsis

```
// In header: <boost/intrusive/options.hpp>

template<bool Enabled>
struct linear {
};
```

Description

This option setter specifies if the list container should use a linear implementation instead of a circular one.

Struct template cache_last

boost::intrusive::cache_last

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/intrusive/options.hpp>

template<bool Enabled> 
struct cache_last {
};
```

Description

This option setter specifies if the list container should use a linear implementation instead of a circular one.

Struct template bucket_traits

boost::intrusive::bucket_traits

Synopsis

```
// In header: <boost/intrusive/options.hpp>

template<typename BucketTraits> 
struct bucket_traits {
};
```

Description

This option setter specifies the bucket traits class for unordered associative containers. When this option is specified, instead of using the default bucket traits, a user defined holder will be defined

Struct template store_hash

boost::intrusive::store_hash

Synopsis

```
// In header: <boost/intrusive/options.hpp>

template<bool Enabled> 
struct store_hash {
};
```

Description

This option setter specifies if the unordered hook should offer room to store the hash value. Storing the hash in the hook will speed up rehashing processes in applications where rehashing is frequent, rehashing might throw or the value is heavy to hash.

Struct template optimize_multikey

boost::intrusive::optimize_multikey

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/intrusive/options.hpp>

template<bool Enabled>
struct optimize_multikey {
};
```

Description

This option setter specifies if the unordered hook should offer room to store another link to another node with the same key. Storing this link will speed up lookups and insertions on unordered_multiset containers with a great number of elements with the same key.

Struct template power_2_buckets

boost::intrusive::power_2_buckets

Synopsis

```
// In header: <boost/intrusive/options.hpp>

template<bool Enabled>
struct power_2_buckets {
};
```

Description

This option setter specifies if the bucket array will be always power of two. This allows using masks instead of the default modulo operation to determine the bucket number from the hash value, leading to better performance. In debug mode, if power of two buckets mode is activated, the bucket length will be checked to through assertions to assure the bucket length is power of two.

Struct template cache_begin

boost::intrusive::cache_begin

Synopsis

```
// In header: <boost/intrusive/options.hpp>

template<bool Enabled>
struct cache_begin {
};
```

Description

This option setter specifies if the container will cache a pointer to the first non-empty bucket so that begin() is always constant-time. This is specially helpful when we can have containers with a few elements but with big bucket arrays (that is, hashtables with low load factors).

Struct template compare_hash

boost::intrusive::compare_hash

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/intrusive/options.hpp>

template<bool Enabled>
struct compare_hash {
};
```

Description

This option setter specifies if the container will compare the hash value before comparing objects. This option can't be specified if store_hash<> is not true. This is specially helpful when we have containers with a high load factor. and the comparison function is much more expensive that comparing already stored hash values.

Struct template incremental

boost::intrusive::incremental

Synopsis

```
// In header: <boost/intrusive/options.hpp>

template<bool Enabled>
struct incremental {
};
```

Description

This option setter specifies if the hash container will use incremental hashing. With incremental hashing the cost of hash table expansion is spread out across each hash table insertion operation, as opposed to be incurred all at once. Therefore linear hashing is well suited for interactive applications or real-time appplications where the worst-case insertion time of non-incremental hash containers (rehashing the whole bucket array) is not admisible.

Header <boost/intrusive/parent_from_member.hpp>

```
namespace boost {
  namespace intrusive {
    template<typename Parent, typename Member>
      Parent * get_parent_from_member(Member *, const Member Parent::*);
    template<typename Parent, typename Member>
      const Parent *
      get_parent_from_member(const Member *, const Member Parent::*);
  }
}
```

Function template get_parent_from_member

boost::intrusive::get_parent_from_member

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/intrusive/parent_from_member.hpp>


template<typename Parent, typename Member>
  Parent * get_parent_from_member(Member * member,
                                  const Member Parent::* ptr_to_member);
```

Description

Given a pointer to a member and its corresponding pointer to data member, this function returns the pointer of the parent containing that member. Note: this function does not work with pointer to members that rely on virtual inheritance.

Function template get_parent_from_member

boost::intrusive::get_parent_from_member

Synopsis

```
// In header: <boost/intrusive/parent_from_member.hpp>


template<typename Parent, typename Member>
  const Parent *
  get_parent_from_member(const Member * member,
                         const Member Parent::* ptr_to_member);
```

Description

Given a const pointer to a member and its corresponding const pointer to data member, this function returns the const pointer of the parent containing that member. Note: this function does not work with pointer to members that rely on virtual inheritance.

Header <boost/intrusive/pointer_plus_bits.hpp>

```
namespace boost {
  namespace intrusive {
    template<typename VoidPointer, std::size_t Alignment>
      struct max_pointer_plus_bits;

    template<std::size_t Alignment>
      struct max_pointer_plus_bits<void *, Alignment>;

    template<typename Pointer, std::size_t NumBits> struct pointer_plus_bits;

    template<typename T, std::size_t NumBits>
      struct pointer_plus_bits<T *, NumBits>;
  }
}
```

Struct template max_pointer_plus_bits

boost::intrusive::max_pointer_plus_bits

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/intrusive/pointer_plus_bits.hpp>

template<typename VoidPointer, std::size_t Alignment>
struct max_pointer_plus_bits {

  // public data members
  static const std::size_t value;
};
```

Description

This trait class is used to know if a pointer can embed extra bits of information if it's going to be used to point to objects with an alignment of "Alignment" bytes.

Struct template max_pointer_plus_bits<void *, Alignment>

boost::intrusive::max_pointer_plus_bits<void *, Alignment>

Synopsis

```
// In header: <boost/intrusive/pointer_plus_bits.hpp>

template<std::size_t Alignment>
struct max_pointer_plus_bits<void *, Alignment> {

  // public data members
  static const std::size_t value;
};
```

Description

This is a specialization for raw pointers. Raw pointers can embed extra bits in the lower bits if the alignment is multiple of 2pow(NumBits).

Struct template pointer_plus_bits

boost::intrusive::pointer_plus_bits

Synopsis

```
// In header: <boost/intrusive/pointer_plus_bits.hpp>

template<typename Pointer, std::size_t NumBits>
struct pointer_plus_bits {
};
```

Description

This is class that is supposed to have static methods to embed extra bits of information in a pointer. This is a declaration and there is no default implementation, because operations to embed the bits change with every pointer type.

An implementation that detects that a pointer type whose has_pointer_plus_bits<>::value is non-zero can make use of these operations to embed the bits in the pointer.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Struct template pointer_plus_bits<T *, NumBits>

boost::intrusive::pointer_plus_bits<T *, NumBits>

Synopsis

```
// In header: <boost/intrusive/pointer_plus_bits.hpp>

template<typename T, std::size_t NumBits> 
struct pointer_plus_bits<T *, NumBits> {
  // types
  typedef T * pointer;

  // public static functions
  static pointer get_pointer(pointer);
  static void set_pointer(pointer &, pointer);
  static std::size_t get_bits(pointer);
  static void set_bits(pointer &, std::size_t);

  // public data members
  static const std::size_t Mask;
};
```

Description

This is the specialization to embed extra bits of information in a raw pointer. The extra bits are stored in the lower bits of the pointer.

`pointer_plus_bits` public static functions

1.
```
static pointer get_pointer(pointer n);
```

2.
```
static void set_pointer(pointer & n, pointer p);
```

3.
```
static std::size_t get_bits(pointer n);
```

4.
```
static void set_bits(pointer & n, std::size_t c);
```

Header <boost/intrusive/pointer_traits.hpp>

```
namespace boost {
  namespace intrusive {
    template<typename Ptr> struct pointer_traits;

    template<typename T> struct pointer_traits<T *>;
  }
}
```

Struct template pointer_traits

boost::intrusive::pointer_traits

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/intrusive/pointer_traits.hpp>

template<typename Ptr>
struct pointer_traits {
  // types
  typedef Ptr              pointer;
  typedef unspecified_type element_type;
  typedef unspecified_type difference_type;
  typedef unspecified      rebind;
  typedef element_type &   reference;

  // public static functions
  static pointer pointer_to(reference);
  template<typename UPtr> static pointer static_cast_from(const UPtr &);
  template<typename UPtr> static pointer const_cast_from(const UPtr &);
  template<typename UPtr> static pointer dynamic_cast_from(const UPtr &);
};
```

Description

pointer_traits is the implementation of C++11 std::pointer_traits class with some extensions like castings.

pointer_traits supplies a uniform interface to certain attributes of pointer-like types.

`pointer_traits` public types

1. typedef Ptr pointer;

   The pointer type queried by this pointer_traits instantiation

2. typedef unspecified_type element_type;

   Ptr::element_type if such a type exists; otherwise, T if Ptr is a class template instantiation of the form SomePointer<T, Args>, where Args is zero or more type arguments ; otherwise , the specialization is ill-formed.

3. typedef unspecified_type difference_type;

   Ptr::difference_type if such a type exists; otherwise, std::ptrdiff_t.

4. typedef unspecified rebind;

   Ptr::rebind<U> if such a type exists; otherwise, SomePointer<U, Args> if Ptr is a class template instantiation of the form SomePointer<T, Args>, where Args is zero or more type arguments ; otherwise, the instantiation of rebind is ill-formed.

   For portable code for C++03 and C++11,
   <preformatted>typename rebind_pointer<U>::type</preformatted>
   shall be used instead of rebind<U> to obtain a pointer to U.

5. typedef element_type & reference;

   Ptr::rebind<U> if such a type exists; otherwise, SomePointer<U, Args> if Ptr is a class template instantiation of the form SomePointer<T, Args>, where Args is zero or more type arguments ; otherwise, the instantiation of rebind is ill-formed.

`pointer_traits` public static functions

1.
   ```
   static pointer pointer_to(reference r);
   ```

   Remark: If element_type is (possibly cv-qualified) void, r type is unspecified; otherwise, it is element_type &.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Returns: A dereferenceable pointer to r obtained by calling Ptr::pointer_to(r). Non-standard extension: If such function does not exist, returns pointer(addressof(r));

2.
```
template<typename UPtr> static pointer static_cast_from(const UPtr & uptr);
```

Remark: Non-standard extension.

Returns: A dereferenceable pointer to r obtained by calling Ptr::static_cast_from(r). If such function does not exist, returns pointer_to(static_cast<element_type&>(*uptr))

3.
```
template<typename UPtr> static pointer const_cast_from(const UPtr & uptr);
```

Remark: Non-standard extension.

Returns: A dereferenceable pointer to r obtained by calling Ptr::const_cast_from(r). If such function does not exist, returns pointer_to(const_cast<element_type&>(*uptr))

4.
```
template<typename UPtr> static pointer dynamic_cast_from(const UPtr & uptr);
```

Remark: Non-standard extension.

Returns: A dereferenceable pointer to r obtained by calling Ptr::dynamic_cast_from(r). If such function does not exist, returns pointer_to(*dynamic_cast<element_type>*(&*uptr))

Struct template pointer_traits<T *>

boost::intrusive::pointer_traits<T *>

Synopsis

```
// In header: <boost/intrusive/pointer_traits.hpp>

template<typename T>
struct pointer_traits<T *> {
  // types
  typedef T              element_type;
  typedef T *            pointer;
  typedef std::ptrdiff_t difference_type;
  typedef T &            reference;
  typedef U *            rebind;

  // member classes/structs/unions
  template<typename U>
  struct rebind_pointer {
    // types
    typedef U * type;
  };

  // public static functions
  static pointer pointer_to(reference);
  template<typename U> static pointer static_cast_from(U *);
  template<typename U> static pointer const_cast_from(U *);
  template<typename U> static pointer dynamic_cast_from(U *);
};
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Description

Specialization of pointer_traits for raw pointers

`pointer_traits` public types

1. typedef U * rebind;

   typedef for
   <preformatted>U *</preformatted>

   For portable code for C++03 and C++11,
   <preformatted>typename rebind_pointer<U>::type</preformatted>
   shall be used instead of rebind<U> to obtain a pointer to U.

`pointer_traits` public static functions

1.
   ```
   static pointer pointer_to(reference r);
   ```

   Returns: addressof(r)

2.
   ```
   template<typename U> static pointer static_cast_from(U * uptr);
   ```

   Returns: static_cast<pointer>(uptr)

3.
   ```
   template<typename U> static pointer const_cast_from(U * uptr);
   ```

   Returns: const_cast<pointer>(uptr)

4.
   ```
   template<typename U> static pointer dynamic_cast_from(U * uptr);
   ```

   Returns: dynamic_cast<pointer>(uptr)

Struct template rebind_pointer

boost::intrusive::pointer_traits<T *>::rebind_pointer

Synopsis

```
// In header: <boost/intrusive/pointer_traits.hpp>


template<typename U>
struct rebind_pointer {
  // types
  typedef U * type;
};
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Header <boost/intrusive/priority_compare.hpp>

```
namespace boost {
  namespace intrusive {
    template<typename T> struct priority_compare;
  }
}
```

Struct template priority_compare

boost::intrusive::priority_compare

Synopsis

```
// In header: <boost/intrusive/priority_compare.hpp>

template<typename T>
struct priority_compare : public std::binary_function< T, T, bool > {

  // public member functions
  bool operator()(const T &, const T &) const;
};
```

Description

`priority_compare` public member functions

1.
```
bool operator()(const T & val, const T & val2) const;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Header <boost/intrusive/rbtree.hpp>

```
namespace boost {
  namespace intrusive {
    template<typename T, class... Options> class rbtree;

    template<typename T, class... Options> struct make_rbtree;
    template<typename T, class... Options> 
      bool operator<(const rbtree< T, Options...> & x, 
                     const rbtree< T, Options...> & y);
    template<typename T, class... Options> 
      bool operator==(const rbtree< T, Options...> & x, 
                      const rbtree< T, Options...> & y);
    template<typename T, class... Options> 
      bool operator!=(const rbtree< T, Options...> & x, 
                      const rbtree< T, Options...> & y);
    template<typename T, class... Options> 
      bool operator>(const rbtree< T, Options...> & x, 
                     const rbtree< T, Options...> & y);
    template<typename T, class... Options> 
      bool operator<=(const rbtree< T, Options...> & x, 
                      const rbtree< T, Options...> & y);
    template<typename T, class... Options> 
      bool operator>=(const rbtree< T, Options...> & x, 
                      const rbtree< T, Options...> & y);
    template<typename T, class... Options> 
      void swap(rbtree< T, Options...> & x, rbtree< T, Options...> & y);
  }
}
```

Class template rbtree

boost::intrusive::rbtree

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/intrusive/rbtree.hpp>

template<typename T, class... Options> 
class rbtree {
public:
  // types
  typedef Config::value_traits                                 value_traits;           
  typedef real_value_traits::pointer                           pointer;                
  typedef real_value_traits::const_pointer                     const_pointer;          
  typedef pointer_traits< pointer >::element_type              value_type;             
  typedef value_type                                           key_type;               
  typedef pointer_traits< pointer >::reference                 reference;              
  typedef pointer_traits< const_pointer >::reference           const_reference;        
  typedef pointer_traits< const_pointer >::difference_type difference_type;        
  typedef Config::size_type                                    size_type;              
  typedef Config::compare                                      value_compare;          
  typedef value_compare                                        key_compare;            
  typedef tree_iterator< rbtree, false >                       iterator;               
  typedef tree_iterator< rbtree, true >                        const_iterator;         
  typedef unspecified                                          reverse_iterator;       
  typedef unspecified                                          const_reverse_iterator; 
  typedef real_value_traits::node_traits                       node_traits;            
  typedef node_traits::node                                    node;                   
  typedef node_traits::node_ptr                                node_ptr;               
  typedef node_traits::const_node_ptr                          const_node_ptr;         
  typedef rbtree_algorithms< node_traits >                     node_algorithms;        
  typedef node_algorithms::insert_commit_data                  insert_commit_data;     

  // construct/copy/destruct
  explicit rbtree(const value_compare & = value_compare(), 
                  const value_traits & = value_traits());
  template<typename Iterator> 
    rbtree(bool, Iterator, Iterator, const value_compare & = value_compare(), 
           const value_traits & = value_traits());
  rbtree(BOOST_RV_REF(rbtree));
  rbtree& operator=(BOOST_RV_REF(rbtree));
  ~rbtree();

  // public member functions
  const real_value_traits & get_real_value_traits() const;
  real_value_traits & get_real_value_traits();
  iterator begin();
  const_iterator begin() const;
  const_iterator cbegin() const;
  iterator end();
  const_iterator end() const;
  const_iterator cend() const;
  reverse_iterator rbegin();
  const_reverse_iterator rbegin() const;
  const_reverse_iterator crbegin() const;
  reverse_iterator rend();
  const_reverse_iterator rend() const;
  const_reverse_iterator crend() const;
  value_compare value_comp() const;
  bool empty() const;
  size_type size() const;
  void swap(rbtree &);
  iterator insert_equal(reference);
  iterator insert_equal(const_iterator, reference);
  template<typename Iterator> void insert_equal(Iterator, Iterator);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
std::pair< iterator, bool > insert_unique(reference);
iterator insert_unique(const_iterator, reference);
template<typename Iterator> void insert_unique(Iterator, Iterator);
template<typename KeyType, typename KeyValueCompare> 
  std::pair< iterator, bool > 
  insert_unique_check(const KeyType &, KeyValueCompare, 
                      insert_commit_data &);
template<typename KeyType, typename KeyValueCompare> 
  std::pair< iterator, bool > 
  insert_unique_check(const_iterator, const KeyType &, KeyValueCompare, 
                      insert_commit_data &);
iterator insert_unique_commit(reference, const insert_commit_data &);
iterator insert_before(const_iterator, reference);
void push_back(reference);
void push_front(reference);
iterator erase(const_iterator);
iterator erase(const_iterator, const_iterator);
size_type erase(const_reference);
template<typename KeyType, typename KeyValueCompare> 
  size_type erase(const KeyType &, KeyValueCompare);
template<typename Disposer> 
  iterator erase_and_dispose(const_iterator, Disposer);
template<typename Disposer> 
  size_type erase_and_dispose(const_reference, Disposer);
template<typename Disposer> 
  iterator erase_and_dispose(const_iterator, const_iterator, Disposer);
template<typename KeyType, typename KeyValueCompare, typename Disposer> 
  size_type erase_and_dispose(const KeyType &, KeyValueCompare, Disposer);
void clear();
template<typename Disposer> void clear_and_dispose(Disposer);
size_type count(const_reference) const;
template<typename KeyType, typename KeyValueCompare> 
  size_type count(const KeyType &, KeyValueCompare) const;
iterator lower_bound(const_reference);
const_iterator lower_bound(const_reference) const;
template<typename KeyType, typename KeyValueCompare> 
  iterator lower_bound(const KeyType &, KeyValueCompare);
template<typename KeyType, typename KeyValueCompare> 
  const_iterator lower_bound(const KeyType &, KeyValueCompare) const;
iterator upper_bound(const_reference);
template<typename KeyType, typename KeyValueCompare> 
  iterator upper_bound(const KeyType &, KeyValueCompare);
const_iterator upper_bound(const_reference) const;
template<typename KeyType, typename KeyValueCompare> 
  const_iterator upper_bound(const KeyType &, KeyValueCompare) const;
iterator find(const_reference);
template<typename KeyType, typename KeyValueCompare> 
  iterator find(const KeyType &, KeyValueCompare);
const_iterator find(const_reference) const;
template<typename KeyType, typename KeyValueCompare> 
  const_iterator find(const KeyType &, KeyValueCompare) const;
std::pair< iterator, iterator > equal_range(const_reference);
template<typename KeyType, typename KeyValueCompare> 
  std::pair< iterator, iterator > 
  equal_range(const KeyType &, KeyValueCompare);
std::pair< const_iterator, const_iterator > 
equal_range(const_reference) const;
template<typename KeyType, typename KeyValueCompare> 
  std::pair< const_iterator, const_iterator > 
  equal_range(const KeyType &, KeyValueCompare) const;
std::pair< iterator, iterator > 
bounded_range(const_reference, const_reference, bool, bool);
template<typename KeyType, typename KeyValueCompare>
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
    std::pair< iterator, iterator >
    bounded_range(const KeyType &, const KeyType &, KeyValueCompare, bool,
                  bool);
  std::pair< const_iterator, const_iterator >
  bounded_range(const_reference, const_reference, bool, bool) const;
  template<typename KeyType, typename KeyValueCompare>
    std::pair< const_iterator, const_iterator >
    bounded_range(const KeyType &, const KeyType &, KeyValueCompare, bool,
                  bool) const;
  template<typename Cloner, typename Disposer>
    void clone_from(const rbtree &, Cloner, Disposer);
  pointer unlink_leftmost_without_rebalance();
  void replace_node(iterator, reference);
  iterator iterator_to(reference);
  const_iterator iterator_to(const_reference) const;

  // public static functions
  static rbtree & container_from_end_iterator(iterator);
  static const rbtree & container_from_end_iterator(const_iterator);
  static rbtree & container_from_iterator(iterator);
  static const rbtree & container_from_iterator(const_iterator);
  static iterator s_iterator_to(reference);
  static const_iterator s_iterator_to(const_reference);
  static void init_node(reference);
  static void remove_node(reference);

  // private static functions
  static rbtree & priv_container_from_end_iterator(const const_iterator &);
  static rbtree & priv_container_from_iterator(const const_iterator &);

  // public data members
  static const bool constant_time_size;
  static const bool stateful_value_traits;
};
```

Description

The class template rbtree is an intrusive red-black tree container, that is used to construct intrusive set and multiset containers. The no-throw guarantee holds only, if the value_compare object doesn't throw.

The template parameter `T` is the type to be managed by the container. The user can specify additional options and if no options are provided default options are used.

The container supports the following options: `base_hook<>/member_hook<>/value_traits<>`, `constant_time_size<>`, `size_type<>` and `compare<>`.

`rbtree` public construct/copy/destruct

1.
```
explicit rbtree(const value_compare & cmp = value_compare(),
                const value_traits & v_traits = value_traits());
```

Effects: Constructs an empty tree.

Complexity: Constant.

Throws: If value_traits::node_traits::node constructor throws (this does not happen with predefined Boost.Intrusive hooks) or the copy constructorof the value_compare object throws. Basic guarantee.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

2.
```
template<typename Iterator>
  rbtree(bool unique, Iterator b, Iterator e,
         const value_compare & cmp = value_compare(),
         const value_traits & v_traits = value_traits());
```

Requires: Dereferencing iterator must yield an lvalue of type value_type. cmp must be a comparison function that induces a strict weak ordering.

Effects: Constructs an empty tree and inserts elements from [b, e).

Complexity: Linear in N if [b, e) is already sorted using comp and otherwise N * log N, where N is the distance between first and last.

Throws: If value_traits::node_traits::node constructor throws (this does not happen with predefined Boost.Intrusive hooks) or the copy constructor/operator() of the value_compare object throws. Basic guarantee.

3.
```
rbtree(BOOST_RV_REF(rbtree) x);
```

Effects: to-do

4.
```
rbtree& operator=(BOOST_RV_REF(rbtree) x);
```

Effects: to-do

5.
```
~rbtree();
```

Effects: Detaches all elements from this. The objects in the set are not deleted (i.e. no destructors are called), but the nodes according to the `value_traits` template parameter are reinitialized and thus can be reused.

Complexity: Linear to elements contained in *this.

Throws: Nothing.

`rbtree` public member functions

1.
```
const real_value_traits & get_real_value_traits() const;
```

2.
```
real_value_traits & get_real_value_traits();
```

3.
```
iterator begin();
```

Effects: Returns an iterator pointing to the beginning of the tree.

Complexity: Constant.

Throws: Nothing.

4.
```
const_iterator begin() const;
```

Effects: Returns a const_iterator pointing to the beginning of the tree.

Complexity: Constant.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: Nothing.

5.
```
const_iterator cbegin() const;
```

Effects: Returns a const_iterator pointing to the beginning of the tree.

Complexity: Constant.

Throws: Nothing.

6.
```
iterator end();
```

Effects: Returns an iterator pointing to the end of the tree.

Complexity: Constant.

Throws: Nothing.

7.
```
const_iterator end() const;
```

Effects: Returns a const_iterator pointing to the end of the tree.

Complexity: Constant.

Throws: Nothing.

8.
```
const_iterator cend() const;
```

Effects: Returns a const_iterator pointing to the end of the tree.

Complexity: Constant.

Throws: Nothing.

9.
```
reverse_iterator rbegin();
```

Effects: Returns a reverse_iterator pointing to the beginning of the reversed tree.

Complexity: Constant.

Throws: Nothing.

10.
```
const_reverse_iterator rbegin() const;
```

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed tree.

Complexity: Constant.

Throws: Nothing.

11.
```
const_reverse_iterator crbegin() const;
```

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed tree.

Complexity: Constant.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: Nothing.

12.
```
reverse_iterator rend();
```

Effects: Returns a reverse_iterator pointing to the end of the reversed tree.

Complexity: Constant.

Throws: Nothing.

13.
```
const_reverse_iterator rend() const;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed tree.

Complexity: Constant.

Throws: Nothing.

14.
```
const_reverse_iterator crend() const;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed tree.

Complexity: Constant.

Throws: Nothing.

15.
```
value_compare value_comp() const;
```

Effects: Returns the value_compare object used by the tree.

Complexity: Constant.

Throws: If value_compare copy-constructor throws.

16.
```
bool empty() const;
```

Effects: Returns true if the container is empty.

Complexity: Constant.

Throws: Nothing.

17.
```
size_type size() const;
```

Effects: Returns the number of elements stored in the tree.

Complexity: Linear to elements contained in *this if constant-time size option is disabled. Constant time otherwise.

Throws: Nothing.

18.
```
void swap(rbtree & other);
```

Effects: Swaps the contents of two rbtrees.

Complexity: Constant.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: If the comparison functor's swap call throws.

19.
```
iterator insert_equal(reference value);
```

Requires: value must be an lvalue

Effects: Inserts value into the tree before the upper bound.

Complexity: Average complexity for insert element is at most logarithmic.

Throws: If the internal value_compare ordering function throws. Strong guarantee.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

20.
```
iterator insert_equal(const_iterator hint, reference value);
```

Requires: value must be an lvalue, and "hint" must be a valid iterator.

Effects: Inserts x into the tree, using "hint" as a hint to where it will be inserted. If "hint" is the upper_bound the insertion takes constant time (two comparisons in the worst case)

Complexity: Logarithmic in general, but it is amortized constant time if t is inserted immediately before hint.

Throws: If the internal value_compare ordering function throws. Strong guarantee.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

21.
```
template<typename Iterator> void insert_equal(Iterator b, Iterator e);
```

Requires: Dereferencing iterator must yield an lvalue of type value_type.

Effects: Inserts a each element of a range into the tree before the upper bound of the key of each element.

Complexity: Insert range is in general O(N * log(N)), where N is the size of the range. However, it is linear in N if the range is already sorted by value_comp().

Throws: Nothing.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

22.
```
std::pair< iterator, bool > insert_unique(reference value);
```

Requires: value must be an lvalue

Effects: Inserts value into the tree if the value is not already present.

Complexity: Average complexity for insert element is at most logarithmic.

Throws: Nothing.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

23.
```
iterator insert_unique(const_iterator hint, reference value);
```

Requires: value must be an lvalue, and "hint" must be a valid iterator

Effects: Tries to insert x into the tree, using "hint" as a hint to where it will be inserted.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Logarithmic in general, but it is amortized constant time (two comparisons in the worst case) if t is inserted immediately before hint.

Throws: Nothing.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

24.
```
template<typename Iterator> void insert_unique(Iterator b, Iterator e);
```

Requires: Dereferencing iterator must yield an lvalue of type value_type.

Effects: Tries to insert each element of a range into the tree.

Complexity: Insert range is in general O(N * log(N)), where N is the size of the range. However, it is linear in N if the range is already sorted by value_comp().

Throws: Nothing.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

25.
```
template<typename KeyType, typename KeyValueCompare>
  std::pair< iterator, bool >
  insert_unique_check(const KeyType & key, KeyValueCompare key_value_comp,
                      insert_commit_data & commit_data);
```

Requires: key_value_comp must be a comparison function that induces the same strict weak ordering as value_compare. The difference is that key_value_comp compares an arbitrary key with the contained values.

Effects: Checks if a value can be inserted in the container, using a user provided key instead of the value itself.

Returns: If there is an equivalent value returns a pair containing an iterator to the already present value and false. If the value can be inserted returns true in the returned pair boolean and fills "commit_data" that is meant to be used with the "insert_commit" function.

Complexity: Average complexity is at most logarithmic.

Throws: If the key_value_comp ordering function throws. Strong guarantee.

Notes: This function is used to improve performance when constructing a value_type is expensive: if there is an equivalent value the constructed object must be discarded. Many times, the part of the node that is used to impose the order is much cheaper to construct than the value_type and this function offers the possibility to use that part to check if the insertion will be successful.

If the check is successful, the user can construct the value_type and use "insert_commit" to insert the object in constant-time. This gives a total logarithmic complexity to the insertion: check(O(log(N)) + commit(O(1)).

"commit_data" remains valid for a subsequent "insert_commit" only if no more objects are inserted or erased from the container.

26.
```
template<typename KeyType, typename KeyValueCompare>
  std::pair< iterator, bool >
  insert_unique_check(const_iterator hint, const KeyType & key,
                      KeyValueCompare key_value_comp,
                      insert_commit_data & commit_data);
```

Requires: key_value_comp must be a comparison function that induces the same strict weak ordering as value_compare. The difference is that key_value_comp compares an arbitrary key with the contained values.

Effects: Checks if a value can be inserted in the container, using a user provided key instead of the value itself, using "hint" as a hint to where it will be inserted.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Returns: If there is an equivalent value returns a pair containing an iterator to the already present value and false. If the value can be inserted returns true in the returned pair boolean and fills "commit_data" that is meant to be used with the "insert_commit" function.

Complexity: Logarithmic in general, but it's amortized constant time if t is inserted immediately before hint.

Throws: If the key_value_comp ordering function throws. Strong guarantee.

Notes: This function is used to improve performance when constructing a value_type is expensive: if there is an equivalent value the constructed object must be discarded. Many times, the part of the constructing that is used to impose the order is much cheaper to construct than the value_type and this function offers the possibility to use that key to check if the insertion will be successful.

If the check is successful, the user can construct the value_type and use "insert_commit" to insert the object in constant-time. This can give a total constant-time complexity to the insertion: check(O(1)) + commit(O(1)).

"commit_data" remains valid for a subsequent "insert_commit" only if no more objects are inserted or erased from the container.

27.
```
iterator insert_unique_commit(reference value,
                              const insert_commit_data & commit_data);
```

Requires: value must be an lvalue of type value_type. commit_data must have been obtained from a previous call to "insert_check". No objects should have been inserted or erased from the container between the "insert_check" that filled "commit_data" and the call to "insert_commit".

Effects: Inserts the value in the `avl_set` using the information obtained from the "commit_data" that a previous "insert_check" filled.

Returns: An iterator to the newly inserted object.

Complexity: Constant time.

Throws: Nothing.

Notes: This function has only sense if a "insert_check" has been previously executed to fill "commit_data". No value should be inserted or erased between the "insert_check" and "insert_commit" calls.

28.
```
iterator insert_before(const_iterator pos, reference value);
```

Requires: value must be an lvalue, "pos" must be a valid iterator (or end) and must be the succesor of value once inserted according to the predicate

Effects: Inserts x into the tree before "pos".

Complexity: Constant time.

Throws: Nothing.

Note: This function does not check preconditions so if "pos" is not the successor of "value" tree ordering invariant will be broken. This is a low-level function to be used only for performance reasons by advanced users.

29.
```
void push_back(reference value);
```

Requires: value must be an lvalue, and it must be no less than the greatest inserted key

Effects: Inserts x into the tree in the last position.

Complexity: Constant time.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: Nothing.

Note: This function does not check preconditions so if value is less than the greatest inserted key tree ordering invariant will be broken. This function is slightly more efficient than using "insert_before". This is a low-level function to be used only for performance reasons by advanced users.

30.
```
void push_front(reference value);
```

Requires: value must be an lvalue, and it must be no greater than the minimum inserted key

Effects: Inserts x into the tree in the first position.

Complexity: Constant time.

Throws: Nothing.

Note: This function does not check preconditions so if value is greater than the minimum inserted key tree ordering invariant will be broken. This function is slightly more efficient than using "insert_before". This is a low-level function to be used only for performance reasons by advanced users.

31.
```
iterator erase(const_iterator i);
```

Effects: Erases the element pointed to by pos.

Complexity: Average complexity for erase element is constant time.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

32.
```
iterator erase(const_iterator b, const_iterator e);
```

Effects: Erases the range pointed to by b end e.

Complexity: Average complexity for erase range is at most O(log(size() + N)), where N is the number of elements in the range.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

33.
```
size_type erase(const_reference value);
```

Effects: Erases all the elements with the given value.

Returns: The number of erased elements.

Complexity: O(log(size() + N).

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

34.
```
template<typename KeyType, typename KeyValueCompare>
  size_type erase(const KeyType & key, KeyValueCompare comp);
```

Effects: Erases all the elements with the given key. according to the comparison functor "comp".

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Returns: The number of erased elements.

Complexity: O(log(size() + N).

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

35.
```
template<typename Disposer> 
  iterator erase_and_dispose(const_iterator i, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases the element pointed to by pos. Disposer::operator()(pointer) is called for the removed element.

Complexity: Average complexity for erase element is constant time.

Throws: Nothing.

Note: Invalidates the iterators to the erased elements.

36.
```
template<typename Disposer> 
  size_type erase_and_dispose(const_reference value, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases all the elements with the given value. Disposer::operator()(pointer) is called for the removed elements.

Returns: The number of erased elements.

Complexity: O(log(size() + N).

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

37.
```
template<typename Disposer> 
  iterator erase_and_dispose(const_iterator b, const_iterator e, 
                             Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases the range pointed to by b end e. Disposer::operator()(pointer) is called for the removed elements.

Complexity: Average complexity for erase range is at most O(log(size() + N)), where N is the number of elements in the range.

Throws: Nothing.

Note: Invalidates the iterators to the erased elements.

38.
```
template<typename KeyType, typename KeyValueCompare, typename Disposer> 
  size_type erase_and_dispose(const KeyType & key, KeyValueCompare comp, 
                              Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases all the elements with the given key. according to the comparison functor "comp". Disposer::operator()(pointer) is called for the removed elements.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Returns: The number of erased elements.

Complexity: O(log(size() + N).

Throws: Nothing.

Note: Invalidates the iterators to the erased elements.

39.
```
void clear();
```

Effects: Erases all of the elements.

Complexity: Linear to the number of elements on the container. if it's a safe-mode or auto-unlink value_type. Constant time otherwise.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

40.
```
template<typename Disposer> void clear_and_dispose(Disposer disposer);
```

Effects: Erases all of the elements calling disposer(p) for each node to be erased. **Complexity**: Average complexity for is at most O(log(size() + N)), where N is the number of elements in the container.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. Calls N times to disposer functor.

41.
```
size_type count(const_reference value) const;
```

Effects: Returns the number of contained elements with the given value

Complexity: Logarithmic to the number of elements contained plus lineal to number of objects with the given value.

Throws: Nothing.

42.
```
template<typename KeyType, typename KeyValueCompare> 
  size_type count(const KeyType & key, KeyValueCompare comp) const;
```

Effects: Returns the number of contained elements with the given key

Complexity: Logarithmic to the number of elements contained plus lineal to number of objects with the given key.

Throws: Nothing.

43.
```
iterator lower_bound(const_reference value);
```

Effects: Returns an iterator to the first element whose key is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

44.
```
const_iterator lower_bound(const_reference value) const;
```

Effects: Returns an iterator to the first element whose key is not less than k or end() if that element does not exist.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Logarithmic.

Throws: Nothing.

45.
```
template<typename KeyType, typename KeyValueCompare> 
  iterator lower_bound(const KeyType & key, KeyValueCompare comp);
```

Effects: Returns an iterator to the first element whose key is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

46.
```
template<typename KeyType, typename KeyValueCompare> 
  const_iterator lower_bound(const KeyType & key, KeyValueCompare comp) const;
```

Effects: Returns a const iterator to the first element whose key is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

47.
```
iterator upper_bound(const_reference value);
```

Effects: Returns an iterator to the first element whose key is greater than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

48.
```
template<typename KeyType, typename KeyValueCompare> 
  iterator upper_bound(const KeyType & key, KeyValueCompare comp);
```

Effects: Returns an iterator to the first element whose key is greater than k according to comp or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

49.
```
const_iterator upper_bound(const_reference value) const;
```

Effects: Returns an iterator to the first element whose key is greater than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

50.
```
template<typename KeyType, typename KeyValueCompare> 
  const_iterator upper_bound(const KeyType & key, KeyValueCompare comp) const;
```

Effects: Returns an iterator to the first element whose key is greater than k according to comp or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

51.
```
iterator find(const_reference value);
```

Effects: Finds an iterator to the first element whose key is k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

52.
```
template<typename KeyType, typename KeyValueCompare> 
  iterator find(const KeyType & key, KeyValueCompare comp);
```

Effects: Finds an iterator to the first element whose key is k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

53.
```
const_iterator find(const_reference value) const;
```

Effects: Finds a const_iterator to the first element whose key is k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

54.
```
template<typename KeyType, typename KeyValueCompare> 
  const_iterator find(const KeyType & key, KeyValueCompare comp) const;
```

Effects: Finds a const_iterator to the first element whose key is k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

55.
```
std::pair< iterator, iterator > equal_range(const_reference value);
```

Effects: Finds a range containing all elements whose key is k or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: Nothing.

56.
```
template<typename KeyType, typename KeyValueCompare> 
  std::pair< iterator, iterator > 
  equal_range(const KeyType & key, KeyValueCompare comp);
```

Effects: Finds a range containing all elements whose key is k or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: Nothing.

57.
```
std::pair< const_iterator, const_iterator > 
equal_range(const_reference value) const;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Finds a range containing all elements whose key is k or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: Nothing.

58.
```
template<typename KeyType, typename KeyValueCompare>
  std::pair< const_iterator, const_iterator >
  equal_range(const KeyType & key, KeyValueCompare comp) const;
```

Effects: Finds a range containing all elements whose key is k or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: Nothing.

59.
```
std::pair< iterator, iterator >
bounded_range(const_reference lower_value, const_reference upper_value,
              bool left_closed, bool right_closed);
```

Requires: 'lower_value' must not be greater than 'upper_value'. If 'lower_value' == 'upper_value', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key) if left_closed, upper_bound(lower_key) otherwise

second = upper_bound(upper_key) if right_closed, lower_bound(upper_key) otherwise

Complexity: Logarithmic.

Throws: If the predicate throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_value and upper_value.

60.
```
template<typename KeyType, typename KeyValueCompare>
  std::pair< iterator, iterator >
  bounded_range(const KeyType & lower_key, const KeyType & upper_key,
                KeyValueCompare comp, bool left_closed, bool right_closed);
```

Requires: KeyValueCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. 'lower_key' must not be greater than 'upper_key' according to 'comp'. If 'lower_key' == 'upper_key', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key, comp) if left_closed, upper_bound(lower_key, comp) otherwise

second = upper_bound(upper_key, comp) if right_closed, lower_bound(upper_key, comp) otherwise

Complexity: Logarithmic.

Throws: If "comp" throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_key and upper_key.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

61.
```
std::pair< const_iterator, const_iterator >
bounded_range(const_reference lower_value, const_reference upper_value,
              bool left_closed, bool right_closed) const;
```

Requires: 'lower_value' must not be greater than 'upper_value'. If 'lower_value' == 'upper_value', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key) if left_closed, upper_bound(lower_key) otherwise

second = upper_bound(upper_key) if right_closed, lower_bound(upper_key) otherwise

Complexity: Logarithmic.

Throws: If the predicate throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_value and upper_value.

62.
```
template<typename KeyType, typename KeyValueCompare>
  std::pair< const_iterator, const_iterator >
  bounded_range(const KeyType & lower_key, const KeyType & upper_key,
                KeyValueCompare comp, bool left_closed, bool right_closed) const;
```

Requires: KeyValueCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. 'lower_key' must not be greater than 'upper_key' according to 'comp'. If 'lower_key' == 'upper_key', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key, comp) if left_closed, upper_bound(lower_key, comp) otherwise

second = upper_bound(upper_key, comp) if right_closed, lower_bound(upper_key, comp) otherwise

Complexity: Logarithmic.

Throws: If "comp" throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_key and upper_key.

63.
```
template<typename Cloner, typename Disposer>
  void clone_from(const rbtree & src, Cloner cloner, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw. Cloner should yield to nodes equivalent to the original nodes.

Effects: Erases all the elements from *this calling Disposer::operator()(pointer), clones all the elements from src calling Cloner::operator()(const_reference) and inserts them on *this. Copies the predicate from the source container.

If cloner throws, all cloned elements are unlinked and disposed calling Disposer::operator()(pointer).

Complexity: Linear to erased plus inserted elements.

Throws: If cloner throws or predicate copy assignment throws. Basic guarantee.

64.
```
pointer unlink_leftmost_without_rebalance();
```

Effects: Unlinks the leftmost node from the tree.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Average complexity is constant time.

Throws: Nothing.

Notes: This function breaks the tree and the tree can only be used for more unlink_leftmost_without_rebalance calls. This function is normally used to achieve a step by step controlled destruction of the tree.

65.
```
void replace_node(iterator replace_this, reference with_this);
```

Requires: replace_this must be a valid iterator of *this and with_this must not be inserted in any tree.

Effects: Replaces replace_this in its position in the tree with with_this. The tree does not need to be rebalanced.

Complexity: Constant.

Throws: Nothing.

Note: This function will break container ordering invariants if with_this is not equivalent to *replace_this according to the ordering rules. This function is faster than erasing and inserting the node, since no rebalancing or comparison is needed.

66.
```
iterator iterator_to(reference value);
```

Requires: value must be an lvalue and shall be in a set of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid iterator i belonging to the set that points to the value

Complexity: Constant.

Throws: Nothing.

67.
```
const_iterator iterator_to(const_reference value) const;
```

Requires: value must be an lvalue and shall be in a set of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid const_iterator i belonging to the set that points to the value

Complexity: Constant.

Throws: Nothing.

`rbtree` public static functions

1.
```
static rbtree & container_from_end_iterator(iterator end_iterator);
```

Precondition: end_iterator must be a valid end iterator of rbtree.

Effects: Returns a const reference to the rbtree associated to the end iterator

Throws: Nothing.

Complexity: Constant.

2.
```
static const rbtree & container_from_end_iterator(const_iterator end_iterator);
```

Precondition: end_iterator must be a valid end const_iterator of rbtree.

Effects: Returns a const reference to the rbtree associated to the iterator

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: Nothing.

Complexity: Constant.

3.
```
static rbtree & container_from_iterator(iterator it);
```

Precondition: it must be a valid iterator of rbtree.

Effects: Returns a const reference to the tree associated to the iterator

Throws: Nothing.

Complexity: Logarithmic.

4.
```
static const rbtree & container_from_iterator(const_iterator it);
```

Precondition: it must be a valid end const_iterator of rbtree.

Effects: Returns a const reference to the tree associated to the end iterator

Throws: Nothing.

Complexity: Logarithmic.

5.
```
static iterator s_iterator_to(reference value);
```

Requires: value must be an lvalue and shall be in a set of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid iterator i belonging to the set that points to the value

Complexity: Constant.

Throws: Nothing.

Note: This static function is available only if the *value traits* is stateless.

6.
```
static const_iterator s_iterator_to(const_reference value);
```

Requires: value must be an lvalue and shall be in a set of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid const_iterator i belonging to the set that points to the value

Complexity: Constant.

Throws: Nothing.

Note: This static function is available only if the *value traits* is stateless.

7.
```
static void init_node(reference value);
```

Requires: value shall not be in a tree.

Effects: init_node puts the hook of a value in a well-known default state.

Throws: Nothing.

Complexity: Constant time.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Note: This function puts the hook in the well-known default state used by auto_unlink and safe hooks.

8.
```
static void remove_node(reference value);
```

Effects: removes "value" from the container.

Throws: Nothing.

Complexity: Logarithmic time.

Note: This static function is only usable with non-constant time size containers that have stateless comparison functors.

If the user calls this function with a constant time size container or stateful comparison functor a compilation error will be issued.

`rbtree` private static functions

1.
```
static rbtree &
priv_container_from_end_iterator(const const_iterator & end_iterator);
```

2.
```
static rbtree & priv_container_from_iterator(const const_iterator & it);
```

Struct template make_rbtree

boost::intrusive::make_rbtree

Synopsis

```
// In header: <boost/intrusive/rbtree.hpp>

template<typename T, class... Options>
struct make_rbtree {
  // types
  typedef implementation_defined type;
};
```

Description

Helper metafunction to define a `rbtree` that yields to the same type when the same options (either explicitly or implicitly) are used.

Header <boost/intrusive/rbtree_algorithms.hpp>

```
namespace boost {
  namespace intrusive {
    template<typename NodeTraits> class rbtree_algorithms;
  }
}
```

Class template rbtree_algorithms

boost::intrusive::rbtree_algorithms

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/intrusive/rbtree_algorithms.hpp>

template<typename NodeTraits> 
class rbtree_algorithms {
public:
  // types
  typedef NodeTraits                          node_traits;       
  typedef NodeTraits::node                    node;              
  typedef NodeTraits::node_ptr                node_ptr;          
  typedef NodeTraits::const_node_ptr          const_node_ptr;    
  typedef NodeTraits::color                   color;             
  typedef tree_algorithms::insert_commit_data insert_commit_data;

  // public static functions
  static node_ptr begin_node(const const_node_ptr &);
  static node_ptr end_node(const const_node_ptr &);
  static void swap_tree(const node_ptr &, const node_ptr &);
  static void swap_nodes(const node_ptr &, const node_ptr &);
  static void swap_nodes(const node_ptr &, const node_ptr &, const node_ptr &, 
                         const node_ptr &);
  static void replace_node(const node_ptr &, const node_ptr &);
  static void replace_node(const node_ptr &, const node_ptr &, 
                           const node_ptr &);
  static void unlink(const node_ptr &);
  static node_ptr unlink_leftmost_without_rebalance(const node_ptr &);
  static bool unique(const const_node_ptr &);
  static std::size_t count(const const_node_ptr &);
  static std::size_t size(const const_node_ptr &);
  static node_ptr next_node(const node_ptr &);
  static node_ptr prev_node(const node_ptr &);
  static void init(const node_ptr &);
  static void init_header(const node_ptr &);
  static node_ptr erase(const node_ptr &, const node_ptr &);
  template<typename Cloner, typename Disposer> 
    static void clone(const const_node_ptr &, const node_ptr &, Cloner, 
                      Disposer);
  template<typename Disposer> 
    static void clear_and_dispose(const node_ptr &, Disposer);
  template<typename KeyType, typename KeyNodePtrCompare> 
    static node_ptr 
    lower_bound(const const_node_ptr &, const KeyType &, KeyNodePtrCompare);
  template<typename KeyType, typename KeyNodePtrCompare> 
    static node_ptr 
    upper_bound(const const_node_ptr &, const KeyType &, KeyNodePtrCompare);
  template<typename KeyType, typename KeyNodePtrCompare> 
    static node_ptr 
    find(const const_node_ptr &, const KeyType &, KeyNodePtrCompare);
  template<typename KeyType, typename KeyNodePtrCompare> 
    static std::pair< node_ptr, node_ptr > 
    equal_range(const const_node_ptr &, const KeyType &, KeyNodePtrCompare);
  template<typename KeyType, typename KeyNodePtrCompare> 
    static std::pair< node_ptr, node_ptr > 
    bounded_range(const const_node_ptr &, const KeyType &, const KeyType &, 
                  KeyNodePtrCompare, bool, bool);
  template<typename NodePtrCompare> 
    static node_ptr 
    insert_equal_upper_bound(const node_ptr &, const node_ptr &, 
                             NodePtrCompare);
  template<typename NodePtrCompare> 
    static node_ptr 
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
    insert_equal_lower_bound(const node_ptr &, const node_ptr &,
                             NodePtrCompare);
  template<typename NodePtrCompare>
    static node_ptr
    insert_equal(const node_ptr &, const node_ptr &, const node_ptr &,
                 NodePtrCompare);
  static node_ptr
  insert_before(const node_ptr &, const node_ptr &, const node_ptr &);
  static void push_back(const node_ptr &, const node_ptr &);
  static void push_front(const node_ptr &, const node_ptr &);
  template<typename KeyType, typename KeyNodePtrCompare>
    static std::pair< node_ptr, bool >
    insert_unique_check(const const_node_ptr &, const KeyType &,
                        KeyNodePtrCompare, insert_commit_data &);
  template<typename KeyType, typename KeyNodePtrCompare>
    static std::pair< node_ptr, bool >
    insert_unique_check(const const_node_ptr &, const node_ptr &,
                        const KeyType &, KeyNodePtrCompare,
                        insert_commit_data &);
  static void insert_unique_commit(const node_ptr &, const node_ptr &,
                                   const insert_commit_data &);
  static node_ptr get_header(const node_ptr &);
};
```

Description

rbtree_algorithms provides basic algorithms to manipulate nodes forming a red-black tree. The insertion and deletion algorithms are based on those in Cormen, Leiserson, and Rivest, Introduction to Algorithms (MIT Press, 1990), except that

(1) the header node is maintained with links not only to the root but also to the leftmost node of the tree, to enable constant time begin(), and to the rightmost node of the tree, to enable linear time performance when used with the generic set algorithms (set_union, etc.);

(2) when a node being deleted has two children its successor node is relinked into its place, rather than copied, so that the only pointers invalidated are those referring to the deleted node.

rbtree_algorithms is configured with a NodeTraits class, which encapsulates the information about the node to be manipulated. NodeTraits must support the following interface:

Typedefs:

`node`: The type of the node that forms the circular list

`node_ptr`: A pointer to a node

`const_node_ptr`: A pointer to a const node

`color`: The type that can store the color of a node

Static functions:

`static node_ptr get_parent(const_node_ptr n);`

`static void set_parent(node_ptr n, node_ptr parent);`

`static node_ptr get_left(const_node_ptr n);`

`static void set_left(node_ptr n, node_ptr left);`

`static node_ptr get_right(const_node_ptr n);`

`static void set_right(node_ptr n, node_ptr right);`

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
static color get_color(const_node_ptr n);
```

```
static void set_color(node_ptr n, color c);
```

```
static color black();
```

```
static color red();
```

`rbtree_algorithms` public types

1. typedef tree_algorithms::insert_commit_data insert_commit_data;

   This type is the information that will be filled by insert_unique_check

`rbtree_algorithms` public static functions

1.
   ```
   static node_ptr begin_node(const const_node_ptr & header);
   ```

2.
   ```
   static node_ptr end_node(const const_node_ptr & header);
   ```

3.
   ```
   static void swap_tree(const node_ptr & header1, const node_ptr & header2);
   ```

   Requires: header1 and header2 must be the header nodes of two trees.

   Effects: Swaps two trees. After the function header1 will contain links to the second tree and header2 will have links to the first tree.

   Complexity: Constant.

   Throws: Nothing.

4.
   ```
   static void swap_nodes(const node_ptr & node1, const node_ptr & node2);
   ```

   Requires: node1 and node2 can't be header nodes of two trees.

   Effects: Swaps two nodes. After the function node1 will be inserted in the position node2 before the function. node2 will be inserted in the position node1 had before the function.

   Complexity: Logarithmic.

   Throws: Nothing.

   Note: This function will break container ordering invariants if node1 and node2 are not equivalent according to the ordering rules.

   Experimental function

5.
   ```
   static void swap_nodes(const node_ptr & node1, const node_ptr & header1,
                          const node_ptr & node2, const node_ptr & header2);
   ```

   Requires: node1 and node2 can't be header nodes of two trees with header header1 and header2.

   Effects: Swaps two nodes. After the function node1 will be inserted in the position node2 before the function. node2 will be inserted in the position node1 had before the function.

   Complexity: Constant.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: Nothing.

Note: This function will break container ordering invariants if node1 and node2 are not equivalent according to the ordering rules.

Experimental function

6.
```
static void replace_node(const node_ptr & node_to_be_replaced,
                         const node_ptr & new_node);
```

Requires: node_to_be_replaced must be inserted in a tree and new_node must not be inserted in a tree.

Effects: Replaces node_to_be_replaced in its position in the tree with new_node. The tree does not need to be rebalanced

Complexity: Logarithmic.

Throws: Nothing.

Note: This function will break container ordering invariants if new_node is not equivalent to node_to_be_replaced according to the ordering rules. This function is faster than erasing and inserting the node, since no rebalancing and comparison is needed.

Experimental function

7.
```
static void replace_node(const node_ptr & node_to_be_replaced,
                         const node_ptr & header, const node_ptr & new_node);
```

Requires: node_to_be_replaced must be inserted in a tree with header "header" and new_node must not be inserted in a tree.

Effects: Replaces node_to_be_replaced in its position in the tree with new_node. The tree does not need to be rebalanced

Complexity: Constant.

Throws: Nothing.

Note: This function will break container ordering invariants if new_node is not equivalent to node_to_be_replaced according to the ordering rules. This function is faster than erasing and inserting the node, since no rebalancing or comparison is needed.

Experimental function

8.
```
static void unlink(const node_ptr & node);
```

Requires: node is a tree node but not the header.

Effects: Unlinks the node and rebalances the tree.

Complexity: Average complexity is constant time.

Throws: Nothing.

9.
```
static node_ptr unlink_leftmost_without_rebalance(const node_ptr & header);
```

Requires: header is the header of a tree.

Effects: Unlinks the leftmost node from the tree, and updates the header link to the new leftmost node.

Complexity: Average complexity is constant time.

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Notes: This function breaks the tree and the tree can only be used for more unlink_leftmost_without_rebalance calls. This function is normally used to achieve a step by step controlled destruction of the tree.

10.
```
static bool unique(const const_node_ptr & node);
```

Requires: node is a node of the tree or an node initialized by init(...).

Effects: Returns true if the node is initialized by init().

Complexity: Constant time.

Throws: Nothing.

11.
```
static std::size_t count(const const_node_ptr & node);
```

Requires: node is a node of the tree but it's not the header.

Effects: Returns the number of nodes of the subtree.

Complexity: Linear time.

Throws: Nothing.

12.
```
static std::size_t size(const const_node_ptr & header);
```

Requires: header is the header node of the tree.

Effects: Returns the number of nodes above the header.

Complexity: Linear time.

Throws: Nothing.

13.
```
static node_ptr next_node(const node_ptr & p);
```

Requires: p is a node from the tree except the header.

Effects: Returns the next node of the tree.

Complexity: Average constant time.

Throws: Nothing.

14.
```
static node_ptr prev_node(const node_ptr & p);
```

Requires: p is a node from the tree except the leftmost node.

Effects: Returns the previous node of the tree.

Complexity: Average constant time.

Throws: Nothing.

15.
```
static void init(const node_ptr & node);
```

Requires: node must not be part of any tree.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: After the function unique(node) == true.

Complexity: Constant.

Throws: Nothing.

Nodes: If node is inserted in a tree, this function corrupts the tree.

16.
```
static void init_header(const node_ptr & header);
```

Requires: node must not be part of any tree.

Effects: Initializes the header to represent an empty tree. unique(header) == true.

Complexity: Constant.

Throws: Nothing.

Nodes: If node is inserted in a tree, this function corrupts the tree.

17.
```
static node_ptr erase(const node_ptr & header, const node_ptr & z);
```

Requires: header must be the header of a tree, z a node of that tree and z != header.

Effects: Erases node "z" from the tree with header "header".

Complexity: Amortized constant time.

Throws: Nothing.

18.
```
template<typename Cloner, typename Disposer> 
  static void clone(const const_node_ptr & source_header, 
                    const node_ptr & target_header, Cloner cloner, 
                    Disposer disposer);
```

Requires: "cloner" must be a function object taking a node_ptr and returning a new cloned node of it. "disposer" must take a node_ptr and shouldn't throw.

Effects: First empties target tree calling `void disposer::operator()(const node_ptr &)` for every node of the tree except the header.

Then, duplicates the entire tree pointed by "source_header" cloning each source node with `node_ptr Cloner::operator()(const node_ptr &)` to obtain the nodes of the target tree. If "cloner" throws, the cloned target nodes are disposed using `void disposer(const node_ptr &)`.

Complexity: Linear to the number of element of the source tree plus the. number of elements of tree target tree when calling this function.

Throws: If cloner functor throws. If this happens target nodes are disposed.

19.
```
template<typename Disposer> 
  static void clear_and_dispose(const node_ptr & header, Disposer disposer);
```

Requires: "disposer" must be an object function taking a node_ptr parameter and shouldn't throw.

Effects: Empties the target tree calling `void disposer::operator()(const node_ptr &)` for every node of the tree except the header.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Linear to the number of element of the source tree plus the. number of elements of tree target tree when calling this function.

Throws: If cloner functor throws. If this happens target nodes are disposed.

20.
```
template<typename KeyType, typename KeyNodePtrCompare>
  static node_ptr
  lower_bound(const const_node_ptr & header, const KeyType & key,
              KeyNodePtrCompare comp);
```

Requires: "header" must be the header node of a tree. KeyNodePtrCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. KeyNodePtrCompare can compare KeyType with tree's node_ptrs.

Effects: Returns an node_ptr to the first element that is not less than "key" according to "comp" or "header" if that element does not exist.

Complexity: Logarithmic.

Throws: If "comp" throws.

21.
```
template<typename KeyType, typename KeyNodePtrCompare>
  static node_ptr
  upper_bound(const const_node_ptr & header, const KeyType & key,
              KeyNodePtrCompare comp);
```

Requires: "header" must be the header node of a tree. KeyNodePtrCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. KeyNodePtrCompare can compare KeyType with tree's node_ptrs.

Effects: Returns an node_ptr to the first element that is greater than "key" according to "comp" or "header" if that element does not exist.

Complexity: Logarithmic.

Throws: If "comp" throws.

22.
```
template<typename KeyType, typename KeyNodePtrCompare>
  static node_ptr
  find(const const_node_ptr & header, const KeyType & key,
       KeyNodePtrCompare comp);
```

Requires: "header" must be the header node of a tree. KeyNodePtrCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. KeyNodePtrCompare can compare KeyType with tree's node_ptrs.

Effects: Returns an node_ptr to the element that is equivalent to "key" according to "comp" or "header" if that element does not exist.

Complexity: Logarithmic.

Throws: If "comp" throws.

23.
```
template<typename KeyType, typename KeyNodePtrCompare>
  static std::pair< node_ptr, node_ptr >
  equal_range(const const_node_ptr & header, const KeyType & key,
              KeyNodePtrCompare comp);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Requires: "header" must be the header node of a tree. KeyNodePtrCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. KeyNodePtrCompare can compare KeyType with tree's node_ptrs.

Effects: Returns an a pair of node_ptr delimiting a range containing all elements that are equivalent to "key" according to "comp" or an empty range that indicates the position where those elements would be if they there are no equivalent elements.

Complexity: Logarithmic.

Throws: If "comp" throws.

24.
```
template<typename KeyType, typename KeyNodePtrCompare>
  static std::pair< node_ptr, node_ptr >
  bounded_range(const const_node_ptr & header, const KeyType & lower_key,
                const KeyType & upper_key, KeyNodePtrCompare comp,
                bool left_closed, bool right_closed);
```

Requires: "header" must be the header node of a tree. KeyNodePtrCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. KeyNodePtrCompare can compare KeyType with tree's node_ptrs. 'lower_key' must not be greater than 'upper_key' according to 'comp'. If 'lower_key' == 'upper_key', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key) if left_closed, upper_bound(lower_key) otherwise

second = upper_bound(upper_key) if right_closed, lower_bound(upper_key) otherwise

Complexity: Logarithmic.

Throws: If "comp" throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_key and upper_key.

25.
```
template<typename NodePtrCompare>
  static node_ptr
  insert_equal_upper_bound(const node_ptr & h, const node_ptr & new_node,
                           NodePtrCompare comp);
```

Requires: "h" must be the header node of a tree. NodePtrCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. NodePtrCompare compares two node_ptrs.

Effects: Inserts new_node into the tree before the upper bound according to "comp".

Complexity: Average complexity for insert element is at most logarithmic.

Throws: If "comp" throws.

26.
```
template<typename NodePtrCompare>
  static node_ptr
  insert_equal_lower_bound(const node_ptr & h, const node_ptr & new_node,
                           NodePtrCompare comp);
```

Requires: "h" must be the header node of a tree. NodePtrCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. NodePtrCompare compares two node_ptrs.

Effects: Inserts new_node into the tree before the lower bound according to "comp".

Complexity: Average complexity for insert element is at most logarithmic.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: If "comp" throws.

27.
```
template<typename NodePtrCompare>
  static node_ptr
  insert_equal(const node_ptr & header, const node_ptr & hint,
                const node_ptr & new_node, NodePtrCompare comp);
```

Requires: "header" must be the header node of a tree. NodePtrCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. NodePtrCompare compares two node_ptrs. "hint" is node from the "header"'s tree.

Effects: Inserts new_node into the tree, using "hint" as a hint to where it will be inserted. If "hint" is the upper_bound the insertion takes constant time (two comparisons in the worst case).

Complexity: Logarithmic in general, but it is amortized constant time if new_node is inserted immediately before "hint".

Throws: If "comp" throws.

28.
```
static node_ptr
insert_before(const node_ptr & header, const node_ptr & pos,
              const node_ptr & new_node);
```

Requires: "header" must be the header node of a tree. "pos" must be a valid iterator or header (end) node. "pos" must be an iterator pointing to the successor to "new_node" once inserted according to the order of already inserted nodes. This function does not check "pos" and this precondition must be guaranteed by the caller.

Effects: Inserts new_node into the tree before "pos".

Complexity: Constant-time.

Throws: Nothing.

Note: If "pos" is not the successor of the newly inserted "new_node" tree invariants might be broken.

29.
```
static void push_back(const node_ptr & header, const node_ptr & new_node);
```

Requires: "header" must be the header node of a tree. "new_node" must be, according to the used ordering no less than the greatest inserted key.

Effects: Inserts new_node into the tree before "pos".

Complexity: Constant-time.

Throws: Nothing.

Note: If "new_node" is less than the greatest inserted key tree invariants are broken. This function is slightly faster than using "insert_before".

30.
```
static void push_front(const node_ptr & header, const node_ptr & new_node);
```

Requires: "header" must be the header node of a tree. "new_node" must be, according to the used ordering, no greater than the lowest inserted key.

Effects: Inserts new_node into the tree before "pos".

Complexity: Constant-time.

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Note: If "new_node" is greater than the lowest inserted key tree invariants are broken. This function is slightly faster than using "insert_before".

31.
```
template<typename KeyType, typename KeyNodePtrCompare>
  static std::pair< node_ptr, bool >
  insert_unique_check(const const_node_ptr & header, const KeyType & key,
                      KeyNodePtrCompare comp,
                      insert_commit_data & commit_data);
```

Requires: "header" must be the header node of a tree. KeyNodePtrCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. NodePtrCompare compares KeyType with a node_ptr.

Effects: Checks if there is an equivalent node to "key" in the tree according to "comp" and obtains the needed information to realize a constant-time node insertion if there is no equivalent node.

Returns: If there is an equivalent value returns a pair containing a node_ptr to the already present node and false. If there is not equivalent key can be inserted returns true in the returned pair's boolean and fills "commit_data" that is meant to be used with the "insert_commit" function to achieve a constant-time insertion function.

Complexity: Average complexity is at most logarithmic.

Throws: If "comp" throws.

Notes: This function is used to improve performance when constructing a node is expensive and the user does not want to have two equivalent nodes in the tree: if there is an equivalent value the constructed object must be discarded. Many times, the part of the node that is used to impose the order is much cheaper to construct than the node and this function offers the possibility to use that part to check if the insertion will be successful.

If the check is successful, the user can construct the node and use "insert_commit" to insert the node in constant-time. This gives a total logarithmic complexity to the insertion: check(O(log(N)) + commit(O(1)).

"commit_data" remains valid for a subsequent "insert_unique_commit" only if no more objects are inserted or erased from the set.

32.
```
template<typename KeyType, typename KeyNodePtrCompare>
  static std::pair< node_ptr, bool >
  insert_unique_check(const const_node_ptr & header, const node_ptr & hint,
                      const KeyType & key, KeyNodePtrCompare comp,
                      insert_commit_data & commit_data);
```

Requires: "header" must be the header node of a tree. KeyNodePtrCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. NodePtrCompare compares KeyType with a node_ptr. "hint" is node from the "header"'s tree.

Effects: Checks if there is an equivalent node to "key" in the tree according to "comp" using "hint" as a hint to where it should be inserted and obtains the needed information to realize a constant-time node insertion if there is no equivalent node. If "hint" is the upper_bound the function has constant time complexity (two comparisons in the worst case).

Returns: If there is an equivalent value returns a pair containing a node_ptr to the already present node and false. If there is not equivalent key can be inserted returns true in the returned pair's boolean and fills "commit_data" that is meant to be used with the "insert_commit" function to achieve a constant-time insertion function.

Complexity: Average complexity is at most logarithmic, but it is amortized constant time if new_node should be inserted immediately before "hint".

Throws: If "comp" throws.

Notes: This function is used to improve performance when constructing a node is expensive and the user does not want to have two equivalent nodes in the tree: if there is an equivalent value the constructed object must be discarded. Many times, the part of

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

the node that is used to impose the order is much cheaper to construct than the node and this function offers the possibility to use that part to check if the insertion will be successful.

If the check is successful, the user can construct the node and use "insert_commit" to insert the node in constant-time. This gives a total logarithmic complexity to the insertion: check(O(log(N)) + commit(O(1)).

"commit_data" remains valid for a subsequent "insert_unique_commit" only if no more objects are inserted or erased from the set.

33.
```
static void insert_unique_commit(const node_ptr & header,
                                 const node_ptr & new_value,
                                 const insert_commit_data & commit_data);
```

Requires: "header" must be the header node of a tree. "commit_data" must have been obtained from a previous call to "insert_unique_check". No objects should have been inserted or erased from the set between the "insert_unique_check" that filled "commit_data" and the call to "insert_commit".

Effects: Inserts new_node in the set using the information obtained from the "commit_data" that a previous "insert_check" filled.

Complexity: Constant time.

Throws: Nothing.

Notes: This function has only sense if a "insert_unique_check" has been previously executed to fill "commit_data". No value should be inserted or erased between the "insert_check" and "insert_commit" calls.

34.
```
static node_ptr get_header(const node_ptr & n);
```

Requires: "n" must be a node inserted in a tree.

Effects: Returns a pointer to the header node of the tree.

Complexity: Logarithmic.

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Header <boost/intrusive/set.hpp>

```
namespace boost {
  namespace intrusive {
    template<typename T, class... Options> class set;

    template<typename T, class... Options> struct make_set;

    template<typename T, class... Options> class multiset;

    template<typename T, class... Options> struct make_multiset;
    template<typename T, class... Options>
      bool operator!=(const set< T, Options...> & x,
                      const set< T, Options...> & y);
    template<typename T, class... Options>
      bool operator>(const set< T, Options...> & x,
                     const set< T, Options...> & y);
    template<typename T, class... Options>
      bool operator<=(const set< T, Options...> & x,
                      const set< T, Options...> & y);
    template<typename T, class... Options>
      bool operator>=(const set< T, Options...> & x,
                      const set< T, Options...> & y);
    template<typename T, class... Options>
      void swap(set< T, Options...> & x, set< T, Options...> & y);
    template<typename T, class... Options>
      bool operator!=(const multiset< T, Options...> & x,
                      const multiset< T, Options...> & y);
    template<typename T, class... Options>
      bool operator>(const multiset< T, Options...> & x,
                     const multiset< T, Options...> & y);
    template<typename T, class... Options>
      bool operator<=(const multiset< T, Options...> & x,
                      const multiset< T, Options...> & y);
    template<typename T, class... Options>
      bool operator>=(const multiset< T, Options...> & x,
                      const multiset< T, Options...> & y);
    template<typename T, class... Options>
      void swap(multiset< T, Options...> & x, multiset< T, Options...> & y);
  }
}
```

Class template set

boost::intrusive::set

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/intrusive/set.hpp>

template<typename T, class... Options>
class set {
public:
  // types
  typedef implementation_defined::value_type             value_type;
  typedef implementation_defined::value_traits           value_traits;
  typedef implementation_defined::pointer                pointer;
  typedef implementation_defined::const_pointer          const_pointer;
  typedef implementation_defined::reference              reference;
  typedef implementation_defined::const_reference        const_reference;
  typedef implementation_defined::difference_type        difference_type;
  typedef implementation_defined::size_type              size_type;
  typedef implementation_defined::value_compare          value_compare;
  typedef implementation_defined::key_compare            key_compare;
  typedef implementation_defined::iterator               iterator;
  typedef implementation_defined::const_iterator         const_iterator;
  typedef implementation_defined::reverse_iterator       reverse_iterator;
  typedef implementation_defined::const_reverse_iterator const_reverse_iterator;
  typedef implementation_defined::insert_commit_data     insert_commit_data;
  typedef implementation_defined::node_traits            node_traits;
  typedef implementation_defined::node                   node;
  typedef implementation_defined::node_ptr               node_ptr;
  typedef implementation_defined::const_node_ptr         const_node_ptr;
  typedef implementation_defined::node_algorithms        node_algorithms;

  // construct/copy/destruct
  explicit set(const value_compare & = value_compare(), 
               const value_traits & = value_traits());
  template<typename Iterator> 
    set(Iterator, Iterator, const value_compare & = value_compare(), 
        const value_traits & = value_traits());
  set(BOOST_RV_REF(set));
  set& operator=(BOOST_RV_REF(set));
  ~set();

  // public member functions
  iterator begin();
  const_iterator begin() const;
  const_iterator cbegin() const;
  iterator end();
  const_iterator end() const;
  const_iterator cend() const;
  reverse_iterator rbegin();
  const_reverse_iterator rbegin() const;
  const_reverse_iterator crbegin() const;
  reverse_iterator rend();
  const_reverse_iterator rend() const;
  const_reverse_iterator crend() const;
  key_compare key_comp() const;
  value_compare value_comp() const;
  bool empty() const;
  size_type size() const;
  void swap(set &);
  template<typename Cloner, typename Disposer> 
    void clone_from(const set &, Cloner, Disposer);
  std::pair< iterator, bool > insert(reference);
  iterator insert(const_iterator, reference);
  template<typename KeyType, typename KeyValueCompare>
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
    std::pair< iterator, bool >
    insert_check(const KeyType &, KeyValueCompare, insert_commit_data &);
  template<typename KeyType, typename KeyValueCompare>
    std::pair< iterator, bool >
    insert_check(const_iterator, const KeyType &, KeyValueCompare,
                 insert_commit_data &);
  iterator insert_commit(reference, const insert_commit_data &);
  template<typename Iterator> void insert(Iterator, Iterator);
  iterator insert_before(const_iterator, reference);
  void push_back(reference);
  void push_front(reference);
  iterator erase(const_iterator);
  iterator erase(const_iterator, const_iterator);
  size_type erase(const_reference);
  template<typename KeyType, typename KeyValueCompare>
    size_type erase(const KeyType &, KeyValueCompare);
  template<typename Disposer>
    iterator erase_and_dispose(const_iterator, Disposer);
  template<typename Disposer>
    iterator erase_and_dispose(const_iterator, const_iterator, Disposer);
  template<typename Disposer>
    size_type erase_and_dispose(const_reference, Disposer);
  template<typename KeyType, typename KeyValueCompare, typename Disposer>
    size_type erase_and_dispose(const KeyType &, KeyValueCompare, Disposer);
  void clear();
  template<typename Disposer> void clear_and_dispose(Disposer);
  size_type count(const_reference) const;
  template<typename KeyType, typename KeyValueCompare>
    size_type count(const KeyType &, KeyValueCompare) const;
  iterator lower_bound(const_reference);
  template<typename KeyType, typename KeyValueCompare>
    iterator lower_bound(const KeyType &, KeyValueCompare);
  const_iterator lower_bound(const_reference) const;
  template<typename KeyType, typename KeyValueCompare>
    const_iterator lower_bound(const KeyType &, KeyValueCompare) const;
  iterator upper_bound(const_reference);
  template<typename KeyType, typename KeyValueCompare>
    iterator upper_bound(const KeyType &, KeyValueCompare);
  const_iterator upper_bound(const_reference) const;
  template<typename KeyType, typename KeyValueCompare>
    const_iterator upper_bound(const KeyType &, KeyValueCompare) const;
  iterator find(const_reference);
  template<typename KeyType, typename KeyValueCompare>
    iterator find(const KeyType &, KeyValueCompare);
  const_iterator find(const_reference) const;
  template<typename KeyType, typename KeyValueCompare>
    const_iterator find(const KeyType &, KeyValueCompare) const;
  std::pair< iterator, iterator > equal_range(const_reference);
  template<typename KeyType, typename KeyValueCompare>
    std::pair< iterator, iterator >
    equal_range(const KeyType &, KeyValueCompare);
  std::pair< const_iterator, const_iterator >
  equal_range(const_reference) const;
  template<typename KeyType, typename KeyValueCompare>
    std::pair< const_iterator, const_iterator >
    equal_range(const KeyType &, KeyValueCompare) const;
  std::pair< iterator, iterator >
  bounded_range(const_reference, const_reference, bool, bool);
  template<typename KeyType, typename KeyValueCompare>
    std::pair< iterator, iterator >
    bounded_range(const KeyType &, const KeyType &, KeyValueCompare, bool,
                  bool);
  std::pair< const_iterator, const_iterator >
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
  bounded_range(const_reference, const_reference, bool, bool) const;
  template<typename KeyType, typename KeyValueCompare> 
    std::pair< const_iterator, const_iterator > 
    bounded_range(const KeyType &, const KeyType &, KeyValueCompare, bool, 
                  bool) const;
  iterator iterator_to(reference);
  const_iterator iterator_to(const_reference) const;
  pointer unlink_leftmost_without_rebalance();
  void replace_node(iterator, reference);

  // public static functions
  static set & container_from_end_iterator(iterator);
  static const set & container_from_end_iterator(const_iterator);
  static set & container_from_iterator(iterator);
  static const set & container_from_iterator(const_iterator);
  static iterator s_iterator_to(reference);
  static const_iterator s_iterator_to(const_reference);
  static void init_node(reference);

  // public data members
  static const bool constant_time_size;
};
```

Description

The class template set is an intrusive container, that mimics most of the interface of std::set as described in the C++ standard.

The template parameter `T` is the type to be managed by the container. The user can specify additional options and if no options are provided default options are used.

The container supports the following options: `base_hook<>/member_hook<>/value_traits<>`, `constant_time_size<>`, `size_type<>` and `compare<>`.

`set` public construct/copy/destruct

1.
```
explicit set(const value_compare & cmp = value_compare(), 
             const value_traits & v_traits = value_traits());
```

   Effects: Constructs an empty set.

   Complexity: Constant.

   Throws: If value_traits::node_traits::node constructor throws (this does not happen with predefined Boost.Intrusive hooks) or the copy constructor of the value_compare object throws.

2.
```
template<typename Iterator> 
  set(Iterator b, Iterator e, const value_compare & cmp = value_compare(), 
      const value_traits & v_traits = value_traits());
```

   Requires: Dereferencing iterator must yield an lvalue of type value_type. cmp must be a comparison function that induces a strict weak ordering.

   Effects: Constructs an empty set and inserts elements from [b, e).

   Complexity: Linear in N if [b, e) is already sorted using comp and otherwise N * log N, where N is std::distance(last, first).

   Throws: If value_traits::node_traits::node constructor throws (this does not happen with predefined Boost.Intrusive hooks) or the copy constructor/operator() of the value_compare object throws.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

3.
```
set(BOOST_RV_REF(set) x);
```

Effects: to-do

4.
```
set& operator=(BOOST_RV_REF(set) x);
```

Effects: to-do

5.
```
~set();
```

Effects: Detaches all elements from this. The objects in the set are not deleted (i.e. no destructors are called).

Complexity: Linear to the number of elements on the container. if it's a safe-mode or auto-unlink value_type. Constant time otherwise.

Throws: Nothing.

`set` public member functions

1.
```
iterator begin();
```

Effects: Returns an iterator pointing to the beginning of the set.

Complexity: Constant.

Throws: Nothing.

2.
```
const_iterator begin() const;
```

Effects: Returns a const_iterator pointing to the beginning of the set.

Complexity: Constant.

Throws: Nothing.

3.
```
const_iterator cbegin() const;
```

Effects: Returns a const_iterator pointing to the beginning of the set.

Complexity: Constant.

Throws: Nothing.

4.
```
iterator end();
```

Effects: Returns an iterator pointing to the end of the set.

Complexity: Constant.

Throws: Nothing.

5.
```
const_iterator end() const;
```

Effects: Returns a const_iterator pointing to the end of the set.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Constant.

Throws: Nothing.

6.
```
const_iterator cend() const;
```

Effects: Returns a const_iterator pointing to the end of the set.

Complexity: Constant.

Throws: Nothing.

7.
```
reverse_iterator rbegin();
```

Effects: Returns a reverse_iterator pointing to the beginning of the reversed set.

Complexity: Constant.

Throws: Nothing.

8.
```
const_reverse_iterator rbegin() const;
```

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed set.

Complexity: Constant.

Throws: Nothing.

9.
```
const_reverse_iterator crbegin() const;
```

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed set.

Complexity: Constant.

Throws: Nothing.

10.
```
reverse_iterator rend();
```

Effects: Returns a reverse_iterator pointing to the end of the reversed set.

Complexity: Constant.

Throws: Nothing.

11.
```
const_reverse_iterator rend() const;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed set.

Complexity: Constant.

Throws: Nothing.

12.
```
const_reverse_iterator crend() const;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed set.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Constant.

Throws: Nothing.

13.
```
key_compare key_comp() const;
```

Effects: Returns the key_compare object used by the set.

Complexity: Constant.

Throws: If key_compare copy-constructor throws.

14.
```
value_compare value_comp() const;
```

Effects: Returns the value_compare object used by the set.

Complexity: Constant.

Throws: If value_compare copy-constructor throws.

15.
```
bool empty() const;
```

Effects: Returns true if the container is empty.

Complexity: Constant.

Throws: Nothing.

16.
```
size_type size() const;
```

Effects: Returns the number of elements stored in the set.

Complexity: Linear to elements contained in *this if, constant-time size option is enabled. Constant-time otherwise.

Throws: Nothing.

17.
```
void swap(set & other);
```

Effects: Swaps the contents of two sets.

Complexity: Constant.

Throws: If the swap() call for the comparison functor found using ADL throws. Strong guarantee.

18.
```
template<typename Cloner, typename Disposer>
  void clone_from(const set & src, Cloner cloner, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw. Cloner should yield to nodes equivalent to the original nodes.

Effects: Erases all the elements from *this calling Disposer::operator()(pointer), clones all the elements from src calling Cloner::operator()(const_reference) and inserts them on *this. Copies the predicate from the source container.

If cloner throws, all cloned elements are unlinked and disposed calling Disposer::operator()(pointer).

Complexity: Linear to erased plus inserted elements.

Throws: If cloner throws or predicate copy assignment throws. Basic guarantee.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

19.
```
std::pair< iterator, bool > insert(reference value);
```

Requires: value must be an lvalue

Effects: Tries to inserts value into the set.

Returns: If the value is not already present inserts it and returns a pair containing the iterator to the new value and true. If there is an equivalent value returns a pair containing an iterator to the already present value and false.

Complexity: Average complexity for insert element is at most logarithmic.

Throws: If the internal value_compare ordering function throws. Strong guarantee.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

20.
```
iterator insert(const_iterator hint, reference value);
```

Requires: value must be an lvalue

Effects: Tries to to insert x into the set, using "hint" as a hint to where it will be inserted.

Returns: An iterator that points to the position where the new element was inserted into the set.

Complexity: Logarithmic in general, but it's amortized constant time if t is inserted immediately before hint.

Throws: If the internal value_compare ordering function throws. Strong guarantee.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

21.
```
template<typename KeyType, typename KeyValueCompare>
  std::pair< iterator, bool >
  insert_check(const KeyType & key, KeyValueCompare key_value_comp,
               insert_commit_data & commit_data);
```

Requires: key_value_comp must be a comparison function that induces the same strict weak ordering as value_compare. The difference is that key_value_comp compares an arbitrary key with the contained values.

Effects: Checks if a value can be inserted in the set, using a user provided key instead of the value itself.

Returns: If there is an equivalent value returns a pair containing an iterator to the already present value and false. If the value can be inserted returns true in the returned pair boolean and fills "commit_data" that is meant to be used with the "insert_commit" function.

Complexity: Average complexity is at most logarithmic.

Throws: If the key_value_comp ordering function throws. Strong guarantee.

Notes: This function is used to improve performance when constructing a value_type is expensive: if there is an equivalent value the constructed object must be discarded. Many times, the part of the node that is used to impose the order is much cheaper to construct than the value_type and this function offers the possibility to use that part to check if the insertion will be successful.

If the check is successful, the user can construct the value_type and use "insert_commit" to insert the object in constant-time. This gives a total logarithmic complexity to the insertion: check(O(log(N)) + commit(O(1)).

"commit_data" remains valid for a subsequent "insert_commit" only if no more objects are inserted or erased from the set.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

22.
```
template<typename KeyType, typename KeyValueCompare>
  std::pair< iterator, bool >
  insert_check(const_iterator hint, const KeyType & key,
               KeyValueCompare key_value_comp,
               insert_commit_data & commit_data);
```

Requires: key_value_comp must be a comparison function that induces the same strict weak ordering as value_compare. The difference is that key_value_comp compares an arbitrary key with the contained values.

Effects: Checks if a value can be inserted in the set, using a user provided key instead of the value itself, using "hint" as a hint to where it will be inserted.

Returns: If there is an equivalent value returns a pair containing an iterator to the already present value and false. If the value can be inserted returns true in the returned pair boolean and fills "commit_data" that is meant to be used with the "insert_commit" function.

Complexity: Logarithmic in general, but it's amortized constant time if t is inserted immediately before hint.

Throws: If the key_value_comp ordering function throws. Strong guarantee.

Notes: This function is used to improve performance when constructing a value_type is expensive: if there is an equivalent value the constructed object must be discarded. Many times, the part of the constructing that is used to impose the order is much cheaper to construct than the value_type and this function offers the possibility to use that key to check if the insertion will be successful.

If the check is successful, the user can construct the value_type and use "insert_commit" to insert the object in constant-time. This can give a total constant-time complexity to the insertion: check(O(1)) + commit(O(1)).

"commit_data" remains valid for a subsequent "insert_commit" only if no more objects are inserted or erased from the set.

23.
```
iterator insert_commit(reference value,
                       const insert_commit_data & commit_data);
```

Requires: value must be an lvalue of type value_type. commit_data must have been obtained from a previous call to "insert_check". No objects should have been inserted or erased from the set between the "insert_check" that filled "commit_data" and the call to "insert_commit".

Effects: Inserts the value in the set using the information obtained from the "commit_data" that a previous "insert_check" filled.

Returns: An iterator to the newly inserted object.

Complexity: Constant time.

Throws: Nothing.

Notes: This function has only sense if a "insert_check" has been previously executed to fill "commit_data". No value should be inserted or erased between the "insert_check" and "insert_commit" calls.

24.
```
template<typename Iterator> void insert(Iterator b, Iterator e);
```

Requires: Dereferencing iterator must yield an lvalue of type value_type.

Effects: Inserts a range into the set.

Complexity: Insert range is in general O(N * log(N)), where N is the size of the range. However, it is linear in N if the range is already sorted by value_comp().

Throws: If the internal value_compare ordering function throws. Basic guarantee.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

25.
```
iterator insert_before(const_iterator pos, reference value);
```

Requires: value must be an lvalue, "pos" must be a valid iterator (or end) and must be the succesor of value once inserted according to the predicate. "value" must not be equal to any inserted key according to the predicate.

Effects: Inserts x into the tree before "pos".

Complexity: Constant time.

Throws: Nothing.

Note: This function does not check preconditions so if "pos" is not the successor of "value" or "value" is not unique tree ordering and uniqueness invariants will be broken respectively. This is a low-level function to be used only for performance reasons by advanced users.

26.
```
void push_back(reference value);
```

Requires: value must be an lvalue, and it must be greater than any inserted key according to the predicate.

Effects: Inserts x into the tree in the last position.

Complexity: Constant time.

Throws: Nothing.

Note: This function does not check preconditions so if value is less than or equal to the greatest inserted key tree ordering invariant will be broken. This function is slightly more efficient than using "insert_before". This is a low-level function to be used only for performance reasons by advanced users.

27.
```
void push_front(reference value);
```

Requires: value must be an lvalue, and it must be less than any inserted key according to the predicate.

Effects: Inserts x into the tree in the first position.

Complexity: Constant time.

Throws: Nothing.

Note: This function does not check preconditions so if value is greater than or equal to the the mimum inserted key tree ordering or uniqueness invariants will be broken. This function is slightly more efficient than using "insert_before". This is a low-level function to be used only for performance reasons by advanced users.

28.
```
iterator erase(const_iterator i);
```

Effects: Erases the element pointed to by pos.

Complexity: Average complexity is constant time.

Returns: An iterator to the element after the erased element.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

29.
```
iterator erase(const_iterator b, const_iterator e);
```

Effects: Erases the range pointed to by b end e.

Complexity: Average complexity for erase range is at most O(log(size() + N)), where N is the number of elements in the range.

Returns: An iterator to the element after the erased elements.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

30.
```
size_type erase(const_reference value);
```

Effects: Erases all the elements with the given value.

Returns: The number of erased elements.

Complexity: O(log(size()) + this->count(value)).

Throws: If the internal value_compare ordering function throws. Basic guarantee.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

31.
```
template<typename KeyType, typename KeyValueCompare>
  size_type erase(const KeyType & key, KeyValueCompare comp);
```

Effects: Erases all the elements that compare equal with the given key and the given comparison functor.

Returns: The number of erased elements.

Complexity: O(log(size() + this->count(key, comp)).

Throws: If the comp ordering function throws. Basic guarantee.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

32.
```
template<typename Disposer>
  iterator erase_and_dispose(const_iterator i, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases the element pointed to by pos. Disposer::operator()(pointer) is called for the removed element.

Complexity: Average complexity for erase element is constant time.

Returns: An iterator to the element after the erased element.

Throws: Nothing.

Note: Invalidates the iterators to the erased elements.

33.
```
template<typename Disposer>
  iterator erase_and_dispose(const_iterator b, const_iterator e,
                             Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Erases the range pointed to by b end e. Disposer::operator()(pointer) is called for the removed elements.

Complexity: Average complexity for erase range is at most O(log(size() + N)), where N is the number of elements in the range.

Returns: An iterator to the element after the erased elements.

Throws: Nothing.

Note: Invalidates the iterators to the erased elements.

34.
```
template<typename Disposer> 
  size_type erase_and_dispose(const_reference value, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases all the elements with the given value. Disposer::operator()(pointer) is called for the removed elements.

Throws: If the internal value_compare ordering function throws.

Complexity: O(log(size() + this->count(value)). Basic guarantee.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

35.
```
template<typename KeyType, typename KeyValueCompare, typename Disposer> 
  size_type erase_and_dispose(const KeyType & key, KeyValueCompare comp, 
                              Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases all the elements with the given key. according to the comparison functor "comp". Disposer::operator()(pointer) is called for the removed elements.

Returns: The number of erased elements.

Complexity: O(log(size() + this->count(key, comp)).

Throws: If comp ordering function throws. Basic guarantee.

Note: Invalidates the iterators to the erased elements.

36.
```
void clear();
```

Effects: Erases all the elements of the container.

Complexity: Linear to the number of elements on the container. if it's a safe-mode or auto-unlink value_type. Constant time otherwise.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

37.
```
template<typename Disposer> void clear_and_dispose(Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases all the elements of the container.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Linear to the number of elements on the container. Disposer::operator()(pointer) is called for the removed elements.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

38.
```
size_type count(const_reference value) const;
```

Effects: Returns the number of contained elements with the given key

Complexity: Logarithmic to the number of elements contained plus lineal to number of objects with the given key.

Throws: If the internal value_compare ordering function throws.

39.
```
template<typename KeyType, typename KeyValueCompare>
  size_type count(const KeyType & key, KeyValueCompare comp) const;
```

Effects: Returns the number of contained elements with the same key compared with the given comparison functor.

Complexity: Logarithmic to the number of elements contained plus lineal to number of objects with the given key.

Throws: If comp ordering function throws.

40.
```
iterator lower_bound(const_reference value);
```

Effects: Returns an iterator to the first element whose key is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

41.
```
template<typename KeyType, typename KeyValueCompare>
  iterator lower_bound(const KeyType & key, KeyValueCompare comp);
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Returns an iterator to the first element whose key according to the comparison functor is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

42.
```
const_iterator lower_bound(const_reference value) const;
```

Effects: Returns a const iterator to the first element whose key is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

43.
```
template<typename KeyType, typename KeyValueCompare> 
  const_iterator lower_bound(const KeyType & key, KeyValueCompare comp) const;
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Returns a const_iterator to the first element whose key according to the comparison functor is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

44.
```
iterator upper_bound(const_reference value);
```

Effects: Returns an iterator to the first element whose key is greater than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

45.
```
template<typename KeyType, typename KeyValueCompare> 
  iterator upper_bound(const KeyType & key, KeyValueCompare comp);
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Returns an iterator to the first element whose key according to the comparison functor is greater than key or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

46.
```
const_iterator upper_bound(const_reference value) const;
```

Effects: Returns an iterator to the first element whose key is greater than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

47.
```
template<typename KeyType, typename KeyValueCompare> 
  const_iterator upper_bound(const KeyType & key, KeyValueCompare comp) const;
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Returns a const_iterator to the first element whose key according to the comparison functor is greater than key or end() if that element does not exist.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

48.
```
iterator find(const_reference value);
```

Effects: Finds an iterator to the first element whose value is "value" or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

49.
```
template<typename KeyType, typename KeyValueCompare>
  iterator find(const KeyType & key, KeyValueCompare comp);
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Finds an iterator to the first element whose key is "key" according to the comparison functor or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

50.
```
const_iterator find(const_reference value) const;
```

Effects: Finds a const_iterator to the first element whose value is "value" or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

51.
```
template<typename KeyType, typename KeyValueCompare>
  const_iterator find(const KeyType & key, KeyValueCompare comp) const;
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Finds a const_iterator to the first element whose key is "key" according to the comparison functor or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

52.
```
std::pair< iterator, iterator > equal_range(const_reference value);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Finds a range containing all elements whose key is k or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

53.
```
template<typename KeyType, typename KeyValueCompare>
  std::pair< iterator, iterator >
  equal_range(const KeyType & key, KeyValueCompare comp);
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Finds a range containing all elements whose key is k according to the comparison functor or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

54.
```
std::pair< const_iterator, const_iterator >
equal_range(const_reference value) const;
```

Effects: Finds a range containing all elements whose key is k or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

55.
```
template<typename KeyType, typename KeyValueCompare>
  std::pair< const_iterator, const_iterator >
  equal_range(const KeyType & key, KeyValueCompare comp) const;
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Finds a range containing all elements whose key is k according to the comparison functor or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

56.
```
std::pair< iterator, iterator >
bounded_range(const_reference lower_value, const_reference upper_value,
              bool left_closed, bool right_closed);
```

Requires: 'lower_value' must not be greater than 'upper_value'. If 'lower_value' == 'upper_value', ('left_closed' || 'right_closed') must be false.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key) if left_closed, upper_bound(lower_key) otherwise

second = upper_bound(upper_key) if right_closed, lower_bound(upper_key) otherwise

Complexity: Logarithmic.

Throws: If the predicate throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_value and upper_value.

57.
```
template<typename KeyType, typename KeyValueCompare> 
  std::pair< iterator, iterator > 
  bounded_range(const KeyType & lower_key, const KeyType & upper_key, 
                KeyValueCompare comp, bool left_closed, bool right_closed);
```

Requires: KeyValueCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. 'lower_key' must not be greater than 'upper_key' according to 'comp'. If 'lower_key' == 'upper_key', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key, comp) if left_closed, upper_bound(lower_key, comp) otherwise

second = upper_bound(upper_key, comp) if right_closed, lower_bound(upper_key, comp) otherwise

Complexity: Logarithmic.

Throws: If "comp" throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_key and upper_key.

58.
```
std::pair< const_iterator, const_iterator > 
bounded_range(const_reference lower_value, const_reference upper_value, 
              bool left_closed, bool right_closed) const;
```

Requires: 'lower_value' must not be greater than 'upper_value'. If 'lower_value' == 'upper_value', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key) if left_closed, upper_bound(lower_key) otherwise

second = upper_bound(upper_key) if right_closed, lower_bound(upper_key) otherwise

Complexity: Logarithmic.

Throws: If the predicate throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_value and upper_value.

59.
```
template<typename KeyType, typename KeyValueCompare> 
  std::pair< const_iterator, const_iterator > 
  bounded_range(const KeyType & lower_key, const KeyType & upper_key, 
                KeyValueCompare comp, bool left_closed, bool right_closed) const;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Requires: KeyValueCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. 'lower_key' must not be greater than 'upper_key' according to 'comp'. If 'lower_key' == 'upper_key', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key, comp) if left_closed, upper_bound(lower_key, comp) otherwise

second = upper_bound(upper_key, comp) if right_closed, lower_bound(upper_key, comp) otherwise

Complexity: Logarithmic.

Throws: If "comp" throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_key and upper_key.

60.
```
iterator iterator_to(reference value);
```

Requires: value must be an lvalue and shall be in a set of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid iterator i belonging to the set that points to the value

Complexity: Constant.

Throws: Nothing.

61.
```
const_iterator iterator_to(const_reference value) const;
```

Requires: value must be an lvalue and shall be in a set of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid const_iterator i belonging to the set that points to the value

Complexity: Constant.

Throws: Nothing.

62.
```
pointer unlink_leftmost_without_rebalance();
```

Effects: Unlinks the leftmost node from the tree.

Complexity: Average complexity is constant time.

Throws: Nothing.

Notes: This function breaks the tree and the tree can only be used for more unlink_leftmost_without_rebalance calls. This function is normally used to achieve a step by step controlled destruction of the tree.

63.
```
void replace_node(iterator replace_this, reference with_this);
```

Requires: replace_this must be a valid iterator of *this and with_this must not be inserted in any tree.

Effects: Replaces replace_this in its position in the tree with with_this. The tree does not need to be rebalanced.

Complexity: Constant.

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Note: This function will break container ordering invariants if with_this is not equivalent to *replace_this according to the ordering rules. This function is faster than erasing and inserting the node, since no rebalancing or comparison is needed.

`set` public static functions

1.
```
static set & container_from_end_iterator(iterator end_iterator);
```

   Precondition: end_iterator must be a valid end iterator of set.

   Effects: Returns a reference to the set associated to the end iterator

   Throws: Nothing.

   Complexity: Constant.

2.
```
static const set & container_from_end_iterator(const_iterator end_iterator);
```

   Precondition: end_iterator must be a valid end const_iterator of set.

   Effects: Returns a const reference to the set associated to the end iterator

   Throws: Nothing.

   Complexity: Constant.

3.
```
static set & container_from_iterator(iterator it);
```

   Precondition: it must be a valid iterator of set.

   Effects: Returns a reference to the set associated to the iterator

   Throws: Nothing.

   Complexity: Logarithmic.

4.
```
static const set & container_from_iterator(const_iterator it);
```

   Precondition: it must be a valid const_iterator of set.

   Effects: Returns a const reference to the set associated to the iterator

   Throws: Nothing.

   Complexity: Logarithmic.

5.
```
static iterator s_iterator_to(reference value);
```

   Requires: value must be an lvalue and shall be in a set of appropriate type. Otherwise the behavior is undefined.

   Effects: Returns: a valid iterator i belonging to the set that points to the value

   Complexity: Constant.

   Throws: Nothing.

   Note: This static function is available only if the *value traits* is stateless.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

6.
```
static const_iterator s_iterator_to(const_reference value);
```

Requires: value must be an lvalue and shall be in a set of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid const_iterator i belonging to the set that points to the value

Complexity: Constant.

Throws: Nothing.

Note: This static function is available only if the *value traits* is stateless.

7.
```
static void init_node(reference value);
```

Requires: value shall not be in a set/multiset.

Effects: init_node puts the hook of a value in a well-known default state.

Throws: Nothing.

Complexity: Constant time.

Note: This function puts the hook in the well-known default state used by auto_unlink and safe hooks.

Struct template make_set

boost::intrusive::make_set

Synopsis

```
// In header: <boost/intrusive/set.hpp>

template<typename T, class... Options>
struct make_set {
  // types
  typedef implementation_defined type;
};
```

Description

Helper metafunction to define a `set` that yields to the same type when the same options (either explicitly or implicitly) are used.

Class template multiset

boost::intrusive::multiset

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/intrusive/set.hpp>

template<typename T, class... Options>
class multiset {
public:
  // types
  typedef implementation_defined::value_type             value_type;
  typedef implementation_defined::value_traits           value_traits;
  typedef implementation_defined::pointer                pointer;
  typedef implementation_defined::const_pointer          const_pointer;
  typedef implementation_defined::reference              reference;
  typedef implementation_defined::const_reference        const_reference;
  typedef implementation_defined::difference_type        difference_type;
  typedef implementation_defined::size_type              size_type;
  typedef implementation_defined::value_compare          value_compare;
  typedef implementation_defined::key_compare            key_compare;
  typedef implementation_defined::iterator               iterator;
  typedef implementation_defined::const_iterator         const_iterator;
  typedef implementation_defined::reverse_iterator       reverse_iterator;
  typedef implementation_defined::const_reverse_iterator const_reverse_iterator;
  typedef implementation_defined::insert_commit_data     insert_commit_data;
  typedef implementation_defined::node_traits            node_traits;
  typedef implementation_defined::node                   node;
  typedef implementation_defined::node_ptr               node_ptr;
  typedef implementation_defined::const_node_ptr         const_node_ptr;
  typedef implementation_defined::node_algorithms        node_algorithms;

  // construct/copy/destruct
  explicit multiset(const value_compare & = value_compare(),
                    const value_traits & = value_traits());
  template<typename Iterator>
    multiset(Iterator, Iterator, const value_compare & = value_compare(),
             const value_traits & = value_traits());
  multiset(BOOST_RV_REF(multiset));
  multiset& operator=(BOOST_RV_REF(multiset));
  ~multiset();

  // public member functions
  iterator begin();
  const_iterator begin() const;
  const_iterator cbegin() const;
  iterator end();
  const_iterator end() const;
  const_iterator cend() const;
  reverse_iterator rbegin();
  const_reverse_iterator rbegin() const;
  const_reverse_iterator crbegin() const;
  reverse_iterator rend();
  const_reverse_iterator rend() const;
  const_reverse_iterator crend() const;
  key_compare key_comp() const;
  value_compare value_comp() const;
  bool empty() const;
  size_type size() const;
  void swap(multiset &);
  template<typename Cloner, typename Disposer>
    void clone_from(const multiset &, Cloner, Disposer);
  iterator insert(reference);
  iterator insert(const_iterator, reference);
  template<typename Iterator> void insert(Iterator, Iterator);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
iterator insert_before(const_iterator, reference);
void push_back(reference);
void push_front(reference);
iterator erase(const_iterator);
iterator erase(const_iterator, iterator);
size_type erase(const_reference);
template<typename KeyType, typename KeyValueCompare>
  size_type erase(const KeyType &, KeyValueCompare);
template<typename Disposer>
  iterator erase_and_dispose(const_iterator, Disposer);
template<typename Disposer>
  iterator erase_and_dispose(const_iterator, const_iterator, Disposer);
template<typename Disposer>
  size_type erase_and_dispose(const_reference, Disposer);
template<typename KeyType, typename KeyValueCompare, typename Disposer>
  size_type erase_and_dispose(const KeyType &, KeyValueCompare, Disposer);
void clear();
template<typename Disposer> void clear_and_dispose(Disposer);
size_type count(const_reference) const;
template<typename KeyType, typename KeyValueCompare>
  size_type count(const KeyType &, KeyValueCompare) const;
iterator lower_bound(const_reference);
template<typename KeyType, typename KeyValueCompare>
  iterator lower_bound(const KeyType &, KeyValueCompare);
const_iterator lower_bound(const_reference) const;
template<typename KeyType, typename KeyValueCompare>
  const_iterator lower_bound(const KeyType &, KeyValueCompare) const;
iterator upper_bound(const_reference);
template<typename KeyType, typename KeyValueCompare>
  iterator upper_bound(const KeyType &, KeyValueCompare);
const_iterator upper_bound(const_reference) const;
template<typename KeyType, typename KeyValueCompare>
  const_iterator upper_bound(const KeyType &, KeyValueCompare) const;
iterator find(const_reference);
template<typename KeyType, typename KeyValueCompare>
  iterator find(const KeyType &, KeyValueCompare);
const_iterator find(const_reference) const;
template<typename KeyType, typename KeyValueCompare>
  const_iterator find(const KeyType &, KeyValueCompare) const;
std::pair< iterator, iterator > equal_range(const_reference);
template<typename KeyType, typename KeyValueCompare>
  std::pair< iterator, iterator >
  equal_range(const KeyType &, KeyValueCompare);
std::pair< const_iterator, const_iterator >
equal_range(const_reference) const;
template<typename KeyType, typename KeyValueCompare>
  std::pair< const_iterator, const_iterator >
  equal_range(const KeyType &, KeyValueCompare) const;
std::pair< iterator, iterator >
bounded_range(const_reference, const_reference, bool, bool);
template<typename KeyType, typename KeyValueCompare>
  std::pair< iterator, iterator >
  bounded_range(const KeyType &, const KeyType &, KeyValueCompare, bool,
                bool);
std::pair< const_iterator, const_iterator >
bounded_range(const_reference, const_reference, bool, bool) const;
template<typename KeyType, typename KeyValueCompare>
  std::pair< const_iterator, const_iterator >
  bounded_range(const KeyType &, const KeyType &, KeyValueCompare, bool,
                bool) const;
iterator iterator_to(reference);
const_iterator iterator_to(const_reference) const;
pointer unlink_leftmost_without_rebalance();
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
  void replace_node(iterator, reference);

  // public static functions
  static multiset & container_from_end_iterator(iterator);
  static const multiset & container_from_end_iterator(const_iterator);
  static multiset & container_from_iterator(iterator);
  static const multiset & container_from_iterator(const_iterator);
  static iterator s_iterator_to(reference);
  static const_iterator s_iterator_to(const_reference);
  static void init_node(reference);
  static void remove_node(reference);

  // public data members
  static const bool constant_time_size;
};
```

Description

The class template multiset is an intrusive container, that mimics most of the interface of std::multiset as described in the C++ standard.

The template parameter `T` is the type to be managed by the container. The user can specify additional options and if no options are provided default options are used.

The container supports the following options: `base_hook<>/member_hook<>/value_traits<>`, `constant_time_size<>`, `size_type<>` and `compare<>`.

`multiset` public construct/copy/destruct

1.
```
explicit multiset(const value_compare & cmp = value_compare(),
                  const value_traits & v_traits = value_traits());
```

Effects: Constructs an empty multiset.

Complexity: Constant.

Throws: If value_traits::node_traits::node constructor throws (this does not happen with predefined Boost.Intrusive hooks) or the copy constructor/operator() of the value_compare object throws.

2.
```
template<typename Iterator>
  multiset(Iterator b, Iterator e,
           const value_compare & cmp = value_compare(),
           const value_traits & v_traits = value_traits());
```

Requires: Dereferencing iterator must yield an lvalue of type value_type. cmp must be a comparison function that induces a strict weak ordering.

Effects: Constructs an empty multiset and inserts elements from [b, e).

Complexity: Linear in N if [b, e) is already sorted using comp and otherwise N * log N, where N is the distance between first and last

Throws: If value_traits::node_traits::node constructor throws (this does not happen with predefined Boost.Intrusive hooks) or the copy constructor/operator() of the value_compare object throws.

3.
```
multiset(BOOST_RV_REF(multiset) x);
```

Effects: to-do

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

4.
```
multiset& operator=(BOOST_RV_REF(multiset) x);
```

Effects: to-do

5.
```
~multiset();
```

Effects: Detaches all elements from this. The objects in the set are not deleted (i.e. no destructors are called).

Complexity: Linear to the number of elements on the container. if it's a safe-mode or auto-unlink value_type. Constant time otherwise.

Throws: Nothing.

`multiset` public member functions

1.
```
iterator begin();
```

Effects: Returns an iterator pointing to the beginning of the multiset.

Complexity: Constant.

Throws: Nothing.

2.
```
const_iterator begin() const;
```

Effects: Returns a const_iterator pointing to the beginning of the multiset.

Complexity: Constant.

Throws: Nothing.

3.
```
const_iterator cbegin() const;
```

Effects: Returns a const_iterator pointing to the beginning of the multiset.

Complexity: Constant.

Throws: Nothing.

4.
```
iterator end();
```

Effects: Returns an iterator pointing to the end of the multiset.

Complexity: Constant.

Throws: Nothing.

5.
```
const_iterator end() const;
```

Effects: Returns a const_iterator pointing to the end of the multiset.

Complexity: Constant.

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

6.
```
const_iterator cend() const;
```

Effects: Returns a const_iterator pointing to the end of the multiset.

Complexity: Constant.

Throws: Nothing.

7.
```
reverse_iterator rbegin();
```

Effects: Returns a reverse_iterator pointing to the beginning of the reversed multiset.

Complexity: Constant.

Throws: Nothing.

8.
```
const_reverse_iterator rbegin() const;
```

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed multiset.

Complexity: Constant.

Throws: Nothing.

9.
```
const_reverse_iterator crbegin() const;
```

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed multiset.

Complexity: Constant.

Throws: Nothing.

10.
```
reverse_iterator rend();
```

Effects: Returns a reverse_iterator pointing to the end of the reversed multiset.

Complexity: Constant.

Throws: Nothing.

11.
```
const_reverse_iterator rend() const;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed multiset.

Complexity: Constant.

Throws: Nothing.

12.
```
const_reverse_iterator crend() const;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed multiset.

Complexity: Constant.

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

13.
```
key_compare key_comp() const;
```

Effects: Returns the key_compare object used by the multiset.

Complexity: Constant.

Throws: If key_compare copy-constructor throws.

14.
```
value_compare value_comp() const;
```

Effects: Returns the value_compare object used by the multiset.

Complexity: Constant.

Throws: If value_compare copy-constructor throws.

15.
```
bool empty() const;
```

Effects: Returns true if the container is empty.

Complexity: Constant.

Throws: Nothing.

16.
```
size_type size() const;
```

Effects: Returns the number of elements stored in the multiset.

Complexity: Linear to elements contained in *this if, constant-time size option is enabled. Constant-time otherwise.

Throws: Nothing.

17.
```
void swap(multiset & other);
```

Effects: Swaps the contents of two multisets.

Complexity: Constant.

Throws: If the swap() call for the comparison functor found using ADL throws. Strong guarantee.

18.
```
template<typename Cloner, typename Disposer>
  void clone_from(const multiset & src, Cloner cloner, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw. Cloner should yield to nodes equivalent to the original nodes.

Effects: Erases all the elements from *this calling Disposer::operator()(pointer), clones all the elements from src calling Cloner::operator()(const_reference) and inserts them on *this. Copies the predicate from the source container.

If cloner throws, all cloned elements are unlinked and disposed calling Disposer::operator()(pointer).

Complexity: Linear to erased plus inserted elements.

Throws: If cloner throws or predicate copy assignment throws. Basic guarantee.

19.
```
iterator insert(reference value);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Requires: value must be an lvalue

Effects: Inserts value into the multiset.

Returns: An iterator that points to the position where the new element was inserted.

Complexity: Average complexity for insert element is at most logarithmic.

Throws: If the internal value_compare ordering function throws. Strong guarantee.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

20.
```
iterator insert(const_iterator hint, reference value);
```

Requires: value must be an lvalue

Effects: Inserts x into the multiset, using pos as a hint to where it will be inserted.

Returns: An iterator that points to the position where the new element was inserted.

Complexity: Logarithmic in general, but it is amortized constant time if t is inserted immediately before hint.

Throws: If the internal value_compare ordering function throws. Strong guarantee.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

21.
```
template<typename Iterator> void insert(Iterator b, Iterator e);
```

Requires: Dereferencing iterator must yield an lvalue of type value_type.

Effects: Inserts a range into the multiset.

Returns: An iterator that points to the position where the new element was inserted.

Complexity: Insert range is in general O(N * log(N)), where N is the size of the range. However, it is linear in N if the range is already sorted by value_comp().

Throws: If the internal value_compare ordering function throws. Basic guarantee.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

22.
```
iterator insert_before(const_iterator pos, reference value);
```

Requires: value must be an lvalue, "pos" must be a valid iterator (or end) and must be the succesor of value once inserted according to the predicate

Effects: Inserts x into the tree before "pos".

Complexity: Constant time.

Throws: Nothing.

Note: This function does not check preconditions so if "pos" is not the successor of "value" tree ordering invariant will be broken. This is a low-level function to be used only for performance reasons by advanced users.

23.
```
void push_back(reference value);
```

Requires: value must be an lvalue, and it must be no less than the greatest inserted key

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Inserts x into the tree in the last position.

Complexity: Constant time.

Throws: Nothing.

Note: This function does not check preconditions so if value is less than the greatest inserted key tree ordering invariant will be broken. This function is slightly more efficient than using "insert_before". This is a low-level function to be used only for performance reasons by advanced users.

24.
```
void push_front(reference value);
```

Requires: value must be an lvalue, and it must be no greater than the minimum inserted key

Effects: Inserts x into the tree in the first position.

Complexity: Constant time.

Throws: Nothing.

Note: This function does not check preconditions so if value is greater than the minimum inserted key tree ordering invariant will be broken. This function is slightly more efficient than using "insert_before". This is a low-level function to be used only for performance reasons by advanced users.

25.
```
iterator erase(const_iterator i);
```

Effects: Erases the element pointed to by pos.

Complexity: Average complexity is constant time.

Returns: An iterator to the element after the erased element.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

26.
```
iterator erase(const_iterator b, iterator e);
```

Effects: Erases the range pointed to by b end e.

Returns: An iterator to the element after the erased elements.

Complexity: Average complexity for erase range is at most O(log(size() + N)), where N is the number of elements in the range.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

27.
```
size_type erase(const_reference value);
```

Effects: Erases all the elements with the given value.

Returns: The number of erased elements.

Complexity: O(log(size() + this->count(value)).

Throws: If the internal value_compare ordering function throws. Basic guarantee.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

28.
```
template<typename KeyType, typename KeyValueCompare> 
  size_type erase(const KeyType & key, KeyValueCompare comp);
```

Effects: Erases all the elements that compare equal with the given key and the given comparison functor.

Returns: The number of erased elements.

Complexity: O(log(size() + this->count(key, comp)).

Throws: If comp ordering function throws. Basic guarantee.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

29.
```
template<typename Disposer> 
  iterator erase_and_dispose(const_iterator i, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Returns: An iterator to the element after the erased element.

Effects: Erases the element pointed to by pos. Disposer::operator()(pointer) is called for the removed element.

Complexity: Average complexity for erase element is constant time.

Throws: Nothing.

Note: Invalidates the iterators to the erased elements.

30.
```
template<typename Disposer> 
  iterator erase_and_dispose(const_iterator b, const_iterator e, 
                             Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Returns: An iterator to the element after the erased elements.

Effects: Erases the range pointed to by b end e. Disposer::operator()(pointer) is called for the removed elements.

Complexity: Average complexity for erase range is at most O(log(size() + N)), where N is the number of elements in the range.

Throws: Nothing.

Note: Invalidates the iterators to the erased elements.

31.
```
template<typename Disposer> 
  size_type erase_and_dispose(const_reference value, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases all the elements with the given value. Disposer::operator()(pointer) is called for the removed elements.

Returns: The number of erased elements.

Complexity: O(log(size() + this->count(value)).

Throws: If the internal value_compare ordering function throws. Basic guarantee.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

32.
```
template<typename KeyType, typename KeyValueCompare, typename Disposer> 
  size_type erase_and_dispose(const KeyType & key, KeyValueCompare comp, 
                              Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases all the elements with the given key. according to the comparison functor "comp". Disposer::operator()(pointer) is called for the removed elements.

Returns: The number of erased elements.

Complexity: O(log(size() + this->count(key, comp)).

Throws: If comp ordering function throws. Basic guarantee.

Note: Invalidates the iterators to the erased elements.

33.
```
void clear();
```

Effects: Erases all the elements of the container.

Complexity: Linear to the number of elements on the container. if it's a safe-mode or auto-unlink value_type. Constant time otherwise.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

34.
```
template<typename Disposer> void clear_and_dispose(Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases all the elements of the container.

Complexity: Linear to the number of elements on the container. Disposer::operator()(pointer) is called for the removed elements.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

35.
```
size_type count(const_reference value) const;
```

Effects: Returns the number of contained elements with the given key

Complexity: Logarithmic to the number of elements contained plus lineal to number of objects with the given key.

Throws: If the internal value_compare ordering function throws.

36.
```
template<typename KeyType, typename KeyValueCompare> 
  size_type count(const KeyType & key, KeyValueCompare comp) const;
```

Effects: Returns the number of contained elements with the same key compared with the given comparison functor.

Complexity: Logarithmic to the number of elements contained plus lineal to number of objects with the given key.

Throws: If comp ordering function throws.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

37.
```
iterator lower_bound(const_reference value);
```

Effects: Returns an iterator to the first element whose key is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

38.
```
template<typename KeyType, typename KeyValueCompare> 
  iterator lower_bound(const KeyType & key, KeyValueCompare comp);
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Returns an iterator to the first element whose key according to the comparison functor is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

39.
```
const_iterator lower_bound(const_reference value) const;
```

Effects: Returns a const iterator to the first element whose key is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

40.
```
template<typename KeyType, typename KeyValueCompare> 
  const_iterator lower_bound(const KeyType & key, KeyValueCompare comp) const;
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Returns a const_iterator to the first element whose key according to the comparison functor is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

41.
```
iterator upper_bound(const_reference value);
```

Effects: Returns an iterator to the first element whose key is greater than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

42.
```
template<typename KeyType, typename KeyValueCompare>
  iterator upper_bound(const KeyType & key, KeyValueCompare comp);
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Returns an iterator to the first element whose key according to the comparison functor is greater than key or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

43.
```
const_iterator upper_bound(const_reference value) const;
```

Effects: Returns an iterator to the first element whose key is greater than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

44.
```
template<typename KeyType, typename KeyValueCompare>
  const_iterator upper_bound(const KeyType & key, KeyValueCompare comp) const;
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Returns a const_iterator to the first element whose key according to the comparison functor is greater than key or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

45.
```
iterator find(const_reference value);
```

Effects: Finds an iterator to the first element whose value is "value" or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

46.
```
template<typename KeyType, typename KeyValueCompare>
  iterator find(const KeyType & key, KeyValueCompare comp);
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Finds an iterator to the first element whose key is "key" according to the comparison functor or end() if that element does not exist.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

47.
```
const_iterator find(const_reference value) const;
```

Effects: Finds a const_iterator to the first element whose value is "value" or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

48.
```
template<typename KeyType, typename KeyValueCompare>
  const_iterator find(const KeyType & key, KeyValueCompare comp) const;
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Finds a const_iterator to the first element whose key is "key" according to the comparison functor or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

49.
```
std::pair< iterator, iterator > equal_range(const_reference value);
```

Effects: Finds a range containing all elements whose key is k or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

50.
```
template<typename KeyType, typename KeyValueCompare>
  std::pair< iterator, iterator >
  equal_range(const KeyType & key, KeyValueCompare comp);
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Finds a range containing all elements whose key is k according to the comparison functor or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

51.
```
std::pair< const_iterator, const_iterator >
equal_range(const_reference value) const;
```

Effects: Finds a range containing all elements whose key is k or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

52.
```
template<typename KeyType, typename KeyValueCompare>
  std::pair< const_iterator, const_iterator >
  equal_range(const KeyType & key, KeyValueCompare comp) const;
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Finds a range containing all elements whose key is k according to the comparison functor or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

53.
```
std::pair< iterator, iterator >
bounded_range(const_reference lower_value, const_reference upper_value,
              bool left_closed, bool right_closed);
```

Requires: 'lower_value' must not be greater than 'upper_value'. If 'lower_value' == 'upper_value', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key) if left_closed, upper_bound(lower_key) otherwise

second = upper_bound(upper_key) if right_closed, lower_bound(upper_key) otherwise

Complexity: Logarithmic.

Throws: If the predicate throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_value and upper_value.

54.
```
template<typename KeyType, typename KeyValueCompare>
  std::pair< iterator, iterator >
  bounded_range(const KeyType & lower_key, const KeyType & upper_key,
                KeyValueCompare comp, bool left_closed, bool right_closed);
```

Requires: KeyValueCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. 'lower_key' must not be greater than 'upper_key' according to 'comp'. If 'lower_key' == 'upper_key', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key, comp) if left_closed, upper_bound(lower_key, comp) otherwise

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

second = upper_bound(upper_key, comp) if right_closed, lower_bound(upper_key, comp) otherwise

Complexity: Logarithmic.

Throws: If "comp" throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_key and upper_key.

55.
```
std::pair< const_iterator, const_iterator >
bounded_range(const_reference lower_value, const_reference upper_value,
              bool left_closed, bool right_closed) const;
```

Requires: 'lower_value' must not be greater than 'upper_value'. If 'lower_value' == 'upper_value', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key) if left_closed, upper_bound(lower_key) otherwise

second = upper_bound(upper_key) if right_closed, lower_bound(upper_key) otherwise

Complexity: Logarithmic.

Throws: If the predicate throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_value and upper_value.

56.
```
template<typename KeyType, typename KeyValueCompare>
  std::pair< const_iterator, const_iterator >
  bounded_range(const KeyType & lower_key, const KeyType & upper_key,
                KeyValueCompare comp, bool left_closed, bool right_closed) const;
```

Requires: KeyValueCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. 'lower_key' must not be greater than 'upper_key' according to 'comp'. If 'lower_key' == 'upper_key', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key, comp) if left_closed, upper_bound(lower_key, comp) otherwise

second = upper_bound(upper_key, comp) if right_closed, lower_bound(upper_key, comp) otherwise

Complexity: Logarithmic.

Throws: If "comp" throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_key and upper_key.

57.
```
iterator iterator_to(reference value);
```

Requires: value must be an lvalue and shall be in a set of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid iterator i belonging to the set that points to the value

Complexity: Constant.

Throws: Nothing.

58.
```
const_iterator iterator_to(const_reference value) const;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Requires: value must be an lvalue and shall be in a set of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid const_iterator i belonging to the set that points to the value

Complexity: Constant.

Throws: Nothing.

59.
```
pointer unlink_leftmost_without_rebalance();
```

Effects: Unlinks the leftmost node from the tree.

Complexity: Average complexity is constant time.

Throws: Nothing.

Notes: This function breaks the tree and the tree can only be used for more unlink_leftmost_without_rebalance calls. This function is normally used to achieve a step by step controlled destruction of the tree.

60.
```
void replace_node(iterator replace_this, reference with_this);
```

Requires: replace_this must be a valid iterator of *this and with_this must not be inserted in any tree.

Effects: Replaces replace_this in its position in the tree with with_this. The tree does not need to be rebalanced.

Complexity: Constant.

Throws: Nothing.

Note: This function will break container ordering invariants if with_this is not equivalent to *replace_this according to the ordering rules. This function is faster than erasing and inserting the node, since no rebalancing or comparison is needed.

`multiset` public static functions

1.
```
static multiset & container_from_end_iterator(iterator end_iterator);
```

Precondition: end_iterator must be a valid end iterator of multiset.

Effects: Returns a const reference to the multiset associated to the end iterator

Throws: Nothing.

Complexity: Constant.

2.
```
static const multiset &
container_from_end_iterator(const_iterator end_iterator);
```

Precondition: end_iterator must be a valid end const_iterator of multiset.

Effects: Returns a const reference to the multiset associated to the end iterator

Throws: Nothing.

Complexity: Constant.

3.
```
static multiset & container_from_iterator(iterator it);
```

Precondition: it must be a valid iterator of multiset.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Returns a const reference to the multiset associated to the iterator

Throws: Nothing.

Complexity: Logarithmic.

4.
```
static const multiset & container_from_iterator(const_iterator it);
```

Precondition: it must be a valid const_iterator of multiset.

Effects: Returns a const reference to the multiset associated to the iterator

Throws: Nothing.

Complexity: Logarithmic.

5.
```
static iterator s_iterator_to(reference value);
```

Requires: value must be an lvalue and shall be in a set of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid iterator i belonging to the set that points to the value

Complexity: Constant.

Throws: Nothing.

Note: This static function is available only if the *value traits* is stateless.

6.
```
static const_iterator s_iterator_to(const_reference value);
```

Requires: value must be an lvalue and shall be in a set of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid const_iterator i belonging to the set that points to the value

Complexity: Constant.

Throws: Nothing.

Note: This static function is available only if the *value traits* is stateless.

7.
```
static void init_node(reference value);
```

Requires: value shall not be in a set/multiset.

Effects: init_node puts the hook of a value in a well-known default state.

Throws: Nothing.

Complexity: Constant time.

Note: This function puts the hook in the well-known default state used by auto_unlink and safe hooks.

8.
```
static void remove_node(reference value);
```

Effects: removes "value" from the container.

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Logarithmic time.

Note: This static function is only usable with non-constant time size containers that have stateless comparison functors.

If the user calls this function with a constant time size container or stateful comparison functor a compilation error will be issued.

Struct template make_multiset

boost::intrusive::make_multiset

Synopsis

```
// In header: <boost/intrusive/set.hpp>

template<typename T, class... Options>
struct make_multiset {
  // types
  typedef implementation_defined type;
};
```

Description

Helper metafunction to define a `multiset` that yields to the same type when the same options (either explicitly or implicitly) are used.

Header <boost/intrusive/set_hook.hpp>

```
namespace boost {
  namespace intrusive {
    template<class... Options> struct make_set_base_hook;

    template<class... Options> class set_base_hook;

    template<class... Options> struct make_set_member_hook;

    template<class... Options> class set_member_hook;
  }
}
```

Struct template make_set_base_hook

boost::intrusive::make_set_base_hook

Synopsis

```
// In header: <boost/intrusive/set_hook.hpp>

template<class... Options>
struct make_set_base_hook {
  // types
  typedef implementation_defined type;
};
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Description

Helper metafunction to define a `set_base_hook` that yields to the same type when the same options (either explicitly or implicitly) are used.

Class template set_base_hook

boost::intrusive::set_base_hook

Synopsis

```
// In header: <boost/intrusive/set_hook.hpp>

template<class... Options>
class set_base_hook : public make_set_base_hook::type< O1, O2, O3, O4 > {
public:
  // construct/copy/destruct
  set_base_hook();
  set_base_hook(const set_base_hook &);
  set_base_hook& operator=(const set_base_hook &);
  ~set_base_hook();

  // public member functions
  void swap_nodes(set_base_hook &);
  bool is_linked() const;
  void unlink();
};
```

Description

Derive a class from set_base_hook in order to store objects in in a set/multiset. set_base_hook holds the data necessary to maintain the set/multiset and provides an appropriate value_traits class for set/multiset.

The hook admits the following options: `tag<>`, `void_pointer<>`, `link_mode<>` and `optimize_size<>`.

`tag<>` defines a tag to identify the node. The same tag value can be used in different classes, but if a class is derived from more than one `list_base_hook`, then each `list_base_hook` needs its unique tag.

`void_pointer<>` is the pointer type that will be used internally in the hook and the the container configured to use this hook.

`link_mode<>` will specify the linking mode of the hook (`normal_link`, `auto_unlink` or `safe_link`).

`optimize_size<>` will tell the hook to optimize the hook for size instead of speed.

`set_base_hook` public construct/copy/destruct

1.
   ```
   set_base_hook();
   ```

   Effects: If `link_mode` is `auto_unlink` or `safe_link` initializes the node to an unlinked state.

   Throws: Nothing.

2.
   ```
   set_base_hook(const set_base_hook &);
   ```

   Effects: If `link_mode` is `auto_unlink` or `safe_link` initializes the node to an unlinked state. The argument is ignored.

   Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Rationale: Providing a copy-constructor makes classes using the hook STL-compliant without forcing the user to do some additional work. swap can be used to emulate move-semantics.

3.
```
set_base_hook& operator=(const set_base_hook &);
```

Effects: Empty function. The argument is ignored.

Throws: Nothing.

Rationale: Providing an assignment operator makes classes using the hook STL-compliant without forcing the user to do some additional work. swap can be used to emulate move-semantics.

4.
```
~set_base_hook();
```

Effects: If link_mode is normal_link, the destructor does nothing (ie. no code is generated). If link_mode is safe_link and the object is stored in a set an assertion is raised. If link_mode is auto_unlink and is_linked() is true, the node is unlinked.

Throws: Nothing.

set_base_hook public member functions

1.
```
void swap_nodes(set_base_hook & other);
```

Effects: Swapping two nodes swaps the position of the elements related to those nodes in one or two containers. That is, if the node this is part of the element e1, the node x is part of the element e2 and both elements are included in the containers s1 and s2, then after the swap-operation e1 is in s2 at the position of e2 and e2 is in s1 at the position of e1. If one element is not in a container, then after the swap-operation the other element is not in a container. Iterators to e1 and e2 related to those nodes are invalidated.

Complexity: Constant

Throws: Nothing.

2.
```
bool is_linked() const;
```

Precondition: link_mode must be safe_link or auto_unlink.

Returns: true, if the node belongs to a container, false otherwise. This function can be used to test whether set::iterator_to will return a valid iterator.

Complexity: Constant

3.
```
void unlink();
```

Effects: Removes the node if it's inserted in a container. This function is only allowed if link_mode is auto_unlink.

Throws: Nothing.

Struct template make_set_member_hook

boost::intrusive::make_set_member_hook

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/intrusive/set_hook.hpp>

template<class... Options>
struct make_set_member_hook {
  // types
  typedef implementation_defined type;
};
```

Description

Helper metafunction to define a `set_member_hook` that yields to the same type when the same options (either explicitly or implicitly) are used.

Class template set_member_hook

boost::intrusive::set_member_hook

Synopsis

```
// In header: <boost/intrusive/set_hook.hpp>

template<class... Options>
class set_member_hook : public make_set_member_hook::type< O1, O2, O3, O4 > {
public:
  // construct/copy/destruct
  set_member_hook();
  set_member_hook(const set_member_hook &);
  set_member_hook& operator=(const set_member_hook &);
  ~set_member_hook();

  // public member functions
  void swap_nodes(set_member_hook &);
  bool is_linked() const;
  void unlink();
};
```

Description

Put a public data member set_member_hook in order to store objects of this class in a set/multiset. set_member_hook holds the data necessary for maintaining the set/multiset and provides an appropriate value_traits class for set/multiset.

The hook admits the following options: `void_pointer<>`, `link_mode<>` and `optimize_size<>`.

`void_pointer<>` is the pointer type that will be used internally in the hook and the the container configured to use this hook.

`link_mode<>` will specify the linking mode of the hook (`normal_link`, `auto_unlink` or `safe_link`).

`optimize_size<>` will tell the hook to optimize the hook for size instead of speed.

`set_member_hook` public construct/copy/destruct

1.
```
set_member_hook();
```

   Effects: If `link_mode` is `auto_unlink` or `safe_link` initializes the node to an unlinked state.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: Nothing.

2.
```
set_member_hook(const set_member_hook &);
```

Effects: If `link_mode` is `auto_unlink` or `safe_link` initializes the node to an unlinked state. The argument is ignored.

Throws: Nothing.

Rationale: Providing a copy-constructor makes classes using the hook STL-compliant without forcing the user to do some additional work. `swap` can be used to emulate move-semantics.

3.
```
set_member_hook& operator=(const set_member_hook &);
```

Effects: Empty function. The argument is ignored.

Throws: Nothing.

Rationale: Providing an assignment operator makes classes using the hook STL-compliant without forcing the user to do some additional work. `swap` can be used to emulate move-semantics.

4.
```
~set_member_hook();
```

Effects: If `link_mode` is `normal_link`, the destructor does nothing (ie. no code is generated). If `link_mode` is `safe_link` and the object is stored in a set an assertion is raised. If `link_mode` is `auto_unlink` and `is_linked()` is true, the node is unlinked.

Throws: Nothing.

`set_member_hook` public member functions

1.
```
void swap_nodes(set_member_hook & other);
```

Effects: Swapping two nodes swaps the position of the elements related to those nodes in one or two containers. That is, if the node this is part of the element e1, the node x is part of the element e2 and both elements are included in the containers s1 and s2, then after the swap-operation e1 is in s2 at the position of e2 and e2 is in s1 at the position of e1. If one element is not in a container, then after the swap-operation the other element is not in a container. Iterators to e1 and e2 related to those nodes are invalidated.

Complexity: Constant

Throws: Nothing.

2.
```
bool is_linked() const;
```

Precondition: `link_mode` must be `safe_link` or `auto_unlink`.

Returns: true, if the node belongs to a container, false otherwise. This function can be used to test whether `set::iterator_to` will return a valid iterator.

Complexity: Constant

3.
```
void unlink();
```

Effects: Removes the node if it's inserted in a container. This function is only allowed if `link_mode` is `auto_unlink`.

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Header <boost/intrusive/sg_set.hpp>

```
namespace boost {
  namespace intrusive {
    template<typename T, class... Options> class sg_set;

    template<typename T, class... Options> struct make_sg_set;

    template<typename T, class... Options> class sg_multiset;

    template<typename T, class... Options> struct make_sg_multiset;
    template<typename T, class... Options>
      bool operator!=(const sg_set< T, Options...> & x,
                      const sg_set< T, Options...> & y);
    template<typename T, class... Options>
      bool operator>(const sg_set< T, Options...> & x,
                     const sg_set< T, Options...> & y);
    template<typename T, class... Options>
      bool operator<=(const sg_set< T, Options...> & x,
                      const sg_set< T, Options...> & y);
    template<typename T, class... Options>
      bool operator>=(const sg_set< T, Options...> & x,
                      const sg_set< T, Options...> & y);
    template<typename T, class... Options>
      void swap(sg_set< T, Options...> & x, sg_set< T, Options...> & y);
    template<typename T, class... Options>
      bool operator!=(const sg_multiset< T, Options...> & x,
                      const sg_multiset< T, Options...> & y);
    template<typename T, class... Options>
      bool operator>(const sg_multiset< T, Options...> & x,
                     const sg_multiset< T, Options...> & y);
    template<typename T, class... Options>
      bool operator<=(const sg_multiset< T, Options...> & x,
                      const sg_multiset< T, Options...> & y);
    template<typename T, class... Options>
      bool operator>=(const sg_multiset< T, Options...> & x,
                      const sg_multiset< T, Options...> & y);
    template<typename T, class... Options>
      void swap(sg_multiset< T, Options...> & x,
                sg_multiset< T, Options...> & y);
  }
}
```

Class template sg_set

boost::intrusive::sg_set

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/intrusive/sg_set.hpp>

template<typename T, class... Options>
class sg_set {
public:
  // types
  typedef implementation_defined::value_type             value_type;
  typedef implementation_defined::value_traits           value_traits;
  typedef implementation_defined::pointer                pointer;
  typedef implementation_defined::const_pointer          const_pointer;
  typedef implementation_defined::reference              reference;
  typedef implementation_defined::const_reference        const_reference;
  typedef implementation_defined::difference_type        difference_type;
  typedef implementation_defined::size_type              size_type;
  typedef implementation_defined::value_compare          value_compare;
  typedef implementation_defined::key_compare            key_compare;
  typedef implementation_defined::iterator               iterator;
  typedef implementation_defined::const_iterator         const_iterator;
  typedef implementation_defined::reverse_iterator       reverse_iterator;
  typedef implementation_defined::const_reverse_iterator const_reverse_iterator;
  typedef implementation_defined::insert_commit_data     insert_commit_data;
  typedef implementation_defined::node_traits            node_traits;
  typedef implementation_defined::node                   node;
  typedef implementation_defined::node_ptr               node_ptr;
  typedef implementation_defined::const_node_ptr         const_node_ptr;
  typedef implementation_defined::node_algorithms        node_algorithms;

  // construct/copy/destruct
  explicit sg_set(const value_compare & = value_compare(),
                  const value_traits & = value_traits());
  template<typename Iterator>
    sg_set(Iterator, Iterator, const value_compare & = value_compare(),
           const value_traits & = value_traits());
  sg_set(BOOST_RV_REF(sg_set));
  sg_set& operator=(BOOST_RV_REF(sg_set));
  ~sg_set();

  // public member functions
  iterator begin();
  const_iterator begin() const;
  const_iterator cbegin() const;
  iterator end();
  const_iterator end() const;
  const_iterator cend() const;
  reverse_iterator rbegin();
  const_reverse_iterator rbegin() const;
  const_reverse_iterator crbegin() const;
  reverse_iterator rend();
  const_reverse_iterator rend() const;
  const_reverse_iterator crend() const;
  key_compare key_comp() const;
  value_compare value_comp() const;
  bool empty() const;
  size_type size() const;
  void swap(sg_set &);
  template<typename Cloner, typename Disposer>
    void clone_from(const sg_set &, Cloner, Disposer);
  std::pair< iterator, bool > insert(reference);
  iterator insert(const_iterator, reference);
  template<typename KeyType, typename KeyValueCompare>
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
  std::pair< iterator, bool >
  insert_check(const KeyType &, KeyValueCompare, insert_commit_data &);
template<typename KeyType, typename KeyValueCompare>
  std::pair< iterator, bool >
  insert_check(const_iterator, const KeyType &, KeyValueCompare,
               insert_commit_data &);
iterator insert_commit(reference, const insert_commit_data &);
template<typename Iterator> void insert(Iterator, Iterator);
iterator insert_before(const_iterator, reference);
void push_back(reference);
void push_front(reference);
iterator erase(const_iterator);
iterator erase(const_iterator, const_iterator);
size_type erase(const_reference);
template<typename KeyType, typename KeyValueCompare>
  size_type erase(const KeyType &, KeyValueCompare);
template<typename Disposer>
  iterator erase_and_dispose(const_iterator, Disposer);
template<typename Disposer>
  iterator erase_and_dispose(const_iterator, const_iterator, Disposer);
template<typename Disposer>
  size_type erase_and_dispose(const_reference, Disposer);
template<typename KeyType, typename KeyValueCompare, typename Disposer>
  size_type erase_and_dispose(const KeyType &, KeyValueCompare, Disposer);
void clear();
template<typename Disposer> void clear_and_dispose(Disposer);
size_type count(const_reference) const;
template<typename KeyType, typename KeyValueCompare>
  size_type count(const KeyType &, KeyValueCompare) const;
iterator lower_bound(const_reference);
template<typename KeyType, typename KeyValueCompare>
  iterator lower_bound(const KeyType &, KeyValueCompare);
const_iterator lower_bound(const_reference) const;
template<typename KeyType, typename KeyValueCompare>
  const_iterator lower_bound(const KeyType &, KeyValueCompare) const;
iterator upper_bound(const_reference);
template<typename KeyType, typename KeyValueCompare>
  iterator upper_bound(const KeyType &, KeyValueCompare);
const_iterator upper_bound(const_reference) const;
template<typename KeyType, typename KeyValueCompare>
  const_iterator upper_bound(const KeyType &, KeyValueCompare) const;
iterator find(const_reference);
template<typename KeyType, typename KeyValueCompare>
  iterator find(const KeyType &, KeyValueCompare);
const_iterator find(const_reference) const;
template<typename KeyType, typename KeyValueCompare>
  const_iterator find(const KeyType &, KeyValueCompare) const;
std::pair< iterator, iterator > equal_range(const_reference);
template<typename KeyType, typename KeyValueCompare>
  std::pair< iterator, iterator >
  equal_range(const KeyType &, KeyValueCompare);
std::pair< const_iterator, const_iterator >
equal_range(const_reference) const;
template<typename KeyType, typename KeyValueCompare>
  std::pair< const_iterator, const_iterator >
  equal_range(const KeyType &, KeyValueCompare) const;
std::pair< iterator, iterator >
bounded_range(const_reference, const_reference, bool, bool);
template<typename KeyType, typename KeyValueCompare>
  std::pair< iterator, iterator >
  bounded_range(const KeyType &, const KeyType &, KeyValueCompare, bool,
                bool);
std::pair< const_iterator, const_iterator >
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
    bounded_range(const_reference, const_reference, bool, bool) const;
    template<typename KeyType, typename KeyValueCompare>
      std::pair< const_iterator, const_iterator >
      bounded_range(const KeyType &, const KeyType &, KeyValueCompare, bool,
                    bool) const;
    iterator iterator_to(reference);
    const_iterator iterator_to(const_reference) const;
    pointer unlink_leftmost_without_rebalance();
    void replace_node(iterator, reference);
    void rebalance();
    iterator rebalance_subtree(iterator);
    float balance_factor() const;
    void balance_factor(float);

    // public static functions
    static sg_set & container_from_end_iterator(iterator);
    static const sg_set & container_from_end_iterator(const_iterator);
    static sg_set & container_from_iterator(iterator);
    static const sg_set & container_from_iterator(const_iterator);
    static iterator s_iterator_to(reference);
    static const_iterator s_iterator_to(const_reference);
    static void init_node(reference);
  };
```

Description

The class template sg_set is an intrusive container, that mimics most of the interface of std::set as described in the C++ standard.

The template parameter `T` is the type to be managed by the container. The user can specify additional options and if no options are provided default options are used.

The container supports the following options: `base_hook<>/member_hook<>/value_traits<>`, `constant_time_size<>`, `size_type<>` and `compare<>`.

`sg_set` public construct/copy/destruct

1.
```
explicit sg_set(const value_compare & cmp = value_compare(),
                const value_traits & v_traits = value_traits());
```

Effects: Constructs an empty `sg_set`.

Complexity: Constant.

Throws: If value_traits::node_traits::node constructor throws (this does not happen with predefined Boost.Intrusive hooks) or the copy constructor of the value_compare object throws.

2.
```
template<typename Iterator>
  sg_set(Iterator b, Iterator e, const value_compare & cmp = value_compare(),
         const value_traits & v_traits = value_traits());
```

Requires: Dereferencing iterator must yield an lvalue of type value_type. cmp must be a comparison function that induces a strict weak ordering.

Effects: Constructs an empty `sg_set` and inserts elements from [b, e).

Complexity: Linear in N if [b, e) is already sorted using comp and otherwise N * log N, where N is std::distance(last, first).

Throws: If value_traits::node_traits::node constructor throws (this does not happen with predefined Boost.Intrusive hooks) or the copy constructor/operator() of the value_compare object throws.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

3.
```
sg_set(BOOST_RV_REF(sg_set) x);
```

   Effects: to-do

4.
```
sg_set& operator=(BOOST_RV_REF(sg_set) x);
```

   Effects: to-do

5.
```
~sg_set();
```

   Effects: Detaches all elements from this. The objects in the `sg_set` are not deleted (i.e. no destructors are called).

   Complexity: Linear to the number of elements on the container. if it's a safe-mode or auto-unlink value_type. Constant time otherwise.

   Throws: Nothing.

`sg_set` public member functions

1.
```
iterator begin();
```

   Effects: Returns an iterator pointing to the beginning of the `sg_set`.

   Complexity: Constant.

   Throws: Nothing.

2.
```
const_iterator begin() const;
```

   Effects: Returns a const_iterator pointing to the beginning of the `sg_set`.

   Complexity: Constant.

   Throws: Nothing.

3.
```
const_iterator cbegin() const;
```

   Effects: Returns a const_iterator pointing to the beginning of the `sg_set`.

   Complexity: Constant.

   Throws: Nothing.

4.
```
iterator end();
```

   Effects: Returns an iterator pointing to the end of the `sg_set`.

   Complexity: Constant.

   Throws: Nothing.

5.
```
const_iterator end() const;
```

   Effects: Returns a const_iterator pointing to the end of the `sg_set`.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Constant.

Throws: Nothing.

6.
```
const_iterator cend() const;
```

Effects: Returns a const_iterator pointing to the end of the `sg_set`.

Complexity: Constant.

Throws: Nothing.

7.
```
reverse_iterator rbegin();
```

Effects: Returns a reverse_iterator pointing to the beginning of the reversed `sg_set`.

Complexity: Constant.

Throws: Nothing.

8.
```
const_reverse_iterator rbegin() const;
```

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed `sg_set`.

Complexity: Constant.

Throws: Nothing.

9.
```
const_reverse_iterator crbegin() const;
```

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed `sg_set`.

Complexity: Constant.

Throws: Nothing.

10.
```
reverse_iterator rend();
```

Effects: Returns a reverse_iterator pointing to the end of the reversed `sg_set`.

Complexity: Constant.

Throws: Nothing.

11.
```
const_reverse_iterator rend() const;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed `sg_set`.

Complexity: Constant.

Throws: Nothing.

12.
```
const_reverse_iterator crend() const;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed `sg_set`.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Constant.

Throws: Nothing.

13.
```
key_compare key_comp() const;
```

Effects: Returns the key_compare object used by the `sg_set`.

Complexity: Constant.

Throws: If key_compare copy-constructor throws.

14.
```
value_compare value_comp() const;
```

Effects: Returns the value_compare object used by the `sg_set`.

Complexity: Constant.

Throws: If value_compare copy-constructor throws.

15.
```
bool empty() const;
```

Effects: Returns true if the container is empty.

Complexity: Constant.

Throws: Nothing.

16.
```
size_type size() const;
```

Effects: Returns the number of elements stored in the `sg_set`.

Complexity: Linear to elements contained in *this if, constant-time size option is enabled. Constant-time otherwise.

Throws: Nothing.

17.
```
void swap(sg_set & other);
```

Effects: Swaps the contents of two sets.

Complexity: Constant.

Throws: If the swap() call for the comparison functor found using ADL throws. Strong guarantee.

18.
```
template<typename Cloner, typename Disposer> 
  void clone_from(const sg_set & src, Cloner cloner, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw. Cloner should yield to nodes equivalent to the original nodes.

Effects: Erases all the elements from *this calling Disposer::operator()(pointer), clones all the elements from src calling Cloner::operator()(const_reference) and inserts them on *this. Copies the predicate from the source container.

If cloner throws, all cloned elements are unlinked and disposed calling Disposer::operator()(pointer).

Complexity: Linear to erased plus inserted elements.

Throws: If cloner throws or predicate copy assignment throws. Basic guarantee.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

19.
```
std::pair< iterator, bool > insert(reference value);
```

Requires: value must be an lvalue

Effects: Tries to inserts value into the `sg_set`.

Returns: If the value is not already present inserts it and returns a pair containing the iterator to the new value and true. If there is an equivalent value returns a pair containing an iterator to the already present value and false.

Complexity: Average complexity for insert element is at most logarithmic.

Throws: If the internal value_compare ordering function throws. Strong guarantee.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

20.
```
iterator insert(const_iterator hint, reference value);
```

Requires: value must be an lvalue

Effects: Tries to to insert x into the `sg_set`, using "hint" as a hint to where it will be inserted.

Returns: An iterator that points to the position where the new element was inserted into the `sg_set`.

Complexity: Logarithmic in general, but it's amortized constant time if t is inserted immediately before hint.

Throws: If the internal value_compare ordering function throws. Strong guarantee.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

21.
```
template<typename KeyType, typename KeyValueCompare>
  std::pair< iterator, bool >
  insert_check(const KeyType & key, KeyValueCompare key_value_comp,
               insert_commit_data & commit_data);
```

Requires: key_value_comp must be a comparison function that induces the same strict weak ordering as value_compare. The difference is that key_value_comp compares an arbitrary key with the contained values.

Effects: Checks if a value can be inserted in the `sg_set`, using a user provided key instead of the value itself.

Returns: If there is an equivalent value returns a pair containing an iterator to the already present value and false. If the value can be inserted returns true in the returned pair boolean and fills "commit_data" that is meant to be used with the "insert_commit" function.

Complexity: Average complexity is at most logarithmic.

Throws: If the key_value_comp ordering function throws. Strong guarantee.

Notes: This function is used to improve performance when constructing a value_type is expensive: if there is an equivalent value the constructed object must be discarded. Many times, the part of the node that is used to impose the order is much cheaper to construct than the value_type and this function offers the possibility to use that part to check if the insertion will be successful.

If the check is successful, the user can construct the value_type and use "insert_commit" to insert the object in constant-time. This gives a total logarithmic complexity to the insertion: check(O(log(N)) + commit(O(1)).

"commit_data" remains valid for a subsequent "insert_commit" only if no more objects are inserted or erased from the `sg_set`.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

22.
```
template<typename KeyType, typename KeyValueCompare>
  std::pair< iterator, bool >
  insert_check(const_iterator hint, const KeyType & key,
               KeyValueCompare key_value_comp,
               insert_commit_data & commit_data);
```

Requires: key_value_comp must be a comparison function that induces the same strict weak ordering as value_compare. The difference is that key_value_comp compares an arbitrary key with the contained values.

Effects: Checks if a value can be inserted in the `sg_set`, using a user provided key instead of the value itself, using "hint" as a hint to where it will be inserted.

Returns: If there is an equivalent value returns a pair containing an iterator to the already present value and false. If the value can be inserted returns true in the returned pair boolean and fills "commit_data" that is meant to be used with the "insert_commit" function.

Complexity: Logarithmic in general, but it's amortized constant time if t is inserted immediately before hint.

Throws: If the key_value_comp ordering function throws. Strong guarantee.

Notes: This function is used to improve performance when constructing a value_type is expensive: if there is an equivalent value the constructed object must be discarded. Many times, the part of the constructing that is used to impose the order is much cheaper to construct than the value_type and this function offers the possibility to use that key to check if the insertion will be successful.

If the check is successful, the user can construct the value_type and use "insert_commit" to insert the object in constant-time. This can give a total constant-time complexity to the insertion: check(O(1)) + commit(O(1)).

"commit_data" remains valid for a subsequent "insert_commit" only if no more objects are inserted or erased from the `sg_set`.

23.
```
iterator insert_commit(reference value,
                       const insert_commit_data & commit_data);
```

Requires: value must be an lvalue of type value_type. commit_data must have been obtained from a previous call to "insert_check". No objects should have been inserted or erased from the `sg_set` between the "insert_check" that filled "commit_data" and the call to "insert_commit".

Effects: Inserts the value in the `sg_set` using the information obtained from the "commit_data" that a previous "insert_check" filled.

Returns: An iterator to the newly inserted object.

Complexity: Constant time.

Throws: Nothing.

Notes: This function has only sense if a "insert_check" has been previously executed to fill "commit_data". No value should be inserted or erased between the "insert_check" and "insert_commit" calls.

24.
```
template<typename Iterator> void insert(Iterator b, Iterator e);
```

Requires: Dereferencing iterator must yield an lvalue of type value_type.

Effects: Inserts a range into the `sg_set`.

Complexity: Insert range is in general O(N * log(N)), where N is the size of the range. However, it is linear in N if the range is already sorted by value_comp().

Throws: If the internal value_compare ordering function throws. Basic guarantee.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

25.
```
iterator insert_before(const_iterator pos, reference value);
```

Requires: value must be an lvalue, "pos" must be a valid iterator (or end) and must be the succesor of value once inserted according to the predicate. "value" must not be equal to any inserted key according to the predicate.

Effects: Inserts x into the tree before "pos".

Complexity: Constant time.

Throws: Nothing.

Note: This function does not check preconditions so if "pos" is not the successor of "value" or "value" is not unique tree ordering and uniqueness invariants will be broken respectively. This is a low-level function to be used only for performance reasons by advanced users.

26.
```
void push_back(reference value);
```

Requires: value must be an lvalue, and it must be greater than any inserted key according to the predicate.

Effects: Inserts x into the tree in the last position.

Complexity: Constant time.

Throws: Nothing.

Note: This function does not check preconditions so if value is less than or equal to the greatest inserted key tree ordering invariant will be broken. This function is slightly more efficient than using "insert_before". This is a low-level function to be used only for performance reasons by advanced users.

27.
```
void push_front(reference value);
```

Requires: value must be an lvalue, and it must be less than any inserted key according to the predicate.

Effects: Inserts x into the tree in the first position.

Complexity: Constant time.

Throws: Nothing.

Note: This function does not check preconditions so if value is greater than or equal to the the mimum inserted key tree ordering or uniqueness invariants will be broken. This function is slightly more efficient than using "insert_before". This is a low-level function to be used only for performance reasons by advanced users.

28.
```
iterator erase(const_iterator i);
```

Effects: Erases the element pointed to by pos.

Complexity: Average complexity is constant time.

Returns: An iterator to the element after the erased element.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

29.
```
iterator erase(const_iterator b, const_iterator e);
```

Effects: Erases the range pointed to by b end e.

Complexity: Average complexity for erase range is at most O(log(size() + N)), where N is the number of elements in the range.

Returns: An iterator to the element after the erased elements.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

30.
```
size_type erase(const_reference value);
```

Effects: Erases all the elements with the given value.

Returns: The number of erased elements.

Complexity: O(log(size()) + this->count(value)).

Throws: If the internal value_compare ordering function throws. Basic guarantee.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

31.
```
template<typename KeyType, typename KeyValueCompare>
  size_type erase(const KeyType & key, KeyValueCompare comp);
```

Effects: Erases all the elements that compare equal with the given key and the given comparison functor.

Returns: The number of erased elements.

Complexity: O(log(size() + this->count(key, comp)).

Throws: If the comp ordering function throws. Basic guarantee.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

32.
```
template<typename Disposer>
  iterator erase_and_dispose(const_iterator i, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases the element pointed to by pos. Disposer::operator()(pointer) is called for the removed element.

Complexity: Average complexity for erase element is constant time.

Returns: An iterator to the element after the erased element.

Throws: Nothing.

Note: Invalidates the iterators to the erased elements.

33.
```
template<typename Disposer>
  iterator erase_and_dispose(const_iterator b, const_iterator e,
                             Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Erases the range pointed to by b end e. Disposer::operator()(pointer) is called for the removed elements.

Complexity: Average complexity for erase range is at most O(log(size() + N)), where N is the number of elements in the range.

Returns: An iterator to the element after the erased elements.

Throws: Nothing.

Note: Invalidates the iterators to the erased elements.

34.
```
template<typename Disposer> 
  size_type erase_and_dispose(const_reference value, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases all the elements with the given value. Disposer::operator()(pointer) is called for the removed elements.

Throws: If the internal value_compare ordering function throws.

Complexity: O(log(size() + this->count(value)). Basic guarantee.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

35.
```
template<typename KeyType, typename KeyValueCompare, typename Disposer> 
  size_type erase_and_dispose(const KeyType & key, KeyValueCompare comp, 
                              Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases all the elements with the given key. according to the comparison functor "comp". Disposer::operator()(pointer) is called for the removed elements.

Returns: The number of erased elements.

Complexity: O(log(size() + this->count(key, comp)).

Throws: If comp ordering function throws. Basic guarantee.

Note: Invalidates the iterators to the erased elements.

36.
```
void clear();
```

Effects: Erases all the elements of the container.

Complexity: Linear to the number of elements on the container. if it's a safe-mode or auto-unlink value_type. Constant time otherwise.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

37.
```
template<typename Disposer> void clear_and_dispose(Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases all the elements of the container.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Linear to the number of elements on the container. Disposer::operator()(pointer) is called for the removed elements.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

38.
```
size_type count(const_reference value) const;
```

Effects: Returns the number of contained elements with the given key

Complexity: Logarithmic to the number of elements contained plus lineal to number of objects with the given key.

Throws: If the internal value_compare ordering function throws.

39.
```
template<typename KeyType, typename KeyValueCompare>
  size_type count(const KeyType & key, KeyValueCompare comp) const;
```

Effects: Returns the number of contained elements with the same key compared with the given comparison functor.

Complexity: Logarithmic to the number of elements contained plus lineal to number of objects with the given key.

Throws: If comp ordering function throws.

40.
```
iterator lower_bound(const_reference value);
```

Effects: Returns an iterator to the first element whose key is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

41.
```
template<typename KeyType, typename KeyValueCompare>
  iterator lower_bound(const KeyType & key, KeyValueCompare comp);
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Returns an iterator to the first element whose key according to the comparison functor is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

42.
```
const_iterator lower_bound(const_reference value) const;
```

Effects: Returns a const iterator to the first element whose key is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

43.
```
template<typename KeyType, typename KeyValueCompare>
  const_iterator lower_bound(const KeyType & key, KeyValueCompare comp) const;
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Returns a const_iterator to the first element whose key according to the comparison functor is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

44.
```
iterator upper_bound(const_reference value);
```

Effects: Returns an iterator to the first element whose key is greater than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

45.
```
template<typename KeyType, typename KeyValueCompare>
  iterator upper_bound(const KeyType & key, KeyValueCompare comp);
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Returns an iterator to the first element whose key according to the comparison functor is greater than key or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

46.
```
const_iterator upper_bound(const_reference value) const;
```

Effects: Returns an iterator to the first element whose key is greater than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

47.
```
template<typename KeyType, typename KeyValueCompare>
  const_iterator upper_bound(const KeyType & key, KeyValueCompare comp) const;
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Returns a const_iterator to the first element whose key according to the comparison functor is greater than key or end() if that element does not exist.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

48.
```
iterator find(const_reference value);
```

Effects: Finds an iterator to the first element whose value is "value" or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

49.
```
template<typename KeyType, typename KeyValueCompare>
  iterator find(const KeyType & key, KeyValueCompare comp);
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Finds an iterator to the first element whose key is "key" according to the comparison functor or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

50.
```
const_iterator find(const_reference value) const;
```

Effects: Finds a const_iterator to the first element whose value is "value" or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

51.
```
template<typename KeyType, typename KeyValueCompare>
  const_iterator find(const KeyType & key, KeyValueCompare comp) const;
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Finds a const_iterator to the first element whose key is "key" according to the comparison functor or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

52.
```
std::pair< iterator, iterator > equal_range(const_reference value);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Finds a range containing all elements whose key is k or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

53.
```
template<typename KeyType, typename KeyValueCompare> 
  std::pair< iterator, iterator > 
  equal_range(const KeyType & key, KeyValueCompare comp);
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Finds a range containing all elements whose key is k according to the comparison functor or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

54.
```
std::pair< const_iterator, const_iterator > 
equal_range(const_reference value) const;
```

Effects: Finds a range containing all elements whose key is k or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

55.
```
template<typename KeyType, typename KeyValueCompare> 
  std::pair< const_iterator, const_iterator > 
  equal_range(const KeyType & key, KeyValueCompare comp) const;
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Finds a range containing all elements whose key is k according to the comparison functor or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

56.
```
std::pair< iterator, iterator > 
bounded_range(const_reference lower_value, const_reference upper_value, 
              bool left_closed, bool right_closed);
```

Requires: 'lower_value' must not be greater than 'upper_value'. If 'lower_value' == 'upper_value', ('left_closed' || 'right_closed') must be false.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key) if left_closed, upper_bound(lower_key) otherwise

second = upper_bound(upper_key) if right_closed, lower_bound(upper_key) otherwise

Complexity: Logarithmic.

Throws: If the predicate throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_value and upper_value.

57.
```
template<typename KeyType, typename KeyValueCompare> 
  std::pair< iterator, iterator > 
  bounded_range(const KeyType & lower_key, const KeyType & upper_key, 
                KeyValueCompare comp, bool left_closed, bool right_closed);
```

Requires: KeyValueCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. 'lower_key' must not be greater than 'upper_key' according to 'comp'. If 'lower_key' == 'upper_key', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key, comp) if left_closed, upper_bound(lower_key, comp) otherwise

second = upper_bound(upper_key, comp) if right_closed, lower_bound(upper_key, comp) otherwise

Complexity: Logarithmic.

Throws: If "comp" throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_key and upper_key.

58.
```
std::pair< const_iterator, const_iterator > 
bounded_range(const_reference lower_value, const_reference upper_value, 
              bool left_closed, bool right_closed) const;
```

Requires: 'lower_value' must not be greater than 'upper_value'. If 'lower_value' == 'upper_value', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key) if left_closed, upper_bound(lower_key) otherwise

second = upper_bound(upper_key) if right_closed, lower_bound(upper_key) otherwise

Complexity: Logarithmic.

Throws: If the predicate throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_value and upper_value.

59.
```
template<typename KeyType, typename KeyValueCompare> 
  std::pair< const_iterator, const_iterator > 
  bounded_range(const KeyType & lower_key, const KeyType & upper_key, 
                KeyValueCompare comp, bool left_closed, bool right_closed) const;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Requires: KeyValueCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. 'lower_key' must not be greater than 'upper_key' according to 'comp'. If 'lower_key' == 'upper_key', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key, comp) if left_closed, upper_bound(lower_key, comp) otherwise

second = upper_bound(upper_key, comp) if right_closed, lower_bound(upper_key, comp) otherwise

Complexity: Logarithmic.

Throws: If "comp" throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_key and upper_key.

60.
```
iterator iterator_to(reference value);
```

Requires: value must be an lvalue and shall be in a `sg_set` of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid iterator i belonging to the `sg_set` that points to the value

Complexity: Constant.

Throws: Nothing.

61.
```
const_iterator iterator_to(const_reference value) const;
```

Requires: value must be an lvalue and shall be in a `sg_set` of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid const_iterator i belonging to the `sg_set` that points to the value

Complexity: Constant.

Throws: Nothing.

62.
```
pointer unlink_leftmost_without_rebalance();
```

Effects: Unlinks the leftmost node from the tree.

Complexity: Average complexity is constant time.

Throws: Nothing.

Notes: This function breaks the tree and the tree can only be used for more unlink_leftmost_without_rebalance calls. This function is normally used to achieve a step by step controlled destruction of the tree.

63.
```
void replace_node(iterator replace_this, reference with_this);
```

Requires: replace_this must be a valid iterator of *this and with_this must not be inserted in any tree.

Effects: Replaces replace_this in its position in the tree with with_this. The tree does not need to be rebalanced.

Complexity: Constant.

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Note: This function will break container ordering invariants if with_this is not equivalent to *replace_this according to the ordering rules. This function is faster than erasing and inserting the node, since no rebalancing or comparison is needed.

64.
```
void rebalance();
```

Effects: Rebalances the tree.

Throws: Nothing.

Complexity: Linear.

65.
```
iterator rebalance_subtree(iterator root);
```

Requires: old_root is a node of a tree.

Effects: Rebalances the subtree rooted at old_root.

Returns: The new root of the subtree.

Throws: Nothing.

Complexity: Linear to the elements in the subtree.

66.
```
float balance_factor() const;
```

Returns: The balance factor (alpha) used in this tree

Throws: Nothing.

Complexity: Constant.

67.
```
void balance_factor(float new_alpha);
```

Requires: new_alpha must be a value between 0.5 and 1.0

Effects: Establishes a new balance factor (alpha) and rebalances the tree if the new balance factor is stricter (less) than the old factor.

Throws: Nothing.

Complexity: Linear to the elements in the subtree.

`sg_set` public static functions

1.
```
static sg_set & container_from_end_iterator(iterator end_iterator);
```

Precondition: end_iterator must be a valid end iterator of `sg_set`.

Effects: Returns a const reference to the `sg_set` associated to the end iterator

Throws: Nothing.

Complexity: Constant.

2.
```
static const sg_set & container_from_end_iterator(const_iterator end_iterator);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Precondition: end_iterator must be a valid end const_iterator of `sg_set`.

Effects: Returns a const reference to the `sg_set` associated to the end iterator

Throws: Nothing.

Complexity: Constant.

3.
```
static sg_set & container_from_iterator(iterator it);
```

Precondition: it must be a valid iterator of set.

Effects: Returns a reference to the set associated to the iterator

Throws: Nothing.

Complexity: Logarithmic.

4.
```
static const sg_set & container_from_iterator(const_iterator it);
```

Precondition: it must be a valid const_iterator of set.

Effects: Returns a const reference to the set associated to the iterator

Throws: Nothing.

Complexity: Logarithmic.

5.
```
static iterator s_iterator_to(reference value);
```

Requires: value must be an lvalue and shall be in a `sg_set` of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid iterator i belonging to the `sg_set` that points to the value

Complexity: Constant.

Throws: Nothing.

Note: This static function is available only if the *value traits* is stateless.

6.
```
static const_iterator s_iterator_to(const_reference value);
```

Requires: value must be an lvalue and shall be in a `sg_set` of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid const_iterator i belonging to the `sg_set` that points to the value

Complexity: Constant.

Throws: Nothing.

Note: This static function is available only if the *value traits* is stateless.

7.
```
static void init_node(reference value);
```

Requires: value shall not be in a sg_set/sg_multiset.

Effects: init_node puts the hook of a value in a well-known default state.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: Nothing.

Complexity: Constant time.

Note: This function puts the hook in the well-known default state used by auto_unlink and safe hooks.

Struct template make_sg_set

boost::intrusive::make_sg_set

Synopsis

```
// In header: <boost/intrusive/sg_set.hpp>

template<typename T, class... Options> 
struct make_sg_set {
  // types
  typedef implementation_defined type;
};
```

Description

Helper metafunction to define a `sg_set` that yields to the same type when the same options (either explicitly or implicitly) are used.

Class template sg_multiset

boost::intrusive::sg_multiset

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/intrusive/sg_set.hpp>

template<typename T, class... Options>
class sg_multiset {
public:
  // types
  typedef implementation_defined::value_type             value_type;
  typedef implementation_defined::value_traits           value_traits;
  typedef implementation_defined::pointer                pointer;
  typedef implementation_defined::const_pointer          const_pointer;
  typedef implementation_defined::reference              reference;
  typedef implementation_defined::const_reference        const_reference;
  typedef implementation_defined::difference_type        difference_type;
  typedef implementation_defined::size_type              size_type;
  typedef implementation_defined::value_compare          value_compare;
  typedef implementation_defined::key_compare            key_compare;
  typedef implementation_defined::iterator               iterator;
  typedef implementation_defined::const_iterator         const_iterator;
  typedef implementation_defined::reverse_iterator       reverse_iterator;
  typedef implementation_defined::const_reverse_iterator const_reverse_iterator;
  typedef implementation_defined::insert_commit_data     insert_commit_data;
  typedef implementation_defined::node_traits            node_traits;
  typedef implementation_defined::node                   node;
  typedef implementation_defined::node_ptr               node_ptr;
  typedef implementation_defined::const_node_ptr         const_node_ptr;
  typedef implementation_defined::node_algorithms        node_algorithms;

  // construct/copy/destruct
  explicit sg_multiset(const value_compare & = value_compare(),
                       const value_traits & = value_traits());
  template<typename Iterator>
    sg_multiset(Iterator, Iterator, const value_compare & = value_compare(),
                const value_traits & = value_traits());
  sg_multiset(BOOST_RV_REF(sg_multiset));
  sg_multiset& operator=(BOOST_RV_REF(sg_multiset));
  ~sg_multiset();

  // public member functions
  iterator begin();
  const_iterator begin() const;
  const_iterator cbegin() const;
  iterator end();
  const_iterator end() const;
  const_iterator cend() const;
  reverse_iterator rbegin();
  const_reverse_iterator rbegin() const;
  const_reverse_iterator crbegin() const;
  reverse_iterator rend();
  const_reverse_iterator rend() const;
  const_reverse_iterator crend() const;
  key_compare key_comp() const;
  value_compare value_comp() const;
  bool empty() const;
  size_type size() const;
  void swap(sg_multiset &);
  template<typename Cloner, typename Disposer>
    void clone_from(const sg_multiset &, Cloner, Disposer);
  iterator insert(reference);
  iterator insert(const_iterator, reference);
  template<typename Iterator> void insert(Iterator, Iterator);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
iterator insert_before(const_iterator, reference);
void push_back(reference);
void push_front(reference);
iterator erase(const_iterator);
iterator erase(const_iterator, const_iterator);
size_type erase(const_reference);
template<typename KeyType, typename KeyValueCompare> 
  size_type erase(const KeyType &, KeyValueCompare);
template<typename Disposer> 
  iterator erase_and_dispose(const_iterator, Disposer);
template<typename Disposer> 
  iterator erase_and_dispose(const_iterator, const_iterator, Disposer);
template<typename Disposer> 
  size_type erase_and_dispose(const_reference, Disposer);
template<typename KeyType, typename KeyValueCompare, typename Disposer> 
  size_type erase_and_dispose(const KeyType &, KeyValueCompare, Disposer);
void clear();
template<typename Disposer> void clear_and_dispose(Disposer);
size_type count(const_reference) const;
template<typename KeyType, typename KeyValueCompare> 
  size_type count(const KeyType &, KeyValueCompare) const;
iterator lower_bound(const_reference);
template<typename KeyType, typename KeyValueCompare> 
  iterator lower_bound(const KeyType &, KeyValueCompare);
const_iterator lower_bound(const_reference) const;
template<typename KeyType, typename KeyValueCompare> 
  const_iterator lower_bound(const KeyType &, KeyValueCompare) const;
iterator upper_bound(const_reference);
template<typename KeyType, typename KeyValueCompare> 
  iterator upper_bound(const KeyType &, KeyValueCompare);
const_iterator upper_bound(const_reference) const;
template<typename KeyType, typename KeyValueCompare> 
  const_iterator upper_bound(const KeyType &, KeyValueCompare) const;
iterator find(const_reference);
template<typename KeyType, typename KeyValueCompare> 
  iterator find(const KeyType &, KeyValueCompare);
const_iterator find(const_reference) const;
template<typename KeyType, typename KeyValueCompare> 
  const_iterator find(const KeyType &, KeyValueCompare) const;
std::pair< iterator, iterator > equal_range(const_reference);
template<typename KeyType, typename KeyValueCompare> 
  std::pair< iterator, iterator > 
  equal_range(const KeyType &, KeyValueCompare);
std::pair< const_iterator, const_iterator > 
equal_range(const_reference) const;
template<typename KeyType, typename KeyValueCompare> 
  std::pair< const_iterator, const_iterator > 
  equal_range(const KeyType &, KeyValueCompare) const;
std::pair< iterator, iterator > 
bounded_range(const_reference, const_reference, bool, bool);
template<typename KeyType, typename KeyValueCompare> 
  std::pair< iterator, iterator > 
  bounded_range(const KeyType &, const KeyType &, KeyValueCompare, bool, 
                bool);
std::pair< const_iterator, const_iterator > 
bounded_range(const_reference, const_reference, bool, bool) const;
template<typename KeyType, typename KeyValueCompare> 
  std::pair< const_iterator, const_iterator > 
  bounded_range(const KeyType &, const KeyType &, KeyValueCompare, bool, 
                bool) const;
iterator iterator_to(reference);
const_iterator iterator_to(const_reference) const;
pointer unlink_leftmost_without_rebalance();
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
  void replace_node(iterator, reference);
  void rebalance();
  iterator rebalance_subtree(iterator);
  float balance_factor() const;
  void balance_factor(float);

  // public static functions
  static sg_multiset & container_from_end_iterator(iterator);
  static const sg_multiset & container_from_end_iterator(const_iterator);
  static sg_multiset & container_from_iterator(iterator);
  static const sg_multiset & container_from_iterator(const_iterator);
  static iterator s_iterator_to(reference);
  static const_iterator s_iterator_to(const_reference);
  static void init_node(reference);
};
```

Description

The class template sg_multiset is an intrusive container, that mimics most of the interface of std::sg_multiset as described in the C++ standard.

The template parameter `T` is the type to be managed by the container. The user can specify additional options and if no options are provided default options are used.

The container supports the following options: `base_hook<>/member_hook<>/value_traits<>`, `constant_time_size<>`, `size_type<>` and `compare<>`.

`sg_multiset` public construct/copy/destruct

1.
```
explicit sg_multiset(const value_compare & cmp = value_compare(),
                     const value_traits & v_traits = value_traits());
```

Effects: Constructs an empty `sg_multiset`.

Complexity: Constant.

Throws: If value_traits::node_traits::node constructor throws (this does not happen with predefined Boost.Intrusive hooks) or the copy constructor/operator() of the value_compare object throws.

2.
```
template<typename Iterator>
  sg_multiset(Iterator b, Iterator e,
              const value_compare & cmp = value_compare(),
              const value_traits & v_traits = value_traits());
```

Requires: Dereferencing iterator must yield an lvalue of type value_type. cmp must be a comparison function that induces a strict weak ordering.

Effects: Constructs an empty `sg_multiset` and inserts elements from [b, e).

Complexity: Linear in N if [b, e) is already sorted using comp and otherwise N * log N, where N is the distance between first and last

Throws: If value_traits::node_traits::node constructor throws (this does not happen with predefined Boost.Intrusive hooks) or the copy constructor/operator() of the value_compare object throws.

3.
```
sg_multiset(BOOST_RV_REF(sg_multiset) x);
```

Effects: to-do

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

4.
```
sg_multiset& operator=(BOOST_RV_REF(sg_multiset) x);
```

Effects: to-do

5.
```
~sg_multiset();
```

Effects: Detaches all elements from this. The objects in the `sg_multiset` are not deleted (i.e. no destructors are called).

Complexity: Linear to the number of elements on the container. if it's a safe-mode or auto-unlink value_type. Constant time otherwise.

Throws: Nothing.

`sg_multiset` public member functions

1.
```
iterator begin();
```

Effects: Returns an iterator pointing to the beginning of the `sg_multiset`.

Complexity: Constant.

Throws: Nothing.

2.
```
const_iterator begin() const;
```

Effects: Returns a const_iterator pointing to the beginning of the `sg_multiset`.

Complexity: Constant.

Throws: Nothing.

3.
```
const_iterator cbegin() const;
```

Effects: Returns a const_iterator pointing to the beginning of the `sg_multiset`.

Complexity: Constant.

Throws: Nothing.

4.
```
iterator end();
```

Effects: Returns an iterator pointing to the end of the `sg_multiset`.

Complexity: Constant.

Throws: Nothing.

5.
```
const_iterator end() const;
```

Effects: Returns a const_iterator pointing to the end of the `sg_multiset`.

Complexity: Constant.

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

6.
```
const_iterator cend() const;
```

Effects: Returns a const_iterator pointing to the end of the `sg_multiset`.

Complexity: Constant.

Throws: Nothing.

7.
```
reverse_iterator rbegin();
```

Effects: Returns a reverse_iterator pointing to the beginning of the reversed `sg_multiset`.

Complexity: Constant.

Throws: Nothing.

8.
```
const_reverse_iterator rbegin() const;
```

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed `sg_multiset`.

Complexity: Constant.

Throws: Nothing.

9.
```
const_reverse_iterator crbegin() const;
```

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed `sg_multiset`.

Complexity: Constant.

Throws: Nothing.

10.
```
reverse_iterator rend();
```

Effects: Returns a reverse_iterator pointing to the end of the reversed `sg_multiset`.

Complexity: Constant.

Throws: Nothing.

11.
```
const_reverse_iterator rend() const;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed `sg_multiset`.

Complexity: Constant.

Throws: Nothing.

12.
```
const_reverse_iterator crend() const;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed `sg_multiset`.

Complexity: Constant.

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

13.
```
key_compare key_comp() const;
```

Effects: Returns the key_compare object used by the `sg_multiset`.

Complexity: Constant.

Throws: If key_compare copy-constructor throws.

14.
```
value_compare value_comp() const;
```

Effects: Returns the value_compare object used by the `sg_multiset`.

Complexity: Constant.

Throws: If value_compare copy-constructor throws.

15.
```
bool empty() const;
```

Effects: Returns true if the container is empty.

Complexity: Constant.

Throws: Nothing.

16.
```
size_type size() const;
```

Effects: Returns the number of elements stored in the `sg_multiset`.

Complexity: Linear to elements contained in *this if, constant-time size option is enabled. Constant-time otherwise.

Throws: Nothing.

17.
```
void swap(sg_multiset & other);
```

Effects: Swaps the contents of two sg_multisets.

Complexity: Constant.

Throws: If the swap() call for the comparison functor found using ADL throws. Strong guarantee.

18.
```
template<typename Cloner, typename Disposer> 
  void clone_from(const sg_multiset & src, Cloner cloner, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw. Cloner should yield to nodes equivalent to the original nodes.

Effects: Erases all the elements from *this calling Disposer::operator()(pointer), clones all the elements from src calling Cloner::operator()(const_reference) and inserts them on *this. Copies the predicate from the source container.

If cloner throws, all cloned elements are unlinked and disposed calling Disposer::operator()(pointer).

Complexity: Linear to erased plus inserted elements.

Throws: If cloner throws or predicate copy assignment throws. Basic guarantee.

19.
```
iterator insert(reference value);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Requires: value must be an lvalue

Effects: Inserts value into the `sg_multiset`.

Returns: An iterator that points to the position where the new element was inserted.

Complexity: Average complexity for insert element is at most logarithmic.

Throws: If the internal value_compare ordering function throws. Strong guarantee.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

20.
```
iterator insert(const_iterator hint, reference value);
```

Requires: value must be an lvalue

Effects: Inserts x into the `sg_multiset`, using pos as a hint to where it will be inserted.

Returns: An iterator that points to the position where the new element was inserted.

Complexity: Logarithmic in general, but it is amortized constant time if t is inserted immediately before hint.

Throws: If the internal value_compare ordering function throws. Strong guarantee.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

21.
```
template<typename Iterator> void insert(Iterator b, Iterator e);
```

Requires: Dereferencing iterator must yield an lvalue of type value_type.

Effects: Inserts a range into the `sg_multiset`.

Returns: An iterator that points to the position where the new element was inserted.

Complexity: Insert range is in general O(N * log(N)), where N is the size of the range. However, it is linear in N if the range is already sorted by value_comp().

Throws: If the internal value_compare ordering function throws. Basic guarantee.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

22.
```
iterator insert_before(const_iterator pos, reference value);
```

Requires: value must be an lvalue, "pos" must be a valid iterator (or end) and must be the succesor of value once inserted according to the predicate

Effects: Inserts x into the tree before "pos".

Complexity: Constant time.

Throws: Nothing.

Note: This function does not check preconditions so if "pos" is not the successor of "value" tree ordering invariant will be broken. This is a low-level function to be used only for performance reasons by advanced users.

23.
```
void push_back(reference value);
```

Requires: value must be an lvalue, and it must be no less than the greatest inserted key

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Inserts x into the tree in the last position.

Complexity: Constant time.

Throws: Nothing.

Note: This function does not check preconditions so if value is less than the greatest inserted key tree ordering invariant will be broken. This function is slightly more efficient than using "insert_before". This is a low-level function to be used only for performance reasons by advanced users.

24.
```
void push_front(reference value);
```

Requires: value must be an lvalue, and it must be no greater than the minimum inserted key

Effects: Inserts x into the tree in the first position.

Complexity: Constant time.

Throws: Nothing.

Note: This function does not check preconditions so if value is greater than the minimum inserted key tree ordering invariant will be broken. This function is slightly more efficient than using "insert_before". This is a low-level function to be used only for performance reasons by advanced users.

25.
```
iterator erase(const_iterator i);
```

Effects: Erases the element pointed to by pos.

Complexity: Average complexity is constant time.

Returns: An iterator to the element after the erased element.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

26.
```
iterator erase(const_iterator b, const_iterator e);
```

Effects: Erases the range pointed to by b end e.

Returns: An iterator to the element after the erased elements.

Complexity: Average complexity for erase range is at most O(log(size() + N)), where N is the number of elements in the range.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

27.
```
size_type erase(const_reference value);
```

Effects: Erases all the elements with the given value.

Returns: The number of erased elements.

Complexity: O(log(size() + this->count(value)).

Throws: If the internal value_compare ordering function throws. Basic guarantee.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

28.
```
template<typename KeyType, typename KeyValueCompare>
  size_type erase(const KeyType & key, KeyValueCompare comp);
```

Effects: Erases all the elements that compare equal with the given key and the given comparison functor.

Returns: The number of erased elements.

Complexity: O(log(size() + this->count(key, comp)).

Throws: If comp ordering function throws. Basic guarantee.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

29.
```
template<typename Disposer>
  iterator erase_and_dispose(const_iterator i, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Returns: An iterator to the element after the erased element.

Effects: Erases the element pointed to by pos. Disposer::operator()(pointer) is called for the removed element.

Complexity: Average complexity for erase element is constant time.

Throws: Nothing.

Note: Invalidates the iterators to the erased elements.

30.
```
template<typename Disposer>
  iterator erase_and_dispose(const_iterator b, const_iterator e,
                             Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Returns: An iterator to the element after the erased elements.

Effects: Erases the range pointed to by b end e. Disposer::operator()(pointer) is called for the removed elements.

Complexity: Average complexity for erase range is at most O(log(size() + N)), where N is the number of elements in the range.

Throws: Nothing.

Note: Invalidates the iterators to the erased elements.

31.
```
template<typename Disposer>
  size_type erase_and_dispose(const_reference value, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases all the elements with the given value. Disposer::operator()(pointer) is called for the removed elements.

Returns: The number of erased elements.

Complexity: O(log(size() + this->count(value)).

Throws: If the internal value_compare ordering function throws. Basic guarantee.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

32.
```
template<typename KeyType, typename KeyValueCompare, typename Disposer>
  size_type erase_and_dispose(const KeyType & key, KeyValueCompare comp,
                              Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases all the elements with the given key. according to the comparison functor "comp". Disposer::operator()(pointer) is called for the removed elements.

Returns: The number of erased elements.

Complexity: O(log(size() + this->count(key, comp)).

Throws: If comp ordering function throws. Basic guarantee.

Note: Invalidates the iterators to the erased elements.

33.
```
void clear();
```

Effects: Erases all the elements of the container.

Complexity: Linear to the number of elements on the container. if it's a safe-mode or auto-unlink value_type. Constant time otherwise.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

34.
```
template<typename Disposer> void clear_and_dispose(Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases all the elements of the container.

Complexity: Linear to the number of elements on the container. Disposer::operator()(pointer) is called for the removed elements.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

35.
```
size_type count(const_reference value) const;
```

Effects: Returns the number of contained elements with the given key

Complexity: Logarithmic to the number of elements contained plus lineal to number of objects with the given key.

Throws: If the internal value_compare ordering function throws.

36.
```
template<typename KeyType, typename KeyValueCompare>
  size_type count(const KeyType & key, KeyValueCompare comp) const;
```

Effects: Returns the number of contained elements with the same key compared with the given comparison functor.

Complexity: Logarithmic to the number of elements contained plus lineal to number of objects with the given key.

Throws: If comp ordering function throws.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

37.
```
iterator lower_bound(const_reference value);
```

Effects: Returns an iterator to the first element whose key is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

38.
```
template<typename KeyType, typename KeyValueCompare>
  iterator lower_bound(const KeyType & key, KeyValueCompare comp);
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Returns an iterator to the first element whose key according to the comparison functor is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

39.
```
const_iterator lower_bound(const_reference value) const;
```

Effects: Returns a const iterator to the first element whose key is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

40.
```
template<typename KeyType, typename KeyValueCompare>
  const_iterator lower_bound(const KeyType & key, KeyValueCompare comp) const;
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Returns a const_iterator to the first element whose key according to the comparison functor is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

41.
```
iterator upper_bound(const_reference value);
```

Effects: Returns an iterator to the first element whose key is greater than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

42.
```
template<typename KeyType, typename KeyValueCompare>
  iterator upper_bound(const KeyType & key, KeyValueCompare comp);
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Returns an iterator to the first element whose key according to the comparison functor is greater than key or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

43.
```
const_iterator upper_bound(const_reference value) const;
```

Effects: Returns an iterator to the first element whose key is greater than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

44.
```
template<typename KeyType, typename KeyValueCompare>
  const_iterator upper_bound(const KeyType & key, KeyValueCompare comp) const;
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Returns a const_iterator to the first element whose key according to the comparison functor is greater than key or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

45.
```
iterator find(const_reference value);
```

Effects: Finds an iterator to the first element whose value is "value" or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

46.
```
template<typename KeyType, typename KeyValueCompare>
  iterator find(const KeyType & key, KeyValueCompare comp);
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Finds an iterator to the first element whose key is "key" according to the comparison functor or end() if that element does not exist.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

47.
```
const_iterator find(const_reference value) const;
```

Effects: Finds a const_iterator to the first element whose value is "value" or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

48.
```
template<typename KeyType, typename KeyValueCompare>
  const_iterator find(const KeyType & key, KeyValueCompare comp) const;
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Finds a const_iterator to the first element whose key is "key" according to the comparison functor or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

49.
```
std::pair< iterator, iterator > equal_range(const_reference value);
```

Effects: Finds a range containing all elements whose key is k or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

50.
```
template<typename KeyType, typename KeyValueCompare>
  std::pair< iterator, iterator >
  equal_range(const KeyType & key, KeyValueCompare comp);
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Finds a range containing all elements whose key is k according to the comparison functor or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

51.
```
std::pair< const_iterator, const_iterator >
equal_range(const_reference value) const;
```

Effects: Finds a range containing all elements whose key is k or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

52.
```
template<typename KeyType, typename KeyValueCompare>
  std::pair< const_iterator, const_iterator >
  equal_range(const KeyType & key, KeyValueCompare comp) const;
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Finds a range containing all elements whose key is k according to the comparison functor or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

53.
```
std::pair< iterator, iterator >
bounded_range(const_reference lower_value, const_reference upper_value,
              bool left_closed, bool right_closed);
```

Requires: 'lower_value' must not be greater than 'upper_value'. If 'lower_value' == 'upper_value', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key) if left_closed, upper_bound(lower_key) otherwise

second = upper_bound(upper_key) if right_closed, lower_bound(upper_key) otherwise

Complexity: Logarithmic.

Throws: If the predicate throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_value and upper_value.

54.
```
template<typename KeyType, typename KeyValueCompare>
  std::pair< iterator, iterator >
  bounded_range(const KeyType & lower_key, const KeyType & upper_key,
                KeyValueCompare comp, bool left_closed, bool right_closed);
```

Requires: KeyValueCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. 'lower_key' must not be greater than 'upper_key' according to 'comp'. If 'lower_key' == 'upper_key', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key, comp) if left_closed, upper_bound(lower_key, comp) otherwise

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

second = upper_bound(upper_key, comp) if right_closed, lower_bound(upper_key, comp) otherwise

Complexity: Logarithmic.

Throws: If "comp" throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_key and upper_key.

55.
```
std::pair< const_iterator, const_iterator >
bounded_range(const_reference lower_value, const_reference upper_value,
              bool left_closed, bool right_closed) const;
```

Requires: 'lower_value' must not be greater than 'upper_value'. If 'lower_value' == 'upper_value', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key) if left_closed, upper_bound(lower_key) otherwise

second = upper_bound(upper_key) if right_closed, lower_bound(upper_key) otherwise

Complexity: Logarithmic.

Throws: If the predicate throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_value and upper_value.

56.
```
template<typename KeyType, typename KeyValueCompare>
  std::pair< const_iterator, const_iterator >
  bounded_range(const KeyType & lower_key, const KeyType & upper_key,
                KeyValueCompare comp, bool left_closed, bool right_closed) const;
```

Requires: KeyValueCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. 'lower_key' must not be greater than 'upper_key' according to 'comp'. If 'lower_key' == 'upper_key', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key, comp) if left_closed, upper_bound(lower_key, comp) otherwise

second = upper_bound(upper_key, comp) if right_closed, lower_bound(upper_key, comp) otherwise

Complexity: Logarithmic.

Throws: If "comp" throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_key and upper_key.

57.
```
iterator iterator_to(reference value);
```

Requires: value must be an lvalue and shall be in a `sg_multiset` of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid iterator i belonging to the `sg_multiset` that points to the value

Complexity: Constant.

Throws: Nothing.

58.
```
const_iterator iterator_to(const_reference value) const;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Requires: value must be an lvalue and shall be in a `sg_multiset` of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid const_iterator i belonging to the `sg_multiset` that points to the value

Complexity: Constant.

Throws: Nothing.

59.
```
pointer unlink_leftmost_without_rebalance();
```

Effects: Unlinks the leftmost node from the tree.

Complexity: Average complexity is constant time.

Throws: Nothing.

Notes: This function breaks the tree and the tree can only be used for more unlink_leftmost_without_rebalance calls. This function is normally used to achieve a step by step controlled destruction of the tree.

60.
```
void replace_node(iterator replace_this, reference with_this);
```

Requires: replace_this must be a valid iterator of *this and with_this must not be inserted in any tree.

Effects: Replaces replace_this in its position in the tree with with_this. The tree does not need to be rebalanced.

Complexity: Constant.

Throws: Nothing.

Note: This function will break container ordering invariants if with_this is not equivalent to *replace_this according to the ordering rules. This function is faster than erasing and inserting the node, since no rebalancing or comparison is needed.

61.
```
void rebalance();
```

Effects: Rebalances the tree.

Throws: Nothing.

Complexity: Linear.

62.
```
iterator rebalance_subtree(iterator root);
```

Requires: old_root is a node of a tree.

Effects: Rebalances the subtree rooted at old_root.

Returns: The new root of the subtree.

Throws: Nothing.

Complexity: Linear to the elements in the subtree.

63.
```
float balance_factor() const;
```

Returns: The balance factor (alpha) used in this tree

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Constant.

64.
```
void balance_factor(float new_alpha);
```

Requires: new_alpha must be a value between 0.5 and 1.0

Effects: Establishes a new balance factor (alpha) and rebalances the tree if the new balance factor is stricter (less) than the old factor.

Throws: Nothing.

Complexity: Linear to the elements in the subtree.

`sg_multiset` public static functions

1.
```
static sg_multiset & container_from_end_iterator(iterator end_iterator);
```

Precondition: end_iterator must be a valid end iterator of `sg_multiset`.

Effects: Returns a const reference to the `sg_multiset` associated to the end iterator

Throws: Nothing.

Complexity: Constant.

2.
```
static const sg_multiset &
container_from_end_iterator(const_iterator end_iterator);
```

Precondition: end_iterator must be a valid end const_iterator of `sg_multiset`.

Effects: Returns a const reference to the `sg_multiset` associated to the end iterator

Throws: Nothing.

Complexity: Constant.

3.
```
static sg_multiset & container_from_iterator(iterator it);
```

Precondition: it must be a valid iterator of multiset.

Effects: Returns a const reference to the multiset associated to the iterator

Throws: Nothing.

Complexity: Constant.

4.
```
static const sg_multiset & container_from_iterator(const_iterator it);
```

Precondition: it must be a valid const_iterator of multiset.

Effects: Returns a const reference to the multiset associated to the iterator

Throws: Nothing.

Complexity: Constant.

5.
```
static iterator s_iterator_to(reference value);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Requires: value must be an lvalue and shall be in a `sg_multiset` of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid iterator i belonging to the `sg_multiset` that points to the value

Complexity: Constant.

Throws: Nothing.

Note: This static function is available only if the *value traits* is stateless.

6.
```
static const_iterator s_iterator_to(const_reference value);
```

Requires: value must be an lvalue and shall be in a `sg_multiset` of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid const_iterator i belonging to the `sg_multiset` that points to the value

Complexity: Constant.

Throws: Nothing.

Note: This static function is available only if the *value traits* is stateless.

7.
```
static void init_node(reference value);
```

Requires: value shall not be in a sg_multiset/sg_multiset.

Effects: init_node puts the hook of a value in a well-known default state.

Throws: Nothing.

Complexity: Constant time.

Note: This function puts the hook in the well-known default state used by auto_unlink and safe hooks.

Struct template make_sg_multiset

boost::intrusive::make_sg_multiset

Synopsis

```
// In header: <boost/intrusive/sg_set.hpp>

template<typename T, class... Options>
struct make_sg_multiset {
  // types
  typedef implementation_defined type;
};
```

Description

Helper metafunction to define a `sg_multiset` that yields to the same type when the same options (either explicitly or implicitly) are used.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Header <boost/intrusive/sgtree.hpp>

```
namespace boost {
  namespace intrusive {
    template<typename T, class... Options> class sgtree;

    template<typename T, class... Options> struct make_sgtree;
    template<typename T, class... Options>
      bool operator<(const sgtree< T, Options...> & x,
                     const sgtree< T, Options...> & y);
    template<typename T, class... Options>
      bool operator==(const sgtree< T, Options...> & x,
                      const sgtree< T, Options...> & y);
    template<typename T, class... Options>
      bool operator!=(const sgtree< T, Options...> & x,
                      const sgtree< T, Options...> & y);
    template<typename T, class... Options>
      bool operator>(const sgtree< T, Options...> & x,
                     const sgtree< T, Options...> & y);
    template<typename T, class... Options>
      bool operator<=(const sgtree< T, Options...> & x,
                      const sgtree< T, Options...> & y);
    template<typename T, class... Options>
      bool operator>=(const sgtree< T, Options...> & x,
                      const sgtree< T, Options...> & y);
    template<typename T, class... Options>
      void swap(sgtree< T, Options...> & x, sgtree< T, Options...> & y);
  }
}
```

Class template sgtree

boost::intrusive::sgtree

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/intrusive/sgtree.hpp>

template<typename T, class... Options>
class sgtree {
public:
  // types
  typedef Config::value_traits                                                    value_traits; ↲

  typedef real_value_traits::pointer                                              pointer;      ↲

  typedef real_value_traits::const_pointer                                        const_pointer; ↲

  typedef pointer_traits< pointer >::element_type                                 value_type;   ↲

  typedef value_type                                                              key_type;     ↲

  typedef pointer_traits< pointer >::reference                                    reference;    ↲

  typedef pointer_traits< const_pointer >::reference                                const_refer↲
ence;
  typedef pointer_traits< const_pointer >::difference_type                        difference_type; ↲

  typedef Config::size_type                                                       size_type;    ↲

  typedef Config::compare                                                         value_compare; ↲

  typedef value_compare                                                           key_compare;  ↲

  typedef tree_iterator< sgtree, false >                                          iterator;     ↲

  typedef tree_iterator< sgtree, true >                                             const_iterat↲
or;
  typedef unspecified                                                             reverse_iterat↲
or;
  typedef unspecified                                                         const_reverse_iter↲
ator;
  typedef real_value_traits::node_traits                                          node_traits;  ↲

  typedef node_traits::node                                                       node;         ↲

  typedef pointer_traits< pointer >::template rebind_pointer< node >::type        node_ptr;     ↲

  typedef pointer_traits< pointer >::template rebind_pointer< const node >::type const_node_ptr; ↲

  typedef sgtree_algorithms< node_traits >                                        node_algorithms; ↲

  typedef node_algorithms::insert_commit_data                                       insert_com↲
mit_data;

  // construct/copy/destruct
  explicit sgtree(const value_compare & = value_compare(),
                  const value_traits & = value_traits());
  template<typename Iterator>
    sgtree(bool, Iterator, Iterator, const value_compare & = value_compare(),
           const value_traits & = value_traits());
  sgtree(BOOST_RV_REF(sgtree));
  sgtree& operator=(BOOST_RV_REF(sgtree));
  ~sgtree();

  // public member functions
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
const real_value_traits & get_real_value_traits() const;
real_value_traits & get_real_value_traits();
iterator begin();
const_iterator begin() const;
const_iterator cbegin() const;
iterator end();
const_iterator end() const;
const_iterator cend() const;
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
const_reverse_iterator crbegin() const;
reverse_iterator rend();
const_reverse_iterator rend() const;
const_reverse_iterator crend() const;
value_compare value_comp() const;
bool empty() const;
size_type size() const;
void swap(sgtree &);
iterator insert_equal(reference);
iterator insert_equal(const_iterator, reference);
template<typename Iterator> void insert_equal(Iterator, Iterator);
std::pair< iterator, bool > insert_unique(reference);
iterator insert_unique(const_iterator, reference);
template<typename Iterator> void insert_unique(Iterator, Iterator);
template<typename KeyType, typename KeyValueCompare>
  std::pair< iterator, bool >
  insert_unique_check(const KeyType &, KeyValueCompare,
                      insert_commit_data &);
template<typename KeyType, typename KeyValueCompare>
  std::pair< iterator, bool >
  insert_unique_check(const_iterator, const KeyType &, KeyValueCompare,
                      insert_commit_data &);
iterator insert_unique_commit(reference, const insert_commit_data &);
iterator insert_before(const_iterator, reference);
void push_back(reference);
void push_front(reference);
iterator erase(const_iterator);
iterator erase(const_iterator, const_iterator);
size_type erase(const_reference);
template<typename KeyType, typename KeyValueCompare>
  size_type erase(const KeyType &, KeyValueCompare);
template<typename Disposer>
  iterator erase_and_dispose(const_iterator, Disposer);
template<typename Disposer>
  iterator erase_and_dispose(const_iterator, const_iterator, Disposer);
template<typename Disposer>
  size_type erase_and_dispose(const_reference, Disposer);
template<typename KeyType, typename KeyValueCompare, typename Disposer>
  size_type erase_and_dispose(const KeyType &, KeyValueCompare, Disposer);
void clear();
template<typename Disposer> void clear_and_dispose(Disposer);
size_type count(const_reference) const;
template<typename KeyType, typename KeyValueCompare>
  size_type count(const KeyType &, KeyValueCompare) const;
iterator lower_bound(const_reference);
const_iterator lower_bound(const_reference) const;
template<typename KeyType, typename KeyValueCompare>
  iterator lower_bound(const KeyType &, KeyValueCompare);
template<typename KeyType, typename KeyValueCompare>
  const_iterator lower_bound(const KeyType &, KeyValueCompare) const;
iterator upper_bound(const_reference);
template<typename KeyType, typename KeyValueCompare>
  iterator upper_bound(const KeyType &, KeyValueCompare);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
  const_iterator upper_bound(const_reference) const;
  template<typename KeyType, typename KeyValueCompare> 
    const_iterator upper_bound(const KeyType &, KeyValueCompare) const;
  iterator find(const_reference);
  template<typename KeyType, typename KeyValueCompare> 
    iterator find(const KeyType &, KeyValueCompare);
  const_iterator find(const_reference) const;
  template<typename KeyType, typename KeyValueCompare> 
    const_iterator find(const KeyType &, KeyValueCompare) const;
  std::pair< iterator, iterator > equal_range(const_reference);
  template<typename KeyType, typename KeyValueCompare> 
    std::pair< iterator, iterator > 
    equal_range(const KeyType &, KeyValueCompare);
  std::pair< const_iterator, const_iterator > 
  equal_range(const_reference) const;
  template<typename KeyType, typename KeyValueCompare> 
    std::pair< const_iterator, const_iterator > 
    equal_range(const KeyType &, KeyValueCompare) const;
  std::pair< iterator, iterator > 
  bounded_range(const_reference, const_reference, bool, bool);
  template<typename KeyType, typename KeyValueCompare> 
    std::pair< iterator, iterator > 
    bounded_range(const KeyType &, const KeyType &, KeyValueCompare, bool, 
                  bool);
  std::pair< const_iterator, const_iterator > 
  bounded_range(const_reference, const_reference, bool, bool) const;
  template<typename KeyType, typename KeyValueCompare> 
    std::pair< const_iterator, const_iterator > 
    bounded_range(const KeyType &, const KeyType &, KeyValueCompare, bool, 
                  bool) const;
  template<typename Cloner, typename Disposer> 
    void clone_from(const sgtree &, Cloner, Disposer);
  pointer unlink_leftmost_without_rebalance();
  void replace_node(iterator, reference);
  iterator iterator_to(reference);
  const_iterator iterator_to(const_reference) const;
  void rebalance();
  iterator rebalance_subtree(iterator);
  float balance_factor() const;
  void balance_factor(float);

  // public static functions
  static sgtree & container_from_end_iterator(iterator);
  static const sgtree & container_from_end_iterator(const_iterator);
  static sgtree & container_from_iterator(iterator);
  static const sgtree & container_from_iterator(const_iterator);
  static iterator s_iterator_to(reference);
  static const_iterator s_iterator_to(const_reference);
  static void init_node(reference);

  // private static functions
  static sgtree & priv_container_from_end_iterator(const const_iterator &);
  static sgtree & priv_container_from_iterator(const const_iterator &);

  // public data members
  static const bool floating_point;
  static const bool constant_time_size;
  static const bool stateful_value_traits;
};
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Description

The class template sgtree is an intrusive scapegoat tree container, that is used to construct intrusive sg_set and sg_multiset containers. The no-throw guarantee holds only, if the value_compare object doesn't throw.

The template parameter `T` is the type to be managed by the container. The user can specify additional options and if no options are provided default options are used.

The container supports the following options: `base_hook<>/member_hook<>/value_traits<>`, `floating_point<>`, `size_type<>` and `compare<>`.

`sgtree` public construct/copy/destruct

1.
   ```
   explicit sgtree(const value_compare & cmp = value_compare(),
                   const value_traits & v_traits = value_traits());
   ```

   Effects: Constructs an empty tree.

   Complexity: Constant.

   Throws: If value_traits::node_traits::node constructor throws (this does not happen with predefined Boost.Intrusive hooks) or the copy constructorof the value_compare object throws. Basic guarantee.

2.
   ```
   template<typename Iterator>
     sgtree(bool unique, Iterator b, Iterator e,
            const value_compare & cmp = value_compare(),
            const value_traits & v_traits = value_traits());
   ```

   Requires: Dereferencing iterator must yield an lvalue of type value_type. cmp must be a comparison function that induces a strict weak ordering.

   Effects: Constructs an empty tree and inserts elements from [b, e).

   Complexity: Linear in N if [b, e) is already sorted using comp and otherwise N * log N, where N is the distance between first and last.

   Throws: If value_traits::node_traits::node constructor throws (this does not happen with predefined Boost.Intrusive hooks) or the copy constructor/operator() of the value_compare object throws. Basic guarantee.

3.
   ```
   sgtree(BOOST_RV_REF(sgtree) x);
   ```

   Effects: to-do

4.
   ```
   sgtree& operator=(BOOST_RV_REF(sgtree) x);
   ```

   Effects: to-do

5.
   ```
   ~sgtree();
   ```

   Effects: Detaches all elements from this. The objects in the set are not deleted (i.e. no destructors are called), but the nodes according to the `value_traits` template parameter are reinitialized and thus can be reused.

   Complexity: Linear to elements contained in *this.

   Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

`sgtree` public member functions

1.
```
const real_value_traits & get_real_value_traits() const;
```

2.
```
real_value_traits & get_real_value_traits();
```

3.
```
iterator begin();
```

   Effects: Returns an iterator pointing to the beginning of the tree.

   Complexity: Constant.

   Throws: Nothing.

4.
```
const_iterator begin() const;
```

   Effects: Returns a const_iterator pointing to the beginning of the tree.

   Complexity: Constant.

   Throws: Nothing.

5.
```
const_iterator cbegin() const;
```

   Effects: Returns a const_iterator pointing to the beginning of the tree.

   Complexity: Constant.

   Throws: Nothing.

6.
```
iterator end();
```

   Effects: Returns an iterator pointing to the end of the tree.

   Complexity: Constant.

   Throws: Nothing.

7.
```
const_iterator end() const;
```

   Effects: Returns a const_iterator pointing to the end of the tree.

   Complexity: Constant.

   Throws: Nothing.

8.
```
const_iterator cend() const;
```

   Effects: Returns a const_iterator pointing to the end of the tree.

   Complexity: Constant.

   Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

9.
```
reverse_iterator rbegin();
```

Effects: Returns a reverse_iterator pointing to the beginning of the reversed tree.

Complexity: Constant.

Throws: Nothing.

10.
```
const_reverse_iterator rbegin() const;
```

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed tree.

Complexity: Constant.

Throws: Nothing.

11.
```
const_reverse_iterator crbegin() const;
```

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed tree.

Complexity: Constant.

Throws: Nothing.

12.
```
reverse_iterator rend();
```

Effects: Returns a reverse_iterator pointing to the end of the reversed tree.

Complexity: Constant.

Throws: Nothing.

13.
```
const_reverse_iterator rend() const;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed tree.

Complexity: Constant.

Throws: Nothing.

14.
```
const_reverse_iterator crend() const;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed tree.

Complexity: Constant.

Throws: Nothing.

15.
```
value_compare value_comp() const;
```

Effects: Returns the value_compare object used by the tree.

Complexity: Constant.

Throws: If value_compare copy-constructor throws.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

16.
```
bool empty() const;
```

Effects: Returns true if the container is empty.

Complexity: Constant.

Throws: Nothing.

17.
```
size_type size() const;
```

Effects: Returns the number of elements stored in the tree.

Complexity: Linear to elements contained in *this if constant-time size option is disabled. Constant time otherwise.

Throws: Nothing.

18.
```
void swap(sgtree & other);
```

Effects: Swaps the contents of two sgtrees.

Complexity: Constant.

Throws: If the comparison functor's swap call throws.

19.
```
iterator insert_equal(reference value);
```

Requires: value must be an lvalue

Effects: Inserts value into the tree before the upper bound.

Complexity: Average complexity for insert element is at most logarithmic.

Throws: If the internal value_compare ordering function throws. Strong guarantee.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

20.
```
iterator insert_equal(const_iterator hint, reference value);
```

Requires: value must be an lvalue, and "hint" must be a valid iterator.

Effects: Inserts x into the tree, using "hint" as a hint to where it will be inserted. If "hint" is the upper_bound the insertion takes constant time (two comparisons in the worst case)

Complexity: Logarithmic in general, but it is amortized constant time if t is inserted immediately before hint.

Throws: Nothing.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

21.
```
template<typename Iterator> void insert_equal(Iterator b, Iterator e);
```

Requires: Dereferencing iterator must yield an lvalue of type value_type.

Effects: Inserts a each element of a range into the tree before the upper bound of the key of each element.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Insert range is in general O(N * log(N)), where N is the size of the range. However, it is linear in N if the range is already sorted by value_comp().

Throws: Nothing.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

22.
```
std::pair< iterator, bool > insert_unique(reference value);
```

Requires: value must be an lvalue

Effects: Inserts value into the tree if the value is not already present.

Complexity: Average complexity for insert element is at most logarithmic.

Throws: Nothing.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

23.
```
iterator insert_unique(const_iterator hint, reference value);
```

Requires: value must be an lvalue, and "hint" must be a valid iterator

Effects: Tries to insert x into the tree, using "hint" as a hint to where it will be inserted.

Complexity: Logarithmic in general, but it is amortized constant time (two comparisons in the worst case) if t is inserted immediately before hint.

Throws: Nothing.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

24.
```
template<typename Iterator> void insert_unique(Iterator b, Iterator e);
```

Requires: Dereferencing iterator must yield an lvalue of type value_type.

Effects: Tries to insert each element of a range into the tree.

Complexity: Insert range is in general O(N * log(N)), where N is the size of the range. However, it is linear in N if the range is already sorted by value_comp().

Throws: Nothing.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

25.
```
template<typename KeyType, typename KeyValueCompare> 
  std::pair< iterator, bool > 
  insert_unique_check(const KeyType & key, KeyValueCompare key_value_comp, 
                      insert_commit_data & commit_data);
```

Requires: key_value_comp must be a comparison function that induces the same strict weak ordering as value_compare. The difference is that key_value_comp compares an arbitrary key with the contained values.

Effects: Checks if a value can be inserted in the container, using a user provided key instead of the value itself.

Returns: If there is an equivalent value returns a pair containing an iterator to the already present value and false. If the value can be inserted returns true in the returned pair boolean and fills "commit_data" that is meant to be used with the "insert_commit" function.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Average complexity is at most logarithmic.

Throws: If the key_value_comp ordering function throws. Strong guarantee.

Notes: This function is used to improve performance when constructing a value_type is expensive: if there is an equivalent value the constructed object must be discarded. Many times, the part of the node that is used to impose the order is much cheaper to construct than the value_type and this function offers the possibility to use that part to check if the insertion will be successful.

If the check is successful, the user can construct the value_type and use "insert_commit" to insert the object in constant-time. This gives a total logarithmic complexity to the insertion: check(O(log(N)) + commit(O(1)).

"commit_data" remains valid for a subsequent "insert_commit" only if no more objects are inserted or erased from the container.

26.
```
template<typename KeyType, typename KeyValueCompare> 
  std::pair< iterator, bool > 
  insert_unique_check(const_iterator hint, const KeyType & key, 
                      KeyValueCompare key_value_comp, 
                      insert_commit_data & commit_data);
```

Requires: key_value_comp must be a comparison function that induces the same strict weak ordering as value_compare. The difference is that key_value_comp compares an arbitrary key with the contained values.

Effects: Checks if a value can be inserted in the container, using a user provided key instead of the value itself, using "hint" as a hint to where it will be inserted.

Returns: If there is an equivalent value returns a pair containing an iterator to the already present value and false. If the value can be inserted returns true in the returned pair boolean and fills "commit_data" that is meant to be used with the "insert_commit" function.

Complexity: Logarithmic in general, but it's amortized constant time if t is inserted immediately before hint.

Throws: If the key_value_comp ordering function throws. Strong guarantee.

Notes: This function is used to improve performance when constructing a value_type is expensive: if there is an equivalent value the constructed object must be discarded. Many times, the part of the constructing that is used to impose the order is much cheaper to construct than the value_type and this function offers the possibility to use that key to check if the insertion will be successful.

If the check is successful, the user can construct the value_type and use "insert_commit" to insert the object in constant-time. This can give a total constant-time complexity to the insertion: check(O(1)) + commit(O(1)).

"commit_data" remains valid for a subsequent "insert_commit" only if no more objects are inserted or erased from the container.

27.
```
iterator insert_unique_commit(reference value, 
                              const insert_commit_data & commit_data);
```

Requires: value must be an lvalue of type value_type. commit_data must have been obtained from a previous call to "insert_check". No objects should have been inserted or erased from the container between the "insert_check" that filled "commit_data" and the call to "insert_commit".

Effects: Inserts the value in the `avl_set` using the information obtained from the "commit_data" that a previous "insert_check" filled.

Returns: An iterator to the newly inserted object.

Complexity: Constant time.

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Notes: This function has only sense if a "insert_check" has been previously executed to fill "commit_data". No value should be inserted or erased between the "insert_check" and "insert_commit" calls.

28.
```
iterator insert_before(const_iterator pos, reference value);
```

Requires: value must be an lvalue, "pos" must be a valid iterator (or end) and must be the succesor of value once inserted according to the predicate

Effects: Inserts x into the tree before "pos".

Complexity: Constant time.

Throws: Nothing.

Note: This function does not check preconditions so if "pos" is not the successor of "value" tree ordering invariant will be broken. This is a low-level function to be used only for performance reasons by advanced users.

29.
```
void push_back(reference value);
```

Requires: value must be an lvalue, and it must be no less than the greatest inserted key

Effects: Inserts x into the tree in the last position.

Complexity: Constant time.

Throws: Nothing.

Note: This function does not check preconditions so if value is less than the greatest inserted key tree ordering invariant will be broken. This function is slightly more efficient than using "insert_before". This is a low-level function to be used only for performance reasons by advanced users.

30.
```
void push_front(reference value);
```

Requires: value must be an lvalue, and it must be no greater than the minimum inserted key

Effects: Inserts x into the tree in the first position.

Complexity: Constant time.

Throws: Nothing.

Note: This function does not check preconditions so if value is greater than the minimum inserted key tree ordering invariant will be broken. This function is slightly more efficient than using "insert_before". This is a low-level function to be used only for performance reasons by advanced users.

31.
```
iterator erase(const_iterator i);
```

Effects: Erases the element pointed to by pos.

Complexity: Average complexity for erase element is constant time.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

32.
```
iterator erase(const_iterator b, const_iterator e);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Erases the range pointed to by b end e.

Complexity: Average complexity for erase range is at most O(log(size() + N)), where N is the number of elements in the range.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

33.
```
size_type erase(const_reference value);
```

Effects: Erases all the elements with the given value.

Returns: The number of erased elements.

Complexity: O(log(size() + N).

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

34.
```
template<typename KeyType, typename KeyValueCompare> 
  size_type erase(const KeyType & key, KeyValueCompare comp);
```

Effects: Erases all the elements with the given key. according to the comparison functor "comp".

Returns: The number of erased elements.

Complexity: O(log(size() + N).

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

35.
```
template<typename Disposer> 
  iterator erase_and_dispose(const_iterator i, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases the element pointed to by pos. Disposer::operator()(pointer) is called for the removed element.

Complexity: Average complexity for erase element is constant time.

Throws: Nothing.

Note: Invalidates the iterators to the erased elements.

36.
```
template<typename Disposer> 
  iterator erase_and_dispose(const_iterator b, const_iterator e, 
                             Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases the range pointed to by b end e. Disposer::operator()(pointer) is called for the removed elements.

Complexity: Average complexity for erase range is at most O(log(size() + N)), where N is the number of elements in the range.

Throws: Nothing.

Note: Invalidates the iterators to the erased elements.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

37.
```
template<typename Disposer>
  size_type erase_and_dispose(const_reference value, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases all the elements with the given value. Disposer::operator()(pointer) is called for the removed elements.

Returns: The number of erased elements.

Complexity: O(log(size() + N).

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

38.
```
template<typename KeyType, typename KeyValueCompare, typename Disposer>
  size_type erase_and_dispose(const KeyType & key, KeyValueCompare comp,
                              Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases all the elements with the given key. according to the comparison functor "comp". Disposer::operator()(pointer) is called for the removed elements.

Returns: The number of erased elements.

Complexity: O(log(size() + N).

Throws: Nothing.

Note: Invalidates the iterators to the erased elements.

39.
```
void clear();
```

Effects: Erases all of the elements.

Complexity: Linear to the number of elements on the container. if it's a safe-mode or auto-unlink value_type. Constant time otherwise.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

40.
```
template<typename Disposer> void clear_and_dispose(Disposer disposer);
```

Effects: Erases all of the elements calling disposer(p) for each node to be erased. **Complexity**: Average complexity for is at most O(log(size() + N)), where N is the number of elements in the container.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. Calls N times to disposer functor.

41.
```
size_type count(const_reference value) const;
```

Effects: Returns the number of contained elements with the given value

Complexity: Logarithmic to the number of elements contained plus lineal to number of objects with the given value.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: Nothing.

42.
```
template<typename KeyType, typename KeyValueCompare> 
  size_type count(const KeyType & key, KeyValueCompare comp) const;
```

Effects: Returns the number of contained elements with the given key

Complexity: Logarithmic to the number of elements contained plus lineal to number of objects with the given key.

Throws: Nothing.

43.
```
iterator lower_bound(const_reference value);
```

Effects: Returns an iterator to the first element whose key is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

44.
```
const_iterator lower_bound(const_reference value) const;
```

Effects: Returns an iterator to the first element whose key is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

45.
```
template<typename KeyType, typename KeyValueCompare> 
  iterator lower_bound(const KeyType & key, KeyValueCompare comp);
```

Effects: Returns an iterator to the first element whose key is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

46.
```
template<typename KeyType, typename KeyValueCompare> 
  const_iterator lower_bound(const KeyType & key, KeyValueCompare comp) const;
```

Effects: Returns a const iterator to the first element whose key is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

47.
```
iterator upper_bound(const_reference value);
```

Effects: Returns an iterator to the first element whose key is greater than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

48.
```
template<typename KeyType, typename KeyValueCompare> 
  iterator upper_bound(const KeyType & key, KeyValueCompare comp);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Returns an iterator to the first element whose key is greater than k according to comp or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

49.
```
const_iterator upper_bound(const_reference value) const;
```

Effects: Returns an iterator to the first element whose key is greater than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

50.
```
template<typename KeyType, typename KeyValueCompare>
  const_iterator upper_bound(const KeyType & key, KeyValueCompare comp) const;
```

Effects: Returns an iterator to the first element whose key is greater than k according to comp or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

51.
```
iterator find(const_reference value);
```

Effects: Finds an iterator to the first element whose key is k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

52.
```
template<typename KeyType, typename KeyValueCompare>
  iterator find(const KeyType & key, KeyValueCompare comp);
```

Effects: Finds an iterator to the first element whose key is k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

53.
```
const_iterator find(const_reference value) const;
```

Effects: Finds a const_iterator to the first element whose key is k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

54.
```
template<typename KeyType, typename KeyValueCompare>
  const_iterator find(const KeyType & key, KeyValueCompare comp) const;
```

Effects: Finds a const_iterator to the first element whose key is k or end() if that element does not exist.

Complexity: Logarithmic.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: Nothing.

55.
```
std::pair< iterator, iterator > equal_range(const_reference value);
```

Effects: Finds a range containing all elements whose key is k or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: Nothing.

56.
```
template<typename KeyType, typename KeyValueCompare>
  std::pair< iterator, iterator >
  equal_range(const KeyType & key, KeyValueCompare comp);
```

Effects: Finds a range containing all elements whose key is k or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: Nothing.

57.
```
std::pair< const_iterator, const_iterator >
equal_range(const_reference value) const;
```

Effects: Finds a range containing all elements whose key is k or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: Nothing.

58.
```
template<typename KeyType, typename KeyValueCompare>
  std::pair< const_iterator, const_iterator >
  equal_range(const KeyType & key, KeyValueCompare comp) const;
```

Effects: Finds a range containing all elements whose key is k or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: Nothing.

59.
```
std::pair< iterator, iterator >
bounded_range(const_reference lower_value, const_reference upper_value,
              bool left_closed, bool right_closed);
```

Requires: 'lower_value' must not be greater than 'upper_value'. If 'lower_value' == 'upper_value', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key) if left_closed, upper_bound(lower_key) otherwise

second = upper_bound(upper_key) if right_closed, lower_bound(upper_key) otherwise

Complexity: Logarithmic.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: If the predicate throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_value and upper_value.

60.
```
template<typename KeyType, typename KeyValueCompare> 
  std::pair< iterator, iterator > 
  bounded_range(const KeyType & lower_key, const KeyType & upper_key, 
                KeyValueCompare comp, bool left_closed, bool right_closed);
```

Requires: KeyValueCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. 'lower_key' must not be greater than 'upper_key' according to 'comp'. If 'lower_key' == 'upper_key', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key, comp) if left_closed, upper_bound(lower_key, comp) otherwise

second = upper_bound(upper_key, comp) if right_closed, lower_bound(upper_key, comp) otherwise

Complexity: Logarithmic.

Throws: If "comp" throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_key and upper_key.

61.
```
std::pair< const_iterator, const_iterator > 
bounded_range(const_reference lower_value, const_reference upper_value, 
              bool left_closed, bool right_closed) const;
```

Requires: 'lower_value' must not be greater than 'upper_value'. If 'lower_value' == 'upper_value', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key) if left_closed, upper_bound(lower_key) otherwise

second = upper_bound(upper_key) if right_closed, lower_bound(upper_key) otherwise

Complexity: Logarithmic.

Throws: If the predicate throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_value and upper_value.

62.
```
template<typename KeyType, typename KeyValueCompare> 
  std::pair< const_iterator, const_iterator > 
  bounded_range(const KeyType & lower_key, const KeyType & upper_key, 
                KeyValueCompare comp, bool left_closed, bool right_closed) const;
```

Requires: KeyValueCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. 'lower_key' must not be greater than 'upper_key' according to 'comp'. If 'lower_key' == 'upper_key', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key, comp) if left_closed, upper_bound(lower_key, comp) otherwise

second = upper_bound(upper_key, comp) if right_closed, lower_bound(upper_key, comp) otherwise

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Logarithmic.

Throws: If "comp" throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_key and upper_key.

63.
```
template<typename Cloner, typename Disposer>
  void clone_from(const sgtree & src, Cloner cloner, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw. Cloner should yield to nodes equivalent to the original nodes.

Effects: Erases all the elements from *this calling Disposer::operator()(pointer), clones all the elements from src calling Cloner::operator()(const_reference) and inserts them on *this. Copies the predicate from the source container.

If cloner throws, all cloned elements are unlinked and disposed calling Disposer::operator()(pointer).

Complexity: Linear to erased plus inserted elements.

Throws: If cloner throws or predicate copy assignment throws. Basic guarantee.

64.
```
pointer unlink_leftmost_without_rebalance();
```

Effects: Unlinks the leftmost node from the tree.

Complexity: Average complexity is constant time.

Throws: Nothing.

Notes: This function breaks the tree and the tree can only be used for more unlink_leftmost_without_rebalance calls. This function is normally used to achieve a step by step controlled destruction of the tree.

65.
```
void replace_node(iterator replace_this, reference with_this);
```

Requires: replace_this must be a valid iterator of *this and with_this must not be inserted in any tree.

Effects: Replaces replace_this in its position in the tree with with_this. The tree does not need to be rebalanced.

Complexity: Constant.

Throws: Nothing.

Note: This function will break container ordering invariants if with_this is not equivalent to *replace_this according to the ordering rules. This function is faster than erasing and inserting the node, since no rebalancing or comparison is needed.

66.
```
iterator iterator_to(reference value);
```

Requires: value must be an lvalue and shall be in a set of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid iterator i belonging to the set that points to the value

Complexity: Constant.

Throws: Nothing.

67.
```
const_iterator iterator_to(const_reference value) const;
```

Requires: value must be an lvalue and shall be in a set of appropriate type. Otherwise the behavior is undefined.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Returns: a valid const_iterator i belonging to the set that points to the value

Complexity: Constant.

Throws: Nothing.

68.
```
void rebalance();
```

Effects: Rebalances the tree.

Throws: Nothing.

Complexity: Linear.

69.
```
iterator rebalance_subtree(iterator root);
```

Requires: old_root is a node of a tree.

Effects: Rebalances the subtree rooted at old_root.

Returns: The new root of the subtree.

Throws: Nothing.

Complexity: Linear to the elements in the subtree.

70.
```
float balance_factor() const;
```

Returns: The balance factor (alpha) used in this tree

Throws: Nothing.

Complexity: Constant.

71.
```
void balance_factor(float new_alpha);
```

Requires: new_alpha must be a value between 0.5 and 1.0

Effects: Establishes a new balance factor (alpha) and rebalances the tree if the new balance factor is stricter (less) than the old factor.

Throws: Nothing.

Complexity: Linear to the elements in the subtree.

`sgtree` public static functions

1.
```
static sgtree & container_from_end_iterator(iterator end_iterator);
```

Precondition: end_iterator must be a valid end iterator of sgtree.

Effects: Returns a const reference to the sgtree associated to the end iterator

Throws: Nothing.

Complexity: Constant.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

2.
```
static const sgtree & container_from_end_iterator(const_iterator end_iterator);
```

Precondition: end_iterator must be a valid end const_iterator of sgtree.

Effects: Returns a const reference to the sgtree associated to the end iterator

Throws: Nothing.

Complexity: Constant.

3.
```
static sgtree & container_from_iterator(iterator it);
```

Precondition: it must be a valid iterator of rbtree.

Effects: Returns a const reference to the tree associated to the iterator

Throws: Nothing.

Complexity: Logarithmic.

4.
```
static const sgtree & container_from_iterator(const_iterator it);
```

Precondition: it must be a valid end const_iterator of rbtree.

Effects: Returns a const reference to the tree associated to the iterator

Throws: Nothing.

Complexity: Logarithmic.

5.
```
static iterator s_iterator_to(reference value);
```

Requires: value must be an lvalue and shall be in a set of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid iterator i belonging to the set that points to the value

Complexity: Constant.

Throws: Nothing.

Note: This static function is available only if the *value traits* is stateless.

6.
```
static const_iterator s_iterator_to(const_reference value);
```

Requires: value must be an lvalue and shall be in a set of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid const_iterator i belonging to the set that points to the value

Complexity: Constant.

Throws: Nothing.

Note: This static function is available only if the *value traits* is stateless.

7.
```
static void init_node(reference value);
```

Requires: value shall not be in a tree.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: init_node puts the hook of a value in a well-known default state.

Throws: Nothing.

Complexity: Constant time.

Note: This function puts the hook in the well-known default state used by auto_unlink and safe hooks.

`sgtree` private static functions

1.
```
static sgtree &
priv_container_from_end_iterator(const const_iterator & end_iterator);
```

2.
```
static sgtree & priv_container_from_iterator(const const_iterator & it);
```

Struct template make_sgtree

boost::intrusive::make_sgtree

Synopsis

```
// In header: <boost/intrusive/sgtree.hpp>

template<typename T, class... Options>
struct make_sgtree {
  // types
  typedef implementation_defined type;
};
```

Description

Helper metafunction to define a `sgtree` that yields to the same type when the same options (either explicitly or implicitly) are used.

Header <boost/intrusive/sgtree_algorithms.hpp>

```
namespace boost {
  namespace intrusive {
    template<typename NodeTraits> class sgtree_algorithms;
  }
}
```

Class template sgtree_algorithms

boost::intrusive::sgtree_algorithms

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/intrusive/sgtree_algorithms.hpp>

template<typename NodeTraits> 
class sgtree_algorithms {
public:
  // types
  typedef NodeTraits::node           node;          
  typedef NodeTraits                 node_traits;   
  typedef NodeTraits::node_ptr       node_ptr;      
  typedef NodeTraits::const_node_ptr const_node_ptr;

  // member classes/structs/unions

  struct insert_commit_data : public insert_commit_data {

    // public data members
    std::size_t depth;
  };

  // public static functions
  static node_ptr begin_node(const const_node_ptr &);
  static node_ptr end_node(const const_node_ptr &);
  static void swap_tree(const node_ptr &, const node_ptr &);
  static void swap_nodes(const node_ptr &, const node_ptr &);
  static void swap_nodes(const node_ptr &, const node_ptr &, const node_ptr &, 
                         const node_ptr &);
  static void replace_node(const node_ptr &, const node_ptr &);
  static void replace_node(const node_ptr &, const node_ptr &, 
                           const node_ptr &);
  static void unlink(const node_ptr &);
  static node_ptr unlink_leftmost_without_rebalance(const node_ptr &);
  static bool unique(const const_node_ptr &);
  static std::size_t count(const const_node_ptr &);
  static std::size_t size(const const_node_ptr &);
  static node_ptr next_node(const node_ptr &);
  static node_ptr prev_node(const node_ptr &);
  static void init(const node_ptr &);
  static void init_header(const node_ptr &);
  template<typename AlphaByMaxSize> 
    static node_ptr 
    erase(const node_ptr &, const node_ptr &, std::size_t, std::size_t &, 
          AlphaByMaxSize);
  template<typename Cloner, typename Disposer> 
    static void clone(const const_node_ptr &, const node_ptr &, Cloner, 
                      Disposer);
  template<typename Disposer> 
    static void clear_and_dispose(const node_ptr &, Disposer);
  template<typename KeyType, typename KeyNodePtrCompare> 
    static node_ptr 
    lower_bound(const const_node_ptr &, const KeyType &, KeyNodePtrCompare);
  template<typename KeyType, typename KeyNodePtrCompare> 
    static node_ptr 
    upper_bound(const const_node_ptr &, const KeyType &, KeyNodePtrCompare);
  template<typename KeyType, typename KeyNodePtrCompare> 
    static node_ptr 
    find(const const_node_ptr &, const KeyType &, KeyNodePtrCompare);
  template<typename KeyType, typename KeyNodePtrCompare> 
    static std::pair< node_ptr, node_ptr > 
    equal_range(const const_node_ptr &, const KeyType &, KeyNodePtrCompare);
  template<typename KeyType, typename KeyNodePtrCompare> 
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
    static std::pair< node_ptr, node_ptr >
    bounded_range(const const_node_ptr &, const KeyType &, const KeyType &,
                  KeyNodePtrCompare, bool, bool);
  template<typename NodePtrCompare, typename H_Alpha>
    static node_ptr
    insert_equal_upper_bound(const node_ptr &, const node_ptr &,
                             NodePtrCompare, std::size_t, H_Alpha,
                             std::size_t &);
  template<typename NodePtrCompare, typename H_Alpha>
    static node_ptr
    insert_equal_lower_bound(const node_ptr &, const node_ptr &,
                             NodePtrCompare, std::size_t, H_Alpha,
                             std::size_t &);
  template<typename NodePtrCompare, typename H_Alpha>
    static node_ptr
    insert_equal(const node_ptr &, const node_ptr &, const node_ptr &,
                 NodePtrCompare, std::size_t, H_Alpha, std::size_t &);
  template<typename KeyType, typename KeyNodePtrCompare>
    static std::pair< node_ptr, bool >
    insert_unique_check(const const_node_ptr &, const KeyType &,
                        KeyNodePtrCompare, insert_commit_data &);
  template<typename H_Alpha>
    static node_ptr
    insert_before(const node_ptr &, const node_ptr &, const node_ptr &,
                  std::size_t, H_Alpha, std::size_t &);
  template<typename H_Alpha>
    static void push_back(const node_ptr &, const node_ptr &, std::size_t,
                          H_Alpha, std::size_t &);
  template<typename H_Alpha>
    static void push_front(const node_ptr &, const node_ptr &, std::size_t,
                           H_Alpha, std::size_t &);
  template<typename KeyType, typename KeyNodePtrCompare>
    static std::pair< node_ptr, bool >
    insert_unique_check(const const_node_ptr &, const node_ptr &,
                        const KeyType &, KeyNodePtrCompare,
                        insert_commit_data &);
  template<typename H_Alpha>
    static void insert_unique_commit(const node_ptr &, const node_ptr &,
                                     const insert_commit_data &, std::size_t,
                                     H_Alpha, std::size_t &);
  static void rebalance(const node_ptr &);
  static node_ptr rebalance_subtree(const node_ptr &);
  static node_ptr get_header(const node_ptr &);
};
```

Description

sgtree_algorithms is configured with a NodeTraits class, which encapsulates the information about the node to be manipulated. NodeTraits must support the following interface:

Typedefs:

`node`: The type of the node that forms the circular list

`node_ptr`: A pointer to a node

`const_node_ptr`: A pointer to a const node

Static functions:

```
static node_ptr get_parent(const_node_ptr n);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
static void set_parent(node_ptr n, node_ptr parent);
```

```
static node_ptr get_left(const_node_ptr n);
```

```
static void set_left(node_ptr n, node_ptr left);
```

```
static node_ptr get_right(const_node_ptr n);
```

```
static void set_right(node_ptr n, node_ptr right);
```

`sgtree_algorithms` public static functions

1.
   ```
   static node_ptr begin_node(const const_node_ptr & header);
   ```

2.
   ```
   static node_ptr end_node(const const_node_ptr & header);
   ```

3.
   ```
   static void swap_tree(const node_ptr & header1, const node_ptr & header2);
   ```

   Requires: header1 and header2 must be the header nodes of two trees.

   Effects: Swaps two trees. After the function header1 will contain links to the second tree and header2 will have links to the first tree.

   Complexity: Constant.

   Throws: Nothing.

4.
   ```
   static void swap_nodes(const node_ptr & node1, const node_ptr & node2);
   ```

   Requires: node1 and node2 can't be header nodes of two trees.

   Effects: Swaps two nodes. After the function node1 will be inserted in the position node2 before the function. node2 will be inserted in the position node1 had before the function.

   Complexity: Logarithmic.

   Throws: Nothing.

   Note: This function will break container ordering invariants if node1 and node2 are not equivalent according to the ordering rules.

   Experimental function

5.
   ```
   static void swap_nodes(const node_ptr & node1, const node_ptr & header1,
                          const node_ptr & node2, const node_ptr & header2);
   ```

   Requires: node1 and node2 can't be header nodes of two trees with header header1 and header2.

   Effects: Swaps two nodes. After the function node1 will be inserted in the position node2 before the function. node2 will be inserted in the position node1 had before the function.

   Complexity: Constant.

   Throws: Nothing.

   Note: This function will break container ordering invariants if node1 and node2 are not equivalent according to the ordering rules.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Experimental function

6.
```
static void replace_node(const node_ptr & node_to_be_replaced,
                         const node_ptr & new_node);
```

Requires: node_to_be_replaced must be inserted in a tree and new_node must not be inserted in a tree.

Effects: Replaces node_to_be_replaced in its position in the tree with new_node. The tree does not need to be rebalanced

Complexity: Logarithmic.

Throws: Nothing.

Note: This function will break container ordering invariants if new_node is not equivalent to node_to_be_replaced according to the ordering rules. This function is faster than erasing and inserting the node, since no rebalancing and comparison is needed.

Experimental function

7.
```
static void replace_node(const node_ptr & node_to_be_replaced,
                         const node_ptr & header, const node_ptr & new_node);
```

Requires: node_to_be_replaced must be inserted in a tree with header "header" and new_node must not be inserted in a tree.

Effects: Replaces node_to_be_replaced in its position in the tree with new_node. The tree does not need to be rebalanced

Complexity: Constant.

Throws: Nothing.

Note: This function will break container ordering invariants if new_node is not equivalent to node_to_be_replaced according to the ordering rules. This function is faster than erasing and inserting the node, since no rebalancing or comparison is needed.

Experimental function

8.
```
static void unlink(const node_ptr & node);
```

Requires: node is a tree node but not the header.

Effects: Unlinks the node and rebalances the tree.

Complexity: Average complexity is constant time.

Throws: Nothing.

9.
```
static node_ptr unlink_leftmost_without_rebalance(const node_ptr & header);
```

Requires: header is the header of a tree.

Effects: Unlinks the leftmost node from the tree, and updates the header link to the new leftmost node.

Complexity: Average complexity is constant time.

Throws: Nothing.

Notes: This function breaks the tree and the tree can only be used for more unlink_leftmost_without_rebalance calls. This function is normally used to achieve a step by step controlled destruction of the tree.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

10.
```
static bool unique(const const_node_ptr & node);
```

Requires: node is a node of the tree or an node initialized by init(...).

Effects: Returns true if the node is initialized by init().

Complexity: Constant time.

Throws: Nothing.

11.
```
static std::size_t count(const const_node_ptr & node);
```

Requires: node is a node of the tree but it's not the header.

Effects: Returns the number of nodes of the subtree.

Complexity: Linear time.

Throws: Nothing.

12.
```
static std::size_t size(const const_node_ptr & header);
```

Requires: header is the header node of the tree.

Effects: Returns the number of nodes above the header.

Complexity: Linear time.

Throws: Nothing.

13.
```
static node_ptr next_node(const node_ptr & p);
```

Requires: p is a node from the tree except the header.

Effects: Returns the next node of the tree.

Complexity: Average constant time.

Throws: Nothing.

14.
```
static node_ptr prev_node(const node_ptr & p);
```

Requires: p is a node from the tree except the leftmost node.

Effects: Returns the previous node of the tree.

Complexity: Average constant time.

Throws: Nothing.

15.
```
static void init(const node_ptr & node);
```

Requires: node must not be part of any tree.

Effects: After the function unique(node) == true.

Complexity: Constant.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: Nothing.

Nodes: If node is inserted in a tree, this function corrupts the tree.

16.
```
static void init_header(const node_ptr & header);
```

Requires: node must not be part of any tree.

Effects: Initializes the header to represent an empty tree. unique(header) == true.

Complexity: Constant.

Throws: Nothing.

Nodes: If node is inserted in a tree, this function corrupts the tree.

17.
```
template<typename AlphaByMaxSize>
  static node_ptr
  erase(const node_ptr & header, const node_ptr & z, std::size_t tree_size,
        std::size_t & max_tree_size, AlphaByMaxSize alpha_by_maxsize);
```

Requires: header must be the header of a tree, z a node of that tree and z != header.

Effects: Erases node "z" from the tree with header "header".

Complexity: Amortized constant time.

Throws: Nothing.

18.
```
template<typename Cloner, typename Disposer>
  static void clone(const const_node_ptr & source_header,
                    const node_ptr & target_header, Cloner cloner,
                    Disposer disposer);
```

Requires: "cloner" must be a function object taking a node_ptr and returning a new cloned node of it. "disposer" must take a node_ptr and shouldn't throw.

Effects: First empties target tree calling `void disposer::operator()(const node_ptr &)` for every node of the tree except the header.

Then, duplicates the entire tree pointed by "source_header" cloning each source node with `node_ptr Cloner::operator()(const node_ptr &)` to obtain the nodes of the target tree. If "cloner" throws, the cloned target nodes are disposed using `void disposer(const node_ptr &)`.

Complexity: Linear to the number of element of the source tree plus the. number of elements of tree target tree when calling this function.

Throws: If cloner functor throws. If this happens target nodes are disposed.

19.
```
template<typename Disposer>
  static void clear_and_dispose(const node_ptr & header, Disposer disposer);
```

Requires: "disposer" must be an object function taking a node_ptr parameter and shouldn't throw.

Effects: Empties the target tree calling `void disposer::operator()(const node_ptr &)` for every node of the tree except the header.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Linear to the number of element of the source tree plus the. number of elements of tree target tree when calling this function.

Throws: If cloner functor throws. If this happens target nodes are disposed.

20.
```
template<typename KeyType, typename KeyNodePtrCompare> 
  static node_ptr 
  lower_bound(const const_node_ptr & header, const KeyType & key, 
              KeyNodePtrCompare comp);
```

Requires: "header" must be the header node of a tree. KeyNodePtrCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. KeyNodePtrCompare can compare KeyType with tree's node_ptrs.

Effects: Returns an node_ptr to the first element that is not less than "key" according to "comp" or "header" if that element does not exist.

Complexity: Logarithmic.

Throws: If "comp" throws.

21.
```
template<typename KeyType, typename KeyNodePtrCompare> 
  static node_ptr 
  upper_bound(const const_node_ptr & header, const KeyType & key, 
              KeyNodePtrCompare comp);
```

Requires: "header" must be the header node of a tree. KeyNodePtrCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. KeyNodePtrCompare can compare KeyType with tree's node_ptrs.

Effects: Returns an node_ptr to the first element that is greater than "key" according to "comp" or "header" if that element does not exist.

Complexity: Logarithmic.

Throws: If "comp" throws.

22.
```
template<typename KeyType, typename KeyNodePtrCompare> 
  static node_ptr 
  find(const const_node_ptr & header, const KeyType & key, 
       KeyNodePtrCompare comp);
```

Requires: "header" must be the header node of a tree. KeyNodePtrCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. KeyNodePtrCompare can compare KeyType with tree's node_ptrs.

Effects: Returns an node_ptr to the element that is equivalent to "key" according to "comp" or "header" if that element does not exist.

Complexity: Logarithmic.

Throws: If "comp" throws.

23.
```
template<typename KeyType, typename KeyNodePtrCompare> 
  static std::pair< node_ptr, node_ptr > 
  equal_range(const const_node_ptr & header, const KeyType & key, 
              KeyNodePtrCompare comp);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Requires: "header" must be the header node of a tree. KeyNodePtrCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. KeyNodePtrCompare can compare KeyType with tree's node_ptrs.

Effects: Returns an a pair of node_ptr delimiting a range containing all elements that are equivalent to "key" according to "comp" or an empty range that indicates the position where those elements would be if they there are no equivalent elements.

Complexity: Logarithmic.

Throws: If "comp" throws.

24.
```
template<typename KeyType, typename KeyNodePtrCompare>
  static std::pair< node_ptr, node_ptr >
  bounded_range(const const_node_ptr & header, const KeyType & lower_key,
                const KeyType & upper_key, KeyNodePtrCompare comp,
                bool left_closed, bool right_closed);
```

Requires: "header" must be the header node of a tree. KeyNodePtrCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. KeyNodePtrCompare can compare KeyType with tree's node_ptrs. 'lower_key' must not be greater than 'upper_key' according to 'comp'. If 'lower_key' == 'upper_key', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key) if left_closed, upper_bound(lower_key) otherwise

second = upper_bound(upper_key) if right_closed, lower_bound(upper_key) otherwise

Complexity: Logarithmic.

Throws: If "comp" throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_key and upper_key.

25.
```
template<typename NodePtrCompare, typename H_Alpha>
  static node_ptr
  insert_equal_upper_bound(const node_ptr & h, const node_ptr & new_node,
                           NodePtrCompare comp, std::size_t tree_size,
                           H_Alpha h_alpha, std::size_t & max_tree_size);
```

Requires: "h" must be the header node of a tree. NodePtrCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. NodePtrCompare compares two node_ptrs.

Effects: Inserts new_node into the tree before the upper bound according to "comp".

Complexity: Average complexity for insert element is at most logarithmic.

Throws: If "comp" throws.

26.
```
template<typename NodePtrCompare, typename H_Alpha>
  static node_ptr
  insert_equal_lower_bound(const node_ptr & h, const node_ptr & new_node,
                           NodePtrCompare comp, std::size_t tree_size,
                           H_Alpha h_alpha, std::size_t & max_tree_size);
```

Requires: "h" must be the header node of a tree. NodePtrCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. NodePtrCompare compares two node_ptrs.

Effects: Inserts new_node into the tree before the lower bound according to "comp".

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Average complexity for insert element is at most logarithmic.

Throws: If "comp" throws.

27.
```
template<typename NodePtrCompare, typename H_Alpha>
  static node_ptr
  insert_equal(const node_ptr & header, const node_ptr & hint,
               const node_ptr & new_node, NodePtrCompare comp,
               std::size_t tree_size, H_Alpha h_alpha,
               std::size_t & max_tree_size);
```

Requires: "header" must be the header node of a tree. NodePtrCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. NodePtrCompare compares two node_ptrs. "hint" is node from the "header"'s tree.

Effects: Inserts new_node into the tree, using "hint" as a hint to where it will be inserted. If "hint" is the upper_bound the insertion takes constant time (two comparisons in the worst case).

Complexity: Logarithmic in general, but it is amortized constant time if new_node is inserted immediately before "hint".

Throws: If "comp" throws.

28.
```
template<typename KeyType, typename KeyNodePtrCompare>
  static std::pair< node_ptr, bool >
  insert_unique_check(const const_node_ptr & header, const KeyType & key,
                      KeyNodePtrCompare comp,
                      insert_commit_data & commit_data);
```

Requires: "header" must be the header node of a tree. KeyNodePtrCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. NodePtrCompare compares KeyType with a node_ptr.

Effects: Checks if there is an equivalent node to "key" in the tree according to "comp" and obtains the needed information to realize a constant-time node insertion if there is no equivalent node.

Returns: If there is an equivalent value returns a pair containing a node_ptr to the already present node and false. If there is not equivalent key can be inserted returns true in the returned pair's boolean and fills "commit_data" that is meant to be used with the "insert_commit" function to achieve a constant-time insertion function.

Complexity: Average complexity is at most logarithmic.

Throws: If "comp" throws.

Notes: This function is used to improve performance when constructing a node is expensive and the user does not want to have two equivalent nodes in the tree: if there is an equivalent value the constructed object must be discarded. Many times, the part of the node that is used to impose the order is much cheaper to construct than the node and this function offers the possibility to use that part to check if the insertion will be successful.

If the check is successful, the user can construct the node and use "insert_commit" to insert the node in constant-time. This gives a total logarithmic complexity to the insertion: check(O(log(N)) + commit(O(1)).

"commit_data" remains valid for a subsequent "insert_unique_commit" only if no more objects are inserted or erased from the set.

29.
```
template<typename H_Alpha>
  static node_ptr
  insert_before(const node_ptr & header, const node_ptr & pos,
                const node_ptr & new_node, std::size_t tree_size,
                H_Alpha h_alpha, std::size_t & max_tree_size);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Requires: "header" must be the header node of a tree. "pos" must be a valid iterator or header (end) node. "pos" must be an iterator pointing to the successor to "new_node" once inserted according to the order of already inserted nodes. This function does not check "pos" and this precondition must be guaranteed by the caller.

Effects: Inserts new_node into the tree before "pos".

Complexity: Constant-time.

Throws: Nothing.

Note: If "pos" is not the successor of the newly inserted "new_node" tree invariants might be broken.

30.
```
template<typename H_Alpha> 
  static void push_back(const node_ptr & header, const node_ptr & new_node, 
                        std::size_t tree_size, H_Alpha h_alpha, 
                        std::size_t & max_tree_size);
```

Requires: "header" must be the header node of a tree. "new_node" must be, according to the used ordering no less than the greatest inserted key.

Effects: Inserts new_node into the tree before "pos".

Complexity: Constant-time.

Throws: Nothing.

Note: If "new_node" is less than the greatest inserted key tree invariants are broken. This function is slightly faster than using "insert_before".

31.
```
template<typename H_Alpha> 
  static void push_front(const node_ptr & header, const node_ptr & new_node, 
                         std::size_t tree_size, H_Alpha h_alpha, 
                         std::size_t & max_tree_size);
```

Requires: "header" must be the header node of a tree. "new_node" must be, according to the used ordering, no greater than the lowest inserted key.

Effects: Inserts new_node into the tree before "pos".

Complexity: Constant-time.

Throws: Nothing.

Note: If "new_node" is greater than the lowest inserted key tree invariants are broken. This function is slightly faster than using "insert_before".

32.
```
template<typename KeyType, typename KeyNodePtrCompare> 
  static std::pair< node_ptr, bool > 
  insert_unique_check(const const_node_ptr & header, const node_ptr & hint, 
                      const KeyType & key, KeyNodePtrCompare comp, 
                      insert_commit_data & commit_data);
```

Requires: "header" must be the header node of a tree. KeyNodePtrCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. NodePtrCompare compares KeyType with a node_ptr. "hint" is node from the "header"'s tree.

Effects: Checks if there is an equivalent node to "key" in the tree according to "comp" using "hint" as a hint to where it should be inserted and obtains the needed information to realize a constant-time node insertion if there is no equivalent node. If "hint" is the upper_bound the function has constant time complexity (two comparisons in the worst case).

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Returns: If there is an equivalent value returns a pair containing a node_ptr to the already present node and false. If there is not equivalent key can be inserted returns true in the returned pair's boolean and fills "commit_data" that is meant to be used with the "insert_commit" function to achieve a constant-time insertion function.

Complexity: Average complexity is at most logarithmic, but it is amortized constant time if new_node should be inserted immediately before "hint".

Throws: If "comp" throws.

Notes: This function is used to improve performance when constructing a node is expensive and the user does not want to have two equivalent nodes in the tree: if there is an equivalent value the constructed object must be discarded. Many times, the part of the node that is used to impose the order is much cheaper to construct than the node and this function offers the possibility to use that part to check if the insertion will be successful.

If the check is successful, the user can construct the node and use "insert_commit" to insert the node in constant-time. This gives a total logarithmic complexity to the insertion: check(O(log(N)) + commit(O(1)).

"commit_data" remains valid for a subsequent "insert_unique_commit" only if no more objects are inserted or erased from the set.

33.
```
template<typename H_Alpha> 
  static void insert_unique_commit(const node_ptr & header, 
                                   const node_ptr & new_value, 
                                   const insert_commit_data & commit_data, 
                                   std::size_t tree_size, H_Alpha h_alpha, 
                                   std::size_t & max_tree_size);
```

Requires: "header" must be the header node of a tree. "commit_data" must have been obtained from a previous call to "insert_unique_check". No objects should have been inserted or erased from the set between the "insert_unique_check" that filled "commit_data" and the call to "insert_commit".

Effects: Inserts new_node in the set using the information obtained from the "commit_data" that a previous "insert_check" filled.

Complexity: Constant time.

Throws: Nothing.

Notes: This function has only sense if a "insert_unique_check" has been previously executed to fill "commit_data". No value should be inserted or erased between the "insert_check" and "insert_commit" calls.

34.
```
static void rebalance(const node_ptr & header);
```

Requires: header must be the header of a tree.

Effects: Rebalances the tree.

Throws: Nothing.

Complexity: Linear.

35.
```
static node_ptr rebalance_subtree(const node_ptr & old_root);
```

Requires: old_root is a node of a tree.

Effects: Rebalances the subtree rooted at old_root.

Returns: The new root of the subtree.

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Linear.

36.
```
static node_ptr get_header(const node_ptr & n);
```

Requires: "n" must be a node inserted in a tree.

Effects: Returns a pointer to the header node of the tree.

Complexity: Logarithmic.

Throws: Nothing.

Struct insert_commit_data

boost::intrusive::sgtree_algorithms::insert_commit_data

Synopsis

```
// In header: <boost/intrusive/sgtree_algorithms.hpp>



struct insert_commit_data : public insert_commit_data {

  // public data members
  std::size_t depth;
};
```

Description

This type is the information that will be filled by insert_unique_check

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Header <boost/intrusive/slist.hpp>

```
namespace boost {
  namespace intrusive {
    template<typename T, class... Options> class slist;

    template<typename T, class... Options> struct make_slist;
    template<typename T, class... Options>
      bool operator<(const slist< T, Options...> & x,
                     const slist< T, Options...> & y);
    template<typename T, class... Options>
      bool operator==(const slist< T, Options...> & x,
                      const slist< T, Options...> & y);
    template<typename T, class... Options>
      bool operator!=(const slist< T, Options...> & x,
                      const slist< T, Options...> & y);
    template<typename T, class... Options>
      bool operator>(const slist< T, Options...> & x,
                     const slist< T, Options...> & y);
    template<typename T, class... Options>
      bool operator<=(const slist< T, Options...> & x,
                      const slist< T, Options...> & y);
    template<typename T, class... Options>
      bool operator>=(const slist< T, Options...> & x,
                      const slist< T, Options...> & y);
    template<typename T, class... Options>
      void swap(slist< T, Options...> & x, slist< T, Options...> & y);
  }
}
```

Class template slist

boost::intrusive::slist

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/intrusive/slist.hpp>

template<typename T, class... Options>
class slist {
public:
  // types
  typedef Config::value_traits                           value_traits;
  typedef real_value_traits::pointer                     pointer;
  typedef real_value_traits::const_pointer               const_pointer;
  typedef pointer_traits< pointer >::element_type        value_type;
  typedef pointer_traits< pointer >::reference           reference;
  typedef pointer_traits< const_pointer >::reference const_reference;
  typedef pointer_traits< pointer >::difference_type difference_type;
  typedef Config::size_type                              size_type;
  typedef slist_iterator< slist, false >                 iterator;
  typedef slist_iterator< slist, true >                  const_iterator;
  typedef real_value_traits::node_traits                 node_traits;
  typedef node_traits::node                              node;
  typedef node_traits::node_ptr                          node_ptr;
  typedef node_traits::const_node_ptr                    const_node_ptr;
  typedef unspecified                                    node_algorithms;

  // construct/copy/destruct
  explicit slist(const value_traits & = value_traits());
  template<typename Iterator>
    slist(Iterator, Iterator, const value_traits & = value_traits());
  slist(BOOST_RV_REF(slist));
  slist& operator=(BOOST_RV_REF(slist));
  ~slist();

  // public member functions
  const real_value_traits & get_real_value_traits() const;
  real_value_traits & get_real_value_traits();
  void clear();
  template<typename Disposer> void clear_and_dispose(Disposer);
  void push_front(reference);
  void push_back(reference);
  void pop_front();
  template<typename Disposer> void pop_front_and_dispose(Disposer);
  reference front();
  const_reference front() const;
  reference back();
  const_reference back() const;
  iterator begin();
  const_iterator begin() const;
  const_iterator cbegin() const;
  iterator end();
  const_iterator end() const;
  const_iterator cend() const;
  iterator before_begin();
  const_iterator before_begin() const;
  const_iterator cbefore_begin() const;
  iterator last();
  const_iterator last() const;
  const_iterator clast() const;
  size_type size() const;
  bool empty() const;
  void swap(slist &);
  void shift_backwards(size_type = 1);
  void shift_forward(size_type = 1);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
template<typename Cloner, typename Disposer> 
  void clone_from(const slist &, Cloner, Disposer);
iterator insert_after(const_iterator, reference);
template<typename Iterator> 
  void insert_after(const_iterator, Iterator, Iterator);
iterator insert(const_iterator, reference);
template<typename Iterator> void insert(const_iterator, Iterator, Iterator);
iterator erase_after(const_iterator);
iterator erase_after(const_iterator, const_iterator);
iterator erase_after(const_iterator, const_iterator, size_type);
iterator erase(const_iterator);
iterator erase(const_iterator, const_iterator);
iterator erase(const_iterator, const_iterator, size_type);
template<typename Disposer> 
  iterator erase_after_and_dispose(const_iterator, Disposer);
template<typename Disposer> 
  iterator erase_after_and_dispose(const_iterator, const_iterator, Disposer);
template<typename Disposer> 
  iterator erase_and_dispose(const_iterator, Disposer);
template<typename Disposer> 
  iterator erase_and_dispose(const_iterator, const_iterator, Disposer);
template<typename Iterator> void assign(Iterator, Iterator);
template<typename Iterator, typename Disposer> 
  void dispose_and_assign(Disposer, Iterator, Iterator);
void splice_after(const_iterator, slist &, const_iterator * = 0);
void splice_after(const_iterator, slist &, const_iterator);
void splice_after(const_iterator, slist &, const_iterator, const_iterator);
void splice_after(const_iterator, slist &, const_iterator, const_iterator, 
                  size_type);
void splice(const_iterator, slist &, const_iterator * = 0);
void splice(const_iterator, slist &, const_iterator);
void splice(const_iterator, slist &, const_iterator, const_iterator);
void splice(const_iterator, slist &, const_iterator, const_iterator, 
            size_type);
template<typename Predicate> void sort(Predicate);
void sort();
template<typename Predicate> 
  void merge(slist &, Predicate, const_iterator * = 0);
void merge(slist &);
void reverse();
void remove(const_reference);
template<typename Disposer> 
  void remove_and_dispose(const_reference, Disposer);
template<typename Pred> void remove_if(Pred);
template<typename Pred, typename Disposer> 
  void remove_and_dispose_if(Pred, Disposer);
void unique();
template<typename BinaryPredicate> void unique(BinaryPredicate);
template<typename Disposer> void unique_and_dispose(Disposer);
template<typename BinaryPredicate, typename Disposer> 
  void unique_and_dispose(BinaryPredicate, Disposer);
iterator iterator_to(reference);
const_iterator iterator_to(const_reference) const;
iterator previous(iterator);
const_iterator previous(const_iterator) const;
iterator previous(const_iterator, iterator);
const_iterator previous(const_iterator, const_iterator) const;

// public static functions
static slist & container_from_end_iterator(iterator);
static const slist & container_from_end_iterator(const_iterator);
static iterator s_iterator_to(reference);
static const_iterator s_iterator_to(const_reference);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
  // private member functions
  void priv_splice_after(const node_ptr &, slist &, const node_ptr &,
                         const node_ptr &);
  void priv_incorporate_after(const node_ptr &, const node_ptr &,
                              const node_ptr &);
  void priv_reverse(unspecified);
  void priv_reverse(unspecified);
  void priv_shift_backwards(size_type, unspecified);
  void priv_shift_backwards(size_type, unspecified);
  void priv_shift_forward(size_type, unspecified);
  void priv_shift_forward(size_type, unspecified);

  // private static functions
  static void priv_swap_cache_last(slist *, slist *);
  static void priv_swap_lists(const node_ptr &, const node_ptr &, unspecified);
  static void priv_swap_lists(const node_ptr &, const node_ptr &, unspecified);
  static slist & priv_container_from_end_iterator(const const_iterator &);

  // public data members
  static const bool constant_time_size;
  static const bool stateful_value_traits;
  static const bool linear;
  static const bool cache_last;
};
```

Description

The class template slist is an intrusive container, that encapsulates a singly-linked list. You can use such a list to squeeze the last bit of performance from your application. Unfortunately, the little gains come with some huge drawbacks. A lot of member functions can't be implemented as efficiently as for standard containers. To overcome this limitation some other member functions with rather unusual semantics have to be introduced.

The template parameter `T` is the type to be managed by the container. The user can specify additional options and if no options are provided default options are used.

The container supports the following options: `base_hook<>/member_hook<>/value_traits<>`, `constant_time_size<>`, `size_type<>`, `linear<>` and `cache_last<>`.

The iterators of slist are forward iterators. slist provides a static function called "previous" to compute the previous iterator of a given iterator. This function has linear complexity. To improve the usability esp. with the '*_after' functions, ++end() == begin() and previous(begin()) == end() are defined. An new special function "before_begin()" is defined, which returns an iterator that points one less the beginning of the list: ++before_begin() == begin()

`slist` public construct/copy/destruct

1.
```
explicit slist(const value_traits & v_traits = value_traits());
```

   Effects: constructs an empty list.

   Complexity: Constant

   Throws: If value_traits::node_traits::node constructor throws (this does not happen with predefined Boost.Intrusive hooks).

2.
```
template<typename Iterator>
  slist(Iterator b, Iterator e,
        const value_traits & v_traits = value_traits());
```

   Requires: Dereferencing iterator must yield an lvalue of type value_type.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Constructs a list equal to [b ,e).

Complexity: Linear in std::distance(b, e). No copy constructors are called.

Throws: If value_traits::node_traits::node constructor throws (this does not happen with predefined Boost.Intrusive hooks).

3.
```
slist(BOOST_RV_REF(slist) x);
```

Effects: to-do

4.
```
slist& operator=(BOOST_RV_REF(slist) x);
```

Effects: to-do

5.
```
~slist();
```

Effects: If it's a safe-mode or auto-unlink value, the destructor does nothing (ie. no code is generated). Otherwise it detaches all elements from this. In this case the objects in the list are not deleted (i.e. no destructors are called), but the hooks according to the `value_traits` template parameter are set to their default value.

Complexity: Linear to the number of elements in the list, if it's a safe-mode or auto-unlink value. Otherwise constant.

`slist` public member functions

1.
```
const real_value_traits & get_real_value_traits() const;
```

2.
```
real_value_traits & get_real_value_traits();
```

3.
```
void clear();
```

Effects: Erases all the elements of the container.

Throws: Nothing.

Complexity: Linear to the number of elements of the list. if it's a safe-mode or auto-unlink value_type. Constant time otherwise.

Note: Invalidates the iterators (but not the references) to the erased elements.

4.
```
template<typename Disposer> void clear_and_dispose(Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases all the elements of the container Disposer::operator()(pointer) is called for the removed elements.

Throws: Nothing.

Complexity: Linear to the number of elements of the list.

Note: Invalidates the iterators to the erased elements.

5.
```
void push_front(reference value);
```

Requires: value must be an lvalue.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Inserts the value in the front of the list. No copy constructors are called.

Throws: Nothing.

Complexity: Constant.

Note: Does not affect the validity of iterators and references.

6.
```
void push_back(reference value);
```

Requires: value must be an lvalue.

Effects: Inserts the value in the back of the list. No copy constructors are called.

Throws: Nothing.

Complexity: Constant.

Note: Does not affect the validity of iterators and references. This function is only available is cache_last<> is true.

7.
```
void pop_front();
```

Effects: Erases the first element of the list. No destructors are called.

Throws: Nothing.

Complexity: Constant.

Note: Invalidates the iterators (but not the references) to the erased element.

8.
```
template<typename Disposer> void pop_front_and_dispose(Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases the first element of the list. Disposer::operator()(pointer) is called for the removed element.

Throws: Nothing.

Complexity: Constant.

Note: Invalidates the iterators to the erased element.

9.
```
reference front();
```

Effects: Returns a reference to the first element of the list.

Throws: Nothing.

Complexity: Constant.

10.
```
const_reference front() const;
```

Effects: Returns a const_reference to the first element of the list.

Throws: Nothing.

Complexity: Constant.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

11.
```
reference back();
```

Effects: Returns a reference to the last element of the list.

Throws: Nothing.

Complexity: Constant.

Note: Does not affect the validity of iterators and references. This function is only available is cache_last<> is true.

12.
```
const_reference back() const;
```

Effects: Returns a const_reference to the last element of the list.

Throws: Nothing.

Complexity: Constant.

Note: Does not affect the validity of iterators and references. This function is only available is cache_last<> is true.

13.
```
iterator begin();
```

Effects: Returns an iterator to the first element contained in the list.

Throws: Nothing.

Complexity: Constant.

14.
```
const_iterator begin() const;
```

Effects: Returns a const_iterator to the first element contained in the list.

Throws: Nothing.

Complexity: Constant.

15.
```
const_iterator cbegin() const;
```

Effects: Returns a const_iterator to the first element contained in the list.

Throws: Nothing.

Complexity: Constant.

16.
```
iterator end();
```

Effects: Returns an iterator to the end of the list.

Throws: Nothing.

Complexity: Constant.

17.
```
const_iterator end() const;
```

Effects: Returns a const_iterator to the end of the list.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: Nothing.

Complexity: Constant.

18.
```
const_iterator cend() const;
```

Effects: Returns a const_iterator to the end of the list.

Throws: Nothing.

Complexity: Constant.

19.
```
iterator before_begin();
```

Effects: Returns an iterator that points to a position before the first element. Equivalent to "end()"

Throws: Nothing.

Complexity: Constant.

20.
```
const_iterator before_begin() const;
```

Effects: Returns an iterator that points to a position before the first element. Equivalent to "end()"

Throws: Nothing.

Complexity: Constant.

21.
```
const_iterator cbefore_begin() const;
```

Effects: Returns an iterator that points to a position before the first element. Equivalent to "end()"

Throws: Nothing.

Complexity: Constant.

22.
```
iterator last();
```

Effects: Returns an iterator to the last element contained in the list.

Throws: Nothing.

Complexity: Constant.

Note: This function is present only if cached_last<> option is true.

23.
```
const_iterator last() const;
```

Effects: Returns a const_iterator to the last element contained in the list.

Throws: Nothing.

Complexity: Constant.

Note: This function is present only if cached_last<> option is true.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

24.
```
const_iterator clast() const;
```

Effects: Returns a const_iterator to the last element contained in the list.

Throws: Nothing.

Complexity: Constant.

Note: This function is present only if cached_last<> option is true.

25.
```
size_type size() const;
```

Effects: Returns the number of the elements contained in the list.

Throws: Nothing.

Complexity: Linear to the number of elements contained in the list. if `constant_time_size` is false. Constant time otherwise.

Note: Does not affect the validity of iterators and references.

26.
```
bool empty() const;
```

Effects: Returns true if the list contains no elements.

Throws: Nothing.

Complexity: Constant.

Note: Does not affect the validity of iterators and references.

27.
```
void swap(slist & other);
```

Effects: Swaps the elements of x and *this.

Throws: Nothing.

Complexity: Linear to the number of elements of both lists. Constant-time if linear<> and/or cache_last<> options are used.

Note: Does not affect the validity of iterators and references.

28.
```
void shift_backwards(size_type n = 1);
```

Effects: Moves backwards all the elements, so that the first element becomes the second, the second becomes the third... the last element becomes the first one.

Throws: Nothing.

Complexity: Linear to the number of elements plus the number shifts.

Note: Iterators Does not affect the validity of iterators and references.

29.
```
void shift_forward(size_type n = 1);
```

Effects: Moves forward all the elements, so that the second element becomes the first, the third becomes the second... the first element becomes the last one.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: Nothing.

Complexity: Linear to the number of elements plus the number shifts.

Note: Does not affect the validity of iterators and references.

30.
```
template<typename Cloner, typename Disposer> 
  void clone_from(const slist & src, Cloner cloner, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw. Cloner should yield to nodes equivalent to the original nodes.

Effects: Erases all the elements from *this calling Disposer::operator()(pointer), clones all the elements from src calling Cloner::operator()(const_reference) and inserts them on *this.

If cloner throws, all cloned elements are unlinked and disposed calling Disposer::operator()(pointer).

Complexity: Linear to erased plus inserted elements.

Throws: If cloner throws.

31.
```
iterator insert_after(const_iterator prev_p, reference value);
```

Requires: value must be an lvalue and prev_p must point to an element contained by the list or to end().

Effects: Inserts the value after the position pointed by prev_p. No copy constructor is called.

Returns: An iterator to the inserted element.

Throws: Nothing.

Complexity: Constant.

Note: Does not affect the validity of iterators and references.

32.
```
template<typename Iterator> 
  void insert_after(const_iterator prev_p, Iterator f, Iterator l);
```

Requires: Dereferencing iterator must yield an lvalue of type value_type and prev_p must point to an element contained by the list or to the end node.

Effects: Inserts the [f, l) after the position prev_p.

Throws: Nothing.

Complexity: Linear to the number of elements inserted.

Note: Does not affect the validity of iterators and references.

33.
```
iterator insert(const_iterator p, reference value);
```

Requires: value must be an lvalue and p must point to an element contained by the list or to end().

Effects: Inserts the value before the position pointed by p. No copy constructor is called.

Throws: Nothing.

Complexity: Linear to the number of elements before p. Constant-time if cache_last<> is true and p == end().

Note: Does not affect the validity of iterators and references.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

34.
```
template<typename Iterator>
  void insert(const_iterator p, Iterator b, Iterator e);
```

Requires: Dereferencing iterator must yield an lvalue of type value_type and p must point to an element contained by the list or to the end node.

Effects: Inserts the pointed by b and e before the position p. No copy constructors are called.

Throws: Nothing.

Complexity: Linear to the number of elements inserted plus linear to the elements before b. Linear to the number of elements to insert if cache_last<> option is true and p == end().

Note: Does not affect the validity of iterators and references.

35.
```
iterator erase_after(const_iterator prev);
```

Effects: Erases the element after the element pointed by prev of the list. No destructors are called.

Returns: the first element remaining beyond the removed elements, or end() if no such element exists.

Throws: Nothing.

Complexity: Constant.

Note: Invalidates the iterators (but not the references) to the erased element.

36.
```
iterator erase_after(const_iterator before_f, const_iterator l);
```

Effects: Erases the range (before_f, l) from the list. No destructors are called.

Returns: the first element remaining beyond the removed elements, or end() if no such element exists.

Throws: Nothing.

Complexity: Linear to the number of erased elements if it's a safe-mode , auto-unlink value or constant-time size is activated. Constant time otherwise.

Note: Invalidates the iterators (but not the references) to the erased element.

37.
```
iterator erase_after(const_iterator before_f, const_iterator l, size_type n);
```

Effects: Erases the range (before_f, l) from the list. n must be std::distance(before_f, l) - 1. No destructors are called.

Returns: the first element remaining beyond the removed elements, or end() if no such element exists.

Throws: Nothing.

Complexity: constant-time if `link_mode` is normal_link. Linear to the elements (l - before_f) otherwise.

Note: Invalidates the iterators (but not the references) to the erased element.

38.
```
iterator erase(const_iterator i);
```

Effects: Erases the element pointed by i of the list. No destructors are called.

Returns: the first element remaining beyond the removed element, or end() if no such element exists.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: Nothing.

Complexity: Linear to the elements before i.

Note: Invalidates the iterators (but not the references) to the erased element.

39.
```
iterator erase(const_iterator f, const_iterator l);
```

Requires: f and l must be valid iterator to elements in *this.

Effects: Erases the range pointed by b and e. No destructors are called.

Returns: the first element remaining beyond the removed elements, or end() if no such element exists.

Throws: Nothing.

Complexity: Linear to the elements before l.

Note: Invalidates the iterators (but not the references) to the erased elements.

40.
```
iterator erase(const_iterator f, const_iterator l, size_type n);
```

Effects: Erases the range [f, l) from the list. n must be std::distance(f, l). No destructors are called.

Returns: the first element remaining beyond the removed elements, or end() if no such element exists.

Throws: Nothing.

Complexity: linear to the elements before f if `link_mode` is normal_link and `constant_time_size` is activated. Linear to the elements before l otherwise.

Note: Invalidates the iterators (but not the references) to the erased element.

41.
```
template<typename Disposer>
  iterator erase_after_and_dispose(const_iterator prev, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases the element after the element pointed by prev of the list. Disposer::operator()(pointer) is called for the removed element.

Returns: the first element remaining beyond the removed elements, or end() if no such element exists.

Throws: Nothing.

Complexity: Constant.

Note: Invalidates the iterators to the erased element.

42.
```
template<typename Disposer>
  iterator erase_after_and_dispose(const_iterator before_f, const_iterator l,
                                   Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases the range (before_f, l) from the list. Disposer::operator()(pointer) is called for the removed elements.

Returns: the first element remaining beyond the removed elements, or end() if no such element exists.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: Nothing.

Complexity: Lineal to the elements (l - before_f + 1).

Note: Invalidates the iterators to the erased element.

43.
```
template<typename Disposer> 
  iterator erase_and_dispose(const_iterator i, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases the element pointed by i of the list. No destructors are called. Disposer::operator()(pointer) is called for the removed element.

Returns: the first element remaining beyond the removed element, or end() if no such element exists.

Throws: Nothing.

Complexity: Linear to the elements before i.

Note: Invalidates the iterators (but not the references) to the erased element.

44.
```
template<typename Disposer> 
  iterator erase_and_dispose(const_iterator f, const_iterator l, 
                             Disposer disposer);
```

Requires: f and l must be valid iterator to elements in *this. Disposer::operator()(pointer) shouldn't throw.

Effects: Erases the range pointed by b and e. No destructors are called. Disposer::operator()(pointer) is called for the removed elements.

Returns: the first element remaining beyond the removed elements, or end() if no such element exists.

Throws: Nothing.

Complexity: Linear to the number of erased elements plus linear to the elements before f.

Note: Invalidates the iterators (but not the references) to the erased elements.

45.
```
template<typename Iterator> void assign(Iterator b, Iterator e);
```

Requires: Dereferencing iterator must yield an lvalue of type value_type.

Effects: Clears the list and inserts the range pointed by b and e. No destructors or copy constructors are called.

Throws: Nothing.

Complexity: Linear to the number of elements inserted plus linear to the elements contained in the list if it's a safe-mode or auto-unlink value. Linear to the number of elements inserted in the list otherwise.

Note: Invalidates the iterators (but not the references) to the erased elements.

46.
```
template<typename Iterator, typename Disposer> 
  void dispose_and_assign(Disposer disposer, Iterator b, Iterator e);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Requires: Dereferencing iterator must yield an lvalue of type value_type.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Clears the list and inserts the range pointed by b and e. No destructors or copy constructors are called. Disposer::operator()(pointer) is called for the removed elements.

Throws: Nothing.

Complexity: Linear to the number of elements inserted plus linear to the elements contained in the list.

Note: Invalidates the iterators (but not the references) to the erased elements.

47.
```
void splice_after(const_iterator prev, slist & x, const_iterator * l = 0);
```

Requires: prev must point to an element contained by this list or to the before_begin() element

Effects: Transfers all the elements of list x to this list, after the the element pointed by prev. No destructors or copy constructors are called.

Returns: Nothing.

Throws: Nothing.

Complexity: In general, linear to the elements contained in x. Constant-time if cache_last<> option is true and also constant-time if linear<> option is true "this" is empty and "l" is not used.

Note: Iterators of values obtained from list x now point to elements of this list. Iterators of this list and all the references are not invalidated.

Additional note: If the optional parameter "l" is provided, it will be assigned to the last spliced element or prev if x is empty. This iterator can be used as new "prev" iterator for a new splice_after call. that will splice new values after the previously spliced values.

48.
```
void splice_after(const_iterator prev_pos, slist & x, const_iterator prev_ele);
```

Requires: prev must point to an element contained by this list or to the before_begin() element. prev_ele must point to an element contained in list x or must be x.before_begin().

Effects: Transfers the element after prev_ele, from list x to this list, after the element pointed by prev. No destructors or copy constructors are called.

Throws: Nothing.

Complexity: Constant.

Note: Iterators of values obtained from list x now point to elements of this list. Iterators of this list and all the references are not invalidated.

49.
```
void splice_after(const_iterator prev_pos, slist & x, const_iterator before_f,
                  const_iterator before_l);
```

Requires: prev_pos must be a dereferenceable iterator in *this or be before_begin(), and before_f and before_l belong to x and ++before_f != x.end() && before_l != x.end().

Effects: Transfers the range (before_f, before_l] from list x to this list, after the element pointed by prev_pos. No destructors or copy constructors are called.

Throws: Nothing.

Complexity: Linear to the number of elements transferred if `constant_time_size` is true. Constant-time otherwise.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Note: Iterators of values obtained from list x now point to elements of this list. Iterators of this list and all the references are not invalidated.

50.
```
void splice_after(const_iterator prev_pos, slist & x, const_iterator before_f,
                  const_iterator before_l, size_type n);
```

Requires: prev_pos must be a dereferenceable iterator in *this or be before_begin(), and before_f and before_l belong to x and ++before_f != x.end() && before_l != x.end() and n == std::distance(before_f, before_l).

Effects: Transfers the range (before_f, before_l] from list x to this list, after the element pointed by p. No destructors or copy constructors are called.

Throws: Nothing.

Complexity: Constant time.

Note: Iterators of values obtained from list x now point to elements of this list. Iterators of this list and all the references are not invalidated.

51.
```
void splice(const_iterator it, slist & x, const_iterator * l = 0);
```

Requires: it is an iterator to an element in *this.

Effects: Transfers all the elements of list x to this list, before the the element pointed by it. No destructors or copy constructors are called.

Returns: Nothing.

Throws: Nothing.

Complexity: Linear to the elements contained in x plus linear to the elements before it. Linear to the elements before it if cache_last<> option is true. Constant-time if cache_last<> option is true and it == end().

Note: Iterators of values obtained from list x now point to elements of this list. Iterators of this list and all the references are not invalidated.

Additional note: If the optional parameter "l" is provided, it will be assigned to the last spliced element or prev if x is empty. This iterator can be used as new "prev" iterator for a new splice_after call. that will splice new values after the previously spliced values.

52.
```
void splice(const_iterator pos, slist & x, const_iterator elem);
```

Requires: it p must be a valid iterator of *this. elem must point to an element contained in list x.

Effects: Transfers the element elem, from list x to this list, before the element pointed by pos. No destructors or copy constructors are called.

Throws: Nothing.

Complexity: Linear to the elements before pos and before elem. Linear to the elements before elem if cache_last<> option is true and pos == end().

Note: Iterators of values obtained from list x now point to elements of this list. Iterators of this list and all the references are not invalidated.

53.
```
void splice(const_iterator pos, slist & x, const_iterator f, const_iterator l);
```

Requires: pos must be a dereferenceable iterator in *this and f and f belong to x and f and f a valid range on x.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Transfers the range [f, l) from list x to this list, before the element pointed by pos. No destructors or copy constructors are called.

Throws: Nothing.

Complexity: Linear to the sum of elements before pos, f, and l plus linear to the number of elements transferred if `constant_time_size` is true. Linear to the sum of elements before f, and l plus linear to the number of elements transferred if `constant_time_size` is true if cache_last<> is true and pos == end()

Note: Iterators of values obtained from list x now point to elements of this list. Iterators of this list and all the references are not invalidated.

54.
```
void splice(const_iterator pos, slist & x, const_iterator f, const_iterator l,
            size_type n);
```

Requires: pos must be a dereferenceable iterator in *this and f and l belong to x and f and l a valid range on x. n == std::distance(f, l).

Effects: Transfers the range [f, l) from list x to this list, before the element pointed by pos. No destructors or copy constructors are called.

Throws: Nothing.

Complexity: Linear to the sum of elements before pos, f, and l. Linear to the sum of elements before f and l if cache_last<> is true and pos == end().

Note: Iterators of values obtained from list x now point to elements of this list. Iterators of this list and all the references are not invalidated.

55.
```
template<typename Predicate> void sort(Predicate p);
```

Effects: This function sorts the list *this according to std::less<value_type>. The sort is stable, that is, the relative order of equivalent elements is preserved.

Throws: If value_traits::node_traits::node constructor throws (this does not happen with predefined Boost.Intrusive hooks) or the predicate throws. Basic guarantee.

Complexity: The number of comparisons is approximately N log N, where N is the list's size.

Note: Iterators and references are not invalidated

56.
```
void sort();
```

Requires: p must be a comparison function that induces a strict weak ordering and both *this and x must be sorted according to that ordering The lists x and *this must be distinct.

Effects: This function removes all of x's elements and inserts them in order into *this. The merge is stable; that is, if an element from *this is equivalent to one from x, then the element from *this will precede the one from x.

Throws: If value_traits::node_traits::node constructor throws (this does not happen with predefined Boost.Intrusive hooks) or std::less<value_type> throws. Basic guarantee.

Complexity: This function is linear time: it performs at most size() + x.size() - 1 comparisons.

Note: Iterators and references are not invalidated.

57.
```
template<typename Predicate>
  void merge(slist & x, Predicate p, const_iterator * l = 0);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Requires: p must be a comparison function that induces a strict weak ordering and both *this and x must be sorted according to that ordering The lists x and *this must be distinct.

Effects: This function removes all of x's elements and inserts them in order into *this. The merge is stable; that is, if an element from *this is equivalent to one from x, then the element from *this will precede the one from x.

Returns: Nothing.

Throws: If the predicate throws. Basic guarantee.

Complexity: This function is linear time: it performs at most size() + x.size() - 1 comparisons.

Note: Iterators and references are not invalidated.

Additional note: If optional "l" argument is passed, it is assigned to an iterator to the last transferred value or end() is x is empty.

58.
```
void merge(slist & x);
```

Effects: This function removes all of x's elements and inserts them in order into *this according to std::less<value_type>. The merge is stable; that is, if an element from *this is equivalent to one from x, then the element from *this will precede the one from x.

Throws: if std::less<value_type> throws. Basic guarantee.

Complexity: This function is linear time: it performs at most size() + x.size() - 1 comparisons.

Note: Iterators and references are not invalidated

59.
```
void reverse();
```

Effects: Reverses the order of elements in the list.

Throws: Nothing.

Complexity: This function is linear to the contained elements.

Note: Iterators and references are not invalidated

60.
```
void remove(const_reference value);
```

Effects: Removes all the elements that compare equal to value. No destructors are called.

Throws: If std::equal_to<value_type> throws. Basic guarantee.

Complexity: Linear time. It performs exactly size() comparisons for equality.

Note: The relative order of elements that are not removed is unchanged, and iterators to elements that are not removed remain valid. This function is linear time: it performs exactly size() comparisons for equality.

61.
```
template<typename Disposer>
  void remove_and_dispose(const_reference value, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Removes all the elements that compare equal to value. Disposer::operator()(pointer) is called for every removed element.

Throws: If std::equal_to<value_type> throws. Basic guarantee.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Linear time. It performs exactly size() comparisons for equality.

Note: The relative order of elements that are not removed is unchanged, and iterators to elements that are not removed remain valid.

62.
```
template<typename Pred> void remove_if(Pred pred);
```

Effects: Removes all the elements for which a specified predicate is satisfied. No destructors are called.

Throws: If pred throws. Basic guarantee.

Complexity: Linear time. It performs exactly size() calls to the predicate.

Note: The relative order of elements that are not removed is unchanged, and iterators to elements that are not removed remain valid.

63.
```
template<typename Pred, typename Disposer>
  void remove_and_dispose_if(Pred pred, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Removes all the elements for which a specified predicate is satisfied. Disposer::operator()(pointer) is called for every removed element.

Throws: If pred throws. Basic guarantee.

Complexity: Linear time. It performs exactly size() comparisons for equality.

Note: The relative order of elements that are not removed is unchanged, and iterators to elements that are not removed remain valid.

64.
```
void unique();
```

Effects: Removes adjacent duplicate elements or adjacent elements that are equal from the list. No destructors are called.

Throws: If std::equal_to<value_type> throws. Basic guarantee.

Complexity: Linear time (size()-1) comparisons calls to pred()).

Note: The relative order of elements that are not removed is unchanged, and iterators to elements that are not removed remain valid.

65.
```
template<typename BinaryPredicate> void unique(BinaryPredicate pred);
```

Effects: Removes adjacent duplicate elements or adjacent elements that satisfy some binary predicate from the list. No destructors are called.

Throws: If the predicate throws. Basic guarantee.

Complexity: Linear time (size()-1) comparisons equality comparisons.

Note: The relative order of elements that are not removed is unchanged, and iterators to elements that are not removed remain valid.

66.
```
template<typename Disposer> void unique_and_dispose(Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Removes adjacent duplicate elements or adjacent elements that satisfy some binary predicate from the list. Disposer::operator()(pointer) is called for every removed element.

Throws: If std::equal_to<value_type> throws. Basic guarantee.

Complexity: Linear time (size()-1) comparisons equality comparisons.

Note: The relative order of elements that are not removed is unchanged, and iterators to elements that are not removed remain valid.

67.
```
template<typename BinaryPredicate, typename Disposer>
  void unique_and_dispose(BinaryPredicate pred, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Removes adjacent duplicate elements or adjacent elements that satisfy some binary predicate from the list. Disposer::operator()(pointer) is called for every removed element.

Throws: If the predicate throws. Basic guarantee.

Complexity: Linear time (size()-1) comparisons equality comparisons.

Note: The relative order of elements that are not removed is unchanged, and iterators to elements that are not removed remain valid.

68.
```
iterator iterator_to(reference value);
```

Requires: value must be a reference to a value inserted in a list.

Effects: This function returns a const_iterator pointing to the element

Throws: Nothing.

Complexity: Constant time.

Note: Iterators and references are not invalidated.

69.
```
const_iterator iterator_to(const_reference value) const;
```

Requires: value must be a const reference to a value inserted in a list.

Effects: This function returns an iterator pointing to the element.

Throws: Nothing.

Complexity: Constant time.

Note: Iterators and references are not invalidated.

70.
```
iterator previous(iterator i);
```

Returns: The iterator to the element before i in the list. Returns the end-iterator, if either i is the begin-iterator or the list is empty.

Throws: Nothing.

Complexity: Linear to the number of elements before i. Constant if cache_last<> is true and i == end().

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

71.
```
const_iterator previous(const_iterator i) const;
```

Returns: The const_iterator to the element before i in the list. Returns the end-const_iterator, if either i is the begin-const_iterator or the list is empty.

Throws: Nothing.

Complexity: Linear to the number of elements before i. Constant if cache_last<> is true and i == end().

72.
```
iterator previous(const_iterator prev_from, iterator i);
```

Returns: The iterator to the element before i in the list, starting the search on element after prev_from. Returns the end-iterator, if either i is the begin-iterator or the list is empty.

Throws: Nothing.

Complexity: Linear to the number of elements before i. Constant if cache_last<> is true and i == end().

73.
```
const_iterator previous(const_iterator prev_from, const_iterator i) const;
```

Returns: The const_iterator to the element before i in the list, starting the search on element after prev_from. Returns the end-const_iterator, if either i is the begin-const_iterator or the list is empty.

Throws: Nothing.

Complexity: Linear to the number of elements before i. Constant if cache_last<> is true and i == end().

`slist` public static functions

1.
```
static slist & container_from_end_iterator(iterator end_iterator);
```

Precondition: end_iterator must be a valid end iterator of slist.

Effects: Returns a const reference to the slist associated to the end iterator

Throws: Nothing.

Complexity: Constant.

2.
```
static const slist & container_from_end_iterator(const_iterator end_iterator);
```

Precondition: end_iterator must be a valid end const_iterator of slist.

Effects: Returns a const reference to the slist associated to the end iterator

Throws: Nothing.

Complexity: Constant.

3.
```
static iterator s_iterator_to(reference value);
```

Requires: value must be a reference to a value inserted in a list.

Effects: This function returns a const_iterator pointing to the element

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Constant time.

Note: Iterators and references are not invalidated. This static function is available only if the *value traits* is stateless.

4.
```
static const_iterator s_iterator_to(const_reference value);
```

Requires: value must be a const reference to a value inserted in a list.

Effects: This function returns an iterator pointing to the element.

Throws: Nothing.

Complexity: Constant time.

Note: Iterators and references are not invalidated. This static function is available only if the *value traits* is stateless.

`slist` private member functions

1.
```
void priv_splice_after(const node_ptr & prev_pos_n, slist & x, 
                       const node_ptr & before_f_n, 
                       const node_ptr & before_l_n);
```

2.
```
void priv_incorporate_after(const node_ptr & prev_pos_n, 
                            const node_ptr & first_n, 
                            const node_ptr & before_l_n);
```

3.
```
void priv_reverse(unspecified);
```

4.
```
void priv_reverse(unspecified);
```

5.
```
void priv_shift_backwards(size_type n, unspecified);
```

6.
```
void priv_shift_backwards(size_type n, unspecified);
```

7.
```
void priv_shift_forward(size_type n, unspecified);
```

8.
```
void priv_shift_forward(size_type n, unspecified);
```

`slist` private static functions

1.
```
static void priv_swap_cache_last(slist * this_impl, slist * other_impl);
```

2.
```
static void priv_swap_lists(const node_ptr & this_node, 
                            const node_ptr & other_node, unspecified);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

3.
```
static void priv_swap_lists(const node_ptr & this_node,
                            const node_ptr & other_node, unspecified);
```

4.
```
static slist &
priv_container_from_end_iterator(const const_iterator & end_iterator);
```

Struct template make_slist

boost::intrusive::make_slist

Synopsis

```
// In header: <boost/intrusive/slist.hpp>

template<typename T, class... Options>
struct make_slist {
  // types
  typedef implementation_defined type;
};
```

Description

Helper metafunction to define a `slist` that yields to the same type when the same options (either explicitly or implicitly) are used.

Header <boost/intrusive/slist_hook.hpp>

```
namespace boost {
  namespace intrusive {
    template<class... Options> struct make_slist_base_hook;

    template<class... Options> class slist_base_hook;

    template<class... Options> struct make_slist_member_hook;

    template<class... Options> class slist_member_hook;
  }
}
```

Struct template make_slist_base_hook

boost::intrusive::make_slist_base_hook

Synopsis

```
// In header: <boost/intrusive/slist_hook.hpp>

template<class... Options>
struct make_slist_base_hook {
  // types
  typedef implementation_defined type;
};
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Description

Helper metafunction to define a `slist_base_hook` that yields to the same type when the same options (either explicitly or implicitly) are used.

Class template slist_base_hook

boost::intrusive::slist_base_hook

Synopsis

```
// In header: <boost/intrusive/slist_hook.hpp>

template<class... Options>
class slist_base_hook : public make_slist_base_hook::type< O1, O2, O3 > {
public:
  // construct/copy/destruct
  slist_base_hook();
  slist_base_hook(const slist_base_hook &);
  slist_base_hook& operator=(const slist_base_hook &);
  ~slist_base_hook();

  // public member functions
  void swap_nodes(slist_base_hook &);
  bool is_linked() const;
  void unlink();
};
```

Description

Derive a class from slist_base_hook in order to store objects in in an list. slist_base_hook holds the data necessary to maintain the list and provides an appropriate value_traits class for list.

The hook admits the following options: `tag<>`, `void_pointer<>` and `link_mode<>`.

`tag<>` defines a tag to identify the node. The same tag value can be used in different classes, but if a class is derived from more than one `list_base_hook`, then each `list_base_hook` needs its unique tag.

`link_mode<>` will specify the linking mode of the hook (`normal_link`, `auto_unlink` or `safe_link`).

`void_pointer<>` is the pointer type that will be used internally in the hook and the the container configured to use this hook.

`slist_base_hook` public construct/copy/destruct

1.
   ```
   slist_base_hook();
   ```

   Effects: If `link_mode` is `auto_unlink` or `safe_link` initializes the node to an unlinked state.

   Throws: Nothing.

2.
   ```
   slist_base_hook(const slist_base_hook &);
   ```

   Effects: If `link_mode` is `auto_unlink` or `safe_link` initializes the node to an unlinked state. The argument is ignored.

   Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Rationale: Providing a copy-constructor makes classes using the hook STL-compliant without forcing the user to do some additional work. swap can be used to emulate move-semantics.

3.
```
slist_base_hook& operator=(const slist_base_hook &);
```

Effects: Empty function. The argument is ignored.

Throws: Nothing.

Rationale: Providing an assignment operator makes classes using the hook STL-compliant without forcing the user to do some additional work. swap can be used to emulate move-semantics.

4.
```
~slist_base_hook();
```

Effects: If link_mode is normal_link, the destructor does nothing (ie. no code is generated). If link_mode is safe_link and the object is stored in an slist an assertion is raised. If link_mode is auto_unlink and is_linked() is true, the node is unlinked.

Throws: Nothing.

slist_base_hook public member functions

1.
```
void swap_nodes(slist_base_hook & other);
```

Effects: Swapping two nodes swaps the position of the elements related to those nodes in one or two containers. That is, if the node this is part of the element e1, the node x is part of the element e2 and both elements are included in the containers s1 and s2, then after the swap-operation e1 is in s2 at the position of e2 and e2 is in s1 at the position of e1. If one element is not in a container, then after the swap-operation the other element is not in a container. Iterators to e1 and e2 related to those nodes are invalidated.

Complexity: Constant

Throws: Nothing.

2.
```
bool is_linked() const;
```

Precondition: link_mode must be safe_link or auto_unlink.

Returns: true, if the node belongs to a container, false otherwise. This function can be used to test whether slist::iterator_to will return a valid iterator.

Complexity: Constant

3.
```
void unlink();
```

Effects: Removes the node if it's inserted in a container. This function is only allowed if link_mode is auto_unlink.

Throws: Nothing.

Struct template make_slist_member_hook

boost::intrusive::make_slist_member_hook

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/intrusive/slist_hook.hpp>

template<class... Options>
struct make_slist_member_hook {
  // types
  typedef implementation_defined type;
};
```

Description

Helper metafunction to define a `slist_member_hook` that yields to the same type when the same options (either explicitly or implicitly) are used.

Class template slist_member_hook

boost::intrusive::slist_member_hook

Synopsis

```
// In header: <boost/intrusive/slist_hook.hpp>

template<class... Options>
class slist_member_hook : public make_slist_member_hook::type< O1, O2, O3 > {
public:
  // construct/copy/destruct
  slist_member_hook();
  slist_member_hook(const slist_member_hook &);
  slist_member_hook& operator=(const slist_member_hook &);
  ~slist_member_hook();

  // public member functions
  void swap_nodes(slist_member_hook &);
  bool is_linked() const;
  void unlink();
};
```

Description

Put a public data member slist_member_hook in order to store objects of this class in an list. slist_member_hook holds the data necessary for maintaining the list and provides an appropriate value_traits class for list.

The hook admits the following options: `void_pointer<>` and `link_mode<>`.

`link_mode<>` will specify the linking mode of the hook (`normal_link`, `auto_unlink` or `safe_link`).

`void_pointer<>` is the pointer type that will be used internally in the hook and the the container configured to use this hook.

`slist_member_hook` public construct/copy/destruct

1.
```
slist_member_hook();
```

Effects: If `link_mode` is `auto_unlink` or `safe_link` initializes the node to an unlinked state.

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

2.
```
slist_member_hook(const slist_member_hook &);
```

   Effects: If link_mode is auto_unlink or safe_link initializes the node to an unlinked state. The argument is ignored.

   Throws: Nothing.

   Rationale: Providing a copy-constructor makes classes using the hook STL-compliant without forcing the user to do some additional work. swap can be used to emulate move-semantics.

3.
```
slist_member_hook& operator=(const slist_member_hook &);
```

   Effects: Empty function. The argument is ignored.

   Throws: Nothing.

   Rationale: Providing an assignment operator makes classes using the hook STL-compliant without forcing the user to do some additional work. swap can be used to emulate move-semantics.

4.
```
~slist_member_hook();
```

   Effects: If link_mode is normal_link, the destructor does nothing (ie. no code is generated). If link_mode is safe_link and the object is stored in an slist an assertion is raised. If link_mode is auto_unlink and is_linked() is true, the node is unlinked.

   Throws: Nothing.

slist_member_hook public member functions

1.
```
void swap_nodes(slist_member_hook & other);
```

   Effects: Swapping two nodes swaps the position of the elements related to those nodes in one or two containers. That is, if the node this is part of the element e1, the node x is part of the element e2 and both elements are included in the containers s1 and s2, then after the swap-operation e1 is in s2 at the position of e2 and e2 is in s1 at the position of e1. If one element is not in a container, then after the swap-operation the other element is not in a container. Iterators to e1 and e2 related to those nodes are invalidated.

   Complexity: Constant

   Throws: Nothing.

2.
```
bool is_linked() const;
```

   Precondition: link_mode must be safe_link or auto_unlink.

   Returns: true, if the node belongs to a container, false otherwise. This function can be used to test whether slist::iterator_to will return a valid iterator.

   Complexity: Constant

3.
```
void unlink();
```

   Effects: Removes the node if it's inserted in a container. This function is only allowed if link_mode is auto_unlink.

   Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Header <boost/intrusive/splay_set.hpp>

```
namespace boost {
  namespace intrusive {
    template<typename T, class... Options> class splay_set;

    template<typename T, class... Options> struct make_splay_set;

    template<typename T, class... Options> class splay_multiset;

    template<typename T, class... Options> struct make_splay_multiset;
    template<typename T, class... Options>
      bool operator!=(const splay_set< T, Options...> & x,
                      const splay_set< T, Options...> & y);
    template<typename T, class... Options>
      bool operator>(const splay_set< T, Options...> & x,
                     const splay_set< T, Options...> & y);
    template<typename T, class... Options>
      bool operator<=(const splay_set< T, Options...> & x,
                      const splay_set< T, Options...> & y);
    template<typename T, class... Options>
      bool operator>=(const splay_set< T, Options...> & x,
                      const splay_set< T, Options...> & y);
    template<typename T, class... Options>
      void swap(splay_set< T, Options...> & x, splay_set< T, Options...> & y);
    template<typename T, class... Options>
      bool operator!=(const splay_multiset< T, Options...> & x,
                      const splay_multiset< T, Options...> & y);
    template<typename T, class... Options>
      bool operator>(const splay_multiset< T, Options...> & x,
                     const splay_multiset< T, Options...> & y);
    template<typename T, class... Options>
      bool operator<=(const splay_multiset< T, Options...> & x,
                      const splay_multiset< T, Options...> & y);
    template<typename T, class... Options>
      bool operator>=(const splay_multiset< T, Options...> & x,
                      const splay_multiset< T, Options...> & y);
    template<typename T, class... Options>
      void swap(splay_multiset< T, Options...> & x,
                splay_multiset< T, Options...> & y);
  }
}
```

Class template splay_set

boost::intrusive::splay_set

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/intrusive/splay_set.hpp>

template<typename T, class... Options>
class splay_set {
public:
  // types
  typedef implementation_defined::value_type             value_type;
  typedef implementation_defined::value_traits           value_traits;
  typedef implementation_defined::pointer                pointer;
  typedef implementation_defined::const_pointer          const_pointer;
  typedef implementation_defined::reference              reference;
  typedef implementation_defined::const_reference        const_reference;
  typedef implementation_defined::difference_type        difference_type;
  typedef implementation_defined::size_type              size_type;
  typedef implementation_defined::value_compare          value_compare;
  typedef implementation_defined::key_compare            key_compare;
  typedef implementation_defined::iterator               iterator;
  typedef implementation_defined::const_iterator         const_iterator;
  typedef implementation_defined::reverse_iterator       reverse_iterator;
  typedef implementation_defined::const_reverse_iterator const_reverse_iterator;
  typedef implementation_defined::insert_commit_data     insert_commit_data;
  typedef implementation_defined::node_traits            node_traits;
  typedef implementation_defined::node                   node;
  typedef implementation_defined::node_ptr               node_ptr;
  typedef implementation_defined::const_node_ptr         const_node_ptr;
  typedef implementation_defined::node_algorithms        node_algorithms;

  // construct/copy/destruct
  explicit splay_set(const value_compare & = value_compare(),
                     const value_traits & = value_traits());
  template<typename Iterator>
    splay_set(Iterator, Iterator, const value_compare & = value_compare(),
              const value_traits & = value_traits());
  splay_set(BOOST_RV_REF(splay_set));
  splay_set& operator=(BOOST_RV_REF(splay_set));
  ~splay_set();

  // public member functions
  iterator begin();
  const_iterator begin() const;
  const_iterator cbegin() const;
  iterator end();
  const_iterator end() const;
  const_iterator cend() const;
  reverse_iterator rbegin();
  const_reverse_iterator rbegin() const;
  const_reverse_iterator crbegin() const;
  reverse_iterator rend();
  const_reverse_iterator rend() const;
  const_reverse_iterator crend() const;
  key_compare key_comp() const;
  value_compare value_comp() const;
  bool empty() const;
  size_type size() const;
  void swap(splay_set &);
  template<typename Cloner, typename Disposer>
    void clone_from(const splay_set &, Cloner, Disposer);
  std::pair< iterator, bool > insert(reference);
  iterator insert(const_iterator, reference);
  template<typename KeyType, typename KeyValueCompare>
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
    std::pair< iterator, bool >
    insert_check(const KeyType &, KeyValueCompare, insert_commit_data &);
  template<typename KeyType, typename KeyValueCompare>
    std::pair< iterator, bool >
    insert_check(const_iterator, const KeyType &, KeyValueCompare,
                 insert_commit_data &);
  iterator insert_commit(reference, const insert_commit_data &);
  template<typename Iterator> void insert(Iterator, Iterator);
  iterator erase(const_iterator);
  iterator erase(const_iterator, const_iterator);
  size_type erase(const_reference);
  template<typename KeyType, typename KeyValueCompare>
    size_type erase(const KeyType &, KeyValueCompare);
  template<typename Disposer>
    iterator erase_and_dispose(const_iterator, Disposer);
  template<typename Disposer>
    iterator erase_and_dispose(const_iterator, const_iterator, Disposer);
  template<typename Disposer>
    size_type erase_and_dispose(const_reference, Disposer);
  template<typename KeyType, typename KeyValueCompare, typename Disposer>
    size_type erase_and_dispose(const KeyType &, KeyValueCompare, Disposer);
  void clear();
  template<typename Disposer> void clear_and_dispose(Disposer);
  size_type count(const_reference);
  template<typename KeyType, typename KeyValueCompare>
    size_type count(const KeyType &, KeyValueCompare);
  size_type count_dont_splay(const_reference) const;
  template<typename KeyType, typename KeyValueCompare>
    size_type count_dont_splay(const KeyType &, KeyValueCompare) const;
  iterator lower_bound(const_reference);
  template<typename KeyType, typename KeyValueCompare>
    iterator lower_bound(const KeyType &, KeyValueCompare);
  const_iterator lower_bound_dont_splay(const_reference) const;
  template<typename KeyType, typename KeyValueCompare>
    const_iterator
    lower_bound_dont_splay(const KeyType &, KeyValueCompare) const;
  iterator upper_bound(const_reference);
  template<typename KeyType, typename KeyValueCompare>
    iterator upper_bound(const KeyType &, KeyValueCompare);
  const_iterator upper_bound_dont_splay(const_reference) const;
  template<typename KeyType, typename KeyValueCompare>
    const_iterator
    upper_bound_dont_splay(const KeyType &, KeyValueCompare) const;
  iterator find(const_reference);
  template<typename KeyType, typename KeyValueCompare>
    iterator find(const KeyType &, KeyValueCompare);
  const_iterator find_dont_splay(const_reference) const;
  template<typename KeyType, typename KeyValueCompare>
    const_iterator find_dont_splay(const KeyType &, KeyValueCompare) const;
  std::pair< iterator, iterator > equal_range(const_reference);
  template<typename KeyType, typename KeyValueCompare>
    std::pair< iterator, iterator >
    equal_range(const KeyType &, KeyValueCompare);
  std::pair< const_iterator, const_iterator >
  equal_range_dont_splay(const_reference) const;
  template<typename KeyType, typename KeyValueCompare>
    std::pair< const_iterator, const_iterator >
    equal_range_dont_splay(const KeyType &, KeyValueCompare) const;
  std::pair< iterator, iterator >
  bounded_range(const_reference, const_reference, bool, bool);
  template<typename KeyType, typename KeyValueCompare>
    std::pair< iterator, iterator >
    bounded_range(const KeyType &, const KeyType &, KeyValueCompare, bool,
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
                bool);
  std::pair< const_iterator, const_iterator >
  bounded_range_dont_splay_dont_splay(const_reference, const_reference, bool,
                                      bool) const;
  template<typename KeyType, typename KeyValueCompare>
    std::pair< const_iterator, const_iterator >
    bounded_range_dont_splay(const KeyType &, const KeyType &,
                             KeyValueCompare, bool, bool) const;
  iterator iterator_to(reference);
  const_iterator iterator_to(const_reference) const;
  pointer unlink_leftmost_without_rebalance();
  void replace_node(iterator, reference);
  void splay_up(iterator);
  template<typename KeyType, typename KeyNodePtrCompare>
    iterator splay_down(const KeyType &, KeyNodePtrCompare);
  iterator splay_down(const value_type &);
  void rebalance();
  iterator rebalance_subtree(iterator);

  // public static functions
  static splay_set & container_from_end_iterator(iterator);
  static const splay_set & container_from_end_iterator(const_iterator);
  static splay_set & container_from_iterator(iterator);
  static const splay_set & container_from_iterator(const_iterator);
  static iterator s_iterator_to(reference);
  static const_iterator s_iterator_to(const_reference);
  static void init_node(reference);

  // public data members
  static const bool constant_time_size;
};
```

Description

The class template splay_set is an intrusive container, that mimics most of the interface of std::set as described in the C++ standard.

The template parameter `T` is the type to be managed by the container. The user can specify additional options and if no options are provided default options are used.

The container supports the following options: `base_hook<>/member_hook<>/value_traits<>`, `constant_time_size<>`, `size_type<>` and `compare<>`.

`splay_set` public construct/copy/destruct

1.
```
explicit splay_set(const value_compare & cmp = value_compare(),
                   const value_traits & v_traits = value_traits());
```

   Effects: Constructs an empty `splay_set`.

   Complexity: Constant.

   Throws: If value_traits::node_traits::node constructor throws (this does not happen with predefined Boost.Intrusive hooks) or the copy constructor of the value_compare object throws.

2.
```
template<typename Iterator>
  splay_set(Iterator b, Iterator e,
            const value_compare & cmp = value_compare(),
            const value_traits & v_traits = value_traits());
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Requires: Dereferencing iterator must yield an lvalue of type value_type. cmp must be a comparison function that induces a strict weak ordering.

Effects: Constructs an empty `splay_set` and inserts elements from [b, e).

Complexity: Linear in N if [b, e) is already sorted using comp and otherwise amortized N * log N, where N is std::distance(last, first).

Throws: If value_traits::node_traits::node constructor throws (this does not happen with predefined Boost.Intrusive hooks) or the copy constructor/operator() of the value_compare object throws.

3.
```
splay_set(BOOST_RV_REF(splay_set) x);
```

   Effects: to-do

4.
```
splay_set& operator=(BOOST_RV_REF(splay_set) x);
```

   Effects: to-do

5.
```
~splay_set();
```

   Effects: Detaches all elements from this. The objects in the `splay_set` are not deleted (i.e. no destructors are called).

   Complexity: Linear to the number of elements on the container. if it's a safe-mode or auto-unlink value_type. Constant time otherwise.

   Throws: Nothing.

`splay_set` public member functions

1.
```
iterator begin();
```

   Effects: Returns an iterator pointing to the beginning of the `splay_set`.

   Complexity: Constant.

   Throws: Nothing.

2.
```
const_iterator begin() const;
```

   Effects: Returns a const_iterator pointing to the beginning of the `splay_set`.

   Complexity: Constant.

   Throws: Nothing.

3.
```
const_iterator cbegin() const;
```

   Effects: Returns a const_iterator pointing to the beginning of the `splay_set`.

   Complexity: Constant.

   Throws: Nothing.

4.
```
iterator end();
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Returns an iterator pointing to the end of the `splay_set`.

Complexity: Constant.

Throws: Nothing.

5.
```
const_iterator end() const;
```

Effects: Returns a const_iterator pointing to the end of the `splay_set`.

Complexity: Constant.

Throws: Nothing.

6.
```
const_iterator cend() const;
```

Effects: Returns a const_iterator pointing to the end of the `splay_set`.

Complexity: Constant.

Throws: Nothing.

7.
```
reverse_iterator rbegin();
```

Effects: Returns a reverse_iterator pointing to the beginning of the reversed `splay_set`.

Complexity: Constant.

Throws: Nothing.

8.
```
const_reverse_iterator rbegin() const;
```

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed `splay_set`.

Complexity: Constant.

Throws: Nothing.

9.
```
const_reverse_iterator crbegin() const;
```

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed `splay_set`.

Complexity: Constant.

Throws: Nothing.

10.
```
reverse_iterator rend();
```

Effects: Returns a reverse_iterator pointing to the end of the reversed `splay_set`.

Complexity: Constant.

Throws: Nothing.

11.
```
const_reverse_iterator rend() const;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Returns a const_reverse_iterator pointing to the end of the reversed `splay_set`.

Complexity: Constant.

Throws: Nothing.

12.
```
const_reverse_iterator crend() const;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed `splay_set`.

Complexity: Constant.

Throws: Nothing.

13.
```
key_compare key_comp() const;
```

Effects: Returns the key_compare object used by the `splay_set`.

Complexity: Constant.

Throws: If key_compare copy-constructor throws.

14.
```
value_compare value_comp() const;
```

Effects: Returns the value_compare object used by the `splay_set`.

Complexity: Constant.

Throws: If value_compare copy-constructor throws.

15.
```
bool empty() const;
```

Effects: Returns true if the container is empty.

Complexity: Constant.

Throws: Nothing.

16.
```
size_type size() const;
```

Effects: Returns the number of elements stored in the `splay_set`.

Complexity: Linear to elements contained in *this if, constant-time size option is enabled. Constant-time otherwise.

Throws: Nothing.

17.
```
void swap(splay_set & other);
```

Effects: Swaps the contents of two splay_sets.

Complexity: Constant.

Throws: If the swap() call for the comparison functor found using ADL throws. Strong guarantee.

18.
```
template<typename Cloner, typename Disposer> 
  void clone_from(const splay_set & src, Cloner cloner, Disposer disposer);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Requires: Disposer::operator()(pointer) shouldn't throw. Cloner should yield to nodes equivalent to the original nodes.

Effects: Erases all the elements from *this calling Disposer::operator()(pointer), clones all the elements from src calling Cloner::operator()(const_reference) and inserts them on *this. Copies the predicate from the source container.

If cloner throws, all cloned elements are unlinked and disposed calling Disposer::operator()(pointer).

Complexity: Linear to erased plus inserted elements.

Throws: If cloner throws or predicate copy assignment throws. Basic guarantee.

19.
```
std::pair< iterator, bool > insert(reference value);
```

Requires: value must be an lvalue

Effects: Tries to inserts value into the `splay_set`.

Returns: If the value is not already present inserts it and returns a pair containing the iterator to the new value and true. If there is an equivalent value returns a pair containing an iterator to the already present value and false.

Complexity: Amortized logarithmic.

Throws: If the internal value_compare ordering function throws. Strong guarantee.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

20.
```
iterator insert(const_iterator hint, reference value);
```

Requires: value must be an lvalue

Effects: Tries to to insert x into the `splay_set`, using "hint" as a hint to where it will be inserted.

Returns: An iterator that points to the position where the new element was inserted into the `splay_set`.

Complexity: Amortized logarithmic in general, but it's amortized constant time if t is inserted immediately before hint.

Throws: If the internal value_compare ordering function throws. Strong guarantee.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

21.
```
template<typename KeyType, typename KeyValueCompare> 
  std::pair< iterator, bool > 
  insert_check(const KeyType & key, KeyValueCompare key_value_comp, 
               insert_commit_data & commit_data);
```

Requires: key_value_comp must be a comparison function that induces the same strict weak ordering as value_compare. The difference is that key_value_comp compares an arbitrary key with the contained values.

Effects: Checks if a value can be inserted in the `splay_set`, using a user provided key instead of the value itself.

Returns: If there is an equivalent value returns a pair containing an iterator to the already present value and false. If the value can be inserted returns true in the returned pair boolean and fills "commit_data" that is meant to be used with the "insert_commit" function.

Complexity: Amortized logarithmic.

Throws: If the key_value_comp ordering function throws. Strong guarantee.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Notes: This function is used to improve performance when constructing a value_type is expensive: if there is an equivalent value the constructed object must be discarded. Many times, the part of the node that is used to impose the order is much cheaper to construct than the value_type and this function offers the possibility to use that part to check if the insertion will be successful.

If the check is successful, the user can construct the value_type and use "insert_commit" to insert the object in constant-time. This gives a total logarithmic complexity to the insertion: check(O(log(N)) + commit(O(1)).

"commit_data" remains valid for a subsequent "insert_commit" only if no more objects are inserted or erased from the `splay_set`.

22.
```
template<typename KeyType, typename KeyValueCompare>
  std::pair< iterator, bool >
  insert_check(const_iterator hint, const KeyType & key,
               KeyValueCompare key_value_comp,
               insert_commit_data & commit_data);
```

Requires: key_value_comp must be a comparison function that induces the same strict weak ordering as value_compare. The difference is that key_value_comp compares an arbitrary key with the contained values.

Effects: Checks if a value can be inserted in the `splay_set`, using a user provided key instead of the value itself, using "hint" as a hint to where it will be inserted.

Returns: If there is an equivalent value returns a pair containing an iterator to the already present value and false. If the value can be inserted returns true in the returned pair boolean and fills "commit_data" that is meant to be used with the "insert_commit" function.

Complexity: Amortized logarithmic in general, but it's amortized constant time if t is inserted immediately before hint.

Throws: If the key_value_comp ordering function throws. Strong guarantee.

Notes: This function is used to improve performance when constructing a value_type is expensive: if there is an equivalent value the constructed object must be discarded. Many times, the part of the constructing that is used to impose the order is much cheaper to construct than the value_type and this function offers the possibility to use that key to check if the insertion will be successful.

If the check is successful, the user can construct the value_type and use "insert_commit" to insert the object in constant-time. This can give a total constant-time complexity to the insertion: check(O(1)) + commit(O(1)).

"commit_data" remains valid for a subsequent "insert_commit" only if no more objects are inserted or erased from the `splay_set`.

23.
```
iterator insert_commit(reference value,
                       const insert_commit_data & commit_data);
```

Requires: value must be an lvalue of type value_type. commit_data must have been obtained from a previous call to "insert_check". No objects should have been inserted or erased from the `splay_set` between the "insert_check" that filled "commit_data" and the call to "insert_commit".

Effects: Inserts the value in the `splay_set` using the information obtained from the "commit_data" that a previous "insert_check" filled.

Returns: An iterator to the newly inserted object.

Complexity: Constant time.

Throws: Nothing.

Notes: This function has only sense if a "insert_check" has been previously executed to fill "commit_data". No value should be inserted or erased between the "insert_check" and "insert_commit" calls.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

24.
```
template<typename Iterator> void insert(Iterator b, Iterator e);
```

Requires: Dereferencing iterator must yield an lvalue of type value_type.

Effects: Inserts a range into the `splay_set`.

Complexity: Insert range is amortized O(N * log(N)), where N is the size of the range. However, it is linear in N if the range is already sorted by value_comp().

Throws: If the internal value_compare ordering function throws. Basic guarantee.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

25.
```
iterator erase(const_iterator i);
```

Effects: Erases the element pointed to by pos.

Complexity: Average complexity is constant time.

Returns: An iterator to the element after the erased element.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

26.
```
iterator erase(const_iterator b, const_iterator e);
```

Effects: Erases the range pointed to by b end e.

Complexity: Average complexity for erase range is amortized O(log(size() + N)), where N is the number of elements in the range.

Returns: An iterator to the element after the erased elements.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

27.
```
size_type erase(const_reference value);
```

Effects: Erases all the elements with the given value.

Returns: The number of erased elements.

Complexity: Amortized O(log(size()) + this->count(value)).

Throws: If the internal value_compare ordering function throws. Basic guarantee.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

28.
```
template<typename KeyType, typename KeyValueCompare> 
  size_type erase(const KeyType & key, KeyValueCompare comp);
```

Effects: Erases all the elements that compare equal with the given key and the given comparison functor.

Returns: The number of erased elements.

Complexity: Amortized O(log(size() + this->count(key, comp)).

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: If the comp ordering function throws. Basic guarantee.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

29.
```
template<typename Disposer> 
  iterator erase_and_dispose(const_iterator i, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases the element pointed to by pos. Disposer::operator()(pointer) is called for the removed element.

Complexity: Average complexity for erase element is constant time.

Returns: An iterator to the element after the erased element.

Throws: Nothing.

Note: Invalidates the iterators to the erased elements.

30.
```
template<typename Disposer> 
  iterator erase_and_dispose(const_iterator b, const_iterator e, 
                             Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases the range pointed to by b end e. Disposer::operator()(pointer) is called for the removed elements.

Complexity: Average complexity for erase range is at most O(log(size() + N)), where N is the number of elements in the range.

Returns: An iterator to the element after the erased elements.

Throws: Nothing.

Note: Invalidates the iterators to the erased elements.

31.
```
template<typename Disposer> 
  size_type erase_and_dispose(const_reference value, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases all the elements with the given value. Disposer::operator()(pointer) is called for the removed elements.

Throws: If the internal value_compare ordering function throws.

Complexity: Amortized O(log(size() + this->count(value)). Basic guarantee.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

32.
```
template<typename KeyType, typename KeyValueCompare, typename Disposer> 
  size_type erase_and_dispose(const KeyType & key, KeyValueCompare comp, 
                              Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases all the elements with the given key. according to the comparison functor "comp". Disposer::operator()(pointer) is called for the removed elements.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Returns: The number of erased elements.

Complexity: Amortized O(log(size() + this->count(key, comp)).

Throws: If comp ordering function throws. Basic guarantee.

Note: Invalidates the iterators to the erased elements.

33.
```
void clear();
```

Effects: Erases all the elements of the container.

Complexity: Linear to the number of elements on the container. if it's a safe-mode or auto-unlink value_type. Constant time otherwise.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

34.
```
template<typename Disposer> void clear_and_dispose(Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases all the elements of the container.

Complexity: Linear to the number of elements on the container. Disposer::operator()(pointer) is called for the removed elements.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

35.
```
size_type count(const_reference value);
```

Effects: Returns the number of contained elements with the given key

Complexity: Amortized logarithmic to the number of elements contained plus lineal to number of objects with the given key.

Throws: If the internal value_compare ordering function throws.

36.
```
template<typename KeyType, typename KeyValueCompare> 
  size_type count(const KeyType & key, KeyValueCompare comp);
```

Effects: Returns the number of contained elements with the same key compared with the given comparison functor.

Complexity: Amortized logarithmic to the number of elements contained plus lineal to number of objects with the given key.

Throws: If comp ordering function throws.

37.
```
size_type count_dont_splay(const_reference value) const;
```

Effects: Returns the number of contained elements with the given key

Complexity: Logarithmic to the number of elements contained plus lineal to number of objects with the given key.

Throws: If the internal value_compare ordering function throws.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

38.
```
template<typename KeyType, typename KeyValueCompare> 
  size_type count_dont_splay(const KeyType & key, KeyValueCompare comp) const;
```

Effects: Returns the number of contained elements with the same key compared with the given comparison functor.

Complexity: Logarithmic to the number of elements contained plus lineal to number of objects with the given key.

Throws: If comp ordering function throws.

39.
```
iterator lower_bound(const_reference value);
```

Effects: Returns an iterator to the first element whose key is not less than k or end() if that element does not exist.

Complexity: Amortized logarithmic.

Throws: If the internal value_compare ordering function throws.

40.
```
template<typename KeyType, typename KeyValueCompare> 
  iterator lower_bound(const KeyType & key, KeyValueCompare comp);
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Returns an iterator to the first element whose key according to the comparison functor is not less than k or end() if that element does not exist.

Complexity: Amortized logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

41.
```
const_iterator lower_bound_dont_splay(const_reference value) const;
```

Effects: Returns a const iterator to the first element whose key is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

42.
```
template<typename KeyType, typename KeyValueCompare> 
  const_iterator 
  lower_bound_dont_splay(const KeyType & key, KeyValueCompare comp) const;
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Returns a const_iterator to the first element whose key according to the comparison functor is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

43.
```
iterator upper_bound(const_reference value);
```

Effects: Returns an iterator to the first element whose key is greater than k or end() if that element does not exist.

Complexity: Amortized logarithmic.

Throws: If the internal value_compare ordering function throws.

44.
```
template<typename KeyType, typename KeyValueCompare> 
  iterator upper_bound(const KeyType & key, KeyValueCompare comp);
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Returns an iterator to the first element whose key according to the comparison functor is greater than key or end() if that element does not exist.

Complexity: Amortized logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

45.
```
const_iterator upper_bound_dont_splay(const_reference value) const;
```

Effects: Returns an iterator to the first element whose key is greater than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

46.
```
template<typename KeyType, typename KeyValueCompare> 
  const_iterator 
  upper_bound_dont_splay(const KeyType & key, KeyValueCompare comp) const;
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Returns a const_iterator to the first element whose key according to the comparison functor is greater than key or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

47.
```
iterator find(const_reference value);
```

Effects: Finds an iterator to the first element whose value is "value" or end() if that element does not exist.

Complexity: Amortized logarithmic.

Throws: If the internal value_compare ordering function throws.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

48.
```
template<typename KeyType, typename KeyValueCompare> 
  iterator find(const KeyType & key, KeyValueCompare comp);
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Finds an iterator to the first element whose key is "key" according to the comparison functor or end() if that element does not exist.

Complexity: Amortized logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

49.
```
const_iterator find_dont_splay(const_reference value) const;
```

Effects: Finds a const_iterator to the first element whose value is "value" or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

50.
```
template<typename KeyType, typename KeyValueCompare> 
  const_iterator 
  find_dont_splay(const KeyType & key, KeyValueCompare comp) const;
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Finds a const_iterator to the first element whose key is "key" according to the comparison functor or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

51.
```
std::pair< iterator, iterator > equal_range(const_reference value);
```

Effects: Finds a range containing all elements whose key is k or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Amortized logarithmic.

Throws: If the internal value_compare ordering function throws.

52.
```
template<typename KeyType, typename KeyValueCompare> 
  std::pair< iterator, iterator > 
  equal_range(const KeyType & key, KeyValueCompare comp);
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Finds a range containing all elements whose key is k according to the comparison functor or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Amortized logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

53.
```
std::pair< const_iterator, const_iterator >
equal_range_dont_splay(const_reference value) const;
```

Effects: Finds a range containing all elements whose key is k or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

54.
```
template<typename KeyType, typename KeyValueCompare>
  std::pair< const_iterator, const_iterator >
  equal_range_dont_splay(const KeyType & key, KeyValueCompare comp) const;
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Finds a range containing all elements whose key is k according to the comparison functor or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

55.
```
std::pair< iterator, iterator >
bounded_range(const_reference lower_value, const_reference upper_value,
              bool left_closed, bool right_closed);
```

Requires: 'lower_value' must not be greater than 'upper_value'. If 'lower_value' == 'upper_value', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key) if left_closed, upper_bound(lower_key) otherwise

second = upper_bound(upper_key) if right_closed, lower_bound(upper_key) otherwise

Complexity: Logarithmic.

Throws: If the predicate throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_value and upper_value.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

56.
```
template<typename KeyType, typename KeyValueCompare>
  std::pair< iterator, iterator >
  bounded_range(const KeyType & lower_key, const KeyType & upper_key,
                KeyValueCompare comp, bool left_closed, bool right_closed);
```

Requires: KeyValueCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. 'lower_key' must not be greater than 'upper_key' according to 'comp'. If 'lower_key' == 'upper_key', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key, comp) if left_closed, upper_bound(lower_key, comp) otherwise

second = upper_bound(upper_key, comp) if right_closed, lower_bound(upper_key, comp) otherwise

Complexity: Logarithmic.

Throws: If "comp" throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_key and upper_key.

57.
```
std::pair< const_iterator, const_iterator >
bounded_range_dont_splay_dont_splay(const_reference lower_value,
                                    const_reference upper_value,
                                    bool left_closed, bool right_closed) const;
```

Requires: 'lower_value' must not be greater than 'upper_value'. If 'lower_value' == 'upper_value', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key) if left_closed, upper_bound(lower_key) otherwise

second = upper_bound(upper_key) if right_closed, lower_bound(upper_key) otherwise

Complexity: Logarithmic.

Throws: If the predicate throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_value and upper_value.

58.
```
template<typename KeyType, typename KeyValueCompare>
  std::pair< const_iterator, const_iterator >
  bounded_range_dont_splay(const KeyType & lower_key,
                           const KeyType & upper_key, KeyValueCompare comp,
                           bool left_closed, bool right_closed) const;
```

Requires: KeyValueCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. 'lower_key' must not be greater than 'upper_key' according to 'comp'. If 'lower_key' == 'upper_key', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key, comp) if left_closed, upper_bound(lower_key, comp) otherwise

second = upper_bound(upper_key, comp) if right_closed, lower_bound(upper_key, comp) otherwise

Complexity: Logarithmic.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: If "comp" throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_key and upper_key.

59.
```
iterator iterator_to(reference value);
```

Requires: value must be an lvalue and shall be in a `splay_set` of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid iterator i belonging to the `splay_set` that points to the value

Complexity: Constant.

Throws: Nothing.

60.
```
const_iterator iterator_to(const_reference value) const;
```

Requires: value must be an lvalue and shall be in a `splay_set` of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid const_iterator i belonging to the `splay_set` that points to the value

Complexity: Constant.

Throws: Nothing.

61.
```
pointer unlink_leftmost_without_rebalance();
```

Effects: Unlinks the leftmost node from the tree.

Complexity: Average complexity is constant time.

Throws: Nothing.

Notes: This function breaks the tree and the tree can only be used for more unlink_leftmost_without_rebalance calls. This function is normally used to achieve a step by step controlled destruction of the tree.

62.
```
void replace_node(iterator replace_this, reference with_this);
```

Requires: replace_this must be a valid iterator of *this and with_this must not be inserted in any tree.

Effects: Replaces replace_this in its position in the tree with with_this. The tree does not need to be rebalanced.

Complexity: Constant.

Throws: Nothing.

Note: This function will break container ordering invariants if with_this is not equivalent to *replace_this according to the ordering rules. This function is faster than erasing and inserting the node, since no rebalancing or comparison is needed.

63.
```
void splay_up(iterator i);
```

Requires: i must be a valid iterator of *this.

Effects: Rearranges the splay set so that the element pointed by i is placed as the root of the tree, improving future searches of this value.

Complexity: Amortized logarithmic.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: Nothing.

64.
```
template<typename KeyType, typename KeyNodePtrCompare>
  iterator splay_down(const KeyType & key, KeyNodePtrCompare comp);
```

Effects: Rearranges the splay set so that if *this stores an element with a key equivalent to value the element is placed as the root of the tree. If the element is not present returns the last node compared with the key. If the tree is empty, end() is returned.

Complexity: Amortized logarithmic.

Returns: An iterator to the new root of the tree, end() if the tree is empty.

Throws: If the comparison functor throws.

65.
```
iterator splay_down(const value_type & value);
```

Effects: Rearranges the splay set so that if *this stores an element with a key equivalent to value the element is placed as the root of the tree.

Complexity: Amortized logarithmic.

Returns: An iterator to the new root of the tree, end() if the tree is empty.

Throws: If the predicate throws.

66.
```
void rebalance();
```

Effects: Rebalances the tree.

Throws: Nothing.

Complexity: Linear.

67.
```
iterator rebalance_subtree(iterator root);
```

Requires: old_root is a node of a tree.

Effects: Rebalances the subtree rooted at old_root.

Returns: The new root of the subtree.

Throws: Nothing.

Complexity: Linear to the elements in the subtree.

`splay_set` public static functions

1.
```
static splay_set & container_from_end_iterator(iterator end_iterator);
```

Precondition: end_iterator must be a valid end iterator of `splay_set`.

Effects: Returns a const reference to the `splay_set` associated to the end iterator

Throws: Nothing.

Complexity: Constant.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

2.
```
static const splay_set &
container_from_end_iterator(const_iterator end_iterator);
```

Precondition: end_iterator must be a valid end const_iterator of `splay_set`.

Effects: Returns a const reference to the `splay_set` associated to the end iterator

Throws: Nothing.

Complexity: Constant.

3.
```
static splay_set & container_from_iterator(iterator it);
```

Precondition: it must be a valid iterator of set.

Effects: Returns a reference to the set associated to the iterator

Throws: Nothing.

Complexity: Constant.

4.
```
static const splay_set & container_from_iterator(const_iterator it);
```

Precondition: it must be a valid const_iterator of set.

Effects: Returns a const reference to the set associated to the iterator

Throws: Nothing.

Complexity: Logarithmic.

5.
```
static iterator s_iterator_to(reference value);
```

Requires: value must be an lvalue and shall be in a `splay_set` of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid iterator i belonging to the `splay_set` that points to the value

Complexity: Constant.

Throws: Nothing.

Note: This static function is available only if the *value traits* is stateless.

6.
```
static const_iterator s_iterator_to(const_reference value);
```

Requires: value must be an lvalue and shall be in a `splay_set` of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid const_iterator i belonging to the `splay_set` that points to the value

Complexity: Constant.

Throws: Nothing.

Note: This static function is available only if the *value traits* is stateless.

7.
```
static void init_node(reference value);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Requires: value shall not be in a splay_set/multisplay_set.

Effects: init_node puts the hook of a value in a well-known default state.

Throws: Nothing.

Complexity: Constant time.

Note: This function puts the hook in the well-known default state used by auto_unlink and safe hooks.

Struct template make_splay_set

boost::intrusive::make_splay_set

Synopsis

```
// In header: <boost/intrusive/splay_set.hpp>

template<typename T, class... Options>
struct make_splay_set {
  // types
  typedef implementation_defined type;
};
```

Description

Helper metafunction to define a `splay_set` that yields to the same type when the same options (either explicitly or implicitly) are used.

Class template splay_multiset

boost::intrusive::splay_multiset

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/intrusive/splay_set.hpp>

template<typename T, class... Options>
class splay_multiset {
public:
  // types
  typedef implementation_defined::value_type             value_type;
  typedef implementation_defined::value_traits           value_traits;
  typedef implementation_defined::pointer                pointer;
  typedef implementation_defined::const_pointer          const_pointer;
  typedef implementation_defined::reference              reference;
  typedef implementation_defined::const_reference        const_reference;
  typedef implementation_defined::difference_type        difference_type;
  typedef implementation_defined::size_type              size_type;
  typedef implementation_defined::value_compare          value_compare;
  typedef implementation_defined::key_compare            key_compare;
  typedef implementation_defined::iterator               iterator;
  typedef implementation_defined::const_iterator         const_iterator;
  typedef implementation_defined::reverse_iterator       reverse_iterator;
  typedef implementation_defined::const_reverse_iterator const_reverse_iterator;
  typedef implementation_defined::insert_commit_data     insert_commit_data;
  typedef implementation_defined::node_traits            node_traits;
  typedef implementation_defined::node                   node;
  typedef implementation_defined::node_ptr               node_ptr;
  typedef implementation_defined::const_node_ptr         const_node_ptr;
  typedef implementation_defined::node_algorithms        node_algorithms;

  // construct/copy/destruct
  explicit splay_multiset(const value_compare & = value_compare(),
                          const value_traits & = value_traits());
  template<typename Iterator>
    splay_multiset(Iterator, Iterator,
                   const value_compare & = value_compare(),
                   const value_traits & = value_traits());
  splay_multiset(BOOST_RV_REF(splay_multiset));
  splay_multiset& operator=(BOOST_RV_REF(splay_multiset));
  ~splay_multiset();

  // public member functions
  iterator begin();
  const_iterator begin() const;
  const_iterator cbegin() const;
  iterator end();
  const_iterator end() const;
  const_iterator cend() const;
  reverse_iterator rbegin();
  const_reverse_iterator rbegin() const;
  const_reverse_iterator crbegin() const;
  reverse_iterator rend();
  const_reverse_iterator rend() const;
  const_reverse_iterator crend() const;
  key_compare key_comp() const;
  value_compare value_comp() const;
  bool empty() const;
  size_type size() const;
  void swap(splay_multiset &);
  template<typename Cloner, typename Disposer>
    void clone_from(const splay_multiset &, Cloner, Disposer);
  iterator insert(reference);
  iterator insert(const_iterator, reference);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
template<typename Iterator> void insert(Iterator, Iterator);
iterator erase(const_iterator);
iterator erase(const_iterator, const_iterator);
size_type erase(const_reference);
template<typename KeyType, typename KeyValueCompare>
  size_type erase(const KeyType &, KeyValueCompare);
template<typename Disposer>
  iterator erase_and_dispose(const_iterator, Disposer);
template<typename Disposer>
  iterator erase_and_dispose(const_iterator, const_iterator, Disposer);
template<typename Disposer>
  size_type erase_and_dispose(const_reference, Disposer);
template<typename KeyType, typename KeyValueCompare, typename Disposer>
  size_type erase_and_dispose(const KeyType &, KeyValueCompare, Disposer);
void clear();
template<typename Disposer> void clear_and_dispose(Disposer);
size_type count(const_reference);
template<typename KeyType, typename KeyValueCompare>
  size_type count(const KeyType &, KeyValueCompare);
size_type count_dont_splay(const_reference) const;
template<typename KeyType, typename KeyValueCompare>
  size_type count_dont_splay(const KeyType &, KeyValueCompare) const;
iterator lower_bound(const_reference);
template<typename KeyType, typename KeyValueCompare>
  iterator lower_bound(const KeyType &, KeyValueCompare);
const_iterator lower_bound_dont_splay(const_reference) const;
template<typename KeyType, typename KeyValueCompare>
  const_iterator
  lower_bound_dont_splay(const KeyType &, KeyValueCompare) const;
iterator upper_bound(const_reference);
template<typename KeyType, typename KeyValueCompare>
  iterator upper_bound(const KeyType &, KeyValueCompare);
const_iterator upper_bound_dont_splay(const_reference) const;
template<typename KeyType, typename KeyValueCompare>
  const_iterator
  upper_bound_dont_splay(const KeyType &, KeyValueCompare) const;
iterator find(const_reference);
template<typename KeyType, typename KeyValueCompare>
  iterator find(const KeyType &, KeyValueCompare);
const_iterator find_dont_splay(const_reference) const;
template<typename KeyType, typename KeyValueCompare>
  const_iterator find_dont_splay(const KeyType &, KeyValueCompare) const;
std::pair< iterator, iterator > equal_range(const_reference);
template<typename KeyType, typename KeyValueCompare>
  std::pair< iterator, iterator >
  equal_range(const KeyType &, KeyValueCompare);
std::pair< const_iterator, const_iterator >
equal_range_dont_splay(const_reference) const;
template<typename KeyType, typename KeyValueCompare>
  std::pair< const_iterator, const_iterator >
  equal_range_dont_splay(const KeyType &, KeyValueCompare) const;
std::pair< iterator, iterator >
bounded_range(const_reference, const_reference, bool, bool);
template<typename KeyType, typename KeyValueCompare>
  std::pair< iterator, iterator >
  bounded_range(const KeyType &, const KeyType &, KeyValueCompare, bool,
                bool);
std::pair< const_iterator, const_iterator >
bounded_range_dont_splay(const_reference, const_reference, bool, bool) const;
template<typename KeyType, typename KeyValueCompare>
  std::pair< const_iterator, const_iterator >
  bounded_range_dont_splay(const KeyType &, const KeyType &,
                           KeyValueCompare, bool, bool) const;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
  iterator iterator_to(reference);
  const_iterator iterator_to(const_reference) const;
  pointer unlink_leftmost_without_rebalance();
  void replace_node(iterator, reference);
  void splay_up(iterator);
  template<typename KeyType, typename KeyNodePtrCompare>
    iterator splay_down(const KeyType &, KeyNodePtrCompare);
  iterator splay_down(const value_type &);
  void rebalance();
  iterator rebalance_subtree(iterator);

  // public static functions
  static splay_multiset & container_from_end_iterator(iterator);
  static const splay_multiset & container_from_end_iterator(const_iterator);
  static splay_multiset & container_from_iterator(iterator);
  static const splay_multiset & container_from_iterator(const_iterator);
  static iterator s_iterator_to(reference);
  static const_iterator s_iterator_to(const_reference);
  static void init_node(reference);

  // public data members
  static const bool constant_time_size;
};
```

Description

The class template splay_multiset is an intrusive container, that mimics most of the interface of std::multiset as described in the C++ standard.

The template parameter `T` is the type to be managed by the container. The user can specify additional options and if no options are provided default options are used.

The container supports the following options: `base_hook<>/member_hook<>/value_traits<>`, `constant_time_size<>`, `size_type<>` and `compare<>`.

`splay_multiset` public construct/copy/destruct

1.
```
explicit splay_multiset(const value_compare & cmp = value_compare(),
                        const value_traits & v_traits = value_traits());
```

Effects: Constructs an empty `splay_multiset`.

Complexity: Constant.

Throws: If value_traits::node_traits::node constructor throws (this does not happen with predefined Boost.Intrusive hooks) or the copy constructor/operator() of the value_compare object throws.

2.
```
template<typename Iterator>
  splay_multiset(Iterator b, Iterator e,
                 const value_compare & cmp = value_compare(),
                 const value_traits & v_traits = value_traits());
```

Requires: Dereferencing iterator must yield an lvalue of type value_type. cmp must be a comparison function that induces a strict weak ordering.

Effects: Constructs an empty `splay_multiset` and inserts elements from [b, e).

Complexity: Linear in N if [b, e) is already sorted using comp and otherwise amortized N * log N, where N is the distance between first and last.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: If value_traits::node_traits::node constructor throws (this does not happen with predefined Boost.Intrusive hooks) or the copy constructor/operator() of the value_compare object throws.

3.
```
splay_multiset(BOOST_RV_REF(splay_multiset) x);
```

Effects: to-do

4.
```
splay_multiset& operator=(BOOST_RV_REF(splay_multiset) x);
```

Effects: to-do

5.
```
~splay_multiset();
```

Effects: Detaches all elements from this. The objects in the set are not deleted (i.e. no destructors are called).

Complexity: Linear to the number of elements on the container. if it's a safe-mode or auto-unlink value_type. Constant time otherwise.

Throws: Nothing.

`splay_multiset` public member functions

1.
```
iterator begin();
```

Effects: Returns an iterator pointing to the beginning of the `splay_multiset`.

Complexity: Constant.

Throws: Nothing.

2.
```
const_iterator begin() const;
```

Effects: Returns a const_iterator pointing to the beginning of the `splay_multiset`.

Complexity: Constant.

Throws: Nothing.

3.
```
const_iterator cbegin() const;
```

Effects: Returns a const_iterator pointing to the beginning of the `splay_multiset`.

Complexity: Constant.

Throws: Nothing.

4.
```
iterator end();
```

Effects: Returns an iterator pointing to the end of the `splay_multiset`.

Complexity: Constant.

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

5.
```
const_iterator end() const;
```

   Effects: Returns a const_iterator pointing to the end of the `splay_multiset`.

   Complexity: Constant.

   Throws: Nothing.

6.
```
const_iterator cend() const;
```

   Effects: Returns a const_iterator pointing to the end of the `splay_multiset`.

   Complexity: Constant.

   Throws: Nothing.

7.
```
reverse_iterator rbegin();
```

   Effects: Returns a reverse_iterator pointing to the beginning of the reversed `splay_multiset`.

   Complexity: Constant.

   Throws: Nothing.

8.
```
const_reverse_iterator rbegin() const;
```

   Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed `splay_multiset`.

   Complexity: Constant.

   Throws: Nothing.

9.
```
const_reverse_iterator crbegin() const;
```

   Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed `splay_multiset`.

   Complexity: Constant.

   Throws: Nothing.

10.
```
reverse_iterator rend();
```

   Effects: Returns a reverse_iterator pointing to the end of the reversed `splay_multiset`.

   Complexity: Constant.

   Throws: Nothing.

11.
```
const_reverse_iterator rend() const;
```

   Effects: Returns a const_reverse_iterator pointing to the end of the reversed `splay_multiset`.

   Complexity: Constant.

   Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

12.
```
const_reverse_iterator crend() const;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed `splay_multiset`.

Complexity: Constant.

Throws: Nothing.

13.
```
key_compare key_comp() const;
```

Effects: Returns the key_compare object used by the `splay_multiset`.

Complexity: Constant.

Throws: If key_compare copy-constructor throws.

14.
```
value_compare value_comp() const;
```

Effects: Returns the value_compare object used by the `splay_multiset`.

Complexity: Constant.

Throws: If value_compare copy-constructor throws.

15.
```
bool empty() const;
```

Effects: Returns true if the container is empty.

Complexity: Constant.

Throws: Nothing.

16.
```
size_type size() const;
```

Effects: Returns the number of elements stored in the `splay_multiset`.

Complexity: Linear to elements contained in *this if, constant-time size option is enabled. Constant-time otherwise.

Throws: Nothing.

17.
```
void swap(splay_multiset & other);
```

Effects: Swaps the contents of two splay_multisets.

Complexity: Constant.

Throws: If the swap() call for the comparison functor found using ADL throws. Strong guarantee.

18.
```
template<typename Cloner, typename Disposer> 
  void clone_from(const splay_multiset & src, Cloner cloner, 
                  Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw. Cloner should yield to nodes equivalent to the original nodes.

Effects: Erases all the elements from *this calling Disposer::operator()(pointer), clones all the elements from src calling Cloner::operator()(const_reference) and inserts them on *this. Copies the predicate from the source container.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

If cloner throws, all cloned elements are unlinked and disposed calling Disposer::operator()(pointer).

Complexity: Linear to erased plus inserted elements.

Throws: If cloner throws or predicate copy assignment throws. Basic guarantee.

19.
```
iterator insert(reference value);
```

Requires: value must be an lvalue

Effects: Inserts value into the `splay_multiset`.

Returns: An iterator that points to the position where the new element was inserted.

Complexity: Amortized logarithmic.

Throws: If the internal value_compare ordering function throws. Strong guarantee.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

20.
```
iterator insert(const_iterator hint, reference value);
```

Requires: value must be an lvalue

Effects: Inserts x into the `splay_multiset`, using pos as a hint to where it will be inserted.

Returns: An iterator that points to the position where the new element was inserted.

Complexity: Amortized logarithmic in general, but it is amortized constant time if t is inserted immediately before hint.

Throws: If the internal value_compare ordering function throws. Strong guarantee.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

21.
```
template<typename Iterator> void insert(Iterator b, Iterator e);
```

Requires: Dereferencing iterator must yield an lvalue of type value_type.

Effects: Inserts a range into the `splay_multiset`.

Returns: An iterator that points to the position where the new element was inserted.

Complexity: Insert range is amortized O(N * log(N)), where N is the size of the range. However, it is linear in N if the range is already sorted by value_comp().

Throws: If the internal value_compare ordering function throws. Basic guarantee.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

22.
```
iterator erase(const_iterator i);
```

Effects: Erases the element pointed to by pos.

Complexity: Average complexity is constant time.

Returns: An iterator to the element after the erased element.

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

23.
```
iterator erase(const_iterator b, const_iterator e);
```

Effects: Erases the range pointed to by b end e.

Returns: An iterator to the element after the erased elements.

Complexity: Average complexity for erase range is amortized O(log(size() + N)), where N is the number of elements in the range.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

24.
```
size_type erase(const_reference value);
```

Effects: Erases all the elements with the given value.

Returns: The number of erased elements.

Complexity: Amortized O(log(size() + this->count(value)).

Throws: If the internal value_compare ordering function throws. Basic guarantee.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

25.
```
template<typename KeyType, typename KeyValueCompare>
  size_type erase(const KeyType & key, KeyValueCompare comp);
```

Effects: Erases all the elements that compare equal with the given key and the given comparison functor.

Returns: The number of erased elements.

Complexity: Amortized O(log(size() + this->count(key, comp)).

Throws: If comp ordering function throws. Basic guarantee.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

26.
```
template<typename Disposer>
  iterator erase_and_dispose(const_iterator i, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Returns: An iterator to the element after the erased element.

Effects: Erases the element pointed to by pos. Disposer::operator()(pointer) is called for the removed element.

Complexity: Average complexity for erase element is constant time.

Throws: Nothing.

Note: Invalidates the iterators to the erased elements.

27.
```
template<typename Disposer>
  iterator erase_and_dispose(const_iterator b, const_iterator e,
                             Disposer disposer);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Requires: Disposer::operator()(pointer) shouldn't throw.

Returns: An iterator to the element after the erased elements.

Effects: Erases the range pointed to by b end e. Disposer::operator()(pointer) is called for the removed elements.

Complexity: Average complexity for erase range is amortized O(log(size() + N)), where N is the number of elements in the range.

Throws: Nothing.

Note: Invalidates the iterators to the erased elements.

28.
```
template<typename Disposer> 
  size_type erase_and_dispose(const_reference value, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases all the elements with the given value. Disposer::operator()(pointer) is called for the removed elements.

Returns: The number of erased elements.

Complexity: Amortized O(log(size() + this->count(value)).

Throws: If the internal value_compare ordering function throws. Basic guarantee.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

29.
```
template<typename KeyType, typename KeyValueCompare, typename Disposer> 
  size_type erase_and_dispose(const KeyType & key, KeyValueCompare comp, 
                              Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases all the elements with the given key. according to the comparison functor "comp". Disposer::operator()(pointer) is called for the removed elements.

Returns: The number of erased elements.

Complexity: Amortized O(log(size() + this->count(key, comp)).

Throws: If comp ordering function throws. Basic guarantee.

Note: Invalidates the iterators to the erased elements.

30.
```
void clear();
```

Effects: Erases all the elements of the container.

Complexity: Linear to the number of elements on the container. if it's a safe-mode or auto-unlink value_type. Constant time otherwise.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

31.
```
template<typename Disposer> void clear_and_dispose(Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Erases all the elements of the container.

Complexity: Linear to the number of elements on the container. Disposer::operator()(pointer) is called for the removed elements.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

32.
```
size_type count(const_reference value);
```

Effects: Returns the number of contained elements with the given key

Complexity: Amortized logarithmic to the number of elements contained plus lineal to number of objects with the given key.

Throws: If the internal value_compare ordering function throws.

33.
```
template<typename KeyType, typename KeyValueCompare>
  size_type count(const KeyType & key, KeyValueCompare comp);
```

Effects: Returns the number of contained elements with the same key compared with the given comparison functor.

Complexity: Amortized logarithmic to the number of elements contained plus lineal to number of objects with the given key.

Throws: If comp ordering function throws.

34.
```
size_type count_dont_splay(const_reference value) const;
```

Effects: Returns the number of contained elements with the given key

Complexity: Logarithmic to the number of elements contained plus lineal to number of objects with the given key.

Throws: If the internal value_compare ordering function throws.

35.
```
template<typename KeyType, typename KeyValueCompare>
  size_type count_dont_splay(const KeyType & key, KeyValueCompare comp) const;
```

Effects: Returns the number of contained elements with the same key compared with the given comparison functor.

Complexity: Logarithmic to the number of elements contained plus lineal to number of objects with the given key.

Throws: If comp ordering function throws.

36.
```
iterator lower_bound(const_reference value);
```

Effects: Returns an iterator to the first element whose key is not less than k or end() if that element does not exist.

Complexity: Amortized logarithmic.

Throws: If the internal value_compare ordering function throws.

37.
```
template<typename KeyType, typename KeyValueCompare>
  iterator lower_bound(const KeyType & key, KeyValueCompare comp);
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Returns an iterator to the first element whose key according to the comparison functor is not less than k or end() if that element does not exist.

Complexity: Amortized logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

38.
```
const_iterator lower_bound_dont_splay(const_reference value) const;
```

Effects: Returns a const iterator to the first element whose key is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

39.
```
template<typename KeyType, typename KeyValueCompare>
  const_iterator
  lower_bound_dont_splay(const KeyType & key, KeyValueCompare comp) const;
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Returns a const_iterator to the first element whose key according to the comparison functor is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

40.
```
iterator upper_bound(const_reference value);
```

Effects: Returns an iterator to the first element whose key is greater than k or end() if that element does not exist.

Complexity: Amortized logarithmic.

Throws: If the internal value_compare ordering function throws.

41.
```
template<typename KeyType, typename KeyValueCompare>
  iterator upper_bound(const KeyType & key, KeyValueCompare comp);
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Returns an iterator to the first element whose key according to the comparison functor is greater than key or end() if that element does not exist.

Complexity: Amortized logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

42.
```
const_iterator upper_bound_dont_splay(const_reference value) const;
```

Effects: Returns an iterator to the first element whose key is greater than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

43.
```
template<typename KeyType, typename KeyValueCompare> 
  const_iterator 
  upper_bound_dont_splay(const KeyType & key, KeyValueCompare comp) const;
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Returns a const_iterator to the first element whose key according to the comparison functor is greater than key or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

44.
```
iterator find(const_reference value);
```

Effects: Finds an iterator to the first element whose value is "value" or end() if that element does not exist.

Complexity: Amortized logarithmic.

Throws: If the internal value_compare ordering function throws.

45.
```
template<typename KeyType, typename KeyValueCompare> 
  iterator find(const KeyType & key, KeyValueCompare comp);
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Finds an iterator to the first element whose key is "key" according to the comparison functor or end() if that element does not exist.

Complexity: Amortized logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

46.
```
const_iterator find_dont_splay(const_reference value) const;
```

Effects: Finds a const_iterator to the first element whose value is "value" or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

47.
```
template<typename KeyType, typename KeyValueCompare> 
  const_iterator 
  find_dont_splay(const KeyType & key, KeyValueCompare comp) const;
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Finds a const_iterator to the first element whose key is "key" according to the comparison functor or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

48.
```
std::pair< iterator, iterator > equal_range(const_reference value);
```

Effects: Finds a range containing all elements whose key is k or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Amortized logarithmic.

Throws: If the internal value_compare ordering function throws.

49.
```
template<typename KeyType, typename KeyValueCompare> 
  std::pair< iterator, iterator > 
  equal_range(const KeyType & key, KeyValueCompare comp);
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Finds a range containing all elements whose key is k according to the comparison functor or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Amortized logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

50.
```
std::pair< const_iterator, const_iterator > 
equal_range_dont_splay(const_reference value) const;
```

Effects: Finds a range containing all elements whose key is k or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

51.
```
template<typename KeyType, typename KeyValueCompare> 
  std::pair< const_iterator, const_iterator > 
  equal_range_dont_splay(const KeyType & key, KeyValueCompare comp) const;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Finds a range containing all elements whose key is k according to the comparison functor or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

52.
```
std::pair< iterator, iterator >
bounded_range(const_reference lower_value, const_reference upper_value,
              bool left_closed, bool right_closed);
```

Requires: 'lower_value' must not be greater than 'upper_value'. If 'lower_value' == 'upper_value', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key) if left_closed, upper_bound(lower_key) otherwise

second = upper_bound(upper_key) if right_closed, lower_bound(upper_key) otherwise

Complexity: Logarithmic.

Throws: If the predicate throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_value and upper_value.

53.
```
template<typename KeyType, typename KeyValueCompare>
  std::pair< iterator, iterator >
  bounded_range(const KeyType & lower_key, const KeyType & upper_key,
                KeyValueCompare comp, bool left_closed, bool right_closed);
```

Requires: KeyValueCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. 'lower_key' must not be greater than 'upper_key' according to 'comp'. If 'lower_key' == 'upper_key', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key, comp) if left_closed, upper_bound(lower_key, comp) otherwise

second = upper_bound(upper_key, comp) if right_closed, lower_bound(upper_key, comp) otherwise

Complexity: Logarithmic.

Throws: If "comp" throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_key and upper_key.

54.
```
std::pair< const_iterator, const_iterator >
bounded_range_dont_splay(const_reference lower_value,
                         const_reference upper_value, bool left_closed,
                         bool right_closed) const;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Requires: 'lower_value' must not be greater than 'upper_value'. If 'lower_value' == 'upper_value', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key) if left_closed, upper_bound(lower_key) otherwise

second = upper_bound(upper_key) if right_closed, lower_bound(upper_key) otherwise

Complexity: Logarithmic.

Throws: If the predicate throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_value and upper_value.

55.
```
template<typename KeyType, typename KeyValueCompare>
  std::pair< const_iterator, const_iterator >
  bounded_range_dont_splay(const KeyType & lower_key,
                           const KeyType & upper_key, KeyValueCompare comp,
                           bool left_closed, bool right_closed) const;
```

Requires: KeyValueCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. 'lower_key' must not be greater than 'upper_key' according to 'comp'. If 'lower_key' == 'upper_key', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key, comp) if left_closed, upper_bound(lower_key, comp) otherwise

second = upper_bound(upper_key, comp) if right_closed, lower_bound(upper_key, comp) otherwise

Complexity: Logarithmic.

Throws: If "comp" throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_key and upper_key.

56.
```
iterator iterator_to(reference value);
```

Requires: value must be an lvalue and shall be in a set of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid iterator i belonging to the set that points to the value

Complexity: Constant.

Throws: Nothing.

57.
```
const_iterator iterator_to(const_reference value) const;
```

Requires: value must be an lvalue and shall be in a set of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid const_iterator i belonging to the set that points to the value

Complexity: Constant.

Throws: Nothing.

58.
```
pointer unlink_leftmost_without_rebalance();
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Unlinks the leftmost node from the tree.

Complexity: Average complexity is constant time.

Throws: Nothing.

Notes: This function breaks the tree and the tree can only be used for more unlink_leftmost_without_rebalance calls. This function is normally used to achieve a step by step controlled destruction of the tree.

59.
```
void replace_node(iterator replace_this, reference with_this);
```

Requires: replace_this must be a valid iterator of *this and with_this must not be inserted in any tree.

Effects: Replaces replace_this in its position in the tree with with_this. The tree does not need to be rebalanced.

Complexity: Constant.

Throws: Nothing.

Note: This function will break container ordering invariants if with_this is not equivalent to *replace_this according to the ordering rules. This function is faster than erasing and inserting the node, since no rebalancing or comparison is needed.

60.
```
void splay_up(iterator i);
```

Requires: i must be a valid iterator of *this.

Effects: Rearranges the splay set so that the element pointed by i is placed as the root of the tree, improving future searches of this value.

Complexity: Amortized logarithmic.

Throws: Nothing.

61.
```
template<typename KeyType, typename KeyNodePtrCompare>
  iterator splay_down(const KeyType & key, KeyNodePtrCompare comp);
```

Effects: Rearranges the splay set so that if *this stores an element with a key equivalent to value the element is placed as the root of the tree. If the element is not present returns the last node compared with the key. If the tree is empty, end() is returned.

Complexity: Amortized logarithmic.

Returns: An iterator to the new root of the tree, end() if the tree is empty.

Throws: If the comparison functor throws.

62.
```
iterator splay_down(const value_type & value);
```

Effects: Rearranges the splay set so that if *this stores an element with a key equivalent to value the element is placed as the root of the tree.

Complexity: Amortized logarithmic.

Returns: An iterator to the new root of the tree, end() if the tree is empty.

Throws: If the predicate throws.

63.
```
void rebalance();
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Rebalances the tree.

Throws: Nothing.

Complexity: Linear.

64.
```
iterator rebalance_subtree(iterator root);
```

Requires: old_root is a node of a tree.

Effects: Rebalances the subtree rooted at old_root.

Returns: The new root of the subtree.

Throws: Nothing.

Complexity: Linear to the elements in the subtree.

`splay_multiset` public static functions

1.
```
static splay_multiset & container_from_end_iterator(iterator end_iterator);
```

Precondition: end_iterator must be a valid end iterator of `splay_multiset`.

Effects: Returns a const reference to the `splay_multiset` associated to the end iterator

Throws: Nothing.

Complexity: Constant.

2.
```
static const splay_multiset &
container_from_end_iterator(const_iterator end_iterator);
```

Precondition: end_iterator must be a valid end const_iterator of `splay_multiset`.

Effects: Returns a const reference to the `splay_multiset` associated to the end iterator

Throws: Nothing.

Complexity: Constant.

3.
```
static splay_multiset & container_from_iterator(iterator it);
```

Precondition: it must be a valid iterator of multiset.

Effects: Returns a const reference to the multiset associated to the iterator

Throws: Nothing.

Complexity: Logarithmic.

4.
```
static const splay_multiset & container_from_iterator(const_iterator it);
```

Precondition: it must be a valid const_iterator of multiset.

Effects: Returns a const reference to the multiset associated to the iterator

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Constant.

5.
```
static iterator s_iterator_to(reference value);
```

Requires: value must be an lvalue and shall be in a set of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid iterator i belonging to the set that points to the value

Complexity: Constant.

Throws: Nothing.

Note: This static function is available only if the *value traits* is stateless.

6.
```
static const_iterator s_iterator_to(const_reference value);
```

Requires: value must be an lvalue and shall be in a set of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid const_iterator i belonging to the set that points to the value

Complexity: Constant.

Throws: Nothing.

Note: This static function is available only if the *value traits* is stateless.

7.
```
static void init_node(reference value);
```

Requires: value shall not be in a set/splay_multiset.

Effects: init_node puts the hook of a value in a well-known default state.

Throws: Nothing.

Complexity: Constant time.

Note: This function puts the hook in the well-known default state used by auto_unlink and safe hooks.

Struct template make_splay_multiset

boost::intrusive::make_splay_multiset

Synopsis

```
// In header: <boost/intrusive/splay_set.hpp>

template<typename T, class... Options>
struct make_splay_multiset {
  // types
  typedef implementation_defined type;
};
```

Description

Helper metafunction to define a `splay_multiset` that yields to the same type when the same options (either explicitly or implicitly) are used.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Header <boost/intrusive/splay_set_hook.hpp>

```
namespace boost {
  namespace intrusive {
    template<class... Options> struct make_splay_set_base_hook;

    template<class... Options> class splay_set_base_hook;

    template<class... Options> struct make_splay_set_member_hook;

    template<class... Options> class splay_set_member_hook;
  }
}
```

Struct template make_splay_set_base_hook

boost::intrusive::make_splay_set_base_hook

Synopsis

```
// In header: <boost/intrusive/splay_set_hook.hpp>

template<class... Options>
struct make_splay_set_base_hook {
  // types
  typedef implementation_defined type;
};
```

Description

Helper metafunction to define a `splay_set_base_hook` that yields to the same type when the same options (either explicitly or implicitly) are used.

Class template splay_set_base_hook

boost::intrusive::splay_set_base_hook

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/intrusive/splay_set_hook.hpp>

template<class... Options>
class splay_set_base_hook :
  public make_splay_set_base_hook::type< O1, O2, O3 >
{
public:
  // construct/copy/destruct
  splay_set_base_hook();
  splay_set_base_hook(const splay_set_base_hook &);
  splay_set_base_hook& operator=(const splay_set_base_hook &);
  ~splay_set_base_hook();

  // public member functions
  void swap_nodes(splay_set_base_hook &);
  bool is_linked() const;
  void unlink();
};
```

Description

Derive a class from splay_set_base_hook in order to store objects in in a splay_set/splay_multiset. splay_set_base_hook holds the data necessary to maintain the splay_set/splay_multiset and provides an appropriate value_traits class for splay_set/splay_multiset.

The hook admits the following options: `tag<>`, `void_pointer<>`, `link_mode<>` and `optimize_size<>`.

`tag<>` defines a tag to identify the node. The same tag value can be used in different classes, but if a class is derived from more than one `list_base_hook`, then each `list_base_hook` needs its unique tag.

`void_pointer<>` is the pointer type that will be used internally in the hook and the the container configured to use this hook.

`link_mode<>` will specify the linking mode of the hook (`normal_link`, `auto_unlink` or `safe_link`).

`splay_set_base_hook` public construct/copy/destruct

1.
```
splay_set_base_hook();
```

   Effects: If `link_mode` is `auto_unlink` or `safe_link` initializes the node to an unlinked state.

   Throws: Nothing.

2.
```
splay_set_base_hook(const splay_set_base_hook &);
```

   Effects: If `link_mode` is `auto_unlink` or `safe_link` initializes the node to an unlinked state. The argument is ignored.

   Throws: Nothing.

   Rationale: Providing a copy-constructor makes classes using the hook STL-compliant without forcing the user to do some additional work. `swap` can be used to emulate move-semantics.

3.
```
splay_set_base_hook& operator=(const splay_set_base_hook &);
```

   Effects: Empty function. The argument is ignored.

   Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Rationale: Providing an assignment operator makes classes using the hook STL-compliant without forcing the user to do some additional work. `swap` can be used to emulate move-semantics.

4.
```
~splay_set_base_hook();
```

Effects: If `link_mode` is `normal_link`, the destructor does nothing (ie. no code is generated). If `link_mode` is `safe_link` and the object is stored in a set an assertion is raised. If `link_mode` is `auto_unlink` and `is_linked()` is true, the node is unlinked.

Throws: Nothing.

`splay_set_base_hook` public member functions

1.
```
void swap_nodes(splay_set_base_hook & other);
```

Effects: Swapping two nodes swaps the position of the elements related to those nodes in one or two containers. That is, if the node this is part of the element e1, the node x is part of the element e2 and both elements are included in the containers s1 and s2, then after the swap-operation e1 is in s2 at the position of e2 and e2 is in s1 at the position of e1. If one element is not in a container, then after the swap-operation the other element is not in a container. Iterators to e1 and e2 related to those nodes are invalidated.

Complexity: Constant

Throws: Nothing.

2.
```
bool is_linked() const;
```

Precondition: `link_mode` must be `safe_link` or `auto_unlink`.

Returns: true, if the node belongs to a container, false otherwise. This function can be used to test whether `set::iterator_to` will return a valid iterator.

Complexity: Constant

3.
```
void unlink();
```

Effects: Removes the node if it's inserted in a container. This function is only allowed if `link_mode` is `auto_unlink`.

Throws: Nothing.

Struct template make_splay_set_member_hook

boost::intrusive::make_splay_set_member_hook

Synopsis

```
// In header: <boost/intrusive/splay_set_hook.hpp>

template<class... Options> 
struct make_splay_set_member_hook {
  // types
  typedef implementation_defined type;
};
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Description

Helper metafunction to define a `splay_set_member_hook` that yields to the same type when the same options (either explicitly or implicitly) are used.

Class template splay_set_member_hook

boost::intrusive::splay_set_member_hook

Synopsis

```
// In header: <boost/intrusive/splay_set_hook.hpp>

template<class... Options>
class splay_set_member_hook :
  public make_splay_set_member_hook::type< O1, O2, O3 >
{
public:
  // construct/copy/destruct
  splay_set_member_hook();
  splay_set_member_hook(const splay_set_member_hook &);
  splay_set_member_hook& operator=(const splay_set_member_hook &);
  ~splay_set_member_hook();

  // public member functions
  void swap_nodes(splay_set_member_hook &);
  bool is_linked() const;
  void unlink();
};
```

Description

Put a public data member splay_set_member_hook in order to store objects of this class in a splay_set/splay_multiset. splay_set_member_hook holds the data necessary for maintaining the splay_set/splay_multiset and provides an appropriate value_traits class for splay_set/splay_multiset.

The hook admits the following options: `void_pointer<>`, `link_mode<>` and `optimize_size<>`.

`void_pointer<>` is the pointer type that will be used internally in the hook and the the container configured to use this hook.

`link_mode<>` will specify the linking mode of the hook (`normal_link`, `auto_unlink` or `safe_link`).

`splay_set_member_hook` public construct/copy/destruct

1.
   ```
   splay_set_member_hook();
   ```

   Effects: If `link_mode` is `auto_unlink` or `safe_link` initializes the node to an unlinked state.

   Throws: Nothing.

2.
   ```
   splay_set_member_hook(const splay_set_member_hook &);
   ```

   Effects: If `link_mode` is `auto_unlink` or `safe_link` initializes the node to an unlinked state. The argument is ignored.

   Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Rationale: Providing a copy-constructor makes classes using the hook STL-compliant without forcing the user to do some additional work. swap can be used to emulate move-semantics.

3.
```
splay_set_member_hook& operator=(const splay_set_member_hook &);
```

Effects: Empty function. The argument is ignored.

Throws: Nothing.

Rationale: Providing an assignment operator makes classes using the hook STL-compliant without forcing the user to do some additional work. swap can be used to emulate move-semantics.

4.
```
~splay_set_member_hook();
```

Effects: If link_mode is normal_link, the destructor does nothing (ie. no code is generated). If link_mode is safe_link and the object is stored in a set an assertion is raised. If link_mode is auto_unlink and is_linked() is true, the node is unlinked.

Throws: Nothing.

splay_set_member_hook public member functions

1.
```
void swap_nodes(splay_set_member_hook & other);
```

Effects: Swapping two nodes swaps the position of the elements related to those nodes in one or two containers. That is, if the node this is part of the element e1, the node x is part of the element e2 and both elements are included in the containers s1 and s2, then after the swap-operation e1 is in s2 at the position of e2 and e2 is in s1 at the position of e1. If one element is not in a container, then after the swap-operation the other element is not in a container. Iterators to e1 and e2 related to those nodes are invalidated.

Complexity: Constant

Throws: Nothing.

2.
```
bool is_linked() const;
```

Precondition: link_mode must be safe_link or auto_unlink.

Returns: true, if the node belongs to a container, false otherwise. This function can be used to test whether set::iterator_to will return a valid iterator.

Complexity: Constant

3.
```
void unlink();
```

Effects: Removes the node if it's inserted in a container. This function is only allowed if link_mode is auto_unlink.

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Header <boost/intrusive/splaytree.hpp>

```
namespace boost {
  namespace intrusive {
    template<typename T, class... Options> class splaytree;

    template<typename T, class... Options> struct make_splaytree;
    template<typename T, class... Options>
      bool operator<(const splaytree< T, Options...> & x,
                     const splaytree< T, Options...> & y);
    template<typename T, class... Options>
      bool operator==(const splaytree< T, Options...> & x,
                      const splaytree< T, Options...> & y);
    template<typename T, class... Options>
      bool operator!=(const splaytree< T, Options...> & x,
                      const splaytree< T, Options...> & y);
    template<typename T, class... Options>
      bool operator>(const splaytree< T, Options...> & x,
                     const splaytree< T, Options...> & y);
    template<typename T, class... Options>
      bool operator<=(const splaytree< T, Options...> & x,
                      const splaytree< T, Options...> & y);
    template<typename T, class... Options>
      bool operator>=(const splaytree< T, Options...> & x,
                      const splaytree< T, Options...> & y);
    template<typename T, class... Options>
      void swap(splaytree< T, Options...> & x, splaytree< T, Options...> & y);
  }
}
```

Class template splaytree

boost::intrusive::splaytree

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/intrusive/splaytree.hpp>

template<typename T, class... Options> 
class splaytree {
public:
  // types
  typedef Config::value_traits                                                   value_traits; ↵

  typedef real_value_traits::pointer                                             pointer;       ↵

  typedef real_value_traits::const_pointer                                      const_pointer; ↵

  typedef pointer_traits< pointer >::element_type                                value_type;    ↵

  typedef pointer_traits< pointer >::reference                                   reference;     ↵

  typedef pointer_traits< const_pointer >::reference                               const_refer↵
ence;
  typedef pointer_traits< pointer >::difference_type                          difference_type; ↵

  typedef Config::size_type                                                      size_type;     ↵

  typedef value_type                                                             key_type;      ↵

  typedef Config::compare                                                       value_compare; ↵

  typedef value_compare                                                          key_compare;   ↵

  typedef tree_iterator< splaytree, false >                                      iterator;      ↵

  typedef tree_iterator< splaytree, true >                                         const_iterat↵
or;
  typedef unspecified                                                            reverse_iterat↵
or;
  typedef unspecified                                                        const_reverse_iter↵
ator;
  typedef real_value_traits::node_traits                                         node_traits;   ↵

  typedef node_traits::node                                                      node;          ↵

  typedef pointer_traits< pointer >::template rebind_pointer< node >::type       node_ptr;      ↵

  typedef pointer_traits< pointer >::template rebind_pointer< const node >::type const_node_ptr; ↵

  typedef splaytree_algorithms< node_traits >                                 node_algorithms; ↵

  typedef node_algorithms::insert_commit_data                                      insert_com↵
mit_data;

  // construct/copy/destruct
  explicit splaytree(const value_compare & = value_compare(), 
                     const value_traits & = value_traits());
  template<typename Iterator> 
    splaytree(bool, Iterator, Iterator, 
              const value_compare & = value_compare(), 
              const value_traits & = value_traits());
  splaytree(BOOST_RV_REF(splaytree));
  splaytree& operator=(BOOST_RV_REF(splaytree));
  ~splaytree();
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
// public member functions
const real_value_traits & get_real_value_traits() const;
real_value_traits & get_real_value_traits();
iterator begin();
const_iterator begin() const;
const_iterator cbegin() const;
iterator end();
const_iterator end() const;
const_iterator cend() const;
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
const_reverse_iterator crbegin() const;
reverse_iterator rend();
const_reverse_iterator rend() const;
const_reverse_iterator crend() const;
value_compare value_comp() const;
bool empty() const;
size_type size() const;
void swap(splaytree &);
iterator insert_equal(reference);
iterator insert_equal(const_iterator, reference);
template<typename Iterator> void insert_equal(Iterator, Iterator);
std::pair< iterator, bool > insert_unique(reference);
iterator insert_unique(const_iterator, reference);
template<typename Iterator> void insert_unique(Iterator, Iterator);
template<typename KeyType, typename KeyValueCompare>
  std::pair< iterator, bool >
  insert_unique_check(const KeyType &, KeyValueCompare,
                      insert_commit_data &);
template<typename KeyType, typename KeyValueCompare>
  std::pair< iterator, bool >
  insert_unique_check(const_iterator, const KeyType &, KeyValueCompare,
                      insert_commit_data &);
iterator insert_unique_commit(reference, const insert_commit_data &);
iterator erase(const_iterator);
iterator erase(const_iterator, const_iterator);
size_type erase(const_reference);
template<typename KeyType, typename KeyValueCompare>
  size_type erase(const KeyType &, KeyValueCompare);
template<typename Disposer>
  iterator erase_and_dispose(const_iterator, Disposer);
template<typename Disposer>
  iterator erase_and_dispose(const_iterator, const_iterator, Disposer);
template<typename Disposer>
  size_type erase_and_dispose(const_reference, Disposer);
template<typename KeyType, typename KeyValueCompare, typename Disposer>
  size_type erase_and_dispose(const KeyType &, KeyValueCompare, Disposer);
void clear();
template<typename Disposer> void clear_and_dispose(Disposer);
size_type count(const_reference);
template<typename KeyType, typename KeyValueCompare>
  size_type count(const KeyType &, KeyValueCompare);
size_type count_dont_splay(const_reference) const;
template<typename KeyType, typename KeyValueCompare>
  size_type count_dont_splay(const KeyType &, KeyValueCompare) const;
iterator lower_bound(const_reference);
const_iterator lower_bound_dont_splay(const_reference) const;
template<typename KeyType, typename KeyValueCompare>
  iterator lower_bound(const KeyType &, KeyValueCompare);
template<typename KeyType, typename KeyValueCompare>
  const_iterator
  lower_bound_dont_splay(const KeyType &, KeyValueCompare) const;
iterator upper_bound(const_reference);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
  template<typename KeyType, typename KeyValueCompare>
    iterator upper_bound(const KeyType &, KeyValueCompare);
  const_iterator upper_bound_dont_splay(const_reference) const;
  template<typename KeyType, typename KeyValueCompare>
    const_iterator
    upper_bound_dont_splay(const KeyType &, KeyValueCompare) const;
  iterator find(const_reference);
  template<typename KeyType, typename KeyValueCompare>
    iterator find(const KeyType &, KeyValueCompare);
  const_iterator find_dont_splay(const_reference) const;
  template<typename KeyType, typename KeyValueCompare>
    const_iterator find_dont_splay(const KeyType &, KeyValueCompare) const;
  std::pair< iterator, iterator > equal_range(const_reference);
  template<typename KeyType, typename KeyValueCompare>
    std::pair< iterator, iterator >
    equal_range(const KeyType &, KeyValueCompare);
  std::pair< const_iterator, const_iterator >
  equal_range_dont_splay(const_reference) const;
  template<typename KeyType, typename KeyValueCompare>
    std::pair< const_iterator, const_iterator >
    equal_range_dont_splay(const KeyType &, KeyValueCompare) const;
  std::pair< iterator, iterator >
  bounded_range(const_reference, const_reference, bool, bool);
  template<typename KeyType, typename KeyValueCompare>
    std::pair< iterator, iterator >
    bounded_range(const KeyType &, const KeyType &, KeyValueCompare, bool,
                  bool);
  std::pair< const_iterator, const_iterator >
  bounded_range(const_reference, const_reference, bool, bool) const;
  template<typename KeyType, typename KeyValueCompare>
    std::pair< const_iterator, const_iterator >
    bounded_range(const KeyType &, const KeyType &, KeyValueCompare, bool,
                  bool) const;
  template<typename Cloner, typename Disposer>
    void clone_from(const splaytree &, Cloner, Disposer);
  pointer unlink_leftmost_without_rebalance();
  void splay_up(iterator);
  template<typename KeyType, typename KeyValueCompare>
    iterator splay_down(const KeyType &, KeyValueCompare);
  iterator splay_down(const value_type &);
  void replace_node(iterator, reference);
  iterator iterator_to(reference);
  const_iterator iterator_to(const_reference) const;
  void rebalance();
  iterator rebalance_subtree(iterator);

  // public static functions
  static splaytree & container_from_end_iterator(iterator);
  static const splaytree & container_from_end_iterator(const_iterator);
  static splaytree & container_from_iterator(iterator);
  static const splaytree & container_from_iterator(const_iterator);
  static iterator s_iterator_to(reference);
  static const_iterator s_iterator_to(const_reference);
  static void init_node(reference);

  // private static functions
  static splaytree & priv_container_from_end_iterator(const const_iterator &);
  static splaytree & priv_container_from_iterator(const const_iterator &);

  // public data members
  static const bool constant_time_size;
  static const bool stateful_value_traits;
};
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Description

The class template splaytree is an intrusive splay tree container that is used to construct intrusive splay_set and splay_multiset containers. The no-throw guarantee holds only, if the value_compare object doesn't throw.

The template parameter `T` is the type to be managed by the container. The user can specify additional options and if no options are provided default options are used.

The container supports the following options: `base_hook<>/member_hook<>/value_traits<>`, `constant_time_size<>`, `size_type<>` and `compare<>`.

`splaytree` public construct/copy/destruct

1.
   ```
   explicit splaytree(const value_compare & cmp = value_compare(),
                      const value_traits & v_traits = value_traits());
   ```

   Effects: Constructs an empty tree.

   Complexity: Constant.

   Throws: If value_traits::node_traits::node constructor throws (this does not happen with predefined Boost.Intrusive hooks) or the copy constructorof the value_compare object throws. Basic guarantee.

2.
   ```
   template<typename Iterator>
     splaytree(bool unique, Iterator b, Iterator e,
               const value_compare & cmp = value_compare(),
               const value_traits & v_traits = value_traits());
   ```

   Requires: Dereferencing iterator must yield an lvalue of type value_type. cmp must be a comparison function that induces a strict weak ordering.

   Effects: Constructs an empty tree and inserts elements from [b, e).

   Complexity: Linear in N if [b, e) is already sorted using comp and otherwise amortized N * log N, where N is the distance between first and last.

   Throws: If value_traits::node_traits::node constructor throws (this does not happen with predefined Boost.Intrusive hooks) or the copy constructor/operator() of the value_compare object throws. Basic guarantee.

3.
   ```
   splaytree(BOOST_RV_REF(splaytree) x);
   ```

   Effects: to-do

4.
   ```
   splaytree& operator=(BOOST_RV_REF(splaytree) x);
   ```

   Effects: to-do

5.
   ```
   ~splaytree();
   ```

   Effects: Detaches all elements from this. The objects in the set are not deleted (i.e. no destructors are called), but the nodes according to the `value_traits` template parameter are reinitialized and thus can be reused.

   Complexity: Linear to the number of elements on the container. if it's a safe-mode or auto-unlink value_type. Constant time otherwise.

   Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

`splaytree` public member functions

1.
```
const real_value_traits & get_real_value_traits() const;
```

2.
```
real_value_traits & get_real_value_traits();
```

3.
```
iterator begin();
```

   Effects: Returns an iterator pointing to the beginning of the tree.

   Complexity: Constant.

   Throws: Nothing.

4.
```
const_iterator begin() const;
```

   Effects: Returns a const_iterator pointing to the beginning of the tree.

   Complexity: Constant.

   Throws: Nothing.

5.
```
const_iterator cbegin() const;
```

   Effects: Returns a const_iterator pointing to the beginning of the tree.

   Complexity: Constant.

   Throws: Nothing.

6.
```
iterator end();
```

   Effects: Returns an iterator pointing to the end of the tree.

   Complexity: Constant.

   Throws: Nothing.

7.
```
const_iterator end() const;
```

   Effects: Returns a const_iterator pointing to the end of the tree.

   Complexity: Constant.

   Throws: Nothing.

8.
```
const_iterator cend() const;
```

   Effects: Returns a const_iterator pointing to the end of the tree.

   Complexity: Constant.

   Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

9.
```
reverse_iterator rbegin();
```

Effects: Returns a reverse_iterator pointing to the beginning of the reversed tree.

Complexity: Constant.

Throws: Nothing.

10.
```
const_reverse_iterator rbegin() const;
```

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed tree.

Complexity: Constant.

Throws: Nothing.

11.
```
const_reverse_iterator crbegin() const;
```

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed tree.

Complexity: Constant.

Throws: Nothing.

12.
```
reverse_iterator rend();
```

Effects: Returns a reverse_iterator pointing to the end of the reversed tree.

Complexity: Constant.

Throws: Nothing.

13.
```
const_reverse_iterator rend() const;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed tree.

Complexity: Constant.

Throws: Nothing.

14.
```
const_reverse_iterator crend() const;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed tree.

Complexity: Constant.

Throws: Nothing.

15.
```
value_compare value_comp() const;
```

Effects: Returns the value_compare object used by the tree.

Complexity: Constant.

Throws: If value_compare copy-constructor throws.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

16.
```
bool empty() const;
```

Effects: Returns true if the container is empty.

Complexity: Constant.

Throws: Nothing.

17.
```
size_type size() const;
```

Effects: Returns the number of elements stored in the tree.

Complexity: Linear to elements contained in *this if constant-time size option is disabled. Constant time otherwise.

Throws: Nothing.

18.
```
void swap(splaytree & other);
```

Effects: Swaps the contents of two splaytrees.

Complexity: Constant.

Throws: If the comparison functor's swap call throws.

19.
```
iterator insert_equal(reference value);
```

Requires: value must be an lvalue

Effects: Inserts value into the tree before the lower bound.

Complexity: Average complexity for insert element is amortized logarithmic.

Throws: If the internal value_compare ordering function throws. Strong guarantee.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

20.
```
iterator insert_equal(const_iterator hint, reference value);
```

Requires: value must be an lvalue, and "hint" must be a valid iterator.

Effects: Inserts x into the tree, using "hint" as a hint to where it will be inserted. If "hint" is the upper_bound the insertion takes constant time (two comparisons in the worst case)

Complexity: Amortized logarithmic in general, but it is amortized constant time if t is inserted immediately before hint.

Throws: If the internal value_compare ordering function throws. Strong guarantee.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

21.
```
template<typename Iterator> void insert_equal(Iterator b, Iterator e);
```

Requires: Dereferencing iterator must yield an lvalue of type value_type.

Effects: Inserts a each element of a range into the tree before the upper bound of the key of each element.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Insert range is in general amortized O(N * log(N)), where N is the size of the range. However, it is linear in N if the range is already sorted by value_comp().

Throws: Nothing.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

22.
```
std::pair< iterator, bool > insert_unique(reference value);
```

Requires: value must be an lvalue

Effects: Inserts value into the tree if the value is not already present.

Complexity: Amortized logarithmic.

Throws: Nothing.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

23.
```
iterator insert_unique(const_iterator hint, reference value);
```

Requires: value must be an lvalue, and "hint" must be a valid iterator

Effects: Tries to insert x into the tree, using "hint" as a hint to where it will be inserted.

Complexity: Amortized logarithmic in general, but it is amortized constant time (two comparisons in the worst case) if t is inserted immediately before hint.

Throws: Nothing.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

24.
```
template<typename Iterator> void insert_unique(Iterator b, Iterator e);
```

Requires: Dereferencing iterator must yield an lvalue of type value_type.

Effects: Tries to insert each element of a range into the tree.

Complexity: Insert range is in general amortized O(N * log(N)), where N is the size of the range. However, it is linear in N if the range is already sorted by value_comp().

Throws: Nothing.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

25.
```
template<typename KeyType, typename KeyValueCompare> 
  std::pair< iterator, bool > 
  insert_unique_check(const KeyType & key, KeyValueCompare key_value_comp, 
                      insert_commit_data & commit_data);
```

Requires: key_value_comp must be a comparison function that induces the same strict weak ordering as value_compare. The difference is that key_value_comp compares an arbitrary key with the contained values.

Effects: Checks if a value can be inserted in the container, using a user provided key instead of the value itself.

Returns: If there is an equivalent value returns a pair containing an iterator to the already present value and false. If the value can be inserted returns true in the returned pair boolean and fills "commit_data" that is meant to be used with the "insert_commit" function.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Average complexity is at most logarithmic.

Throws: If the key_value_comp ordering function throws. Strong guarantee.

Notes: This function is used to improve performance when constructing a value_type is expensive: if there is an equivalent value the constructed object must be discarded. Many times, the part of the node that is used to impose the order is much cheaper to construct than the value_type and this function offers the possibility to use that part to check if the insertion will be successful.

If the check is successful, the user can construct the value_type and use "insert_commit" to insert the object in constant-time. This gives a total logarithmic complexity to the insertion: check(O(log(N)) + commit(O(1)).

"commit_data" remains valid for a subsequent "insert_commit" only if no more objects are inserted or erased from the container.

26.
```
template<typename KeyType, typename KeyValueCompare> 
  std::pair< iterator, bool > 
  insert_unique_check(const_iterator hint, const KeyType & key, 
                      KeyValueCompare key_value_comp, 
                      insert_commit_data & commit_data);
```

Requires: key_value_comp must be a comparison function that induces the same strict weak ordering as value_compare. The difference is that key_value_comp compares an arbitrary key with the contained values.

Effects: Checks if a value can be inserted in the container, using a user provided key instead of the value itself, using "hint" as a hint to where it will be inserted.

Returns: If there is an equivalent value returns a pair containing an iterator to the already present value and false. If the value can be inserted returns true in the returned pair boolean and fills "commit_data" that is meant to be used with the "insert_commit" function.

Complexity: Logarithmic in general, but it's amortized constant time if t is inserted immediately before hint.

Throws: If the key_value_comp ordering function throws. Strong guarantee.

Notes: This function is used to improve performance when constructing a value_type is expensive: if there is an equivalent value the constructed object must be discarded. Many times, the part of the constructing that is used to impose the order is much cheaper to construct than the value_type and this function offers the possibility to use that key to check if the insertion will be successful.

If the check is successful, the user can construct the value_type and use "insert_commit" to insert the object in constant-time. This can give a total constant-time complexity to the insertion: check(O(1)) + commit(O(1)).

"commit_data" remains valid for a subsequent "insert_commit" only if no more objects are inserted or erased from the container.

27.
```
iterator insert_unique_commit(reference value, 
                              const insert_commit_data & commit_data);
```

Requires: value must be an lvalue of type value_type. commit_data must have been obtained from a previous call to "insert_check". No objects should have been inserted or erased from the container between the "insert_check" that filled "commit_data" and the call to "insert_commit".

Effects: Inserts the value in the `avl_set` using the information obtained from the "commit_data" that a previous "insert_check" filled.

Returns: An iterator to the newly inserted object.

Complexity: Constant time.

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Notes: This function has only sense if a "insert_check" has been previously executed to fill "commit_data". No value should be inserted or erased between the "insert_check" and "insert_commit" calls.

28.
```
iterator erase(const_iterator i);
```

Effects: Erases the element pointed to by pos.

Complexity: Average complexity for erase element is constant time.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

29.
```
iterator erase(const_iterator b, const_iterator e);
```

Effects: Erases the range pointed to by b end e.

Complexity: Average complexity for erase range is amortized O(log(size() + N)), where N is the number of elements in the range.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

30.
```
size_type erase(const_reference value);
```

Effects: Erases all the elements with the given value.

Returns: The number of erased elements.

Complexity: Amortized O(log(size() + N).

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

31.
```
template<typename KeyType, typename KeyValueCompare>
  size_type erase(const KeyType & key, KeyValueCompare comp);
```

Effects: Erases all the elements with the given key. according to the comparison functor "comp".

Returns: The number of erased elements.

Complexity: Amortized O(log(size() + N).

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

32.
```
template<typename Disposer>
  iterator erase_and_dispose(const_iterator i, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases the element pointed to by pos. Disposer::operator()(pointer) is called for the removed element.

Complexity: Average complexity for erase element is constant time.

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Note: Invalidates the iterators to the erased elements.

33.
```
template<typename Disposer> 
  iterator erase_and_dispose(const_iterator b, const_iterator e, 
                             Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases the range pointed to by b end e. Disposer::operator()(pointer) is called for the removed elements.

Complexity: Average complexity for erase range is amortized O(log(size() + N)), where N is the number of elements in the range.

Throws: Nothing.

Note: Invalidates the iterators to the erased elements.

34.
```
template<typename Disposer> 
  size_type erase_and_dispose(const_reference value, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases all the elements with the given value. Disposer::operator()(pointer) is called for the removed elements.

Returns: The number of erased elements.

Complexity: Amortized O(log(size() + N).

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

35.
```
template<typename KeyType, typename KeyValueCompare, typename Disposer> 
  size_type erase_and_dispose(const KeyType & key, KeyValueCompare comp, 
                              Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases all the elements with the given key. according to the comparison functor "comp". Disposer::operator()(pointer) is called for the removed elements.

Returns: The number of erased elements.

Complexity: Amortized O(log(size() + N).

Throws: Nothing.

Note: Invalidates the iterators to the erased elements.

36.
```
void clear();
```

Effects: Erases all of the elements.

Complexity: Linear to the number of elements on the container. if it's a safe-mode or auto-unlink value_type. Constant time otherwise.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

37.
```
template<typename Disposer> void clear_and_dispose(Disposer disposer);
```

Effects: Erases all of the elements calling disposer(p) for each node to be erased. **Complexity**: Amortized O(log(size() + N)), where N is the number of elements in the container.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. Calls N times to disposer functor.

38.
```
size_type count(const_reference value);
```

Effects: Returns the number of contained elements with the given value

Complexity: Amortized logarithmic to the number of elements contained plus lineal to number of objects with the given value.

Throws: Nothing.

39.
```
template<typename KeyType, typename KeyValueCompare> 
  size_type count(const KeyType & key, KeyValueCompare comp);
```

Effects: Returns the number of contained elements with the given key

Complexity: Amortized logarithmic to the number of elements contained plus lineal to number of objects with the given key.

Throws: Nothing.

40.
```
size_type count_dont_splay(const_reference value) const;
```

Effects: Returns the number of contained elements with the given value

Complexity: Amortized logarithmic to the number of elements contained plus lineal to number of objects with the given value.

Throws: Nothing.

41.
```
template<typename KeyType, typename KeyValueCompare> 
  size_type count_dont_splay(const KeyType & key, KeyValueCompare comp) const;
```

Effects: Returns the number of contained elements with the given key

Complexity: Amortized logarithmic to the number of elements contained plus lineal to number of objects with the given key.

Throws: Nothing.

42.
```
iterator lower_bound(const_reference value);
```

Effects: Returns an iterator to the first element whose key is not less than k or end() if that element does not exist.

Complexity: Amortized logarithmic.

Throws: Nothing.

43.
```
const_iterator lower_bound_dont_splay(const_reference value) const;
```

Effects: Returns an iterator to the first element whose key is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: Nothing.

44.
```
template<typename KeyType, typename KeyValueCompare> 
  iterator lower_bound(const KeyType & key, KeyValueCompare comp);
```

Effects: Returns an iterator to the first element whose key is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

45.
```
template<typename KeyType, typename KeyValueCompare> 
  const_iterator 
  lower_bound_dont_splay(const KeyType & key, KeyValueCompare comp) const;
```

Effects: Returns a const iterator to the first element whose key is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

46.
```
iterator upper_bound(const_reference value);
```

Effects: Returns an iterator to the first element whose key is greater than k or end() if that element does not exist.

Complexity: Amortized logarithmic.

Throws: Nothing.

47.
```
template<typename KeyType, typename KeyValueCompare> 
  iterator upper_bound(const KeyType & key, KeyValueCompare comp);
```

Effects: Returns an iterator to the first element whose key is greater than k according to comp or end() if that element does not exist.

Complexity: Amortized logarithmic.

Throws: Nothing.

48.
```
const_iterator upper_bound_dont_splay(const_reference value) const;
```

Effects: Returns an iterator to the first element whose key is greater than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

49.
```
template<typename KeyType, typename KeyValueCompare> 
  const_iterator 
  upper_bound_dont_splay(const KeyType & key, KeyValueCompare comp) const;
```

Effects: Returns an iterator to the first element whose key is greater than k according to comp or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

50.
```
iterator find(const_reference value);
```

Effects: Finds an iterator to the first element whose key is k or end() if that element does not exist.

Complexity: Amortized logarithmic.

Throws: Nothing.

51.
```
template<typename KeyType, typename KeyValueCompare> 
  iterator find(const KeyType & key, KeyValueCompare comp);
```

Effects: Finds an iterator to the first element whose key is k or end() if that element does not exist.

Complexity: Amortized logarithmic.

Throws: Nothing.

52.
```
const_iterator find_dont_splay(const_reference value) const;
```

Effects: Finds a const_iterator to the first element whose key is k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

53.
```
template<typename KeyType, typename KeyValueCompare> 
  const_iterator 
  find_dont_splay(const KeyType & key, KeyValueCompare comp) const;
```

Effects: Finds a const_iterator to the first element whose key is k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

54.
```
std::pair< iterator, iterator > equal_range(const_reference value);
```

Effects: Finds a range containing all elements whose key is k or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Amortized logarithmic.

Throws: Nothing.

55.
```
template<typename KeyType, typename KeyValueCompare> 
  std::pair< iterator, iterator > 
  equal_range(const KeyType & key, KeyValueCompare comp);
```

Effects: Finds a range containing all elements whose key is k or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Amortized logarithmic.

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

56.
```
std::pair< const_iterator, const_iterator >
equal_range_dont_splay(const_reference value) const;
```

Effects: Finds a range containing all elements whose key is k or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: Nothing.

57.
```
template<typename KeyType, typename KeyValueCompare>
  std::pair< const_iterator, const_iterator >
  equal_range_dont_splay(const KeyType & key, KeyValueCompare comp) const;
```

Effects: Finds a range containing all elements whose key is k or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: Nothing.

58.
```
std::pair< iterator, iterator >
bounded_range(const_reference lower_value, const_reference upper_value,
              bool left_closed, bool right_closed);
```

Requires: 'lower_value' must not be greater than 'upper_value'. If 'lower_value' == 'upper_value', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key) if left_closed, upper_bound(lower_key) otherwise

second = upper_bound(upper_key) if right_closed, lower_bound(upper_key) otherwise

Complexity: Logarithmic.

Throws: If the predicate throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_value and upper_value.

59.
```
template<typename KeyType, typename KeyValueCompare>
  std::pair< iterator, iterator >
  bounded_range(const KeyType & lower_key, const KeyType & upper_key,
                KeyValueCompare comp, bool left_closed, bool right_closed);
```

Requires: KeyValueCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. 'lower_key' must not be greater than 'upper_key' according to 'comp'. If 'lower_key' == 'upper_key', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key, comp) if left_closed, upper_bound(lower_key, comp) otherwise

second = upper_bound(upper_key, comp) if right_closed, lower_bound(upper_key, comp) otherwise

Complexity: Logarithmic.

Throws: If "comp" throws.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_key and upper_key.

60.
```
std::pair< const_iterator, const_iterator >
bounded_range(const_reference lower_value, const_reference upper_value,
              bool left_closed, bool right_closed) const;
```

Requires: 'lower_value' must not be greater than 'upper_value'. If 'lower_value' == 'upper_value', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key) if left_closed, upper_bound(lower_key) otherwise

second = upper_bound(upper_key) if right_closed, lower_bound(upper_key) otherwise

Complexity: Logarithmic.

Throws: If the predicate throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_value and upper_value.

61.
```
template<typename KeyType, typename KeyValueCompare>
  std::pair< const_iterator, const_iterator >
  bounded_range(const KeyType & lower_key, const KeyType & upper_key,
                KeyValueCompare comp, bool left_closed, bool right_closed) const;
```

Requires: KeyValueCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. 'lower_key' must not be greater than 'upper_key' according to 'comp'. If 'lower_key' == 'upper_key', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key, comp) if left_closed, upper_bound(lower_key, comp) otherwise

second = upper_bound(upper_key, comp) if right_closed, lower_bound(upper_key, comp) otherwise

Complexity: Logarithmic.

Throws: If "comp" throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_key and upper_key.

62.
```
template<typename Cloner, typename Disposer>
  void clone_from(const splaytree & src, Cloner cloner, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw. Cloner should yield to nodes equivalent to the original nodes.

Effects: Erases all the elements from *this calling Disposer::operator()(pointer), clones all the elements from src calling Cloner::operator()(const_reference) and inserts them on *this. Copies the predicate from the source container.

If cloner throws, all cloned elements are unlinked and disposed calling Disposer::operator()(pointer).

Complexity: Linear to erased plus inserted elements.

Throws: If cloner throws or predicate copy assignment throws. Basic guarantee.

63.
```
pointer unlink_leftmost_without_rebalance();
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Unlinks the leftmost node from the tree.

Complexity: Average complexity is constant time.

Throws: Nothing.

Notes: This function breaks the tree and the tree can only be used for more unlink_leftmost_without_rebalance calls. This function is normally used to achieve a step by step controlled destruction of the tree.

64.
```
void splay_up(iterator i);
```

Requires: i must be a valid iterator of *this.

Effects: Rearranges the splay set so that the element pointed by i is placed as the root of the tree, improving future searches of this value.

Complexity: Amortized logarithmic.

Throws: Nothing.

65.
```
template<typename KeyType, typename KeyValueCompare> 
  iterator splay_down(const KeyType & key, KeyValueCompare comp);
```

Effects: Rearranges the splay set so that if *this stores an element with a key equivalent to value the element is placed as the root of the tree. If the element is not present returns the last node compared with the key. If the tree is empty, end() is returned.

Complexity: Amortized logarithmic.

Returns: An iterator to the new root of the tree, end() if the tree is empty.

Throws: If the comparison functor throws.

66.
```
iterator splay_down(const value_type & value);
```

Effects: Rearranges the splay set so that if *this stores an element with a key equivalent to value the element is placed as the root of the tree.

Complexity: Amortized logarithmic.

Returns: An iterator to the new root of the tree, end() if the tree is empty.

Throws: If the predicate throws.

67.
```
void replace_node(iterator replace_this, reference with_this);
```

Requires: replace_this must be a valid iterator of *this and with_this must not be inserted in any tree.

Effects: Replaces replace_this in its position in the tree with with_this. The tree does not need to be rebalanced.

Complexity: Constant.

Throws: Nothing.

Note: This function will break container ordering invariants if with_this is not equivalent to *replace_this according to the ordering rules. This function is faster than erasing and inserting the node, since no rebalancing or comparison is needed.

68.
```
iterator iterator_to(reference value);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Requires: value must be an lvalue and shall be in a set of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid iterator i belonging to the set that points to the value

Complexity: Constant.

Throws: Nothing.

69.
```
const_iterator iterator_to(const_reference value) const;
```

Requires: value must be an lvalue and shall be in a set of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid const_iterator i belonging to the set that points to the value

Complexity: Constant.

Throws: Nothing.

70.
```
void rebalance();
```

Effects: Rebalances the tree.

Throws: Nothing.

Complexity: Linear.

71.
```
iterator rebalance_subtree(iterator root);
```

Requires: old_root is a node of a tree.

Effects: Rebalances the subtree rooted at old_root.

Returns: The new root of the subtree.

Throws: Nothing.

Complexity: Linear to the elements in the subtree.

`splaytree` public static functions

1.
```
static splaytree & container_from_end_iterator(iterator end_iterator);
```

Precondition: end_iterator must be a valid end iterator of splaytree.

Effects: Returns a const reference to the splaytree associated to the end iterator

Throws: Nothing.

Complexity: Constant.

2.
```
static const splaytree &
container_from_end_iterator(const_iterator end_iterator);
```

Precondition: end_iterator must be a valid end const_iterator of splaytree.

Effects: Returns a const reference to the splaytree associated to the end iterator

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Constant.

3.
```
static splaytree & container_from_iterator(iterator it);
```

   Precondition: it must be a valid iterator of rbtree.

   Effects: Returns a const reference to the tree associated to the iterator

   Throws: Nothing.

   Complexity: Logarithmic.

4.
```
static const splaytree & container_from_iterator(const_iterator it);
```

   Precondition: it must be a valid end const_iterator of rbtree.

   Effects: Returns a const reference to the tree associated to the iterator

   Throws: Nothing.

   Complexity: Logarithmic.

5.
```
static iterator s_iterator_to(reference value);
```

   Requires: value must be an lvalue and shall be in a set of appropriate type. Otherwise the behavior is undefined.

   Effects: Returns: a valid iterator i belonging to the set that points to the value

   Complexity: Constant.

   Throws: Nothing.

   Note: This static function is available only if the *value traits* is stateless.

6.
```
static const_iterator s_iterator_to(const_reference value);
```

   Requires: value must be an lvalue and shall be in a set of appropriate type. Otherwise the behavior is undefined.

   Effects: Returns: a valid const_iterator i belonging to the set that points to the value

   Complexity: Constant.

   Throws: Nothing.

   Note: This static function is available only if the *value traits* is stateless.

7.
```
static void init_node(reference value);
```

   Requires: value shall not be in a tree.

   Effects: init_node puts the hook of a value in a well-known default state.

   Throws: Nothing.

   Complexity: Constant time.

   Note: This function puts the hook in the well-known default state used by auto_unlink and safe hooks.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

`splaytree` private static functions

1.
   ```
   static splaytree &
   priv_container_from_end_iterator(const const_iterator & end_iterator);
   ```

2.
   ```
   static splaytree & priv_container_from_iterator(const const_iterator & it);
   ```

Struct template make_splaytree

boost::intrusive::make_splaytree

Synopsis

```
// In header: <boost/intrusive/splaytree.hpp>

template<typename T, class... Options>
struct make_splaytree {
  // types
  typedef implementation_defined type;
};
```

Description

Helper metafunction to define a `splaytree` that yields to the same type when the same options (either explicitly or implicitly) are used.

Header <boost/intrusive/splaytree_algorithms.hpp>

```
namespace boost {
  namespace intrusive {
    template<typename NodeTraits> class splaytree_algorithms;
  }
}
```

Class template splaytree_algorithms

boost::intrusive::splaytree_algorithms

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/intrusive/splaytree_algorithms.hpp>

template<typename NodeTraits>
class splaytree_algorithms {
public:
  // types
  typedef NodeTraits::node                    node;
  typedef NodeTraits                          node_traits;
  typedef NodeTraits::node_ptr                node_ptr;
  typedef NodeTraits::const_node_ptr          const_node_ptr;
  typedef tree_algorithms::insert_commit_data insert_commit_data;

  // public static functions
  static node_ptr begin_node(const const_node_ptr &);
  static node_ptr end_node(const const_node_ptr &);
  static bool unique(const const_node_ptr &);
  static void unlink(const node_ptr &);
  static void swap_nodes(const node_ptr &, const node_ptr &);
  static void swap_nodes(const node_ptr &, const node_ptr &, const node_ptr &,
                         const node_ptr &);
  static void replace_node(const node_ptr &, const node_ptr &);
  static void replace_node(const node_ptr &, const node_ptr &,
                           const node_ptr &);
  static node_ptr next_node(const node_ptr &);
  static node_ptr prev_node(const node_ptr &);
  static void init(const node_ptr &);
  static void init_header(const node_ptr &);
  template<typename Disposer>
    static void clear_and_dispose(const node_ptr &, Disposer);
  static std::size_t count(const const_node_ptr &);
  static std::size_t size(const const_node_ptr &);
  static void swap_tree(const node_ptr &, const node_ptr &);
  static void insert_unique_commit(const node_ptr &, const node_ptr &,
                                   const insert_commit_data &);
  template<typename KeyType, typename KeyNodePtrCompare>
    static std::pair< node_ptr, bool >
    insert_unique_check(const node_ptr &, const KeyType &, KeyNodePtrCompare,
                        insert_commit_data &);
  template<typename KeyType, typename KeyNodePtrCompare>
    static std::pair< node_ptr, bool >
    insert_unique_check(const node_ptr &, const node_ptr &, const KeyType &,
                        KeyNodePtrCompare, insert_commit_data &);
  static bool is_header(const const_node_ptr &);
  template<typename KeyType, typename KeyNodePtrCompare>
    static node_ptr
    find(const const_node_ptr &, const KeyType &, KeyNodePtrCompare,
         bool = true);
  template<typename KeyType, typename KeyNodePtrCompare>
    static std::pair< node_ptr, node_ptr >
    equal_range(const const_node_ptr &, const KeyType &, KeyNodePtrCompare,
                bool = true);
  template<typename KeyType, typename KeyNodePtrCompare>
    static std::pair< node_ptr, node_ptr >
    bounded_range(const const_node_ptr &, const KeyType &, const KeyType &,
                  KeyNodePtrCompare, bool, bool, bool = true);
  template<typename KeyType, typename KeyNodePtrCompare>
    static node_ptr
    lower_bound(const const_node_ptr &, const KeyType &, KeyNodePtrCompare,
                bool = true);
  template<typename KeyType, typename KeyNodePtrCompare>
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
    static node_ptr
    upper_bound(const const_node_ptr &, const KeyType &, KeyNodePtrCompare,
                bool = true);
  template<typename NodePtrCompare>
    static node_ptr
    insert_equal(const node_ptr &, const node_ptr &, const node_ptr &,
                 NodePtrCompare);
  static node_ptr
  insert_before(const node_ptr &, const node_ptr &, const node_ptr &);
  static void push_back(const node_ptr &, const node_ptr &);
  static void push_front(const node_ptr &, const node_ptr &);
  template<typename NodePtrCompare>
    static node_ptr
    insert_equal_upper_bound(const node_ptr &, const node_ptr &,
                             NodePtrCompare);
  template<typename NodePtrCompare>
    static node_ptr
    insert_equal_lower_bound(const node_ptr &, const node_ptr &,
                             NodePtrCompare);
  template<typename Cloner, typename Disposer>
    static void clone(const const_node_ptr &, const node_ptr &, Cloner,
                      Disposer);
  static void erase(const node_ptr &, const node_ptr &, bool = true);
  static void splay_up(const node_ptr &, const node_ptr &);
  template<typename KeyType, typename KeyNodePtrCompare>
    static node_ptr
    splay_down(const node_ptr &, const KeyType &, KeyNodePtrCompare);
  static void rebalance(const node_ptr &);
  static node_ptr rebalance_subtree(const node_ptr &);
  static node_ptr get_header(const node_ptr &);
};
```

Description

A splay tree is an implementation of a binary search tree. The tree is self balancing using the splay algorithm as described in

"Self-Adjusting Binary Search Trees by Daniel Dominic Sleator and Robert Endre Tarjan AT&T Bell Laboratories, Murray Hill, NJ Journal of the ACM, Vol 32, no 3, July 1985, pp 652-686 splaytree_algorithms is configured with a NodeTraits class, which encapsulates the information about the node to be manipulated. NodeTraits must support the following interface:

Typedefs:

`node`: The type of the node that forms the circular list

`node_ptr`: A pointer to a node

`const_node_ptr`: A pointer to a const node

Static functions:

`static node_ptr get_parent(const_node_ptr n);`

`static void set_parent(node_ptr n, node_ptr parent);`

`static node_ptr get_left(const_node_ptr n);`

`static void set_left(node_ptr n, node_ptr left);`

`static node_ptr get_right(const_node_ptr n);`

`static void set_right(node_ptr n, node_ptr right);`

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

`splaytree_algorithms` public types

1. typedef tree_algorithms::insert_commit_data insert_commit_data;

   This type is the information that will be filled by insert_unique_check

`splaytree_algorithms` public static functions

1.
```
static node_ptr begin_node(const const_node_ptr & header);
```

2.
```
static node_ptr end_node(const const_node_ptr & header);
```

3.
```
static bool unique(const const_node_ptr & node);
```

   Requires: node is a node of the tree or an node initialized by init(...).

   Effects: Returns true if the node is initialized by init().

   Complexity: Constant time.

   Throws: Nothing.

4.
```
static void unlink(const node_ptr & node);
```

5.
```
static void swap_nodes(const node_ptr & node1, const node_ptr & node2);
```

   Requires: node1 and node2 can't be header nodes of two trees.

   Effects: Swaps two nodes. After the function node1 will be inserted in the position node2 before the function. node2 will be inserted in the position node1 had before the function.

   Complexity: Logarithmic.

   Throws: Nothing.

   Note: This function will break container ordering invariants if node1 and node2 are not equivalent according to the ordering rules.

   Experimental function

6.
```
static void swap_nodes(const node_ptr & node1, const node_ptr & header1,
                       const node_ptr & node2, const node_ptr & header2);
```

   Requires: node1 and node2 can't be header nodes of two trees with header header1 and header2.

   Effects: Swaps two nodes. After the function node1 will be inserted in the position node2 before the function. node2 will be inserted in the position node1 had before the function.

   Complexity: Constant.

   Throws: Nothing.

   Note: This function will break container ordering invariants if node1 and node2 are not equivalent according to the ordering rules.

   Experimental function

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

7.
```
static void replace_node(const node_ptr & node_to_be_replaced,
                         const node_ptr & new_node);
```

Requires: node_to_be_replaced must be inserted in a tree and new_node must not be inserted in a tree.

Effects: Replaces node_to_be_replaced in its position in the tree with new_node. The tree does not need to be rebalanced

Complexity: Logarithmic.

Throws: Nothing.

Note: This function will break container ordering invariants if new_node is not equivalent to node_to_be_replaced according to the ordering rules. This function is faster than erasing and inserting the node, since no rebalancing and comparison is needed.

Experimental function

8.
```
static void replace_node(const node_ptr & node_to_be_replaced,
                         const node_ptr & header, const node_ptr & new_node);
```

Requires: node_to_be_replaced must be inserted in a tree with header "header" and new_node must not be inserted in a tree.

Effects: Replaces node_to_be_replaced in its position in the tree with new_node. The tree does not need to be rebalanced

Complexity: Constant.

Throws: Nothing.

Note: This function will break container ordering invariants if new_node is not equivalent to node_to_be_replaced according to the ordering rules. This function is faster than erasing and inserting the node, since no rebalancing or comparison is needed.

Experimental function

9.
```
static node_ptr next_node(const node_ptr & p);
```

Requires: p is a node from the tree except the header.

Effects: Returns the next node of the tree.

Complexity: Average constant time.

Throws: Nothing.

10.
```
static node_ptr prev_node(const node_ptr & p);
```

Requires: p is a node from the tree except the leftmost node.

Effects: Returns the previous node of the tree.

Complexity: Average constant time.

Throws: Nothing.

11.
```
static void init(const node_ptr & node);
```

Requires: node must not be part of any tree.

Effects: After the function unique(node) == true.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Constant.

Throws: Nothing.

Nodes: If node is inserted in a tree, this function corrupts the tree.

12.
```
static void init_header(const node_ptr & header);
```

Requires: node must not be part of any tree.

Effects: Initializes the header to represent an empty tree. unique(header) == true.

Complexity: Constant.

Throws: Nothing.

Nodes: If node is inserted in a tree, this function corrupts the tree.

13.
```
template<typename Disposer> 
  static void clear_and_dispose(const node_ptr & header, Disposer disposer);
```

Requires: "disposer" must be an object function taking a node_ptr parameter and shouldn't throw.

Effects: Empties the target tree calling `void disposer::operator()(const node_ptr &)` for every node of the tree except the header.

Complexity: Linear to the number of element of the source tree plus the. number of elements of tree target tree when calling this function.

Throws: If cloner functor throws. If this happens target nodes are disposed.

14.
```
static std::size_t count(const const_node_ptr & node);
```

Requires: node is a node of the tree but it's not the header.

Effects: Returns the number of nodes of the subtree.

Complexity: Linear time.

Throws: Nothing.

15.
```
static std::size_t size(const const_node_ptr & header);
```

Requires: header is the header node of the tree.

Effects: Returns the number of nodes above the header.

Complexity: Linear time.

Throws: Nothing.

16.
```
static void swap_tree(const node_ptr & header1, const node_ptr & header2);
```

Requires: header1 and header2 must be the header nodes of two trees.

Effects: Swaps two trees. After the function header1 will contain links to the second tree and header2 will have links to the first tree.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Constant.

Throws: Nothing.

17.
```
static void insert_unique_commit(const node_ptr & header,
                                 const node_ptr & new_value,
                                 const insert_commit_data & commit_data);
```

Requires: "header" must be the header node of a tree. "commit_data" must have been obtained from a previous call to "insert_unique_check". No objects should have been inserted or erased from the set between the "insert_unique_check" that filled "commit_data" and the call to "insert_commit".

Effects: Inserts new_node in the set using the information obtained from the "commit_data" that a previous "insert_check" filled.

Complexity: Constant time.

Throws: Nothing.

Notes: This function has only sense if a "insert_unique_check" has been previously executed to fill "commit_data". No value should be inserted or erased between the "insert_check" and "insert_commit" calls.

18.
```
template<typename KeyType, typename KeyNodePtrCompare>
  static std::pair< node_ptr, bool >
  insert_unique_check(const node_ptr & header, const KeyType & key,
                      KeyNodePtrCompare comp,
                      insert_commit_data & commit_data);
```

Requires: "header" must be the header node of a tree. KeyNodePtrCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. NodePtrCompare compares KeyType with a node_ptr.

Effects: Checks if there is an equivalent node to "key" in the tree according to "comp" and obtains the needed information to realize a constant-time node insertion if there is no equivalent node.

Returns: If there is an equivalent value returns a pair containing a node_ptr to the already present node and false. If there is not equivalent key can be inserted returns true in the returned pair's boolean and fills "commit_data" that is meant to be used with the "insert_commit" function to achieve a constant-time insertion function.

Complexity: Average complexity is at most logarithmic.

Throws: If "comp" throws.

Notes: This function is used to improve performance when constructing a node is expensive and the user does not want to have two equivalent nodes in the tree: if there is an equivalent value the constructed object must be discarded. Many times, the part of the node that is used to impose the order is much cheaper to construct than the node and this function offers the possibility to use that part to check if the insertion will be successful.

If the check is successful, the user can construct the node and use "insert_commit" to insert the node in constant-time. This gives a total logarithmic complexity to the insertion: check(O(log(N)) + commit(O(1)).

"commit_data" remains valid for a subsequent "insert_unique_commit" only if no more objects are inserted or erased from the set.

19.
```
template<typename KeyType, typename KeyNodePtrCompare>
  static std::pair< node_ptr, bool >
  insert_unique_check(const node_ptr & header, const node_ptr & hint,
                      const KeyType & key, KeyNodePtrCompare comp,
                      insert_commit_data & commit_data);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

20.
```
static bool is_header(const const_node_ptr & p);
```

21.
```
template<typename KeyType, typename KeyNodePtrCompare> 
  static node_ptr 
  find(const const_node_ptr & header, const KeyType & key, 
       KeyNodePtrCompare comp, bool splay = true);
```

Requires: "header" must be the header node of a tree. KeyNodePtrCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. KeyNodePtrCompare can compare KeyType with tree's node_ptrs.

Effects: Returns an node_ptr to the element that is equivalent to "key" according to "comp" or "header" if that element does not exist.

Complexity: Logarithmic.

Throws: If "comp" throws.

22.
```
template<typename KeyType, typename KeyNodePtrCompare> 
  static std::pair< node_ptr, node_ptr > 
  equal_range(const const_node_ptr & header, const KeyType & key, 
              KeyNodePtrCompare comp, bool splay = true);
```

Requires: "header" must be the header node of a tree. KeyNodePtrCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. KeyNodePtrCompare can compare KeyType with tree's node_ptrs.

Effects: Returns an a pair of node_ptr delimiting a range containing all elements that are equivalent to "key" according to "comp" or an empty range that indicates the position where those elements would be if they there are no equivalent elements.

Complexity: Logarithmic.

Throws: If "comp" throws.

23.
```
template<typename KeyType, typename KeyNodePtrCompare> 
  static std::pair< node_ptr, node_ptr > 
  bounded_range(const const_node_ptr & header, const KeyType & lower_key, 
                const KeyType & upper_key, KeyNodePtrCompare comp, 
                bool left_closed, bool right_closed, bool splay = true);
```

Requires: "header" must be the header node of a tree. KeyNodePtrCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. KeyNodePtrCompare can compare KeyType with tree's node_ptrs. 'lower_key' must not be greater than 'upper_key' according to 'comp'. If 'lower_key' == 'upper_key', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key) if left_closed, upper_bound(lower_key) otherwise

second = upper_bound(upper_key) if right_closed, lower_bound(upper_key) otherwise

Complexity: Logarithmic.

Throws: If "comp" throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_key and upper_key.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

24.
```
template<typename KeyType, typename KeyNodePtrCompare> 
  static node_ptr 
  lower_bound(const const_node_ptr & header, const KeyType & key, 
              KeyNodePtrCompare comp, bool splay = true);
```

Requires: "header" must be the header node of a tree. KeyNodePtrCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. KeyNodePtrCompare can compare KeyType with tree's node_ptrs.

Effects: Returns an node_ptr to the first element that is not less than "key" according to "comp" or "header" if that element does not exist.

Complexity: Logarithmic.

Throws: If "comp" throws.

25.
```
template<typename KeyType, typename KeyNodePtrCompare> 
  static node_ptr 
  upper_bound(const const_node_ptr & header, const KeyType & key, 
              KeyNodePtrCompare comp, bool splay = true);
```

Requires: "header" must be the header node of a tree. KeyNodePtrCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. KeyNodePtrCompare can compare KeyType with tree's node_ptrs.

Effects: Returns an node_ptr to the first element that is greater than "key" according to "comp" or "header" if that element does not exist.

Complexity: Logarithmic.

Throws: If "comp" throws.

26.
```
template<typename NodePtrCompare> 
  static node_ptr 
  insert_equal(const node_ptr & header, const node_ptr & hint, 
               const node_ptr & new_node, NodePtrCompare comp);
```

Requires: "header" must be the header node of a tree. NodePtrCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. NodePtrCompare compares two node_ptrs. "hint" is node from the "header"'s tree.

Effects: Inserts new_node into the tree, using "hint" as a hint to where it will be inserted. If "hint" is the upper_bound the insertion takes constant time (two comparisons in the worst case).

Complexity: Logarithmic in general, but it is amortized constant time if new_node is inserted immediately before "hint".

Throws: If "comp" throws.

27.
```
static node_ptr 
insert_before(const node_ptr & header, const node_ptr & pos, 
              const node_ptr & new_node);
```

Requires: "header" must be the header node of a tree. "pos" must be a valid iterator or header (end) node. "pos" must be an iterator pointing to the successor to "new_node" once inserted according to the order of already inserted nodes. This function does not check "pos" and this precondition must be guaranteed by the caller.

Effects: Inserts new_node into the tree before "pos".

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Constant-time.

Throws: Nothing.

Note: If "pos" is not the successor of the newly inserted "new_node" tree invariants might be broken.

28.
```
static void push_back(const node_ptr & header, const node_ptr & new_node);
```

Requires: "header" must be the header node of a tree. "new_node" must be, according to the used ordering no less than the greatest inserted key.

Effects: Inserts new_node into the tree before "pos".

Complexity: Constant-time.

Throws: Nothing.

Note: If "new_node" is less than the greatest inserted key tree invariants are broken. This function is slightly faster than using "insert_before".

29.
```
static void push_front(const node_ptr & header, const node_ptr & new_node);
```

Requires: "header" must be the header node of a tree. "new_node" must be, according to the used ordering, no greater than the lowest inserted key.

Effects: Inserts new_node into the tree before "pos".

Complexity: Constant-time.

Throws: Nothing.

Note: If "new_node" is greater than the lowest inserted key tree invariants are broken. This function is slightly faster than using "insert_before".

30.
```
template<typename NodePtrCompare> 
  static node_ptr 
  insert_equal_upper_bound(const node_ptr & header, const node_ptr & new_node, 
                           NodePtrCompare comp);
```

Requires: "header" must be the header node of a tree. NodePtrCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. NodePtrCompare compares two node_ptrs.

Effects: Inserts new_node into the tree before the upper bound according to "comp".

Complexity: Average complexity for insert element is at most logarithmic.

Throws: If "comp" throws.

31.
```
template<typename NodePtrCompare> 
  static node_ptr 
  insert_equal_lower_bound(const node_ptr & header, const node_ptr & new_node, 
                           NodePtrCompare comp);
```

Requires: "header" must be the header node of a tree. NodePtrCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. NodePtrCompare compares two node_ptrs.

Effects: Inserts new_node into the tree before the lower bound according to "comp".

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Average complexity for insert element is at most logarithmic.

Throws: If "comp" throws.

32.
```
template<typename Cloner, typename Disposer>
  static void clone(const const_node_ptr & source_header,
                    const node_ptr & target_header, Cloner cloner,
                    Disposer disposer);
```

Requires: "cloner" must be a function object taking a node_ptr and returning a new cloned node of it. "disposer" must take a node_ptr and shouldn't throw.

Effects: First empties target tree calling `void disposer::operator()(const node_ptr &)` for every node of the tree except the header.

Then, duplicates the entire tree pointed by "source_header" cloning each source node with `node_ptr Cloner::operator()(const node_ptr &)` to obtain the nodes of the target tree. If "cloner" throws, the cloned target nodes are disposed using `void disposer(const node_ptr &)`.

Complexity: Linear to the number of element of the source tree plus the. number of elements of tree target tree when calling this function.

Throws: If cloner functor throws. If this happens target nodes are disposed.

33.
```
static void erase(const node_ptr & header, const node_ptr & z,
                  bool splay = true);
```

34.
```
static void splay_up(const node_ptr & node, const node_ptr & header);
```

35.
```
template<typename KeyType, typename KeyNodePtrCompare>
  static node_ptr
  splay_down(const node_ptr & header, const KeyType & key,
             KeyNodePtrCompare comp);
```

36.
```
static void rebalance(const node_ptr & header);
```

Requires: header must be the header of a tree.

Effects: Rebalances the tree.

Throws: Nothing.

Complexity: Linear.

37.
```
static node_ptr rebalance_subtree(const node_ptr & old_root);
```

Requires: old_root is a node of a tree.

Effects: Rebalances the subtree rooted at old_root.

Returns: The new root of the subtree.

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Linear.

38.
```
static node_ptr get_header(const node_ptr & n);
```

Requires: "n" must be a node inserted in a tree.

Effects: Returns a pointer to the header node of the tree.

Complexity: Logarithmic.

Throws: Nothing.

Header <boost/intrusive/treap.hpp>

```
namespace boost {
  namespace intrusive {
    template<typename T, class... Options> class treap;

    template<typename T, class... Options> struct make_trie;
    template<typename T, class... Options>
      bool operator<(const treap< T, Options...> & x,
                     const treap< T, Options...> & y);
    template<typename T, class... Options>
      bool operator==(const treap< T, Options...> & x,
                      const treap< T, Options...> & y);
    template<typename T, class... Options>
      bool operator!=(const treap< T, Options...> & x,
                      const treap< T, Options...> & y);
    template<typename T, class... Options>
      bool operator>(const treap< T, Options...> & x,
                     const treap< T, Options...> & y);
    template<typename T, class... Options>
      bool operator<=(const treap< T, Options...> & x,
                      const treap< T, Options...> & y);
    template<typename T, class... Options>
      bool operator>=(const treap< T, Options...> & x,
                      const treap< T, Options...> & y);
    template<typename T, class... Options>
      void swap(treap< T, Options...> & x, treap< T, Options...> & y);
  }
}
```

Class template treap

boost::intrusive::treap

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/intrusive/treap.hpp>

template<typename T, class... Options>
class treap {
public:
  // types
  typedef Config::value_traits                                                 value_traits; ↵

  typedef real_value_traits::pointer                                           pointer;      ↵

  typedef real_value_traits::const_pointer                                     const_pointer; ↵

  typedef pointer_traits< pointer >::element_type                              value_type;   ↵

  typedef pointer_traits< pointer >::reference                                 reference;    ↵

  typedef pointer_traits< const_pointer >::reference                             const_refer↵
ence;
  typedef pointer_traits< pointer >::difference_type                        difference_type; ↵

  typedef value_type                                                           key_type;     ↵

  typedef Config::size_type                                                    size_type;    ↵

  typedef Config::compare                                                      value_compare; ↵

  typedef Config::priority_compare                                               priority_com↵
pare;
  typedef value_compare                                                        key_compare;  ↵

  typedef tree_iterator< treap, false >                                        iterator;     ↵

  typedef tree_iterator< treap, true >                                           const_iterat↵
or;
  typedef unspecified                                                          reverse_iterat↵
or;
  typedef unspecified                                                     const_reverse_iter↵
ator;
  typedef real_value_traits::node_traits                                       node_traits;  ↵

  typedef node_traits::node                                                    node;         ↵

  typedef pointer_traits< pointer >::template rebind_pointer< node >::type       node_ptr;   ↵

  typedef pointer_traits< pointer >::template rebind_pointer< const node >::type const_node_ptr; ↵

  typedef treap_algorithms< node_traits >                                   node_algorithms; ↵

  typedef node_algorithms::insert_commit_data                                    insert_com↵
mit_data;

  // construct/copy/destruct
  explicit treap(const value_compare & = value_compare(), 
                 const priority_compare & = priority_compare(), 
                 const value_traits & = value_traits());
  template<typename Iterator> 
    treap(bool, Iterator, Iterator, const value_compare & = value_compare(), 
          const priority_compare & = priority_compare(), 
          const value_traits & = value_traits());
  treap(BOOST_RV_REF(treap));
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
treap& operator=(BOOST_RV_REF(treap));
~treap();

// public member functions
const real_value_traits & get_real_value_traits() const;
real_value_traits & get_real_value_traits();
iterator begin();
const_iterator begin() const;
const_iterator cbegin() const;
iterator end();
const_iterator end() const;
const_iterator cend() const;
iterator top();
const_iterator top() const;
const_iterator ctop() const;
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
const_reverse_iterator crbegin() const;
reverse_iterator rend();
const_reverse_iterator rend() const;
const_reverse_iterator crend() const;
reverse_iterator rtop();
const_reverse_iterator rtop() const;
const_reverse_iterator crtop() const;
value_compare value_comp() const;
priority_compare priority_comp() const;
bool empty() const;
size_type size() const;
void swap(treap &);
iterator insert_equal(reference);
iterator insert_equal(const_iterator, reference);
template<typename Iterator> void insert_equal(Iterator, Iterator);
std::pair< iterator, bool > insert_unique(reference);
iterator insert_unique(const_iterator, reference);
template<typename Iterator> void insert_unique(Iterator, Iterator);
template<typename KeyType, typename KeyValueCompare,
         typename KeyValuePrioCompare>
  std::pair< iterator, bool >
  insert_unique_check(const KeyType &, KeyValueCompare, KeyValuePrioCompare,
                      insert_commit_data &);
template<typename KeyType, typename KeyValueCompare,
         typename KeyValuePrioCompare>
  std::pair< iterator, bool >
  insert_unique_check(const_iterator, const KeyType &, KeyValueCompare,
                      KeyValuePrioCompare, insert_commit_data &);
iterator insert_unique_commit(reference, const insert_commit_data &);
iterator insert_before(const_iterator, reference);
void push_back(reference);
void push_front(reference);
iterator erase(const_iterator);
iterator erase(const_iterator, const_iterator);
size_type erase(const_reference);
template<typename KeyType, typename KeyValueCompare>
  size_type erase(const KeyType &, KeyValueCompare);
template<typename Disposer>
  iterator erase_and_dispose(const_iterator, Disposer);
template<typename Disposer>
  iterator erase_and_dispose(const_iterator, const_iterator, Disposer);
template<typename Disposer>
  size_type erase_and_dispose(const_reference, Disposer);
template<typename KeyType, typename KeyValueCompare, typename Disposer>
  size_type erase_and_dispose(const KeyType &, KeyValueCompare, Disposer);
void clear();
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
template<typename Disposer> void clear_and_dispose(Disposer);
size_type count(const_reference) const;
template<typename KeyType, typename KeyValueCompare>
  size_type count(const KeyType &, KeyValueCompare) const;
iterator lower_bound(const_reference);
const_iterator lower_bound(const_reference) const;
template<typename KeyType, typename KeyValueCompare>
  iterator lower_bound(const KeyType &, KeyValueCompare);
template<typename KeyType, typename KeyValueCompare>
  const_iterator lower_bound(const KeyType &, KeyValueCompare) const;
iterator upper_bound(const_reference);
template<typename KeyType, typename KeyValueCompare>
  iterator upper_bound(const KeyType &, KeyValueCompare);
const_iterator upper_bound(const_reference) const;
template<typename KeyType, typename KeyValueCompare>
  const_iterator upper_bound(const KeyType &, KeyValueCompare) const;
iterator find(const_reference);
template<typename KeyType, typename KeyValueCompare>
  iterator find(const KeyType &, KeyValueCompare);
const_iterator find(const_reference) const;
template<typename KeyType, typename KeyValueCompare>
  const_iterator find(const KeyType &, KeyValueCompare) const;
std::pair< iterator, iterator > equal_range(const_reference);
template<typename KeyType, typename KeyValueCompare>
  std::pair< iterator, iterator >
  equal_range(const KeyType &, KeyValueCompare);
std::pair< const_iterator, const_iterator >
equal_range(const_reference) const;
template<typename KeyType, typename KeyValueCompare>
  std::pair< const_iterator, const_iterator >
  equal_range(const KeyType &, KeyValueCompare) const;
std::pair< iterator, iterator >
bounded_range(const_reference, const_reference, bool, bool);
template<typename KeyType, typename KeyValueCompare>
  std::pair< iterator, iterator >
  bounded_range(const KeyType &, const KeyType &, KeyValueCompare, bool,
                bool);
std::pair< const_iterator, const_iterator >
bounded_range(const_reference, const_reference, bool, bool) const;
template<typename KeyType, typename KeyValueCompare>
  std::pair< const_iterator, const_iterator >
  bounded_range(const KeyType &, const KeyType &, KeyValueCompare, bool,
                bool) const;
template<typename Cloner, typename Disposer>
  void clone_from(const treap &, Cloner, Disposer);
pointer unlink_leftmost_without_rebalance();
void replace_node(iterator, reference);
iterator iterator_to(reference);
const_iterator iterator_to(const_reference) const;

// public static functions
static treap & container_from_end_iterator(iterator);
static const treap & container_from_end_iterator(const_iterator);
static treap & container_from_iterator(iterator);
static const treap & container_from_iterator(const_iterator);
static iterator s_iterator_to(reference);
static const_iterator s_iterator_to(const_reference);
static void init_node(reference);

// private static functions
static treap & priv_container_from_end_iterator(const const_iterator &);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
  static treap & priv_container_from_iterator(const const_iterator &);

  // public data members
  static const bool constant_time_size;
  static const bool stateful_value_traits;
};
```

Description

The class template treap is an intrusive treap container that is used to construct intrusive set and multiset containers. The no-throw guarantee holds only, if the value_compare object and priority_compare object don't throw.

The template parameter `T` is the type to be managed by the container. The user can specify additional options and if no options are provided default options are used.

The container supports the following options: `base_hook<>/member_hook<>/value_traits<>`, `constant_time_size<>`, `size_type<>`, `compare<>` and `priority_compare<>`

`treap` public construct/copy/destruct

1.
```
explicit treap(const value_compare & cmp = value_compare(),
               const priority_compare & pcmp = priority_compare(),
               const value_traits & v_traits = value_traits());
```

   Effects: Constructs an empty treap.

   Complexity: Constant.

   Throws: If value_traits::node_traits::node constructor throws (this does not happen with predefined Boost.Intrusive hooks) or the copy constructor of the value_compare/priority_compare objects throw. Basic guarantee.

2.
```
template<typename Iterator>
  treap(bool unique, Iterator b, Iterator e,
        const value_compare & cmp = value_compare(),
        const priority_compare & pcmp = priority_compare(),
        const value_traits & v_traits = value_traits());
```

   Requires: Dereferencing iterator must yield an lvalue of type value_type. cmp must be a comparison function that induces a strict weak ordering.

   Effects: Constructs an empty treap and inserts elements from [b, e).

   Complexity: Linear in N if [b, e) is already sorted using comp and otherwise N * log N, where N is the distance between first and last.

   Throws: If value_traits::node_traits::node constructor throws (this does not happen with predefined Boost.Intrusive hooks) or the copy constructor/operator() of the value_compare/priority_compare objects throw. Basic guarantee.

3.
```
treap(BOOST_RV_REF(treap) x);
```

   Effects: to-do

4.
```
treap& operator=(BOOST_RV_REF(treap) x);
```

   Effects: to-do

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

5.
```
~treap();
```

Effects: Detaches all elements from this. The objects in the set are not deleted (i.e. no destructors are called), but the nodes according to the `value_traits` template parameter are reinitialized and thus can be reused.

Complexity: Linear to elements contained in *this if constant-time size option is disabled. Constant time otherwise.

Throws: Nothing.

`treap` public member functions

1.
```
const real_value_traits & get_real_value_traits() const;
```

2.
```
real_value_traits & get_real_value_traits();
```

3.
```
iterator begin();
```

Effects: Returns an iterator pointing to the beginning of the treap.

Complexity: Constant.

Throws: Nothing.

4.
```
const_iterator begin() const;
```

Effects: Returns a const_iterator pointing to the beginning of the treap.

Complexity: Constant.

Throws: Nothing.

5.
```
const_iterator cbegin() const;
```

Effects: Returns a const_iterator pointing to the beginning of the treap.

Complexity: Constant.

Throws: Nothing.

6.
```
iterator end();
```

Effects: Returns an iterator pointing to the end of the treap.

Complexity: Constant.

Throws: Nothing.

7.
```
const_iterator end() const;
```

Effects: Returns a const_iterator pointing to the end of the treap.

Complexity: Constant.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: Nothing.

8.
```
const_iterator cend() const;
```

Effects: Returns a const_iterator pointing to the end of the treap.

Complexity: Constant.

Throws: Nothing.

9.
```
iterator top();
```

Effects: Returns an iterator pointing to the highest priority object of the treap.

Complexity: Constant.

Throws: Nothing.

10.
```
const_iterator top() const;
```

Effects: Returns a const_iterator pointing to the highest priority object of the treap..

Complexity: Constant.

Throws: Nothing.

11.
```
const_iterator ctop() const;
```

Effects: Returns a const_iterator pointing to the highest priority object of the treap..

Complexity: Constant.

Throws: Nothing.

12.
```
reverse_iterator rbegin();
```

Effects: Returns a reverse_iterator pointing to the beginning of the reversed treap.

Complexity: Constant.

Throws: Nothing.

13.
```
const_reverse_iterator rbegin() const;
```

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed treap.

Complexity: Constant.

Throws: Nothing.

14.
```
const_reverse_iterator crbegin() const;
```

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed treap.

Complexity: Constant.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: Nothing.

15.
```
reverse_iterator rend();
```

Effects: Returns a reverse_iterator pointing to the end of the reversed treap.

Complexity: Constant.

Throws: Nothing.

16.
```
const_reverse_iterator rend() const;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed treap.

Complexity: Constant.

Throws: Nothing.

17.
```
const_reverse_iterator crend() const;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed treap.

Complexity: Constant.

Throws: Nothing.

18.
```
reverse_iterator rtop();
```

Effects: Returns a reverse_iterator pointing to the highest priority object of the reversed treap.

Complexity: Constant.

Throws: Nothing.

19.
```
const_reverse_iterator rtop() const;
```

Effects: Returns a const_reverse_iterator pointing to the highest priority objec of the reversed treap.

Complexity: Constant.

Throws: Nothing.

20.
```
const_reverse_iterator crtop() const;
```

Effects: Returns a const_reverse_iterator pointing to the highest priority object of the reversed treap.

Complexity: Constant.

Throws: Nothing.

21.
```
value_compare value_comp() const;
```

Effects: Returns the value_compare object used by the treap.

Complexity: Constant.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: If value_compare copy-constructor throws.

22.
```
priority_compare priority_comp() const;
```

Effects: Returns the priority_compare object used by the treap.

Complexity: Constant.

Throws: If priority_compare copy-constructor throws.

23.
```
bool empty() const;
```

Effects: Returns true if the container is empty.

Complexity: Constant.

Throws: Nothing.

24.
```
size_type size() const;
```

Effects: Returns the number of elements stored in the treap.

Complexity: Linear to elements contained in *this if constant-time size option is disabled. Constant time otherwise.

Throws: Nothing.

25.
```
void swap(treap & other);
```

Effects: Swaps the contents of two treaps.

Complexity: Constant.

Throws: If the comparison functor's swap call throws.

26.
```
iterator insert_equal(reference value);
```

Requires: value must be an lvalue

Effects: Inserts value into the treap before the upper bound.

Complexity: Average complexity for insert element is at most logarithmic.

Throws: If the internal value_compare or priority_compare functions throw. Strong guarantee.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

27.
```
iterator insert_equal(const_iterator hint, reference value);
```

Requires: value must be an lvalue, and "hint" must be a valid iterator.

Effects: Inserts x into the treap, using "hint" as a hint to where it will be inserted. If "hint" is the upper_bound the insertion takes constant time (two comparisons in the worst case)

Complexity: Logarithmic in general, but it is amortized constant time if t is inserted immediately before hint.

Throws: If the internal value_compare or priority_compare functions throw. Strong guarantee.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

28.
```
template<typename Iterator> void insert_equal(Iterator b, Iterator e);
```

Requires: Dereferencing iterator must yield an lvalue of type value_type.

Effects: Inserts a each element of a range into the treap before the upper bound of the key of each element.

Complexity: Insert range is in general O(N * log(N)), where N is the size of the range. However, it is linear in N if the range is already sorted by value_comp().

Throws: If the internal value_compare or priority_compare functions throw. Strong guarantee.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

29.
```
std::pair< iterator, bool > insert_unique(reference value);
```

Requires: value must be an lvalue

Effects: Inserts value into the treap if the value is not already present.

Complexity: Average complexity for insert element is at most logarithmic.

Throws: If the internal value_compare or priority_compare functions throw. Strong guarantee.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

30.
```
iterator insert_unique(const_iterator hint, reference value);
```

Requires: value must be an lvalue, and "hint" must be a valid iterator

Effects: Tries to insert x into the treap, using "hint" as a hint to where it will be inserted.

Complexity: Logarithmic in general, but it is amortized constant time (two comparisons in the worst case) if t is inserted immediately before hint.

Throws: If the internal value_compare or priority_compare functions throw. Strong guarantee.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

31.
```
template<typename Iterator> void insert_unique(Iterator b, Iterator e);
```

Requires: Dereferencing iterator must yield an lvalue of type value_type.

Effects: Tries to insert each element of a range into the treap.

Complexity: Insert range is in general O(N * log(N)), where N is the size of the range. However, it is linear in N if the range is already sorted by value_comp().

Throws: If the internal value_compare or priority_compare functions throw. Strong guarantee.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

32.
```
template<typename KeyType, typename KeyValueCompare, 
         typename KeyValuePrioCompare> 
  std::pair< iterator, bool > 
  insert_unique_check(const KeyType & key, KeyValueCompare key_value_comp, 
                      KeyValuePrioCompare key_value_pcomp, 
                      insert_commit_data & commit_data);
```

Requires: key_value_comp must be a comparison function that induces the same strict weak ordering as value_compare. key_value_pcomp must be a comparison function that induces the same strict weak ordering as priority_compare. The difference is that key_value_pcomp and key_value_comp compare an arbitrary key with the contained values.

Effects: Checks if a value can be inserted in the container, using a user provided key instead of the value itself.

Returns: If there is an equivalent value returns a pair containing an iterator to the already present value and false. If the value can be inserted returns true in the returned pair boolean and fills "commit_data" that is meant to be used with the "insert_commit" function.

Complexity: Average complexity is at most logarithmic.

Throws: If the key_value_comp or key_value_pcomp ordering functions throw. Strong guarantee.

Notes: This function is used to improve performance when constructing a value_type is expensive: if there is an equivalent value the constructed object must be discarded. Many times, the part of the node that is used to impose the order is much cheaper to construct than the value_type and this function offers the possibility to use that part to check if the insertion will be successful.

If the check is successful, the user can construct the value_type and use "insert_commit" to insert the object in constant-time. This gives a total logarithmic complexity to the insertion: check(O(log(N)) + commit(O(1)).

"commit_data" remains valid for a subsequent "insert_commit" only if no more objects are inserted or erased from the container.

33.
```
template<typename KeyType, typename KeyValueCompare, 
         typename KeyValuePrioCompare> 
  std::pair< iterator, bool > 
  insert_unique_check(const_iterator hint, const KeyType & key, 
                      KeyValueCompare key_value_comp, 
                      KeyValuePrioCompare key_value_pcomp, 
                      insert_commit_data & commit_data);
```

Requires: key_value_comp must be a comparison function that induces the same strict weak ordering as value_compare. key_value_pcomp must be a comparison function that induces the same strict weak ordering as priority_compare. The difference is that key_value_pcomp and key_value_comp compare an arbitrary key with the contained values.

Effects: Checks if a value can be inserted in the container, using a user provided key instead of the value itself, using "hint" as a hint to where it will be inserted.

Returns: If there is an equivalent value returns a pair containing an iterator to the already present value and false. If the value can be inserted returns true in the returned pair boolean and fills "commit_data" that is meant to be used with the "insert_commit" function.

Complexity: Logarithmic in general, but it's amortized constant time if t is inserted immediately before hint.

Throws: If the key_value_comp or key_value_pcomp ordering functions throw. Strong guarantee.

Notes: This function is used to improve performance when constructing a value_type is expensive: if there is an equivalent value the constructed object must be discarded. Many times, the part of the constructing that is used to impose the order is much cheaper to construct than the value_type and this function offers the possibility to use that key to check if the insertion will be successful.

If the check is successful, the user can construct the value_type and use "insert_commit" to insert the object in constant-time. This can give a total constant-time complexity to the insertion: check(O(1)) + commit(O(1)).

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

"commit_data" remains valid for a subsequent "insert_commit" only if no more objects are inserted or erased from the container.

34.
```
iterator insert_unique_commit(reference value,
                              const insert_commit_data & commit_data);
```

Requires: value must be an lvalue of type value_type. commit_data must have been obtained from a previous call to "insert_check". No objects should have been inserted or erased from the container between the "insert_check" that filled "commit_data" and the call to "insert_commit".

Effects: Inserts the value in the `avl_set` using the information obtained from the "commit_data" that a previous "insert_check" filled.

Returns: An iterator to the newly inserted object.

Complexity: Constant time.

Throws: Nothing

Notes: This function has only sense if a "insert_check" has been previously executed to fill "commit_data". No value should be inserted or erased between the "insert_check" and "insert_commit" calls.

35.
```
iterator insert_before(const_iterator pos, reference value);
```

Requires: value must be an lvalue, "pos" must be a valid iterator (or end) and must be the succesor of value once inserted according to the predicate

Effects: Inserts x into the treap before "pos".

Complexity: Constant time.

Throws: If the internal priority_compare function throws. Strong guarantee.

Note: This function does not check preconditions so if "pos" is not the successor of "value" treap ordering invariant will be broken. This is a low-level function to be used only for performance reasons by advanced users.

36.
```
void push_back(reference value);
```

Requires: value must be an lvalue, and it must be no less than the greatest inserted key

Effects: Inserts x into the treap in the last position.

Complexity: Constant time.

Throws: If the internal priority_compare function throws. Strong guarantee.

Note: This function does not check preconditions so if value is less than the greatest inserted key treap ordering invariant will be broken. This function is slightly more efficient than using "insert_before". This is a low-level function to be used only for performance reasons by advanced users.

37.
```
void push_front(reference value);
```

Requires: value must be an lvalue, and it must be no greater than the minimum inserted key

Effects: Inserts x into the treap in the first position.

Complexity: Constant time.

Throws: If the internal priority_compare function throws. Strong guarantee.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Note: This function does not check preconditions so if value is greater than the minimum inserted key treap ordering invariant will be broken. This function is slightly more efficient than using "insert_before". This is a low-level function to be used only for performance reasons by advanced users.

38.
```
iterator erase(const_iterator i);
```

Effects: Erases the element pointed to by pos.

Complexity: Average complexity for erase element is constant time.

Throws: if the internal priority_compare function throws. Strong guarantee.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

39.
```
iterator erase(const_iterator b, const_iterator e);
```

Effects: Erases the range pointed to by b end e.

Complexity: Average complexity for erase range is at most O(log(size() + N)), where N is the number of elements in the range.

Throws: if the internal priority_compare function throws. Strong guarantee.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

40.
```
size_type erase(const_reference value);
```

Effects: Erases all the elements with the given value.

Returns: The number of erased elements.

Complexity: O(log(size() + N).

Throws: if the internal priority_compare function throws. Strong guarantee.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

41.
```
template<typename KeyType, typename KeyValueCompare>
  size_type erase(const KeyType & key, KeyValueCompare comp);
```

Effects: Erases all the elements with the given key. according to the comparison functor "comp".

Returns: The number of erased elements.

Complexity: O(log(size() + N).

Throws: if the internal priority_compare function throws. Equivalent guarantee to *while(beg != end) erase(beg++);*

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

42.
```
template<typename Disposer>
  iterator erase_and_dispose(const_iterator i, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases the element pointed to by pos. Disposer::operator()(pointer) is called for the removed element.

Complexity: Average complexity for erase element is constant time.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: if the internal priority_compare function throws. Strong guarantee.

Note: Invalidates the iterators to the erased elements.

43.
```
template<typename Disposer>
  iterator erase_and_dispose(const_iterator b, const_iterator e,
                             Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases the range pointed to by b end e. Disposer::operator()(pointer) is called for the removed elements.

Complexity: Average complexity for erase range is at most O(log(size() + N)), where N is the number of elements in the range.

Throws: if the internal priority_compare function throws. Strong guarantee.

Note: Invalidates the iterators to the erased elements.

44.
```
template<typename Disposer>
  size_type erase_and_dispose(const_reference value, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases all the elements with the given value. Disposer::operator()(pointer) is called for the removed elements.

Returns: The number of erased elements.

Complexity: O(log(size() + N).

Throws: if the priority_compare function throws then weak guarantee and heap invariants are broken. The safest thing would be to clear or destroy the container.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

45.
```
template<typename KeyType, typename KeyValueCompare, typename Disposer>
  size_type erase_and_dispose(const KeyType & key, KeyValueCompare comp,
                              Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases all the elements with the given key. according to the comparison functor "comp". Disposer::operator()(pointer) is called for the removed elements.

Returns: The number of erased elements.

Complexity: O(log(size() + N).

Throws: if the priority_compare function throws then weak guarantee and heap invariants are broken. The safest thing would be to clear or destroy the container.

Note: Invalidates the iterators to the erased elements.

46.
```
void clear();
```

Effects: Erases all of the elements.

Complexity: Linear to the number of elements on the container. if it's a safe-mode or auto-unlink value_type. Constant time otherwise.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

47.
```
template<typename Disposer> void clear_and_dispose(Disposer disposer);
```

Effects: Erases all of the elements calling disposer(p) for each node to be erased. **Complexity**: Average complexity for is at most O(log(size() + N)), where N is the number of elements in the container.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. Calls N times to disposer functor.

48.
```
size_type count(const_reference value) const;
```

Effects: Returns the number of contained elements with the given value

Complexity: Logarithmic to the number of elements contained plus lineal to number of objects with the given value.

Throws: Nothing.

49.
```
template<typename KeyType, typename KeyValueCompare>
  size_type count(const KeyType & key, KeyValueCompare comp) const;
```

Effects: Returns the number of contained elements with the given key

Complexity: Logarithmic to the number of elements contained plus lineal to number of objects with the given key.

Throws: Nothing.

50.
```
iterator lower_bound(const_reference value);
```

Effects: Returns an iterator to the first element whose key is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

51.
```
const_iterator lower_bound(const_reference value) const;
```

Effects: Returns an iterator to the first element whose key is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

52.
```
template<typename KeyType, typename KeyValueCompare>
  iterator lower_bound(const KeyType & key, KeyValueCompare comp);
```

Effects: Returns an iterator to the first element whose key is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

53.
```
template<typename KeyType, typename KeyValueCompare>
  const_iterator lower_bound(const KeyType & key, KeyValueCompare comp) const;
```

Effects: Returns a const iterator to the first element whose key is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

54.
```
iterator upper_bound(const_reference value);
```

Effects: Returns an iterator to the first element whose key is greater than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

55.
```
template<typename KeyType, typename KeyValueCompare>
  iterator upper_bound(const KeyType & key, KeyValueCompare comp);
```

Effects: Returns an iterator to the first element whose key is greater than k according to comp or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

56.
```
const_iterator upper_bound(const_reference value) const;
```

Effects: Returns an iterator to the first element whose key is greater than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

57.
```
template<typename KeyType, typename KeyValueCompare>
  const_iterator upper_bound(const KeyType & key, KeyValueCompare comp) const;
```

Effects: Returns an iterator to the first element whose key is greater than k according to comp or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

58.
```
iterator find(const_reference value);
```

Effects: Finds an iterator to the first element whose key is k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

59.
```
template<typename KeyType, typename KeyValueCompare>
  iterator find(const KeyType & key, KeyValueCompare comp);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Finds an iterator to the first element whose key is k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

60.
```
const_iterator find(const_reference value) const;
```

Effects: Finds a const_iterator to the first element whose key is k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

61.
```
template<typename KeyType, typename KeyValueCompare> 
  const_iterator find(const KeyType & key, KeyValueCompare comp) const;
```

Effects: Finds a const_iterator to the first element whose key is k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: Nothing.

62.
```
std::pair< iterator, iterator > equal_range(const_reference value);
```

Effects: Finds a range containing all elements whose key is k or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: Nothing.

63.
```
template<typename KeyType, typename KeyValueCompare> 
  std::pair< iterator, iterator > 
  equal_range(const KeyType & key, KeyValueCompare comp);
```

Effects: Finds a range containing all elements whose key is k or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: Nothing.

64.
```
std::pair< const_iterator, const_iterator > 
equal_range(const_reference value) const;
```

Effects: Finds a range containing all elements whose key is k or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: Nothing.

65.
```
template<typename KeyType, typename KeyValueCompare> 
  std::pair< const_iterator, const_iterator > 
  equal_range(const KeyType & key, KeyValueCompare comp) const;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Finds a range containing all elements whose key is k or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: Nothing.

66.
```
std::pair< iterator, iterator >
bounded_range(const_reference lower_value, const_reference upper_value,
              bool left_closed, bool right_closed);
```

Requires: 'lower_value' must not be greater than 'upper_value'. If 'lower_value' == 'upper_value', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key) if left_closed, upper_bound(lower_key) otherwise

second = upper_bound(upper_key) if right_closed, lower_bound(upper_key) otherwise

Complexity: Logarithmic.

Throws: If the predicate throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_value and upper_value.

67.
```
template<typename KeyType, typename KeyValueCompare>
  std::pair< iterator, iterator >
  bounded_range(const KeyType & lower_key, const KeyType & upper_key,
                KeyValueCompare comp, bool left_closed, bool right_closed);
```

Requires: KeyValueCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. 'lower_key' must not be greater than 'upper_key' according to 'comp'. If 'lower_key' == 'upper_key', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key, comp) if left_closed, upper_bound(lower_key, comp) otherwise

second = upper_bound(upper_key, comp) if right_closed, lower_bound(upper_key, comp) otherwise

Complexity: Logarithmic.

Throws: If "comp" throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_key and upper_key.

68.
```
std::pair< const_iterator, const_iterator >
bounded_range(const_reference lower_value, const_reference upper_value,
              bool left_closed, bool right_closed) const;
```

Requires: 'lower_value' must not be greater than 'upper_value'. If 'lower_value' == 'upper_value', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key) if left_closed, upper_bound(lower_key) otherwise

second = upper_bound(upper_key) if right_closed, lower_bound(upper_key) otherwise

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Logarithmic.

Throws: If the predicate throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_value and upper_value.

69.
```
template<typename KeyType, typename KeyValueCompare> 
  std::pair< const_iterator, const_iterator > 
  bounded_range(const KeyType & lower_key, const KeyType & upper_key, 
                KeyValueCompare comp, bool left_closed, bool right_closed) const;
```

Requires: KeyValueCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. 'lower_key' must not be greater than 'upper_key' according to 'comp'. If 'lower_key' == 'upper_key', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key, comp) if left_closed, upper_bound(lower_key, comp) otherwise

second = upper_bound(upper_key, comp) if right_closed, lower_bound(upper_key, comp) otherwise

Complexity: Logarithmic.

Throws: If "comp" throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_key and upper_key.

70.
```
template<typename Cloner, typename Disposer> 
  void clone_from(const treap & src, Cloner cloner, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw. Cloner should yield to nodes equivalent to the original nodes.

Effects: Erases all the elements from *this calling Disposer::operator()(pointer), clones all the elements from src calling Cloner::operator()(const_reference) and inserts them on *this. Copies the predicate from the source container.

If cloner throws, all cloned elements are unlinked and disposed calling Disposer::operator()(pointer).

Complexity: Linear to erased plus inserted elements.

Throws: If cloner throws or predicate copy assignment throws. Basic guarantee.

71.
```
pointer unlink_leftmost_without_rebalance();
```

Effects: Unlinks the leftmost node from the treap.

Complexity: Average complexity is constant time.

Throws: Nothing.

Notes: This function breaks the treap and the treap can only be used for more unlink_leftmost_without_rebalance calls. This function is normally used to achieve a step by step controlled destruction of the treap.

72.
```
void replace_node(iterator replace_this, reference with_this);
```

Requires: replace_this must be a valid iterator of *this and with_this must not be inserted in any treap.

Effects: Replaces replace_this in its position in the treap with with_this. The treap does not need to be rebalanced.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Constant.

Throws: Nothing.

Note: This function will break container ordering invariants if with_this is not equivalent to *replace_this according to the ordering and priority rules. This function is faster than erasing and inserting the node, since no rebalancing or comparison is needed.

73.
```
iterator iterator_to(reference value);
```

Requires: value must be an lvalue and shall be in a set of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid iterator i belonging to the set that points to the value

Complexity: Constant.

Throws: Nothing.

74.
```
const_iterator iterator_to(const_reference value) const;
```

Requires: value must be an lvalue and shall be in a set of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid const_iterator i belonging to the set that points to the value

Complexity: Constant.

Throws: Nothing.

`treap` public static functions

1.
```
static treap & container_from_end_iterator(iterator end_iterator);
```

Precondition: end_iterator must be a valid end iterator of treap.

Effects: Returns a const reference to the treap associated to the end iterator

Throws: Nothing.

Complexity: Constant.

2.
```
static const treap & container_from_end_iterator(const_iterator end_iterator);
```

Precondition: end_iterator must be a valid end const_iterator of treap.

Effects: Returns a const reference to the treap associated to the iterator

Throws: Nothing.

Complexity: Constant.

3.
```
static treap & container_from_iterator(iterator it);
```

Precondition: it must be a valid iterator of treap.

Effects: Returns a const reference to the treap associated to the iterator

Throws: Nothing.

Complexity: Logarithmic.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

4.
   ```
   static const treap & container_from_iterator(const_iterator it);
   ```

   Precondition: it must be a valid end const_iterator of treap.

   Effects: Returns a const reference to the treap associated to the end iterator

   Throws: Nothing.

   Complexity: Logarithmic.

5.
   ```
   static iterator s_iterator_to(reference value);
   ```

   Requires: value must be an lvalue and shall be in a set of appropriate type. Otherwise the behavior is undefined.

   Effects: Returns: a valid iterator i belonging to the set that points to the value

   Complexity: Constant.

   Throws: Nothing.

   Note: This static function is available only if the *value traits* is stateless.

6.
   ```
   static const_iterator s_iterator_to(const_reference value);
   ```

   Requires: value must be an lvalue and shall be in a set of appropriate type. Otherwise the behavior is undefined.

   Effects: Returns: a valid const_iterator i belonging to the set that points to the value

   Complexity: Constant.

   Throws: Nothing.

   Note: This static function is available only if the *value traits* is stateless.

7.
   ```
   static void init_node(reference value);
   ```

   Requires: value shall not be in a treap.

   Effects: init_node puts the hook of a value in a well-known default state.

   Throws: Nothing.

   Complexity: Constant time.

   Note: This function puts the hook in the well-known default state used by auto_unlink and safe hooks.

`treap` private static functions

1.
   ```
   static treap &
   priv_container_from_end_iterator(const const_iterator & end_iterator);
   ```

2.
   ```
   static treap & priv_container_from_iterator(const const_iterator & it);
   ```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Struct template make_trie

boost::intrusive::make_trie

Synopsis

```
// In header: <boost/intrusive/treap.hpp>

template<typename T, class... Options>
struct make_trie {
  // types
  typedef implementation_defined type;
};
```

Description

Helper metafunction to define a `treap` that yields to the same type when the same options (either explicitly or implicitly) are used.

Header <boost/intrusive/treap_algorithms.hpp>

```
namespace boost {
  namespace intrusive {
    template<typename NodeTraits> class treap_algorithms;
  }
}
```

Class template treap_algorithms

boost::intrusive::treap_algorithms

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/intrusive/treap_algorithms.hpp>

template<typename NodeTraits>
class treap_algorithms {
public:
  // types
  typedef NodeTraits                  node_traits;
  typedef NodeTraits::node            node;
  typedef NodeTraits::node_ptr        node_ptr;
  typedef NodeTraits::const_node_ptr const_node_ptr;

  // member classes/structs/unions

  struct insert_commit_data {
  };

  // public static functions
  static node_ptr begin_node(const const_node_ptr &);
  static node_ptr end_node(const const_node_ptr &);
  static void swap_tree(const node_ptr &, const node_ptr &);
  static void swap_nodes(const node_ptr &, const node_ptr &);
  static void swap_nodes(const node_ptr &, const node_ptr &, const node_ptr &,
                         const node_ptr &);
  static void replace_node(const node_ptr &, const node_ptr &);
  static void replace_node(const node_ptr &, const node_ptr &,
                           const node_ptr &);
  template<typename NodePtrPriorityCompare>
    static void unlink(const node_ptr &, NodePtrPriorityCompare);
  static node_ptr unlink_leftmost_without_rebalance(const node_ptr &);
  static bool unique(const const_node_ptr &);
  static std::size_t count(const const_node_ptr &);
  static std::size_t size(const const_node_ptr &);
  static node_ptr next_node(const node_ptr &);
  static node_ptr prev_node(const node_ptr &);
  static void init(const node_ptr &);
  static void init_header(const node_ptr &);
  template<typename NodePtrPriorityCompare>
    static node_ptr
    erase(const node_ptr &, const node_ptr &, NodePtrPriorityCompare);
  template<typename Cloner, typename Disposer>
    static void clone(const const_node_ptr &, const node_ptr &, Cloner,
                      Disposer);
  template<typename Disposer>
    static void clear_and_dispose(const node_ptr &, Disposer);
  template<typename KeyType, typename KeyNodePtrCompare>
    static node_ptr
    lower_bound(const const_node_ptr &, const KeyType &, KeyNodePtrCompare);
  template<typename KeyType, typename KeyNodePtrCompare>
    static node_ptr
    upper_bound(const const_node_ptr &, const KeyType &, KeyNodePtrCompare);
  template<typename KeyType, typename KeyNodePtrCompare>
    static node_ptr
    find(const const_node_ptr &, const KeyType &, KeyNodePtrCompare);
  template<typename KeyType, typename KeyNodePtrCompare>
    static std::pair< node_ptr, node_ptr >
    equal_range(const const_node_ptr &, const KeyType &, KeyNodePtrCompare);
  template<typename KeyType, typename KeyNodePtrCompare>
    static std::pair< node_ptr, node_ptr >
    bounded_range(const const_node_ptr &, const KeyType &, const KeyType &,
                  KeyNodePtrCompare, bool, bool);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
  template<typename NodePtrCompare, typename NodePtrPriorityCompare> 
    static node_ptr 
    insert_equal_upper_bound(const node_ptr &, const node_ptr &, 
                             NodePtrCompare, NodePtrPriorityCompare);
  template<typename NodePtrCompare, typename NodePtrPriorityCompare> 
    static node_ptr 
    insert_equal_lower_bound(const node_ptr &, const node_ptr &, 
                             NodePtrCompare, NodePtrPriorityCompare);
  template<typename NodePtrCompare, typename NodePtrPriorityCompare> 
    static node_ptr 
    insert_equal(const node_ptr &, const node_ptr &, const node_ptr &, 
                 NodePtrCompare, NodePtrPriorityCompare);
  template<typename NodePtrPriorityCompare> 
    static node_ptr 
    insert_before(const node_ptr &, const node_ptr &, const node_ptr &, 
                  NodePtrPriorityCompare);
  template<typename NodePtrPriorityCompare> 
    static void push_back(const node_ptr &, const node_ptr &, 
                          NodePtrPriorityCompare);
  template<typename NodePtrPriorityCompare> 
    static void push_front(const node_ptr &, const node_ptr &, 
                           NodePtrPriorityCompare);
  template<typename KeyType, typename KeyNodePtrCompare, 
           typename KeyNodePtrPrioCompare> 
    static std::pair< node_ptr, bool > 
    insert_unique_check(const const_node_ptr &, const KeyType &, 
                        KeyNodePtrCompare, KeyNodePtrPrioCompare, 
                        insert_commit_data &);
  template<typename KeyType, typename KeyNodePtrCompare, 
           typename KeyNodePtrPrioCompare> 
    static std::pair< node_ptr, bool > 
    insert_unique_check(const const_node_ptr &, const node_ptr &, 
                        const KeyType &, KeyNodePtrCompare, 
                        KeyNodePtrPrioCompare, insert_commit_data &);
  static void insert_unique_commit(const node_ptr &, const node_ptr &, 
                                   const insert_commit_data &);
  static node_ptr get_header(const node_ptr &);
};
```

Description

treap_algorithms provides basic algorithms to manipulate nodes forming a treap.

(1) the header node is maintained with links not only to the root but also to the leftmost node of the tree, to enable constant time begin(), and to the rightmost node of the tree, to enable linear time performance when used with the generic set algorithms (set_union, etc.);

(2) when a node being deleted has two children its successor node is relinked into its place, rather than copied, so that the only pointers invalidated are those referring to the deleted node.

treap_algorithms is configured with a NodeTraits class, which encapsulates the information about the node to be manipulated. NodeTraits must support the following interface:

Typedefs:

`node`: The type of the node that forms the circular list

`node_ptr`: A pointer to a node

`const_node_ptr`: A pointer to a const node

Static functions:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
static node_ptr get_parent(const_node_ptr n);

static void set_parent(node_ptr n, node_ptr parent);

static node_ptr get_left(const_node_ptr n);

static void set_left(node_ptr n, node_ptr left);

static node_ptr get_right(const_node_ptr n);

static void set_right(node_ptr n, node_ptr right);
```

`treap_algorithms` public static functions

1.
```
static node_ptr begin_node(const const_node_ptr & header);
```

2.
```
static node_ptr end_node(const const_node_ptr & header);
```

3.
```
static void swap_tree(const node_ptr & header1, const node_ptr & header2);
```

   Requires: header1 and header2 must be the header nodes of two trees.

   Effects: Swaps two trees. After the function header1 will contain links to the second tree and header2 will have links to the first tree.

   Complexity: Constant.

   Throws: Nothing.

4.
```
static void swap_nodes(const node_ptr & node1, const node_ptr & node2);
```

   Requires: node1 and node2 can't be header nodes of two trees.

   Effects: Swaps two nodes. After the function node1 will be inserted in the position node2 before the function. node2 will be inserted in the position node1 had before the function.

   Complexity: Logarithmic.

   Throws: Nothing.

   Note: This function will break container ordering invariants if node1 and node2 are not equivalent according to the ordering rules.

   Experimental function

5.
```
static void swap_nodes(const node_ptr & node1, const node_ptr & header1,
                       const node_ptr & node2, const node_ptr & header2);
```

   Requires: node1 and node2 can't be header nodes of two trees with header header1 and header2.

   Effects: Swaps two nodes. After the function node1 will be inserted in the position node2 before the function. node2 will be inserted in the position node1 had before the function.

   Complexity: Constant.

   Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Note: This function will break container ordering invariants if node1 and node2 are not equivalent according to the ordering rules.

Experimental function

6.
```
static void replace_node(const node_ptr & node_to_be_replaced,
                         const node_ptr & new_node);
```

Requires: node_to_be_replaced must be inserted in a tree and new_node must not be inserted in a tree.

Effects: Replaces node_to_be_replaced in its position in the tree with new_node. The tree does not need to be rebalanced

Complexity: Logarithmic.

Throws: Nothing.

Note: This function will break container ordering invariants if new_node is not equivalent to node_to_be_replaced according to the ordering rules. This function is faster than erasing and inserting the node, since no rebalancing and comparison is needed.

Experimental function

7.
```
static void replace_node(const node_ptr & node_to_be_replaced,
                         const node_ptr & header, const node_ptr & new_node);
```

Requires: node_to_be_replaced must be inserted in a tree with header "header" and new_node must not be inserted in a tree.

Effects: Replaces node_to_be_replaced in its position in the tree with new_node. The tree does not need to be rebalanced

Complexity: Constant.

Throws: Nothing.

Note: This function will break container ordering invariants if new_node is not equivalent to node_to_be_replaced according to the ordering rules. This function is faster than erasing and inserting the node, since no rebalancing or comparison is needed.

Experimental function

8.
```
template<typename NodePtrPriorityCompare>
  static void unlink(const node_ptr & node, NodePtrPriorityCompare pcomp);
```

Requires: node is a tree node but not the header.

Effects: Unlinks the node and rebalances the tree.

Complexity: Average complexity is constant time.

Throws: If "pcomp" throws, strong guarantee

9.
```
static node_ptr unlink_leftmost_without_rebalance(const node_ptr & header);
```

Requires: header is the header of a tree.

Effects: Unlinks the leftmost node from the tree, and updates the header link to the new leftmost node.

Complexity: Average complexity is constant time.

Throws: Nothing.

Notes: This function breaks the tree and the tree can only be used for more unlink_leftmost_without_rebalance calls. This function is normally used to achieve a step by step controlled destruction of the tree.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

10.
```
static bool unique(const const_node_ptr & node);
```

Requires: node is a node of the tree or an node initialized by init(...).

Effects: Returns true if the node is initialized by init().

Complexity: Constant time.

Throws: Nothing.

11.
```
static std::size_t count(const const_node_ptr & node);
```

Requires: node is a node of the tree but it's not the header.

Effects: Returns the number of nodes of the subtree.

Complexity: Linear time.

Throws: Nothing.

12.
```
static std::size_t size(const const_node_ptr & header);
```

Requires: header is the header node of the tree.

Effects: Returns the number of nodes above the header.

Complexity: Linear time.

Throws: Nothing.

13.
```
static node_ptr next_node(const node_ptr & p);
```

Requires: p is a node from the tree except the header.

Effects: Returns the next node of the tree.

Complexity: Average constant time.

Throws: Nothing.

14.
```
static node_ptr prev_node(const node_ptr & p);
```

Requires: p is a node from the tree except the leftmost node.

Effects: Returns the previous node of the tree.

Complexity: Average constant time.

Throws: Nothing.

15.
```
static void init(const node_ptr & node);
```

Requires: node must not be part of any tree.

Effects: After the function unique(node) == true.

Complexity: Constant.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: Nothing.

Nodes: If node is inserted in a tree, this function corrupts the tree.

16.
```
static void init_header(const node_ptr & header);
```

Requires: node must not be part of any tree.

Effects: Initializes the header to represent an empty tree. unique(header) == true.

Complexity: Constant.

Throws: Nothing.

Nodes: If node is inserted in a tree, this function corrupts the tree.

17.
```
template<typename NodePtrPriorityCompare>
  static node_ptr
  erase(const node_ptr & header, const node_ptr & z,
        NodePtrPriorityCompare pcomp);
```

Requires: header must be the header of a tree, z a node of that tree and z != header.

Effects: Erases node "z" from the tree with header "header".

Complexity: Amortized constant time.

Throws: If "pcomp" throws, strong guarantee.

18.
```
template<typename Cloner, typename Disposer>
  static void clone(const const_node_ptr & source_header,
                    const node_ptr & target_header, Cloner cloner,
                    Disposer disposer);
```

Requires: "cloner" must be a function object taking a node_ptr and returning a new cloned node of it. "disposer" must take a node_ptr and shouldn't throw.

Effects: First empties target tree calling `void disposer::operator()(const node_ptr &)` for every node of the tree except the header.

Then, duplicates the entire tree pointed by "source_header" cloning each source node with `node_ptr Cloner::operator()(const node_ptr &)` to obtain the nodes of the target tree. If "cloner" throws, the cloned target nodes are disposed using `void disposer(const node_ptr &)`.

Complexity: Linear to the number of element of the source tree plus the. number of elements of tree target tree when calling this function.

Throws: If cloner functor throws. If this happens target nodes are disposed.

19.
```
template<typename Disposer>
  static void clear_and_dispose(const node_ptr & header, Disposer disposer);
```

Requires: "disposer" must be an object function taking a node_ptr parameter and shouldn't throw.

Effects: Empties the target tree calling `void disposer::operator()(const node_ptr &)` for every node of the tree except the header.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Linear to the number of element of the source tree plus the. number of elements of tree target tree when calling this function.

Throws: If cloner functor throws. If this happens target nodes are disposed.

20.
```
template<typename KeyType, typename KeyNodePtrCompare> 
  static node_ptr 
  lower_bound(const const_node_ptr & header, const KeyType & key, 
              KeyNodePtrCompare comp);
```

Requires: "header" must be the header node of a tree. KeyNodePtrCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. KeyNodePtrCompare can compare KeyType with tree's node_ptrs.

Effects: Returns an node_ptr to the first element that is not less than "key" according to "comp" or "header" if that element does not exist.

Complexity: Logarithmic.

Throws: If "comp" throws.

21.
```
template<typename KeyType, typename KeyNodePtrCompare> 
  static node_ptr 
  upper_bound(const const_node_ptr & header, const KeyType & key, 
              KeyNodePtrCompare comp);
```

Requires: "header" must be the header node of a tree. KeyNodePtrCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. KeyNodePtrCompare can compare KeyType with tree's node_ptrs.

Effects: Returns an node_ptr to the first element that is greater than "key" according to "comp" or "header" if that element does not exist.

Complexity: Logarithmic.

Throws: If "comp" throws.

22.
```
template<typename KeyType, typename KeyNodePtrCompare> 
  static node_ptr 
  find(const const_node_ptr & header, const KeyType & key, 
       KeyNodePtrCompare comp);
```

Requires: "header" must be the header node of a tree. KeyNodePtrCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. KeyNodePtrCompare can compare KeyType with tree's node_ptrs.

Effects: Returns an node_ptr to the element that is equivalent to "key" according to "comp" or "header" if that element does not exist.

Complexity: Logarithmic.

Throws: If "comp" throws.

23.
```
template<typename KeyType, typename KeyNodePtrCompare> 
  static std::pair< node_ptr, node_ptr > 
  equal_range(const const_node_ptr & header, const KeyType & key, 
              KeyNodePtrCompare comp);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Requires: "header" must be the header node of a tree. KeyNodePtrCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. KeyNodePtrCompare can compare KeyType with tree's node_ptrs.

Effects: Returns an a pair of node_ptr delimiting a range containing all elements that are equivalent to "key" according to "comp" or an empty range that indicates the position where those elements would be if they there are no equivalent elements.

Complexity: Logarithmic.

Throws: If "comp" throws.

24.
```
template<typename KeyType, typename KeyNodePtrCompare> 
  static std::pair< node_ptr, node_ptr > 
  bounded_range(const const_node_ptr & header, const KeyType & lower_key, 
                const KeyType & upper_key, KeyNodePtrCompare comp, 
                bool left_closed, bool right_closed);
```

Requires: "header" must be the header node of a tree. KeyNodePtrCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. KeyNodePtrCompare can compare KeyType with tree's node_ptrs. 'lower_key' must not be greater than 'upper_key' according to 'comp'. If 'lower_key' == 'upper_key', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key) if left_closed, upper_bound(lower_key) otherwise

second = upper_bound(upper_key) if right_closed, lower_bound(upper_key) otherwise

Complexity: Logarithmic.

Throws: If "comp" throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_key and upper_key.

25.
```
template<typename NodePtrCompare, typename NodePtrPriorityCompare> 
  static node_ptr 
  insert_equal_upper_bound(const node_ptr & h, const node_ptr & new_node, 
                           NodePtrCompare comp, NodePtrPriorityCompare pcomp);
```

Requires: "h" must be the header node of a tree. NodePtrCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. NodePtrCompare compares two node_ptrs. NodePtrPriorityCompare is a priority function object that induces a strict weak ordering compatible with the one used to create the the tree. NodePtrPriorityCompare compares two node_ptrs.

Effects: Inserts new_node into the tree before the upper bound according to "comp" and rotates the tree according to "pcomp".

Complexity: Average complexity for insert element is at most logarithmic.

Throws: If "comp" throw or "pcomp" throw.

26.
```
template<typename NodePtrCompare, typename NodePtrPriorityCompare> 
  static node_ptr 
  insert_equal_lower_bound(const node_ptr & h, const node_ptr & new_node, 
                           NodePtrCompare comp, NodePtrPriorityCompare pcomp);
```

Requires: "h" must be the header node of a tree. NodePtrCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. NodePtrCompare compares two node_ptrs. NodePtrPriorityCompare is a priority function object that induces a strict weak ordering compatible with the one used to create the the tree. NodePtrPriorityCompare compares two node_ptrs.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Inserts new_node into the tree before the upper bound according to "comp" and rotates the tree according to "pcomp".

Complexity: Average complexity for insert element is at most logarithmic.

Throws: If "comp" throws.

27.
```
template<typename NodePtrCompare, typename NodePtrPriorityCompare> 
  static node_ptr 
  insert_equal(const node_ptr & h, const node_ptr & hint, 
               const node_ptr & new_node, NodePtrCompare comp, 
               NodePtrPriorityCompare pcomp);
```

Requires: "header" must be the header node of a tree. NodePtrCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. NodePtrCompare compares two node_ptrs. "hint" is node from the "header"'s tree. NodePtrPriorityCompare is a priority function object that induces a strict weak ordering compatible with the one used to create the the tree. NodePtrPriorityCompare compares two node_ptrs.

Effects: Inserts new_node into the tree, using "hint" as a hint to where it will be inserted. If "hint" is the upper_bound the insertion takes constant time (two comparisons in the worst case). Rotates the tree according to "pcomp".

Complexity: Logarithmic in general, but it is amortized constant time if new_node is inserted immediately before "hint".

Throws: If "comp" throw or "pcomp" throw.

28.
```
template<typename NodePtrPriorityCompare> 
  static node_ptr 
  insert_before(const node_ptr & header, const node_ptr & pos, 
                const node_ptr & new_node, NodePtrPriorityCompare pcomp);
```

Requires: "header" must be the header node of a tree. "pos" must be a valid node of the tree (including header end) node. "pos" must be a node pointing to the successor to "new_node" once inserted according to the order of already inserted nodes. This function does not check "pos" and this precondition must be guaranteed by the caller. NodePtrPriorityCompare is a priority function object that induces a strict weak ordering compatible with the one used to create the the tree. NodePtrPriorityCompare compares two node_ptrs.

Effects: Inserts new_node into the tree before "pos" and rotates the tree according to "pcomp".

Complexity: Constant-time.

Throws: If "pcomp" throws, strong guarantee.

Note: If "pos" is not the successor of the newly inserted "new_node" tree invariants might be broken.

29.
```
template<typename NodePtrPriorityCompare> 
  static void push_back(const node_ptr & header, const node_ptr & new_node, 
                        NodePtrPriorityCompare pcomp);
```

Requires: "header" must be the header node of a tree. "new_node" must be, according to the used ordering no less than the greatest inserted key. NodePtrPriorityCompare is a priority function object that induces a strict weak ordering compatible with the one used to create the the tree. NodePtrPriorityCompare compares two node_ptrs.

Effects: Inserts x into the tree in the last position and rotates the tree according to "pcomp".

Complexity: Constant-time.

Throws: If "pcomp" throws, strong guarantee.

Note: If "new_node" is less than the greatest inserted key tree invariants are broken. This function is slightly faster than using "insert_before".

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

30.
```
template<typename NodePtrPriorityCompare> 
  static void push_front(const node_ptr & header, const node_ptr & new_node, 
                         NodePtrPriorityCompare pcomp);
```

Requires: "header" must be the header node of a tree. "new_node" must be, according to the used ordering, no greater than the lowest inserted key. NodePtrPriorityCompare is a priority function object that induces a strict weak ordering compatible with the one used to create the the tree. NodePtrPriorityCompare compares two node_ptrs.

Effects: Inserts x into the tree in the first position and rotates the tree according to "pcomp".

Complexity: Constant-time.

Throws: If "pcomp" throws, strong guarantee.

Note: If "new_node" is greater than the lowest inserted key tree invariants are broken. This function is slightly faster than using "insert_before".

31.
```
template<typename KeyType, typename KeyNodePtrCompare, 
         typename KeyNodePtrPrioCompare> 
  static std::pair< node_ptr, bool > 
  insert_unique_check(const const_node_ptr & header, const KeyType & key, 
                      KeyNodePtrCompare comp, KeyNodePtrPrioCompare pcomp, 
                      insert_commit_data & commit_data);
```

Requires: "header" must be the header node of a tree. KeyNodePtrCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. NodePtrCompare compares KeyType with a node_ptr.

Effects: Checks if there is an equivalent node to "key" in the tree according to "comp" and obtains the needed information to realize a constant-time node insertion if there is no equivalent node.

Returns: If there is an equivalent value returns a pair containing a node_ptr to the already present node and false. If there is not equivalent key can be inserted returns true in the returned pair's boolean and fills "commit_data" that is meant to be used with the "insert_commit" function to achieve a constant-time insertion function.

Complexity: Average complexity is at most logarithmic.

Throws: If "comp" throws.

Notes: This function is used to improve performance when constructing a node is expensive and the user does not want to have two equivalent nodes in the tree: if there is an equivalent value the constructed object must be discarded. Many times, the part of the node that is used to impose the order is much cheaper to construct than the node and this function offers the possibility to use that part to check if the insertion will be successful.

If the check is successful, the user can construct the node and use "insert_commit" to insert the node in constant-time. This gives a total logarithmic complexity to the insertion: check(O(log(N)) + commit(O(1)).

"commit_data" remains valid for a subsequent "insert_unique_commit" only if no more objects are inserted or erased from the set.

32.
```
template<typename KeyType, typename KeyNodePtrCompare, 
         typename KeyNodePtrPrioCompare> 
  static std::pair< node_ptr, bool > 
  insert_unique_check(const const_node_ptr & header, const node_ptr & hint, 
                      const KeyType & key, KeyNodePtrCompare comp, 
                      KeyNodePtrPrioCompare pcomp, 
                      insert_commit_data & commit_data);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Requires: "header" must be the header node of a tree. KeyNodePtrCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. NodePtrCompare compares KeyType with a node_ptr. "hint" is node from the "header"'s tree.

Effects: Checks if there is an equivalent node to "key" in the tree according to "comp" using "hint" as a hint to where it should be inserted and obtains the needed information to realize a constant-time node insertion if there is no equivalent node. If "hint" is the upper_bound the function has constant time complexity (two comparisons in the worst case).

Returns: If there is an equivalent value returns a pair containing a node_ptr to the already present node and false. If there is not equivalent key can be inserted returns true in the returned pair's boolean and fills "commit_data" that is meant to be used with the "insert_commit" function to achieve a constant-time insertion function.

Complexity: Average complexity is at most logarithmic, but it is amortized constant time if new_node should be inserted immediately before "hint".

Throws: If "comp" throws.

Notes: This function is used to improve performance when constructing a node is expensive and the user does not want to have two equivalent nodes in the tree: if there is an equivalent value the constructed object must be discarded. Many times, the part of the node that is used to impose the order is much cheaper to construct than the node and this function offers the possibility to use that part to check if the insertion will be successful.

If the check is successful, the user can construct the node and use "insert_commit" to insert the node in constant-time. This gives a total logarithmic complexity to the insertion: check(O(log(N)) + commit(O(1)).

"commit_data" remains valid for a subsequent "insert_unique_commit" only if no more objects are inserted or erased from the set.

33.
```
static void insert_unique_commit(const node_ptr & header,
                                 const node_ptr & new_node,
                                 const insert_commit_data & commit_data);
```

Requires: "header" must be the header node of a tree. "commit_data" must have been obtained from a previous call to "insert_unique_check". No objects should have been inserted or erased from the set between the "insert_unique_check" that filled "commit_data" and the call to "insert_commit".

Effects: Inserts new_node in the set using the information obtained from the "commit_data" that a previous "insert_check" filled.

Complexity: Constant time.

Throws: Nothing.

Notes: This function has only sense if a "insert_unique_check" has been previously executed to fill "commit_data". No value should be inserted or erased between the "insert_check" and "insert_commit" calls.

34.
```
static node_ptr get_header(const node_ptr & n);
```

Requires: "n" must be a node inserted in a tree.

Effects: Returns a pointer to the header node of the tree.

Complexity: Logarithmic.

Throws: Nothing.

Struct insert_commit_data

boost::intrusive::treap_algorithms::insert_commit_data

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/intrusive/treap_algorithms.hpp>



struct insert_commit_data {
};
```

Description

This type is the information that will be filled by insert_unique_check

Header <boost/intrusive/treap_set.hpp>

```
namespace boost {
  namespace intrusive {
    template<typename T, class... Options> class treap_set;

    template<typename T, class... Options> struct make_treap_set;

    template<typename T, class... Options> class treap_multiset;

    template<typename T, class... Options> struct make_treap_multiset;
    template<typename T, class... Options>
      bool operator!=(const treap_set< T, Options...> & x,
                      const treap_set< T, Options...> & y);
    template<typename T, class... Options>
      bool operator>(const treap_set< T, Options...> & x,
                     const treap_set< T, Options...> & y);
    template<typename T, class... Options>
      bool operator<=(const treap_set< T, Options...> & x,
                      const treap_set< T, Options...> & y);
    template<typename T, class... Options>
      bool operator>=(const treap_set< T, Options...> & x,
                      const treap_set< T, Options...> & y);
    template<typename T, class... Options>
      void swap(treap_set< T, Options...> & x, treap_set< T, Options...> & y);
    template<typename T, class... Options>
      bool operator!=(const treap_multiset< T, Options...> & x,
                      const treap_multiset< T, Options...> & y);
    template<typename T, class... Options>
      bool operator>(const treap_multiset< T, Options...> & x,
                     const treap_multiset< T, Options...> & y);
    template<typename T, class... Options>
      bool operator<=(const treap_multiset< T, Options...> & x,
                      const treap_multiset< T, Options...> & y);
    template<typename T, class... Options>
      bool operator>=(const treap_multiset< T, Options...> & x,
                      const treap_multiset< T, Options...> & y);
    template<typename T, class... Options>
      void swap(treap_multiset< T, Options...> & x,
                treap_multiset< T, Options...> & y);
  }
}
```

Class template treap_set

boost::intrusive::treap_set

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/intrusive/treap_set.hpp>

template<typename T, class... Options>
class treap_set {
public:
  // types
  typedef implementation_defined::value_type             value_type;
  typedef implementation_defined::value_traits           value_traits;
  typedef implementation_defined::pointer                pointer;
  typedef implementation_defined::const_pointer          const_pointer;
  typedef implementation_defined::reference              reference;
  typedef implementation_defined::const_reference        const_reference;
  typedef implementation_defined::difference_type        difference_type;
  typedef implementation_defined::size_type              size_type;
  typedef implementation_defined::value_compare          value_compare;
  typedef implementation_defined::priority_compare       priority_compare;
  typedef implementation_defined::key_compare            key_compare;
  typedef implementation_defined::iterator               iterator;
  typedef implementation_defined::const_iterator         const_iterator;
  typedef implementation_defined::reverse_iterator       reverse_iterator;
  typedef implementation_defined::const_reverse_iterator const_reverse_iterator;
  typedef implementation_defined::insert_commit_data     insert_commit_data;
  typedef implementation_defined::node_traits            node_traits;
  typedef implementation_defined::node                   node;
  typedef implementation_defined::node_ptr               node_ptr;
  typedef implementation_defined::const_node_ptr         const_node_ptr;
  typedef implementation_defined::node_algorithms        node_algorithms;

  // construct/copy/destruct
  explicit treap_set(const value_compare & = value_compare(),
                     const priority_compare & = priority_compare(),
                     const value_traits & = value_traits());
  template<typename Iterator>
    treap_set(Iterator, Iterator, const value_compare & = value_compare(),
              const priority_compare & = priority_compare(),
              const value_traits & = value_traits());
  treap_set(BOOST_RV_REF(treap_set));
  treap_set& operator=(BOOST_RV_REF(treap_set));
  ~treap_set();

  // public member functions
  iterator begin();
  const_iterator begin() const;
  const_iterator cbegin() const;
  iterator end();
  const_iterator end() const;
  iterator top();
  const_iterator top() const;
  const_iterator ctop() const;
  const_iterator cend() const;
  reverse_iterator rbegin();
  const_reverse_iterator rbegin() const;
  const_reverse_iterator crbegin() const;
  reverse_iterator rend();
  const_reverse_iterator rend() const;
  const_reverse_iterator crend() const;
  reverse_iterator rtop();
  const_reverse_iterator rtop() const;
  const_reverse_iterator crtop() const;
  key_compare key_comp() const;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
value_compare value_comp() const;
priority_compare priority_comp() const;
bool empty() const;
size_type size() const;
void swap(treap_set &);
template<typename Cloner, typename Disposer> 
  void clone_from(const treap_set &, Cloner, Disposer);
std::pair< iterator, bool > insert(reference);
iterator insert(const_iterator, reference);
template<typename KeyType, typename KeyValueCompare, 
         typename KeyValuePriorityCompare> 
  std::pair< iterator, bool > 
  insert_check(const KeyType &, KeyValueCompare, KeyValuePriorityCompare, 
               insert_commit_data &);
template<typename KeyType, typename KeyValueCompare, 
         typename KeyValuePriorityCompare> 
  std::pair< iterator, bool > 
  insert_check(const_iterator, const KeyType &, KeyValueCompare, 
               KeyValuePriorityCompare, insert_commit_data &);
iterator insert_commit(reference, const insert_commit_data &);
template<typename Iterator> void insert(Iterator, Iterator);
iterator insert_before(const_iterator, reference);
void push_back(reference);
void push_front(reference);
iterator erase(const_iterator);
iterator erase(const_iterator, const_iterator);
size_type erase(const_reference);
template<typename KeyType, typename KeyValueCompare> 
  size_type erase(const KeyType &, KeyValueCompare);
template<typename Disposer> 
  iterator erase_and_dispose(const_iterator, Disposer);
template<typename Disposer> 
  iterator erase_and_dispose(const_iterator, const_iterator, Disposer);
template<typename Disposer> 
  size_type erase_and_dispose(const_reference, Disposer);
template<typename KeyType, typename KeyValueCompare, typename Disposer> 
  size_type erase_and_dispose(const KeyType &, KeyValueCompare, Disposer);
void clear();
template<typename Disposer> void clear_and_dispose(Disposer);
size_type count(const_reference) const;
template<typename KeyType, typename KeyValueCompare> 
  size_type count(const KeyType &, KeyValueCompare) const;
iterator lower_bound(const_reference);
template<typename KeyType, typename KeyValueCompare> 
  iterator lower_bound(const KeyType &, KeyValueCompare);
const_iterator lower_bound(const_reference) const;
template<typename KeyType, typename KeyValueCompare> 
  const_iterator lower_bound(const KeyType &, KeyValueCompare) const;
iterator upper_bound(const_reference);
template<typename KeyType, typename KeyValueCompare> 
  iterator upper_bound(const KeyType &, KeyValueCompare);
const_iterator upper_bound(const_reference) const;
template<typename KeyType, typename KeyValueCompare> 
  const_iterator upper_bound(const KeyType &, KeyValueCompare) const;
iterator find(const_reference);
template<typename KeyType, typename KeyValueCompare> 
  iterator find(const KeyType &, KeyValueCompare);
const_iterator find(const_reference) const;
template<typename KeyType, typename KeyValueCompare> 
  const_iterator find(const KeyType &, KeyValueCompare) const;
std::pair< iterator, iterator > equal_range(const_reference);
template<typename KeyType, typename KeyValueCompare> 
  std::pair< iterator, iterator >
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
    equal_range(const KeyType &, KeyValueCompare);
  std::pair< const_iterator, const_iterator >
  equal_range(const_reference) const;
  template<typename KeyType, typename KeyValueCompare>
    std::pair< const_iterator, const_iterator >
    equal_range(const KeyType &, KeyValueCompare) const;
  std::pair< iterator, iterator >
  bounded_range(const_reference, const_reference, bool, bool);
  template<typename KeyType, typename KeyValueCompare>
    std::pair< iterator, iterator >
    bounded_range(const KeyType &, const KeyType &, KeyValueCompare, bool,
                  bool);
  std::pair< const_iterator, const_iterator >
  bounded_range(const_reference, const_reference, bool, bool) const;
  template<typename KeyType, typename KeyValueCompare>
    std::pair< const_iterator, const_iterator >
    bounded_range(const KeyType &, const KeyType &, KeyValueCompare, bool,
                  bool) const;
  iterator iterator_to(reference);
  const_iterator iterator_to(const_reference) const;
  pointer unlink_leftmost_without_rebalance();
  void replace_node(iterator, reference);
  void rebalance();
  iterator rebalance_subtree(iterator);
  float balance_factor() const;
  void balance_factor(float);

  // public static functions
  static treap_set & container_from_end_iterator(iterator);
  static const treap_set & container_from_end_iterator(const_iterator);
  static treap_set & container_from_iterator(iterator);
  static const treap_set & container_from_iterator(const_iterator);
  static iterator s_iterator_to(reference);
  static const_iterator s_iterator_to(const_reference);
  static void init_node(reference);

  // public data members
  static const bool constant_time_size;
};
```

Description

The class template treap_set is an intrusive container, that mimics most of the interface of std::set as described in the C++ standard.

The template parameter `T` is the type to be managed by the container. The user can specify additional options and if no options are provided default options are used.

The container supports the following options: `base_hook<>/member_hook<>/value_traits<>`, `constant_time_size<>`, `size_type<>`, `compare<>` and `priority_compare<>`

`treap_set` public construct/copy/destruct

1.
```
explicit treap_set(const value_compare & cmp = value_compare(),
                   const priority_compare & pcmp = priority_compare(),
                   const value_traits & v_traits = value_traits());
```

Effects: Constructs an empty `treap_set`.

Complexity: Constant.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: If value_traits::node_traits::node constructor throws (this does not happen with predefined Boost.Intrusive hooks) or the copy constructor of the value_compare object throws.

2.
```
template<typename Iterator> 
  treap_set(Iterator b, Iterator e, 
            const value_compare & cmp = value_compare(), 
            const priority_compare & pcmp = priority_compare(), 
            const value_traits & v_traits = value_traits());
```

Requires: Dereferencing iterator must yield an lvalue of type value_type. cmp must be a comparison function that induces a strict weak ordering.

Effects: Constructs an empty `treap_set` and inserts elements from [b, e).

Complexity: Linear in N if [b, e) is already sorted using comp and otherwise N * log N, where N is std::distance(last, first).

Throws: If value_traits::node_traits::node constructor throws (this does not happen with predefined Boost.Intrusive hooks) or the copy constructor/operator() of the value_compare object throws.

3.
```
treap_set(BOOST_RV_REF(treap_set) x);
```

Effects: to-do

4.
```
treap_set& operator=(BOOST_RV_REF(treap_set) x);
```

Effects: to-do

5.
```
~treap_set();
```

Effects: Detaches all elements from this. The objects in the `treap_set` are not deleted (i.e. no destructors are called).

Complexity: Linear to the number of elements on the container. if it's a safe-mode or auto-unlink value_type. Constant time otherwise.

Throws: Nothing.

`treap_set` public member functions

1.
```
iterator begin();
```

Effects: Returns an iterator pointing to the beginning of the `treap_set`.

Complexity: Constant.

Throws: Nothing.

2.
```
const_iterator begin() const;
```

Effects: Returns a const_iterator pointing to the beginning of the `treap_set`.

Complexity: Constant.

Throws: Nothing.

3.
```
const_iterator cbegin() const;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Returns a const_iterator pointing to the beginning of the `treap_set`.

Complexity: Constant.

Throws: Nothing.

4.
```
iterator end();
```

Effects: Returns an iterator pointing to the end of the `treap_set`.

Complexity: Constant.

Throws: Nothing.

5.
```
const_iterator end() const;
```

Effects: Returns a const_iterator pointing to the end of the `treap_set`.

Complexity: Constant.

Throws: Nothing.

6.
```
iterator top();
```

Effects: Returns an iterator pointing to the highest priority object of the tree.

Complexity: Constant.

Throws: Nothing.

7.
```
const_iterator top() const;
```

Effects: Returns a const_iterator pointing to the highest priority object of the tree..

Complexity: Constant.

Throws: Nothing.

8.
```
const_iterator ctop() const;
```

Effects: Returns a const_iterator pointing to the highest priority object of the tree..

Complexity: Constant.

Throws: Nothing.

9.
```
const_iterator cend() const;
```

Effects: Returns a const_iterator pointing to the end of the `treap_set`.

Complexity: Constant.

Throws: Nothing.

10.
```
reverse_iterator rbegin();
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Returns a reverse_iterator pointing to the beginning of the reversed `treap_set`.

Complexity: Constant.

Throws: Nothing.

11.
```
const_reverse_iterator rbegin() const;
```

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed `treap_set`.

Complexity: Constant.

Throws: Nothing.

12.
```
const_reverse_iterator crbegin() const;
```

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed `treap_set`.

Complexity: Constant.

Throws: Nothing.

13.
```
reverse_iterator rend();
```

Effects: Returns a reverse_iterator pointing to the end of the reversed `treap_set`.

Complexity: Constant.

Throws: Nothing.

14.
```
const_reverse_iterator rend() const;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed `treap_set`.

Complexity: Constant.

Throws: Nothing.

15.
```
const_reverse_iterator crend() const;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed `treap_set`.

Complexity: Constant.

Throws: Nothing.

16.
```
reverse_iterator rtop();
```

Effects: Returns a reverse_iterator pointing to the highest priority object of the reversed tree.

Complexity: Constant.

Throws: Nothing.

17.
```
const_reverse_iterator rtop() const;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Returns a const_reverse_iterator pointing to the highest priority objec of the reversed tree.

Complexity: Constant.

Throws: Nothing.

18.
```
const_reverse_iterator crtop() const;
```

Effects: Returns a const_reverse_iterator pointing to the highest priority object of the reversed tree.

Complexity: Constant.

Throws: Nothing.

19.
```
key_compare key_comp() const;
```

Effects: Returns the key_compare object used by the `treap_set`.

Complexity: Constant.

Throws: If key_compare copy-constructor throws.

20.
```
value_compare value_comp() const;
```

Effects: Returns the value_compare object used by the `treap_set`.

Complexity: Constant.

Throws: If value_compare copy-constructor throws.

21.
```
priority_compare priority_comp() const;
```

Effects: Returns the priority_compare object used by the `treap_set`.

Complexity: Constant.

Throws: If priority_compare copy-constructor throws.

22.
```
bool empty() const;
```

Effects: Returns true if the container is empty.

Complexity: Constant.

Throws: Nothing.

23.
```
size_type size() const;
```

Effects: Returns the number of elements stored in the `treap_set`.

Complexity: Linear to elements contained in *this if, constant-time size option is enabled. Constant-time otherwise.

Throws: Nothing.

24.
```
void swap(treap_set & other);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Swaps the contents of two sets.

Complexity: Constant.

Throws: If the swap() call for the comparison functor found using ADL throws. Strong guarantee.

25.
```
template<typename Cloner, typename Disposer>
  void clone_from(const treap_set & src, Cloner cloner, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw. Cloner should yield to nodes equivalent to the original nodes.

Effects: Erases all the elements from *this calling Disposer::operator()(pointer), clones all the elements from src calling Cloner::operator()(const_reference) and inserts them on *this. Copies the predicate from the source container.

If cloner throws, all cloned elements are unlinked and disposed calling Disposer::operator()(pointer).

Complexity: Linear to erased plus inserted elements.

Throws: If cloner throws or predicate copy assignment throws. Basic guarantee.

26.
```
std::pair< iterator, bool > insert(reference value);
```

Requires: value must be an lvalue

Effects: Tries to inserts value into the `treap_set`.

Returns: If the value is not already present inserts it and returns a pair containing the iterator to the new value and true. If there is an equivalent value returns a pair containing an iterator to the already present value and false.

Complexity: Average complexity for insert element is at most logarithmic.

Throws: If the internal value_compare or priority_compare ordering function throw. Strong guarantee.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

27.
```
iterator insert(const_iterator hint, reference value);
```

Requires: value must be an lvalue

Effects: Tries to to insert x into the `treap_set`, using "hint" as a hint to where it will be inserted.

Returns: An iterator that points to the position where the new element was inserted into the `treap_set`.

Complexity: Logarithmic in general, but it's amortized constant time if t is inserted immediately before hint.

Throws: If the internal value_compare or priority_compare ordering functions throw. Strong guarantee.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

28.
```
template<typename KeyType, typename KeyValueCompare,
         typename KeyValuePriorityCompare>
  std::pair< iterator, bool >
  insert_check(const KeyType & key, KeyValueCompare key_value_comp,
               KeyValuePriorityCompare key_value_pcomp,
               insert_commit_data & commit_data);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Requires: key_value_comp must be a comparison function that induces the same strict weak ordering as value_compare. key_value_pcomp must be a comparison function that induces the same strict weak ordering as priority_compare. The difference is that key_value_pcomp and key_value_comp compare an arbitrary key with the contained values.

Effects: Checks if a value can be inserted in the `treap_set`, using a user provided key instead of the value itself.

Returns: If there is an equivalent value returns a pair containing an iterator to the already present value and false. If the value can be inserted returns true in the returned pair boolean and fills "commit_data" that is meant to be used with the "insert_commit" function.

Complexity: Average complexity is at most logarithmic.

Throws: If key_value_comp or key_value_pcomp ordering function throw. Strong guarantee.

Notes: This function is used to improve performance when constructing a value_type is expensive: if there is an equivalent value the constructed object must be discarded. Many times, the part of the node that is used to impose the order is much cheaper to construct than the value_type and this function offers the possibility to use that part to check if the insertion will be successful.

If the check is successful, the user can construct the value_type and use "insert_commit" to insert the object in constant-time. This gives a total logarithmic complexity to the insertion: check(O(log(N)) + commit(O(1)).

"commit_data" remains valid for a subsequent "insert_commit" only if no more objects are inserted or erased from the `treap_set`.

29.
```
template<typename KeyType, typename KeyValueCompare, 
         typename KeyValuePriorityCompare> 
  std::pair< iterator, bool > 
  insert_check(const_iterator hint, const KeyType & key, 
               KeyValueCompare key_value_comp, 
               KeyValuePriorityCompare key_value_pcomp, 
               insert_commit_data & commit_data);
```

Requires: key_value_comp must be a comparison function that induces the same strict weak ordering as value_compare. key_value_pcomp must be a comparison function that induces the same strict weak ordering as priority_compare. The difference is that key_value_pcomp and key_value_comp compare an arbitrary key with the contained values.

Effects: Checks if a value can be inserted in the `treap_set`, using a user provided key instead of the value itself, using "hint" as a hint to where it will be inserted.

Returns: If there is an equivalent value returns a pair containing an iterator to the already present value and false. If the value can be inserted returns true in the returned pair boolean and fills "commit_data" that is meant to be used with the "insert_commit" function.

Complexity: Logarithmic in general, but it's amortized constant time if t is inserted immediately before hint.

Throws: If key_value_comp or key_value_pcomp ordering function throw. Strong guarantee.

Notes: This function is used to improve performance when constructing a value_type is expensive: if there is an equivalent value the constructed object must be discarded. Many times, the part of the constructing that is used to impose the order is much cheaper to construct than the value_type and this function offers the possibility to use that key to check if the insertion will be successful.

If the check is successful, the user can construct the value_type and use "insert_commit" to insert the object in constant-time. This can give a total constant-time complexity to the insertion: check(O(1)) + commit(O(1)).

"commit_data" remains valid for a subsequent "insert_commit" only if no more objects are inserted or erased from the `treap_set`.

30.
```
iterator insert_commit(reference value, 
                       const insert_commit_data & commit_data);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Requires: value must be an lvalue of type value_type. commit_data must have been obtained from a previous call to "insert_check". No objects should have been inserted or erased from the `treap_set` between the "insert_check" that filled "commit_data" and the call to "insert_commit".

Effects: Inserts the value in the `treap_set` using the information obtained from the "commit_data" that a previous "insert_check" filled.

Returns: An iterator to the newly inserted object.

Complexity: Constant time.

Throws: Nothing.

Notes: This function has only sense if a "insert_check" has been previously executed to fill "commit_data". No value should be inserted or erased between the "insert_check" and "insert_commit" calls.

31.
```
template<typename Iterator> void insert(Iterator b, Iterator e);
```

Requires: Dereferencing iterator must yield an lvalue of type value_type.

Effects: Inserts a range into the `treap_set`.

Complexity: Insert range is in general O(N * log(N)), where N is the size of the range. However, it is linear in N if the range is already sorted by value_comp().

Throws: If the internal value_compare or priority_compare ordering function throw. Basic guarantee.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

32.
```
iterator insert_before(const_iterator pos, reference value);
```

Requires: value must be an lvalue, "pos" must be a valid iterator (or end) and must be the succesor of value once inserted according to the predicate. "value" must not be equal to any inserted key according to the predicate.

Effects: Inserts x into the treap before "pos".

Complexity: Constant time.

Throws: If the internal priority_compare function throws. Strong guarantee.

Note: This function does not check preconditions so if "pos" is not the successor of "value" treap ordering invariant will be broken. This is a low-level function to be used only for performance reasons by advanced users.

33.
```
void push_back(reference value);
```

Requires: value must be an lvalue, and it must be greater than any inserted key according to the predicate.

Effects: Inserts x into the treap in the last position.

Complexity: Constant time.

Throws: If the internal priority_compare function throws. Strong guarantee.

Note: This function does not check preconditions so if value is less than the greatest inserted key treap ordering invariant will be broken. This function is slightly more efficient than using "insert_before". This is a low-level function to be used only for performance reasons by advanced users.

34.
```
void push_front(reference value);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Requires: value must be an lvalue, and it must be less than any inserted key according to the predicate.

Effects: Inserts x into the treap in the first position.

Complexity: Constant time.

Throws: If the internal priority_compare function throws. Strong guarantee.

Note: This function does not check preconditions so if value is greater than the minimum inserted key treap ordering invariant will be broken. This function is slightly more efficient than using "insert_before". This is a low-level function to be used only for performance reasons by advanced users.

35.
```
iterator erase(const_iterator i);
```

Effects: Erases the element pointed to by pos.

Complexity: Average complexity is constant time.

Returns: An iterator to the element after the erased element.

Throws: If the internal priority_compare function throws. Strong guarantee.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

36.
```
iterator erase(const_iterator b, const_iterator e);
```

Effects: Erases the range pointed to by b end e.

Complexity: Average complexity for erase range is at most O(log(size() + N)), where N is the number of elements in the range.

Returns: An iterator to the element after the erased elements.

Throws: If the internal priority_compare function throws. Basic guarantee.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

37.
```
size_type erase(const_reference value);
```

Effects: Erases all the elements with the given value.

Returns: The number of erased elements.

Complexity: O(log(size()) + this->count(value)).

Throws: If internal value_compare or priority_compare ordering functions throw. Basic guarantee.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

38.
```
template<typename KeyType, typename KeyValueCompare> 
  size_type erase(const KeyType & key, KeyValueCompare comp);
```

Effects: Erases all the elements that compare equal with the given key and the given comparison functor.

Returns: The number of erased elements.

Complexity: O(log(size() + this->count(key, comp)).

Throws: If comp or internal priority_compare ordering functions throw. Basic guarantee.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

39.
```
template<typename Disposer> 
  iterator erase_and_dispose(const_iterator i, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases the element pointed to by pos. Disposer::operator()(pointer) is called for the removed element.

Complexity: Average complexity for erase element is constant time.

Returns: An iterator to the element after the erased element.

Throws: If the internal priority_compare function throws. Strong guarantee.

Note: Invalidates the iterators to the erased elements.

40.
```
template<typename Disposer> 
  iterator erase_and_dispose(const_iterator b, const_iterator e, 
                             Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases the range pointed to by b end e. Disposer::operator()(pointer) is called for the removed elements.

Complexity: Average complexity for erase range is at most O(log(size() + N)), where N is the number of elements in the range.

Returns: An iterator to the element after the erased elements.

Throws: If the internal priority_compare function throws. Basic guarantee.

Note: Invalidates the iterators to the erased elements.

41.
```
template<typename Disposer> 
  size_type erase_and_dispose(const_reference value, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases all the elements with the given value. Disposer::operator()(pointer) is called for the removed elements.

Throws: If the internal value_compare ordering function throws.

Complexity: O(log(size() + this->count(value)). Basic guarantee.

Throws: If the internal priority_compare function throws. Strong guarantee.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

42.
```
template<typename KeyType, typename KeyValueCompare, typename Disposer> 
  size_type erase_and_dispose(const KeyType & key, KeyValueCompare comp, 
                              Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases all the elements with the given key. according to the comparison functor "comp". Disposer::operator()(pointer) is called for the removed elements.

Returns: The number of erased elements.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: O(log(size() + this->count(key, comp)).

Throws: If comp or internal priority_compare ordering functions throw. Basic guarantee.

Note: Invalidates the iterators to the erased elements.

43.
```
void clear();
```

Effects: Erases all the elements of the container.

Complexity: Linear to the number of elements on the container. if it's a safe-mode or auto-unlink value_type. Constant time otherwise.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

44.
```
template<typename Disposer> void clear_and_dispose(Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases all the elements of the container.

Complexity: Linear to the number of elements on the container. Disposer::operator()(pointer) is called for the removed elements.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

45.
```
size_type count(const_reference value) const;
```

Effects: Returns the number of contained elements with the given key

Complexity: Logarithmic to the number of elements contained plus lineal to number of objects with the given key.

Throws: If the internal value_compare ordering function throws.

46.
```
template<typename KeyType, typename KeyValueCompare> 
  size_type count(const KeyType & key, KeyValueCompare comp) const;
```

Effects: Returns the number of contained elements with the same key compared with the given comparison functor.

Complexity: Logarithmic to the number of elements contained plus lineal to number of objects with the given key.

Throws: If comp ordering function throws.

47.
```
iterator lower_bound(const_reference value);
```

Effects: Returns an iterator to the first element whose key is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

48.
```
template<typename KeyType, typename KeyValueCompare> 
  iterator lower_bound(const KeyType & key, KeyValueCompare comp);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Returns an iterator to the first element whose key according to the comparison functor is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

49.
```
const_iterator lower_bound(const_reference value) const;
```

Effects: Returns a const iterator to the first element whose key is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

50.
```
template<typename KeyType, typename KeyValueCompare>
  const_iterator lower_bound(const KeyType & key, KeyValueCompare comp) const;
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Returns a const_iterator to the first element whose key according to the comparison functor is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

51.
```
iterator upper_bound(const_reference value);
```

Effects: Returns an iterator to the first element whose key is greater than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

52.
```
template<typename KeyType, typename KeyValueCompare>
  iterator upper_bound(const KeyType & key, KeyValueCompare comp);
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Returns an iterator to the first element whose key according to the comparison functor is greater than key or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

53.
```
const_iterator upper_bound(const_reference value) const;
```

Effects: Returns an iterator to the first element whose key is greater than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

54.
```
template<typename KeyType, typename KeyValueCompare>
  const_iterator upper_bound(const KeyType & key, KeyValueCompare comp) const;
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Returns a const_iterator to the first element whose key according to the comparison functor is greater than key or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

55.
```
iterator find(const_reference value);
```

Effects: Finds an iterator to the first element whose value is "value" or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

56.
```
template<typename KeyType, typename KeyValueCompare>
  iterator find(const KeyType & key, KeyValueCompare comp);
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Finds an iterator to the first element whose key is "key" according to the comparison functor or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

57.
```
const_iterator find(const_reference value) const;
```

Effects: Finds a const_iterator to the first element whose value is "value" or end() if that element does not exist.

Complexity: Logarithmic.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: If the internal value_compare ordering function throws.

58.
```
template<typename KeyType, typename KeyValueCompare>
  const_iterator find(const KeyType & key, KeyValueCompare comp) const;
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Finds a const_iterator to the first element whose key is "key" according to the comparison functor or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

59.
```
std::pair< iterator, iterator > equal_range(const_reference value);
```

Effects: Finds a range containing all elements whose key is k or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

60.
```
template<typename KeyType, typename KeyValueCompare>
  std::pair< iterator, iterator >
  equal_range(const KeyType & key, KeyValueCompare comp);
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Finds a range containing all elements whose key is k according to the comparison functor or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

61.
```
std::pair< const_iterator, const_iterator >
equal_range(const_reference value) const;
```

Effects: Finds a range containing all elements whose key is k or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

62.
```
template<typename KeyType, typename KeyValueCompare>
  std::pair< const_iterator, const_iterator >
  equal_range(const KeyType & key, KeyValueCompare comp) const;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Finds a range containing all elements whose key is k according to the comparison functor or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

63.
```
std::pair< iterator, iterator >
bounded_range(const_reference lower_value, const_reference upper_value,
              bool left_closed, bool right_closed);
```

Requires: 'lower_value' must not be greater than 'upper_value'. If 'lower_value' == 'upper_value', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key) if left_closed, upper_bound(lower_key) otherwise

second = upper_bound(upper_key) if right_closed, lower_bound(upper_key) otherwise

Complexity: Logarithmic.

Throws: If the predicate throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_value and upper_value.

64.
```
template<typename KeyType, typename KeyValueCompare>
  std::pair< iterator, iterator >
  bounded_range(const KeyType & lower_key, const KeyType & upper_key,
                KeyValueCompare comp, bool left_closed, bool right_closed);
```

Requires: KeyValueCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. 'lower_key' must not be greater than 'upper_key' according to 'comp'. If 'lower_key' == 'upper_key', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key, comp) if left_closed, upper_bound(lower_key, comp) otherwise

second = upper_bound(upper_key, comp) if right_closed, lower_bound(upper_key, comp) otherwise

Complexity: Logarithmic.

Throws: If "comp" throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_key and upper_key.

65.
```
std::pair< const_iterator, const_iterator >
bounded_range(const_reference lower_value, const_reference upper_value,
              bool left_closed, bool right_closed) const;
```

Requires: 'lower_value' must not be greater than 'upper_value'. If 'lower_value' == 'upper_value', ('left_closed' || 'right_closed') must be false.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key) if left_closed, upper_bound(lower_key) otherwise

second = upper_bound(upper_key) if right_closed, lower_bound(upper_key) otherwise

Complexity: Logarithmic.

Throws: If the predicate throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_value and upper_value.

66.
```
template<typename KeyType, typename KeyValueCompare>
  std::pair< const_iterator, const_iterator >
  bounded_range(const KeyType & lower_key, const KeyType & upper_key,
                KeyValueCompare comp, bool left_closed, bool right_closed) const;
```

Requires: KeyValueCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. 'lower_key' must not be greater than 'upper_key' according to 'comp'. If 'lower_key' == 'upper_key', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key, comp) if left_closed, upper_bound(lower_key, comp) otherwise

second = upper_bound(upper_key, comp) if right_closed, lower_bound(upper_key, comp) otherwise

Complexity: Logarithmic.

Throws: If "comp" throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_key and upper_key.

67.
```
iterator iterator_to(reference value);
```

Requires: value must be an lvalue and shall be in a `treap_set` of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid iterator i belonging to the `treap_set` that points to the value

Complexity: Constant.

Throws: Nothing.

68.
```
const_iterator iterator_to(const_reference value) const;
```

Requires: value must be an lvalue and shall be in a `treap_set` of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid const_iterator i belonging to the `treap_set` that points to the value

Complexity: Constant.

Throws: Nothing.

69.
```
pointer unlink_leftmost_without_rebalance();
```

Effects: Unlinks the leftmost node from the tree.

Complexity: Average complexity is constant time.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: Nothing.

Notes: This function breaks the tree and the tree can only be used for more unlink_leftmost_without_rebalance calls. This function is normally used to achieve a step by step controlled destruction of the tree.

70.
```
void replace_node(iterator replace_this, reference with_this);
```

Requires: replace_this must be a valid iterator of *this and with_this must not be inserted in any tree.

Effects: Replaces replace_this in its position in the tree with with_this. The tree does not need to be rebalanced.

Complexity: Constant.

Throws: Nothing.

Note: This function will break container ordering invariants if with_this is not equivalent to *replace_this according to the ordering rules. This function is faster than erasing and inserting the node, since no rebalancing or comparison is needed.

71.
```
void rebalance();
```

Effects: Rebalances the tree.

Throws: Nothing.

Complexity: Linear.

72.
```
iterator rebalance_subtree(iterator root);
```

Requires: old_root is a node of a tree.

Effects: Rebalances the subtree rooted at old_root.

Returns: The new root of the subtree.

Throws: Nothing.

Complexity: Linear to the elements in the subtree.

73.
```
float balance_factor() const;
```

Returns: The balance factor (alpha) used in this tree

Throws: Nothing.

Complexity: Constant.

74.
```
void balance_factor(float new_alpha);
```

Requires: new_alpha must be a value between 0.5 and 1.0

Effects: Establishes a new balance factor (alpha) and rebalances the tree if the new balance factor is stricter (less) than the old factor.

Throws: Nothing.

Complexity: Linear to the elements in the subtree.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

`treap_set` public static functions

1.
```
static treap_set & container_from_end_iterator(iterator end_iterator);
```

   Precondition: end_iterator must be a valid end iterator of `treap_set`.

   Effects: Returns a const reference to the `treap_set` associated to the end iterator

   Throws: Nothing.

   Complexity: Constant.

2.
```
static const treap_set &
container_from_end_iterator(const_iterator end_iterator);
```

   Precondition: end_iterator must be a valid end const_iterator of `treap_set`.

   Effects: Returns a const reference to the `treap_set` associated to the end iterator

   Throws: Nothing.

   Complexity: Constant.

3.
```
static treap_set & container_from_iterator(iterator it);
```

   Precondition: it must be a valid iterator of set.

   Effects: Returns a reference to the set associated to the iterator

   Throws: Nothing.

   Complexity: Logarithmic.

4.
```
static const treap_set & container_from_iterator(const_iterator it);
```

   Precondition: it must be a valid const_iterator of set.

   Effects: Returns a const reference to the set associated to the iterator

   Throws: Nothing.

   Complexity: Logarithmic.

5.
```
static iterator s_iterator_to(reference value);
```

   Requires: value must be an lvalue and shall be in a `treap_set` of appropriate type. Otherwise the behavior is undefined.

   Effects: Returns: a valid iterator i belonging to the `treap_set` that points to the value

   Complexity: Constant.

   Throws: Nothing.

   Note: This static function is available only if the *value traits* is stateless.

6.
```
static const_iterator s_iterator_to(const_reference value);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Requires: value must be an lvalue and shall be in a `treap_set` of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid const_iterator i belonging to the `treap_set` that points to the value

Complexity: Constant.

Throws: Nothing.

Note: This static function is available only if the *value traits* is stateless.

7.
```
static void init_node(reference value);
```

Requires: value shall not be in a treap_set/treap_multiset.

Effects: init_node puts the hook of a value in a well-known default state.

Throws: Nothing.

Complexity: Constant time.

Note: This function puts the hook in the well-known default state used by auto_unlink and safe hooks.

Struct template make_treap_set

boost::intrusive::make_treap_set

Synopsis

```
// In header: <boost/intrusive/treap_set.hpp>

template<typename T, class... Options>
struct make_treap_set {
  // types
  typedef implementation_defined type;
};
```

Description

Helper metafunction to define a `treap_set` that yields to the same type when the same options (either explicitly or implicitly) are used.

Class template treap_multiset

boost::intrusive::treap_multiset

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/intrusive/treap_set.hpp>

template<typename T, class... Options> 
class treap_multiset {
public:
  // types
  typedef implementation_defined::value_type             value_type;            
  typedef implementation_defined::value_traits           value_traits;          
  typedef implementation_defined::pointer                pointer;               
  typedef implementation_defined::const_pointer          const_pointer;         
  typedef implementation_defined::reference              reference;             
  typedef implementation_defined::const_reference        const_reference;       
  typedef implementation_defined::difference_type        difference_type;       
  typedef implementation_defined::size_type              size_type;             
  typedef implementation_defined::value_compare          value_compare;         
  typedef implementation_defined::priority_compare       priority_compare;      
  typedef implementation_defined::key_compare            key_compare;           
  typedef implementation_defined::iterator               iterator;              
  typedef implementation_defined::const_iterator         const_iterator;        
  typedef implementation_defined::reverse_iterator       reverse_iterator;      
  typedef implementation_defined::const_reverse_iterator const_reverse_iterator;
  typedef implementation_defined::insert_commit_data     insert_commit_data;    
  typedef implementation_defined::node_traits            node_traits;           
  typedef implementation_defined::node                   node;                  
  typedef implementation_defined::node_ptr               node_ptr;              
  typedef implementation_defined::const_node_ptr         const_node_ptr;        
  typedef implementation_defined::node_algorithms        node_algorithms;       

  // construct/copy/destruct
  explicit treap_multiset(const value_compare & = value_compare(), 
                          const priority_compare & = priority_compare(), 
                          const value_traits & = value_traits());
  template<typename Iterator> 
    treap_multiset(Iterator, Iterator, 
                   const value_compare & = value_compare(), 
                   const priority_compare & = priority_compare(), 
                   const value_traits & = value_traits());
  treap_multiset(BOOST_RV_REF(treap_multiset));
  treap_multiset& operator=(BOOST_RV_REF(treap_multiset));
  ~treap_multiset();

  // public member functions
  iterator begin();
  const_iterator begin() const;
  const_iterator cbegin() const;
  iterator end();
  const_iterator end() const;
  const_iterator cend() const;
  iterator top();
  const_iterator top() const;
  const_iterator ctop() const;
  reverse_iterator rbegin();
  const_reverse_iterator rbegin() const;
  const_reverse_iterator crbegin() const;
  reverse_iterator rend();
  const_reverse_iterator rend() const;
  const_reverse_iterator crend() const;
  reverse_iterator rtop();
  const_reverse_iterator rtop() const;
  const_reverse_iterator crtop() const;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
key_compare key_comp() const;
value_compare value_comp() const;
priority_compare priority_comp() const;
bool empty() const;
size_type size() const;
void swap(treap_multiset &);
template<typename Cloner, typename Disposer>
  void clone_from(const treap_multiset &, Cloner, Disposer);
iterator insert(reference);
iterator insert(const_iterator, reference);
template<typename Iterator> void insert(Iterator, Iterator);
iterator insert_before(const_iterator, reference);
void push_back(reference);
void push_front(reference);
iterator erase(const_iterator);
iterator erase(const_iterator, const_iterator);
size_type erase(const_reference);
template<typename KeyType, typename KeyValueCompare>
  size_type erase(const KeyType &, KeyValueCompare);
template<typename Disposer>
  iterator erase_and_dispose(const_iterator, Disposer);
template<typename Disposer>
  iterator erase_and_dispose(const_iterator, const_iterator, Disposer);
template<typename Disposer>
  size_type erase_and_dispose(const_reference, Disposer);
template<typename KeyType, typename KeyValueCompare, typename Disposer>
  size_type erase_and_dispose(const KeyType &, KeyValueCompare, Disposer);
void clear();
template<typename Disposer> void clear_and_dispose(Disposer);
size_type count(const_reference) const;
template<typename KeyType, typename KeyValueCompare>
  size_type count(const KeyType &, KeyValueCompare) const;
iterator lower_bound(const_reference);
template<typename KeyType, typename KeyValueCompare>
  iterator lower_bound(const KeyType &, KeyValueCompare);
const_iterator lower_bound(const_reference) const;
template<typename KeyType, typename KeyValueCompare>
  const_iterator lower_bound(const KeyType &, KeyValueCompare) const;
iterator upper_bound(const_reference);
template<typename KeyType, typename KeyValueCompare>
  iterator upper_bound(const KeyType &, KeyValueCompare);
const_iterator upper_bound(const_reference) const;
template<typename KeyType, typename KeyValueCompare>
  const_iterator upper_bound(const KeyType &, KeyValueCompare) const;
iterator find(const_reference);
template<typename KeyType, typename KeyValueCompare>
  iterator find(const KeyType &, KeyValueCompare);
const_iterator find(const_reference) const;
template<typename KeyType, typename KeyValueCompare>
  const_iterator find(const KeyType &, KeyValueCompare) const;
std::pair< iterator, iterator > equal_range(const_reference);
template<typename KeyType, typename KeyValueCompare>
  std::pair< iterator, iterator >
  equal_range(const KeyType &, KeyValueCompare);
std::pair< const_iterator, const_iterator >
equal_range(const_reference) const;
template<typename KeyType, typename KeyValueCompare>
  std::pair< const_iterator, const_iterator >
  equal_range(const KeyType &, KeyValueCompare) const;
std::pair< iterator, iterator >
bounded_range(const_reference, const_reference, bool, bool);
template<typename KeyType, typename KeyValueCompare>
  std::pair< iterator, iterator >
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
    bounded_range(const KeyType &, const KeyType &, KeyValueCompare, bool, 
                  bool);
  std::pair< const_iterator, const_iterator > 
  bounded_range(const_reference, const_reference, bool, bool) const;
  template<typename KeyType, typename KeyValueCompare> 
    std::pair< const_iterator, const_iterator > 
    bounded_range(const KeyType &, const KeyType &, KeyValueCompare, bool, 
                  bool) const;
  iterator iterator_to(reference);
  const_iterator iterator_to(const_reference) const;
  pointer unlink_leftmost_without_rebalance();
  void replace_node(iterator, reference);
  void rebalance();
  iterator rebalance_subtree(iterator);
  float balance_factor() const;
  void balance_factor(float);

  // public static functions
  static treap_multiset & container_from_end_iterator(iterator);
  static const treap_multiset & container_from_end_iterator(const_iterator);
  static treap_multiset & container_from_iterator(iterator);
  static const treap_multiset & container_from_iterator(const_iterator);
  static iterator s_iterator_to(reference);
  static const_iterator s_iterator_to(const_reference);
  static void init_node(reference);

  // public data members
  static const bool constant_time_size;
};
```

Description

The class template treap_multiset is an intrusive container, that mimics most of the interface of std::treap_multiset as described in the C++ standard.

The template parameter `T` is the type to be managed by the container. The user can specify additional options and if no options are provided default options are used.

The container supports the following options: `base_hook<>/member_hook<>/value_traits<>`, `constant_time_size<>`, `size_type<>`, `compare<>` and `priority_compare<>`

`treap_multiset` public construct/copy/destruct

1.
```
explicit treap_multiset(const value_compare & cmp = value_compare(), 
                        const priority_compare & pcmp = priority_compare(), 
                        const value_traits & v_traits = value_traits());
```

   Effects: Constructs an empty `treap_multiset`.

   Complexity: Constant.

   Throws: If value_traits::node_traits::node constructor throws (this does not happen with predefined Boost.Intrusive hooks) or the copy constructor of the value_compare/priority_compare objects throw.

2.
```
template<typename Iterator> 
  treap_multiset(Iterator b, Iterator e, 
                 const value_compare & cmp = value_compare(), 
                 const priority_compare & pcmp = priority_compare(), 
                 const value_traits & v_traits = value_traits());
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Requires: Dereferencing iterator must yield an lvalue of type value_type. cmp must be a comparison function that induces a strict weak ordering.

Effects: Constructs an empty `treap_multiset` and inserts elements from [b, e).

Complexity: Linear in N if [b, e) is already sorted using comp and otherwise N * log N, where N is the distance between first and last

Throws: If value_traits::node_traits::node constructor throws (this does not happen with predefined Boost.Intrusive hooks) or the copy constructor/operator() of the value_compare/priority_compare objects throw.

3.
```
treap_multiset(BOOST_RV_REF(treap_multiset) x);
```

Effects: to-do

4.
```
treap_multiset& operator=(BOOST_RV_REF(treap_multiset) x);
```

Effects: to-do

5.
```
~treap_multiset();
```

Effects: Detaches all elements from this. The objects in the `treap_multiset` are not deleted (i.e. no destructors are called).

Complexity: Linear to the number of elements on the container. if it's a safe-mode or auto-unlink value_type. Constant time otherwise.

Throws: Nothing.

`treap_multiset` public member functions

1.
```
iterator begin();
```

Effects: Returns an iterator pointing to the beginning of the `treap_multiset`.

Complexity: Constant.

Throws: Nothing.

2.
```
const_iterator begin() const;
```

Effects: Returns a const_iterator pointing to the beginning of the `treap_multiset`.

Complexity: Constant.

Throws: Nothing.

3.
```
const_iterator cbegin() const;
```

Effects: Returns a const_iterator pointing to the beginning of the `treap_multiset`.

Complexity: Constant.

Throws: Nothing.

4.
```
iterator end();
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Returns an iterator pointing to the end of the `treap_multiset`.

Complexity: Constant.

Throws: Nothing.

5.
```
const_iterator end() const;
```

Effects: Returns a const_iterator pointing to the end of the `treap_multiset`.

Complexity: Constant.

Throws: Nothing.

6.
```
const_iterator cend() const;
```

Effects: Returns a const_iterator pointing to the end of the `treap_multiset`.

Complexity: Constant.

Throws: Nothing.

7.
```
iterator top();
```

Effects: Returns an iterator pointing to the highest priority object of the tree.

Complexity: Constant.

Throws: Nothing.

8.
```
const_iterator top() const;
```

Effects: Returns a const_iterator pointing to the highest priority object of the tree..

Complexity: Constant.

Throws: Nothing.

9.
```
const_iterator ctop() const;
```

Effects: Returns a const_iterator pointing to the highest priority object of the tree..

Complexity: Constant.

Throws: Nothing.

10.
```
reverse_iterator rbegin();
```

Effects: Returns a reverse_iterator pointing to the beginning of the reversed `treap_multiset`.

Complexity: Constant.

Throws: Nothing.

11.
```
const_reverse_iterator rbegin() const;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed `treap_multiset`.

Complexity: Constant.

Throws: Nothing.

12.
```
const_reverse_iterator crbegin() const;
```

Effects: Returns a const_reverse_iterator pointing to the beginning of the reversed `treap_multiset`.

Complexity: Constant.

Throws: Nothing.

13.
```
reverse_iterator rend();
```

Effects: Returns a reverse_iterator pointing to the end of the reversed `treap_multiset`.

Complexity: Constant.

Throws: Nothing.

14.
```
const_reverse_iterator rend() const;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed `treap_multiset`.

Complexity: Constant.

Throws: Nothing.

15.
```
const_reverse_iterator crend() const;
```

Effects: Returns a const_reverse_iterator pointing to the end of the reversed `treap_multiset`.

Complexity: Constant.

Throws: Nothing.

16.
```
reverse_iterator rtop();
```

Effects: Returns a reverse_iterator pointing to the highest priority object of the reversed tree.

Complexity: Constant.

Throws: Nothing.

17.
```
const_reverse_iterator rtop() const;
```

Effects: Returns a const_reverse_iterator pointing to the highest priority objec of the reversed tree.

Complexity: Constant.

Throws: Nothing.

18.
```
const_reverse_iterator crtop() const;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Returns a const_reverse_iterator pointing to the highest priority object of the reversed tree.

Complexity: Constant.

Throws: Nothing.

19.
```
key_compare key_comp() const;
```

Effects: Returns the key_compare object used by the `treap_multiset`.

Complexity: Constant.

Throws: If key_compare copy-constructor throws.

20.
```
value_compare value_comp() const;
```

Effects: Returns the value_compare object used by the `treap_multiset`.

Complexity: Constant.

Throws: If value_compare copy-constructor throws.

21.
```
priority_compare priority_comp() const;
```

Effects: Returns the priority_compare object used by the `treap_multiset`.

Complexity: Constant.

Throws: If priority_compare copy-constructor throws.

22.
```
bool empty() const;
```

Effects: Returns true if the container is empty.

Complexity: Constant.

Throws: Nothing.

23.
```
size_type size() const;
```

Effects: Returns the number of elements stored in the `treap_multiset`.

Complexity: Linear to elements contained in *this if, constant-time size option is enabled. Constant-time otherwise.

Throws: Nothing.

24.
```
void swap(treap_multiset & other);
```

Effects: Swaps the contents of two treap_multisets.

Complexity: Constant.

Throws: If the swap() call for the comparison functor found using ADL throws. Strong guarantee.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

25.
```
template<typename Cloner, typename Disposer> 
  void clone_from(const treap_multiset & src, Cloner cloner, 
                  Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw. Cloner should yield to nodes equivalent to the original nodes.

Effects: Erases all the elements from *this calling Disposer::operator()(pointer), clones all the elements from src calling Cloner::operator()(const_reference) and inserts them on *this. Copies the predicate from the source container.

If cloner throws, all cloned elements are unlinked and disposed calling Disposer::operator()(pointer).

Complexity: Linear to erased plus inserted elements.

Throws: If cloner throws or predicate copy assignment throws. Basic guarantee.

26.
```
iterator insert(reference value);
```

Requires: value must be an lvalue

Effects: Inserts value into the `treap_multiset`.

Returns: An iterator that points to the position where the new element was inserted.

Complexity: Average complexity for insert element is at most logarithmic.

Throws: If the internal value_compare or priority_compare ordering function throws. Strong guarantee.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

27.
```
iterator insert(const_iterator hint, reference value);
```

Requires: value must be an lvalue

Effects: Inserts x into the `treap_multiset`, using pos as a hint to where it will be inserted.

Returns: An iterator that points to the position where the new element was inserted.

Complexity: Logarithmic in general, but it is amortized constant time if t is inserted immediately before hint.

Throws: If internal value_compare or priority_compare ordering functions throw. Strong guarantee.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

28.
```
template<typename Iterator> void insert(Iterator b, Iterator e);
```

Requires: Dereferencing iterator must yield an lvalue of type value_type.

Effects: Inserts a range into the `treap_multiset`.

Returns: An iterator that points to the position where the new element was inserted.

Complexity: Insert range is in general O(N * log(N)), where N is the size of the range. However, it is linear in N if the range is already sorted by value_comp().

Throws: If internal value_compare or priority_compare ordering functions throw. Basic guarantee.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

29.
```
iterator insert_before(const_iterator pos, reference value);
```

Requires: value must be an lvalue, "pos" must be a valid iterator (or end) and must be the succesor of value once inserted according to the predicate

Effects: Inserts x into the treap before "pos".

Complexity: Constant time.

Throws: If the internal priority_compare function throws. Strong guarantee.

Note: This function does not check preconditions so if "pos" is not the successor of "value" treap ordering invariant will be broken. This is a low-level function to be used only for performance reasons by advanced users.

30.
```
void push_back(reference value);
```

Requires: value must be an lvalue, and it must be no less than the greatest inserted key.

Effects: Inserts x into the treap in the last position.

Complexity: Constant time.

Throws: If the internal priority_compare function throws. Strong guarantee.

Note: This function does not check preconditions so if value is less than the greatest inserted key treap ordering invariant will be broken. This function is slightly more efficient than using "insert_before". This is a low-level function to be used only for performance reasons by advanced users.

31.
```
void push_front(reference value);
```

Requires: value must be an lvalue, and it must be no greater than the minimum inserted key

Effects: Inserts x into the treap in the first position.

Complexity: Constant time.

Throws: If the internal priority_compare function throws. Strong guarantee.

Note: This function does not check preconditions so if value is greater than the minimum inserted key treap ordering invariant will be broken. This function is slightly more efficient than using "insert_before". This is a low-level function to be used only for performance reasons by advanced users.

32.
```
iterator erase(const_iterator i);
```

Effects: Erases the element pointed to by pos.

Complexity: Average complexity is constant time.

Returns: An iterator to the element after the erased element.

Throws: If the internal priority_compare function throws. Strong guarantee.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

33.
```
iterator erase(const_iterator b, const_iterator e);
```

Effects: Erases the range pointed to by b end e.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Returns: An iterator to the element after the erased elements.

Complexity: Average complexity for erase range is at most O(log(size() + N)), where N is the number of elements in the range.

Throws: If the internal priority_compare function throws. Basic guarantee.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

34.
```
size_type erase(const_reference value);
```

Effects: Erases all the elements with the given value.

Returns: The number of erased elements.

Complexity: O(log(size() + this->count(value)).

Throws: If the internal value_compare or priority_compare ordering functiona throw. Basic guarantee.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

35.
```
template<typename KeyType, typename KeyValueCompare>
  size_type erase(const KeyType & key, KeyValueCompare comp);
```

Effects: Erases all the elements that compare equal with the given key and the given comparison functor.

Returns: The number of erased elements.

Complexity: O(log(size() + this->count(key, comp)).

Throws: If comp or internal priority_compare ordering functions throw. Basic guarantee.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

36.
```
template<typename Disposer>
  iterator erase_and_dispose(const_iterator i, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Returns: An iterator to the element after the erased element.

Effects: Erases the element pointed to by pos. Disposer::operator()(pointer) is called for the removed element.

Complexity: Average complexity for erase element is constant time.

Throws: If the internal priority_compare function throws. Strong guarantee.

Note: Invalidates the iterators to the erased elements.

37.
```
template<typename Disposer>
  iterator erase_and_dispose(const_iterator b, const_iterator e,
                             Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Returns: An iterator to the element after the erased elements.

Effects: Erases the range pointed to by b end e. Disposer::operator()(pointer) is called for the removed elements.

Complexity: Average complexity for erase range is at most O(log(size() + N)), where N is the number of elements in the range.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: If the internal priority_compare function throws. Basic guarantee.

Note: Invalidates the iterators to the erased elements.

38.
```
template<typename Disposer>
  size_type erase_and_dispose(const_reference value, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases all the elements with the given value. Disposer::operator()(pointer) is called for the removed elements.

Returns: The number of erased elements.

Complexity: O(log(size() + this->count(value)).

Throws: If the internal value_compare ordering function throws. Basic guarantee.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

39.
```
template<typename KeyType, typename KeyValueCompare, typename Disposer>
  size_type erase_and_dispose(const KeyType & key, KeyValueCompare comp,
                              Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases all the elements with the given key. according to the comparison functor "comp". Disposer::operator()(pointer) is called for the removed elements.

Returns: The number of erased elements.

Complexity: O(log(size() + this->count(key, comp)).

Throws: If comp or internal priority_compare ordering functions throw. Basic guarantee.

Note: Invalidates the iterators to the erased elements.

40.
```
void clear();
```

Effects: Erases all the elements of the container.

Complexity: Linear to the number of elements on the container. if it's a safe-mode or auto-unlink value_type. Constant time otherwise.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

41.
```
template<typename Disposer> void clear_and_dispose(Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases all the elements of the container.

Complexity: Linear to the number of elements on the container. Disposer::operator()(pointer) is called for the removed elements.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

42.
```
size_type count(const_reference value) const;
```

Effects: Returns the number of contained elements with the given key

Complexity: Logarithmic to the number of elements contained plus lineal to number of objects with the given key.

Throws: If the internal value_compare ordering function throws.

43.
```
template<typename KeyType, typename KeyValueCompare>
  size_type count(const KeyType & key, KeyValueCompare comp) const;
```

Effects: Returns the number of contained elements with the same key compared with the given comparison functor.

Complexity: Logarithmic to the number of elements contained plus lineal to number of objects with the given key.

Throws: If comp ordering function throws.

44.
```
iterator lower_bound(const_reference value);
```

Effects: Returns an iterator to the first element whose key is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

45.
```
template<typename KeyType, typename KeyValueCompare>
  iterator lower_bound(const KeyType & key, KeyValueCompare comp);
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Returns an iterator to the first element whose key according to the comparison functor is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

46.
```
const_iterator lower_bound(const_reference value) const;
```

Effects: Returns a const iterator to the first element whose key is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

47.
```
template<typename KeyType, typename KeyValueCompare>
  const_iterator lower_bound(const KeyType & key, KeyValueCompare comp) const;
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Returns a const_iterator to the first element whose key according to the comparison functor is not less than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

48.
```
iterator upper_bound(const_reference value);
```

Effects: Returns an iterator to the first element whose key is greater than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

49.
```
template<typename KeyType, typename KeyValueCompare>
  iterator upper_bound(const KeyType & key, KeyValueCompare comp);
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Returns an iterator to the first element whose key according to the comparison functor is greater than key or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

50.
```
const_iterator upper_bound(const_reference value) const;
```

Effects: Returns an iterator to the first element whose key is greater than k or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

51.
```
template<typename KeyType, typename KeyValueCompare>
  const_iterator upper_bound(const KeyType & key, KeyValueCompare comp) const;
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Returns a const_iterator to the first element whose key according to the comparison functor is greater than key or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

52.
```
iterator find(const_reference value);
```

Effects: Finds an iterator to the first element whose value is "value" or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

53.
```
template<typename KeyType, typename KeyValueCompare>
  iterator find(const KeyType & key, KeyValueCompare comp);
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Finds an iterator to the first element whose key is "key" according to the comparison functor or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

54.
```
const_iterator find(const_reference value) const;
```

Effects: Finds a const_iterator to the first element whose value is "value" or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

55.
```
template<typename KeyType, typename KeyValueCompare>
  const_iterator find(const KeyType & key, KeyValueCompare comp) const;
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Finds a const_iterator to the first element whose key is "key" according to the comparison functor or end() if that element does not exist.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

56.
```
std::pair< iterator, iterator > equal_range(const_reference value);
```

Effects: Finds a range containing all elements whose key is k or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

57.
```
template<typename KeyType, typename KeyValueCompare>
  std::pair< iterator, iterator >
  equal_range(const KeyType & key, KeyValueCompare comp);
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Finds a range containing all elements whose key is k according to the comparison functor or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

58.
```
std::pair< const_iterator, const_iterator >
equal_range(const_reference value) const;
```

Effects: Finds a range containing all elements whose key is k or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: If the internal value_compare ordering function throws.

59.
```
template<typename KeyType, typename KeyValueCompare>
  std::pair< const_iterator, const_iterator >
  equal_range(const KeyType & key, KeyValueCompare comp) const;
```

Requires: comp must imply the same element order as value_compare. Usually key is the part of the value_type that is used in the ordering functor.

Effects: Finds a range containing all elements whose key is k according to the comparison functor or an empty range that indicates the position where those elements would be if they there is no elements with key k.

Complexity: Logarithmic.

Throws: If comp ordering function throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

60.
```
std::pair< iterator, iterator >
bounded_range(const_reference lower_value, const_reference upper_value,
              bool left_closed, bool right_closed);
```

Requires: 'lower_value' must not be greater than 'upper_value'. If 'lower_value' == 'upper_value', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key) if left_closed, upper_bound(lower_key) otherwise

second = upper_bound(upper_key) if right_closed, lower_bound(upper_key) otherwise

Complexity: Logarithmic.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: If the predicate throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_value and upper_value.

61.
```
template<typename KeyType, typename KeyValueCompare> 
  std::pair< iterator, iterator > 
  bounded_range(const KeyType & lower_key, const KeyType & upper_key, 
                KeyValueCompare comp, bool left_closed, bool right_closed);
```

Requires: KeyValueCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. 'lower_key' must not be greater than 'upper_key' according to 'comp'. If 'lower_key' == 'upper_key', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key, comp) if left_closed, upper_bound(lower_key, comp) otherwise

second = upper_bound(upper_key, comp) if right_closed, lower_bound(upper_key, comp) otherwise

Complexity: Logarithmic.

Throws: If "comp" throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_key and upper_key.

62.
```
std::pair< const_iterator, const_iterator > 
bounded_range(const_reference lower_value, const_reference upper_value, 
              bool left_closed, bool right_closed) const;
```

Requires: 'lower_value' must not be greater than 'upper_value'. If 'lower_value' == 'upper_value', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key) if left_closed, upper_bound(lower_key) otherwise

second = upper_bound(upper_key) if right_closed, lower_bound(upper_key) otherwise

Complexity: Logarithmic.

Throws: If the predicate throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_value and upper_value.

63.
```
template<typename KeyType, typename KeyValueCompare> 
  std::pair< const_iterator, const_iterator > 
  bounded_range(const KeyType & lower_key, const KeyType & upper_key, 
                KeyValueCompare comp, bool left_closed, bool right_closed) const;
```

Requires: KeyValueCompare is a function object that induces a strict weak ordering compatible with the strict weak ordering used to create the the tree. 'lower_key' must not be greater than 'upper_key' according to 'comp'. If 'lower_key' == 'upper_key', ('left_closed' || 'right_closed') must be false.

Effects: Returns an a pair with the following criteria:

first = lower_bound(lower_key, comp) if left_closed, upper_bound(lower_key, comp) otherwise

second = upper_bound(upper_key, comp) if right_closed, lower_bound(upper_key, comp) otherwise

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Logarithmic.

Throws: If "comp" throws.

Note: This function can be more efficient than calling upper_bound and lower_bound for lower_key and upper_key.

64.
```
iterator iterator_to(reference value);
```

Requires: value must be an lvalue and shall be in a `treap_multiset` of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid iterator i belonging to the `treap_multiset` that points to the value

Complexity: Constant.

Throws: Nothing.

65.
```
const_iterator iterator_to(const_reference value) const;
```

Requires: value must be an lvalue and shall be in a `treap_multiset` of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid const_iterator i belonging to the `treap_multiset` that points to the value

Complexity: Constant.

Throws: Nothing.

66.
```
pointer unlink_leftmost_without_rebalance();
```

Effects: Unlinks the leftmost node from the tree.

Complexity: Average complexity is constant time.

Throws: Nothing.

Notes: This function breaks the tree and the tree can only be used for more unlink_leftmost_without_rebalance calls. This function is normally used to achieve a step by step controlled destruction of the tree.

67.
```
void replace_node(iterator replace_this, reference with_this);
```

Requires: replace_this must be a valid iterator of *this and with_this must not be inserted in any tree.

Effects: Replaces replace_this in its position in the tree with with_this. The tree does not need to be rebalanced.

Complexity: Constant.

Throws: Nothing.

Note: This function will break container ordering invariants if with_this is not equivalent to *replace_this according to the ordering rules. This function is faster than erasing and inserting the node, since no rebalancing or comparison is needed.

68.
```
void rebalance();
```

Effects: Rebalances the tree.

Throws: Nothing.

Complexity: Linear.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

69.
```
iterator rebalance_subtree(iterator root);
```

Requires: old_root is a node of a tree.

Effects: Rebalances the subtree rooted at old_root.

Returns: The new root of the subtree.

Throws: Nothing.

Complexity: Linear to the elements in the subtree.

70.
```
float balance_factor() const;
```

Returns: The balance factor (alpha) used in this tree

Throws: Nothing.

Complexity: Constant.

71.
```
void balance_factor(float new_alpha);
```

Requires: new_alpha must be a value between 0.5 and 1.0

Effects: Establishes a new balance factor (alpha) and rebalances the tree if the new balance factor is stricter (less) than the old factor.

Throws: Nothing.

Complexity: Linear to the elements in the subtree.

`treap_multiset` public static functions

1.
```
static treap_multiset & container_from_end_iterator(iterator end_iterator);
```

Precondition: end_iterator must be a valid end iterator of `treap_multiset`.

Effects: Returns a const reference to the `treap_multiset` associated to the end iterator

Throws: Nothing.

Complexity: Constant.

2.
```
static const treap_multiset &
container_from_end_iterator(const_iterator end_iterator);
```

Precondition: end_iterator must be a valid end const_iterator of `treap_multiset`.

Effects: Returns a const reference to the `treap_multiset` associated to the end iterator

Throws: Nothing.

Complexity: Constant.

3.
```
static treap_multiset & container_from_iterator(iterator it);
```

Precondition: it must be a valid iterator of multiset.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Returns a const reference to the multiset associated to the iterator

Throws: Nothing.

Complexity: Constant.

4.
```
static const treap_multiset & container_from_iterator(const_iterator it);
```

Precondition: it must be a valid const_iterator of multiset.

Effects: Returns a const reference to the multiset associated to the iterator

Throws: Nothing.

Complexity: Constant.

5.
```
static iterator s_iterator_to(reference value);
```

Requires: value must be an lvalue and shall be in a `treap_multiset` of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid iterator i belonging to the `treap_multiset` that points to the value

Complexity: Constant.

Throws: Nothing.

Note: This static function is available only if the *value traits* is stateless.

6.
```
static const_iterator s_iterator_to(const_reference value);
```

Requires: value must be an lvalue and shall be in a `treap_multiset` of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid const_iterator i belonging to the `treap_multiset` that points to the value

Complexity: Constant.

Throws: Nothing.

Note: This static function is available only if the *value traits* is stateless.

7.
```
static void init_node(reference value);
```

Requires: value shall not be in a treap_multiset/treap_multiset.

Effects: init_node puts the hook of a value in a well-known default state.

Throws: Nothing.

Complexity: Constant time.

Note: This function puts the hook in the well-known default state used by auto_unlink and safe hooks.

Struct template make_treap_multiset

boost::intrusive::make_treap_multiset

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/intrusive/treap_set.hpp>

template<typename T, class... Options>
struct make_treap_multiset {
  // types
  typedef implementation_defined type;
};
```

Description

Helper metafunction to define a `treap_multiset` that yields to the same type when the same options (either explicitly or implicitly) are used.

Header <boost/intrusive/trivial_value_traits.hpp>

```
namespace boost {
  namespace intrusive {
    template<typename NodeTraits, link_mode_type LinkMode = normal_link>
      struct trivial_value_traits;
  }
}
```

Struct template trivial_value_traits

boost::intrusive::trivial_value_traits

Synopsis

```
// In header: <boost/intrusive/trivial_value_traits.hpp>

template<typename NodeTraits, link_mode_type LinkMode = normal_link>
struct trivial_value_traits {
  // types
  typedef NodeTraits                    node_traits;
  typedef node_traits::node_ptr       node_ptr;
  typedef node_traits::const_node_ptr const_node_ptr;
  typedef node_traits::node           value_type;
  typedef node_ptr                      pointer;
  typedef const_node_ptr                const_pointer;

  // public static functions
  static node_ptr to_node_ptr(value_type &);
  static const_node_ptr to_node_ptr(const value_type &);
  static const pointer & to_value_ptr(const node_ptr &);
  static const const_pointer & to_value_ptr(const const_node_ptr &);

  // public data members
  static const link_mode_type link_mode;
};
```

Description

This value traits template is used to create value traits from user defined node traits where value_traits::value_type and node_traits::node should be equal

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

`trivial_value_traits` public static functions

1.
   ```
   static node_ptr to_node_ptr(value_type & value);
   ```

2.
   ```
   static const_node_ptr to_node_ptr(const value_type & value);
   ```

3.
   ```
   static const pointer & to_value_ptr(const node_ptr & n);
   ```

4.
   ```
   static const const_pointer & to_value_ptr(const const_node_ptr & n);
   ```

Header <boost/intrusive/unordered_set.hpp>

```
namespace boost {
  namespace intrusive {
    template<typename T, class... Options> class unordered_set;

    template<typename T, class... Options> struct make_unordered_set;

    template<typename T, class... Options> class unordered_multiset;

    template<typename T, class... Options> struct make_unordered_multiset;
  }
}
```

Class template unordered_set

boost::intrusive::unordered_set

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/intrusive/unordered_set.hpp>

template<typename T, class... Options> 
class unordered_set {
public:
  // types
  typedef implementation_defined::value_type           value_type;          
  typedef implementation_defined::value_traits         value_traits;        
  typedef implementation_defined::bucket_traits        bucket_traits;       
  typedef implementation_defined::pointer              pointer;             
  typedef implementation_defined::const_pointer        const_pointer;       
  typedef implementation_defined::reference            reference;           
  typedef implementation_defined::const_reference      const_reference;     
  typedef implementation_defined::difference_type      difference_type;     
  typedef implementation_defined::size_type            size_type;           
  typedef implementation_defined::key_type             key_type;            
  typedef implementation_defined::key_equal            key_equal;           
  typedef implementation_defined::hasher               hasher;              
  typedef implementation_defined::bucket_type          bucket_type;         
  typedef implementation_defined::bucket_ptr           bucket_ptr;          
  typedef implementation_defined::iterator             iterator;            
  typedef implementation_defined::const_iterator       const_iterator;      
  typedef implementation_defined::insert_commit_data   insert_commit_data;  
  typedef implementation_defined::local_iterator       local_iterator;      
  typedef implementation_defined::const_local_iterator const_local_iterator;
  typedef implementation_defined::node_traits          node_traits;         
  typedef implementation_defined::node                 node;                
  typedef implementation_defined::node_ptr             node_ptr;            
  typedef implementation_defined::const_node_ptr       const_node_ptr;      
  typedef implementation_defined::node_algorithms      node_algorithms;     

  // construct/copy/destruct
  explicit unordered_set(const bucket_traits &, const hasher & = hasher(), 
                         const key_equal & = key_equal(), 
                         const value_traits & = value_traits());
  template<typename Iterator> 
    unordered_set(Iterator, Iterator, const bucket_traits &, 
                  const hasher & = hasher(), const key_equal & = key_equal(), 
                  const value_traits & = value_traits());
  unordered_set(BOOST_RV_REF(unordered_set));
  unordered_set& operator=(BOOST_RV_REF(unordered_set));
  ~unordered_set();

  // public member functions
  iterator begin();
  const_iterator begin() const;
  const_iterator cbegin() const;
  iterator end();
  const_iterator end() const;
  const_iterator cend() const;
  hasher hash_function() const;
  key_equal key_eq() const;
  bool empty() const;
  size_type size() const;
  void swap(unordered_set &);
  template<typename Cloner, typename Disposer> 
    void clone_from(const unordered_set &, Cloner, Disposer);
  std::pair< iterator, bool > insert(reference);
  template<typename Iterator> void insert(Iterator, Iterator);
  template<typename KeyType, typename KeyHasher, typename KeyValueEqual>
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
  std::pair< iterator, bool >
  insert_check(const KeyType &, KeyHasher, KeyValueEqual,
               insert_commit_data &);
iterator insert_commit(reference, const insert_commit_data &);
void erase(const_iterator);
void erase(const_iterator, const_iterator);
size_type erase(const_reference);
template<typename KeyType, typename KeyHasher, typename KeyValueEqual>
  size_type erase(const KeyType &, KeyHasher, KeyValueEqual);
template<typename Disposer> void erase_and_dispose(const_iterator, Disposer);
template<typename Disposer>
  void erase_and_dispose(const_iterator, const_iterator, Disposer);
template<typename Disposer>
  size_type erase_and_dispose(const_reference, Disposer);
template<typename KeyType, typename KeyHasher, typename KeyValueEqual,
         typename Disposer>
  size_type erase_and_dispose(const KeyType &, KeyHasher, KeyValueEqual,
                              Disposer);
void clear();
template<typename Disposer> void clear_and_dispose(Disposer);
size_type count(const_reference) const;
template<typename KeyType, typename KeyHasher, typename KeyValueEqual>
  size_type count(const KeyType &, KeyHasher, KeyValueEqual) const;
iterator find(const_reference);
template<typename KeyType, typename KeyHasher, typename KeyValueEqual>
  iterator find(const KeyType &, KeyHasher, KeyValueEqual);
const_iterator find(const_reference) const;
template<typename KeyType, typename KeyHasher, typename KeyValueEqual>
  const_iterator find(const KeyType &, KeyHasher, KeyValueEqual) const;
std::pair< iterator, iterator > equal_range(const_reference);
template<typename KeyType, typename KeyHasher, typename KeyValueEqual>
  std::pair< iterator, iterator >
  equal_range(const KeyType &, KeyHasher, KeyValueEqual);
std::pair< const_iterator, const_iterator >
equal_range(const_reference) const;
template<typename KeyType, typename KeyHasher, typename KeyValueEqual>
  std::pair< const_iterator, const_iterator >
  equal_range(const KeyType &, KeyHasher, KeyValueEqual) const;
iterator iterator_to(reference);
const_iterator iterator_to(const_reference) const;
local_iterator local_iterator_to(reference);
const_local_iterator local_iterator_to(const_reference) const;
size_type bucket_count() const;
size_type bucket_size(size_type) const;
size_type bucket(const value_type &) const;
template<typename KeyType, typename KeyHasher>
  size_type bucket(const KeyType &, KeyHasher) const;
bucket_ptr bucket_pointer() const;
local_iterator begin(size_type);
const_local_iterator begin(size_type) const;
const_local_iterator cbegin(size_type) const;
local_iterator end(size_type);
const_local_iterator end(size_type) const;
const_local_iterator cend(size_type) const;
void rehash(const bucket_traits &);
bool incremental_rehash(bool = true);
bool incremental_rehash(const bucket_traits &);
size_type split_count() const;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
  // public static functions
  static local_iterator s_local_iterator_to(reference);
  static const_local_iterator s_local_iterator_to(const_reference);
  static size_type suggested_upper_bucket_count(size_type);
  static size_type suggested_lower_bucket_count(size_type);
};
```

Description

The class template unordered_set is an intrusive container, that mimics most of the interface of std::tr1::unordered_set as described in the C++ TR1.

unordered_set is a semi-intrusive container: each object to be stored in the container must contain a proper hook, but the container also needs additional auxiliary memory to work: unordered_set needs a pointer to an array of type `bucket_type` to be passed in the constructor. This bucket array must have at least the same lifetime as the container. This makes the use of unordered_set more complicated than purely intrusive containers. `bucket_type` is default-constructible, copyable and assignable

The template parameter `T` is the type to be managed by the container. The user can specify additional options and if no options are provided default options are used.

The container supports the following options: `base_hook<>/member_hook<>/value_traits<>`, `constant_time_size<>`, `size_type<>`, `hash<>` and `equal<>` `bucket_traits<>`, `power_2_buckets<>` and `cache_begin<>`.

unordered_set only provides forward iterators but it provides 4 iterator types: iterator and const_iterator to navigate through the whole container and local_iterator and const_local_iterator to navigate through the values stored in a single bucket. Local iterators are faster and smaller.

It's not recommended to use non constant-time size unordered_sets because several key functions, like "empty()", become non-constant time functions. Non constant-time size unordered_sets are mainly provided to support auto-unlink hooks.

unordered_set, unlike std::unordered_set, does not make automatic rehashings nor offers functions related to a load factor. Rehashing can be explicitly requested and the user must provide a new bucket array that will be used from that moment.

Since no automatic rehashing is done, iterators are never invalidated when inserting or erasing elements. Iterators are only invalidated when rehasing.

`unordered_set` public construct/copy/destruct

1.
```
explicit unordered_set(const bucket_traits & b_traits,
                       const hasher & hash_func = hasher(),
                       const key_equal & equal_func = key_equal(),
                       const value_traits & v_traits = value_traits());
```

Requires: buckets must not be being used by any other resource.

Effects: Constructs an empty unordered_set_impl, storing a reference to the bucket array and copies of the hasher and equal functors.

Complexity: Constant.

Throws: If value_traits::node_traits::node constructor throws (this does not happen with predefined Boost.Intrusive hooks) or the copy constructor or invocation of Hash or Equal throws.

Notes: buckets array must be disposed only after *this is disposed.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

2.
```
template<typename Iterator>
  unordered_set(Iterator b, Iterator e, const bucket_traits & b_traits,
                const hasher & hash_func = hasher(),
                const key_equal & equal_func = key_equal(),
                const value_traits & v_traits = value_traits());
```

Requires: buckets must not be being used by any other resource and Dereferencing iterator must yield an lvalue of type value_type.

Effects: Constructs an empty unordered_set and inserts elements from [b, e).

Complexity: If N is std::distance(b, e): Average case is O(N) (with a good hash function and with buckets_len >= N),worst case O(N2).

Throws: If value_traits::node_traits::node constructor throws (this does not happen with predefined Boost.Intrusive hooks) or the copy constructor or invocation of hasher or key_equal throws.

Notes: buckets array must be disposed only after *this is disposed.

3.
```
unordered_set(BOOST_RV_REF(unordered_set) x);
```

Effects: to-do

4.
```
unordered_set& operator=(BOOST_RV_REF(unordered_set) x);
```

Effects: to-do

5.
```
~unordered_set();
```

Effects: Detaches all elements from this. The objects in the unordered_set are not deleted (i.e. no destructors are called).

Complexity: Linear to the number of elements in the unordered_set, if it's a safe-mode or auto-unlink value. Otherwise constant.

Throws: Nothing.

`unordered_set` public member functions

1.
```
iterator begin();
```

Effects: Returns an iterator pointing to the beginning of the unordered_set.

Complexity: Constant time if cache_begin<> is true. Amortized constant time with worst case (empty unordered_set) O(this->bucket_count())

Throws: Nothing.

2.
```
const_iterator begin() const;
```

Effects: Returns a const_iterator pointing to the beginning of the unordered_set.

Complexity: Constant time if cache_begin<> is true. Amortized constant time with worst case (empty unordered_set) O(this->bucket_count())

Throws: Nothing.

3.
```
const_iterator cbegin() const;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Returns a const_iterator pointing to the beginning of the `unordered_set`.

Complexity: Constant time if `cache_begin`<> is true. Amortized constant time with worst case (empty `unordered_set`) O(this->bucket_count())

Throws: Nothing.

4.
```
iterator end();
```

Effects: Returns an iterator pointing to the end of the `unordered_set`.

Complexity: Constant.

Throws: Nothing.

5.
```
const_iterator end() const;
```

Effects: Returns a const_iterator pointing to the end of the `unordered_set`.

Complexity: Constant.

Throws: Nothing.

6.
```
const_iterator cend() const;
```

Effects: Returns a const_iterator pointing to the end of the `unordered_set`.

Complexity: Constant.

Throws: Nothing.

7.
```
hasher hash_function() const;
```

Effects: Returns the hasher object used by the `unordered_set`.

Complexity: Constant.

Throws: If hasher copy-constructor throws.

8.
```
key_equal key_eq() const;
```

Effects: Returns the key_equal object used by the `unordered_set`.

Complexity: Constant.

Throws: If key_equal copy-constructor throws.

9.
```
bool empty() const;
```

Effects: Returns true if the container is empty.

Complexity: if constant-time size and `cache_last` options are disabled, average constant time (worst case, with empty() == true: O(this->bucket_count()). Otherwise constant.

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

10.
```
size_type size() const;
```

Effects: Returns the number of elements stored in the `unordered_set`.

Complexity: Linear to elements contained in *this if constant-time size option is disabled. Constant-time otherwise.

Throws: Nothing.

11.
```
void swap(unordered_set & other);
```

Requires: the hasher and the equality function unqualified swap call should not throw.

Effects: Swaps the contents of two unordered_sets. Swaps also the contained bucket array and equality and hasher functors.

Complexity: Constant.

Throws: If the swap() call for the comparison or hash functors found using ADL throw. Basic guarantee.

12.
```
template<typename Cloner, typename Disposer>
  void clone_from(const unordered_set & src, Cloner cloner, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw. Cloner should yield to nodes that compare equal and produce the same hash than the original node.

Effects: Erases all the elements from *this calling Disposer::operator()(pointer), clones all the elements from src calling Cloner::operator()(const_reference) and inserts them on *this. The hash function and the equality predicate are copied from the source.

If `store_hash` option is true, this method does not use the hash function.

If any operation throws, all cloned elements are unlinked and disposed calling Disposer::operator()(pointer).

Complexity: Linear to erased plus inserted elements.

Throws: If cloner or hasher throw or hash or equality predicate copying throws. Basic guarantee.

13.
```
std::pair< iterator, bool > insert(reference value);
```

Requires: value must be an lvalue

Effects: Tries to inserts value into the `unordered_set`.

Returns: If the value is not already present inserts it and returns a pair containing the iterator to the new value and true. If there is an equivalent value returns a pair containing an iterator to the already present value and false.

Complexity: Average case O(1), worst case O(this->size()).

Throws: If the internal hasher or the equality functor throws. Strong guarantee.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

14.
```
template<typename Iterator> void insert(Iterator b, Iterator e);
```

Requires: Dereferencing iterator must yield an lvalue of type value_type.

Effects: Equivalent to this->insert(t) for each element in [b, e).

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Average case O(N), where N is std::distance(b, e). Worst case O(N*this->size()).

Throws: If the internal hasher or the equality functor throws. Basic guarantee.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

15.
```
template<typename KeyType, typename KeyHasher, typename KeyValueEqual> 
  std::pair< iterator, bool > 
  insert_check(const KeyType & key, KeyHasher hasher, 
               KeyValueEqual key_value_equal, 
               insert_commit_data & commit_data);
```

Requires: "hasher" must be a hash function that induces the same hash values as the stored hasher. The difference is that "hasher" hashes the given key instead of the value_type.

"key_value_equal" must be a equality function that induces the same equality as key_equal. The difference is that "key_value_equal" compares an arbitrary key with the contained values.

Effects: Checks if a value can be inserted in the `unordered_set`, using a user provided key instead of the value itself.

Returns: If there is an equivalent value returns a pair containing an iterator to the already present value and false. If the value can be inserted returns true in the returned pair boolean and fills "commit_data" that is meant to be used with the "insert_commit" function.

Complexity: Average case O(1), worst case O(this->size()).

Throws: If hasher or key_value_equal throw. Strong guarantee.

Notes: This function is used to improve performance when constructing a value_type is expensive: if there is an equivalent value the constructed object must be discarded. Many times, the part of the node that is used to impose the hash or the equality is much cheaper to construct than the value_type and this function offers the possibility to use that the part to check if the insertion will be successful.

If the check is successful, the user can construct the value_type and use "insert_commit" to insert the object in constant-time.

"commit_data" remains valid for a subsequent "insert_commit" only if no more objects are inserted or erased from the `unordered_set`.

After a successful rehashing insert_commit_data remains valid.

16.
```
iterator insert_commit(reference value, 
                       const insert_commit_data & commit_data);
```

Requires: value must be an lvalue of type value_type. commit_data must have been obtained from a previous call to "insert_check". No objects should have been inserted or erased from the `unordered_set` between the "insert_check" that filled "commit_data" and the call to "insert_commit".

Effects: Inserts the value in the `unordered_set` using the information obtained from the "commit_data" that a previous "insert_check" filled.

Returns: An iterator to the newly inserted object.

Complexity: Constant time.

Throws: Nothing.

Notes: This function has only sense if a "insert_check" has been previously executed to fill "commit_data". No value should be inserted or erased between the "insert_check" and "insert_commit" calls.

After a successful rehashing insert_commit_data remains valid.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

17.
```
void erase(const_iterator i);
```

Effects: Erases the element pointed to by i.

Complexity: Average case O(1), worst case O(this->size()).

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased element. No destructors are called.

18.
```
void erase(const_iterator b, const_iterator e);
```

Effects: Erases the range pointed to by b end e.

Complexity: Average case O(std::distance(b, e)), worst case O(this->size()).

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

19.
```
size_type erase(const_reference value);
```

Effects: Erases all the elements with the given value.

Returns: The number of erased elements.

Complexity: Average case O(this->count(value)). Worst case O(this->size()).

Throws: If the internal hasher or the equality functor throws. Basic guarantee.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

20.
```
template<typename KeyType, typename KeyHasher, typename KeyValueEqual>
  size_type erase(const KeyType & key, KeyHasher hash_func,
                  KeyValueEqual equal_func);
```

Requires: "hasher" must be a hash function that induces the same hash values as the stored hasher. The difference is that "hasher" hashes the given key instead of the value_type.

"key_value_equal" must be a equality function that induces the same equality as key_equal. The difference is that "key_value_equal" compares an arbitrary key with the contained values.

Effects: Erases all the elements that have the same hash and compare equal with the given key.

Returns: The number of erased elements.

Complexity: Average case O(this->count(value)). Worst case O(this->size()).

Throws: If hash_func or equal_func throw. Basic guarantee.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

21.
```
template<typename Disposer>
  void erase_and_dispose(const_iterator i, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases the element pointed to by i. Disposer::operator()(pointer) is called for the removed element.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Average case O(1), worst case O(this->size()).

Throws: Nothing.

Note: Invalidates the iterators to the erased elements.

22.
```
template<typename Disposer> 
  void erase_and_dispose(const_iterator b, const_iterator e, 
                         Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases the range pointed to by b end e. Disposer::operator()(pointer) is called for the removed elements.

Complexity: Average case O(std::distance(b, e)), worst case O(this->size()).

Throws: Nothing.

Note: Invalidates the iterators to the erased elements.

23.
```
template<typename Disposer> 
  size_type erase_and_dispose(const_reference value, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases all the elements with the given value. Disposer::operator()(pointer) is called for the removed elements.

Returns: The number of erased elements.

Complexity: Average case O(this->count(value)). Worst case O(this->size()).

Throws: If the internal hasher or the equality functor throws. Basic guarantee.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

24.
```
template<typename KeyType, typename KeyHasher, typename KeyValueEqual, 
         typename Disposer> 
  size_type erase_and_dispose(const KeyType & key, KeyHasher hash_func, 
                              KeyValueEqual equal_func, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases all the elements with the given key. according to the comparison functor "equal_func". Disposer::operator()(pointer) is called for the removed elements.

Returns: The number of erased elements.

Complexity: Average case O(this->count(value)). Worst case O(this->size()).

Throws: If hash_func or equal_func throw. Basic guarantee.

Note: Invalidates the iterators to the erased elements.

25.
```
void clear();
```

Effects: Erases all of the elements.

Complexity: Linear to the number of elements on the container. if it's a safe-mode or auto-unlink value_type. Constant time otherwise.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

26.
```
template<typename Disposer> void clear_and_dispose(Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases all of the elements.

Complexity: Linear to the number of elements on the container. Disposer::operator()(pointer) is called for the removed elements.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

27.
```
size_type count(const_reference value) const;
```

Effects: Returns the number of contained elements with the given value

Complexity: Average case O(1), worst case O(this->size()).

Throws: If the internal hasher or the equality functor throws.

28.
```
template<typename KeyType, typename KeyHasher, typename KeyValueEqual>
  size_type count(const KeyType & key, KeyHasher hash_func,
                  KeyValueEqual equal_func) const;
```

Requires: "hash_func" must be a hash function that induces the same hash values as the stored hasher. The difference is that "hash_func" hashes the given key instead of the value_type.

"equal_func" must be a equality function that induces the same equality as key_equal. The difference is that "equal_func" compares an arbitrary key with the contained values.

Effects: Returns the number of contained elements with the given key

Complexity: Average case O(1), worst case O(this->size()).

Throws: If hash_func or equal_func throw.

29.
```
iterator find(const_reference value);
```

Effects: Finds an iterator to the first element is equal to "value" or end() if that element does not exist.

Complexity: Average case O(1), worst case O(this->size()).

Throws: If the internal hasher or the equality functor throws.

30.
```
template<typename KeyType, typename KeyHasher, typename KeyValueEqual>
  iterator find(const KeyType & key, KeyHasher hash_func,
                KeyValueEqual equal_func);
```

Requires: "hash_func" must be a hash function that induces the same hash values as the stored hasher. The difference is that "hash_func" hashes the given key instead of the value_type.

"equal_func" must be a equality function that induces the same equality as key_equal. The difference is that "equal_func" compares an arbitrary key with the contained values.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Finds an iterator to the first element whose key is "key" according to the given hasher and equality functor or end() if that element does not exist.

Complexity: Average case O(1), worst case O(this->size()).

Throws: If hash_func or equal_func throw.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

31.
```
const_iterator find(const_reference value) const;
```

Effects: Finds a const_iterator to the first element whose key is "key" or end() if that element does not exist.

Complexity: Average case O(1), worst case O(this->size()).

Throws: If the internal hasher or the equality functor throws.

32.
```
template<typename KeyType, typename KeyHasher, typename KeyValueEqual> 
  const_iterator 
  find(const KeyType & key, KeyHasher hash_func, KeyValueEqual equal_func) const;
```

Requires: "hash_func" must be a hash function that induces the same hash values as the stored hasher. The difference is that "hash_func" hashes the given key instead of the value_type.

"equal_func" must be a equality function that induces the same equality as key_equal. The difference is that "equal_func" compares an arbitrary key with the contained values.

Effects: Finds an iterator to the first element whose key is "key" according to the given hasher and equality functor or end() if that element does not exist.

Complexity: Average case O(1), worst case O(this->size()).

Throws: If hash_func or equal_func throw.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

33.
```
std::pair< iterator, iterator > equal_range(const_reference value);
```

Effects: Returns a range containing all elements with values equivalent to value. Returns std::make_pair(this->end(), this->end()) if no such elements exist.

Complexity: Average case O(this->count(value)). Worst case O(this->size()).

Throws: If the internal hasher or the equality functor throws.

34.
```
template<typename KeyType, typename KeyHasher, typename KeyValueEqual> 
  std::pair< iterator, iterator > 
  equal_range(const KeyType & key, KeyHasher hash_func, 
              KeyValueEqual equal_func);
```

Requires: "hash_func" must be a hash function that induces the same hash values as the stored hasher. The difference is that "hash_func" hashes the given key instead of the value_type.

"equal_func" must be a equality function that induces the same equality as key_equal. The difference is that "equal_func" compares an arbitrary key with the contained values.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Returns a range containing all elements with equivalent keys. Returns std::make_pair(this->end(), this->end()) if no such elements exist.

Complexity: Average case O(this->count(key, hash_func, hash_func)). Worst case O(this->size()).

Throws: If hash_func or the equal_func throw.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

35.
```
std::pair< const_iterator, const_iterator >
equal_range(const_reference value) const;
```

Effects: Returns a range containing all elements with values equivalent to value. Returns std::make_pair(this->end(), this->end()) if no such elements exist.

Complexity: Average case O(this->count(value)). Worst case O(this->size()).

Throws: If the internal hasher or the equality functor throws.

36.
```
template<typename KeyType, typename KeyHasher, typename KeyValueEqual>
  std::pair< const_iterator, const_iterator >
  equal_range(const KeyType & key, KeyHasher hash_func,
              KeyValueEqual equal_func) const;
```

Requires: "hash_func" must be a hash function that induces the same hash values as the stored hasher. The difference is that "hash_func" hashes the given key instead of the value_type.

"equal_func" must be a equality function that induces the same equality as key_equal. The difference is that "equal_func" compares an arbitrary key with the contained values.

Effects: Returns a range containing all elements with equivalent keys. Returns std::make_pair(this->end(), this->end()) if no such elements exist.

Complexity: Average case O(this->count(key, hash_func, equal_func)). Worst case O(this->size()).

Throws: If the hash_func or equal_func throw.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

37.
```
iterator iterator_to(reference value);
```

Requires: value must be an lvalue and shall be in a `unordered_set` of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid iterator belonging to the `unordered_set` that points to the value

Complexity: Constant.

Throws: If the internal hash function throws.

38.
```
const_iterator iterator_to(const_reference value) const;
```

Requires: value must be an lvalue and shall be in a `unordered_set` of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid const_iterator belonging to the `unordered_set` that points to the value

Complexity: Constant.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: If the internal hash function throws.

39.
```
local_iterator local_iterator_to(reference value);
```

Requires: value must be an lvalue and shall be in a `unordered_set` of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid local_iterator belonging to the `unordered_set` that points to the value

Complexity: Constant.

Throws: Nothing.

40.
```
const_local_iterator local_iterator_to(const_reference value) const;
```

Requires: value must be an lvalue and shall be in a `unordered_set` of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid const_local_iterator belonging to the `unordered_set` that points to the value

Complexity: Constant.

Throws: Nothing.

41.
```
size_type bucket_count() const;
```

Effects: Returns the number of buckets passed in the constructor or the last rehash function.

Complexity: Constant.

Throws: Nothing.

42.
```
size_type bucket_size(size_type n) const;
```

Requires: n is in the range [0, this->bucket_count()).

Effects: Returns the number of elements in the nth bucket.

Complexity: Constant.

Throws: Nothing.

43.
```
size_type bucket(const value_type & k) const;
```

Effects: Returns the index of the bucket in which elements with keys equivalent to k would be found, if any such element existed.

Complexity: Constant.

Throws: If the hash functor throws.

Note: the return value is in the range [0, this->bucket_count()).

44.
```
template<typename KeyType, typename KeyHasher> 
  size_type bucket(const KeyType & k, KeyHasher hash_func) const;
```

Requires: "hash_func" must be a hash function that induces the same hash values as the stored hasher. The difference is that "hash_func" hashes the given key instead of the value_type.

Effects: Returns the index of the bucket in which elements with keys equivalent to k would be found, if any such element existed.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Constant.

Throws: If hash_func throws.

Note: the return value is in the range [0, this->bucket_count()).

45.
```
bucket_ptr bucket_pointer() const;
```

Effects: Returns the bucket array pointer passed in the constructor or the last rehash function.

Complexity: Constant.

Throws: Nothing.

46.
```
local_iterator begin(size_type n);
```

Requires: n is in the range [0, this->bucket_count()).

Effects: Returns a local_iterator pointing to the beginning of the sequence stored in the bucket n.

Complexity: Constant.

Throws: Nothing.

Note: [this->begin(n), this->end(n)) is a valid range containing all of the elements in the nth bucket.

47.
```
const_local_iterator begin(size_type n) const;
```

Requires: n is in the range [0, this->bucket_count()).

Effects: Returns a const_local_iterator pointing to the beginning of the sequence stored in the bucket n.

Complexity: Constant.

Throws: Nothing.

Note: [this->begin(n), this->end(n)) is a valid range containing all of the elements in the nth bucket.

48.
```
const_local_iterator cbegin(size_type n) const;
```

Requires: n is in the range [0, this->bucket_count()).

Effects: Returns a const_local_iterator pointing to the beginning of the sequence stored in the bucket n.

Complexity: Constant.

Throws: Nothing.

Note: [this->begin(n), this->end(n)) is a valid range containing all of the elements in the nth bucket.

49.
```
local_iterator end(size_type n);
```

Requires: n is in the range [0, this->bucket_count()).

Effects: Returns a local_iterator pointing to the end of the sequence stored in the bucket n.

Complexity: Constant.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: Nothing.

Note: [this->begin(n), this->end(n)) is a valid range containing all of the elements in the nth bucket.

50.
```
const_local_iterator end(size_type n) const;
```

Requires: n is in the range [0, this->bucket_count()).

Effects: Returns a const_local_iterator pointing to the end of the sequence stored in the bucket n.

Complexity: Constant.

Throws: Nothing.

Note: [this->begin(n), this->end(n)) is a valid range containing all of the elements in the nth bucket.

51.
```
const_local_iterator cend(size_type n) const;
```

Requires: n is in the range [0, this->bucket_count()).

Effects: Returns a const_local_iterator pointing to the end of the sequence stored in the bucket n.

Complexity: Constant.

Throws: Nothing.

Note: [this->begin(n), this->end(n)) is a valid range containing all of the elements in the nth bucket.

52.
```
void rehash(const bucket_traits & new_bucket_traits);
```

Requires: new_buckets must be a pointer to a new bucket array or the same as the old bucket array. new_size is the length of the the array pointed by new_buckets. If new_buckets == this->bucket_pointer() n can be bigger or smaller than this->bucket_count().

Effects: Updates the internal reference with the new bucket erases the values from the old bucket and inserts then in the new one.

If `store_hash` option is true, this method does not use the hash function.

Complexity: Average case linear in this->size(), worst case quadratic.

Throws: If the hasher functor throws. Basic guarantee.

53.
```
bool incremental_rehash(bool grow = true);
```

Requires:

Effects:

Complexity:

Throws:

Note: this method is only available if incremental<true> option is activated.

54.
```
bool incremental_rehash(const bucket_traits & new_bucket_traits);
```

Note: this method is only available if incremental<true> option is activated.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

55.
```
size_type split_count() const;
```

Requires:

Effects:

Complexity:

Throws:

`unordered_set` public static functions

1.
```
static local_iterator s_local_iterator_to(reference value);
```

Requires: value must be an lvalue and shall be in a `unordered_set` of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid local_iterator belonging to the `unordered_set` that points to the value

Complexity: Constant.

Throws: Nothing.

Note: This static function is available only if the *value traits* is stateless.

2.
```
static const_local_iterator s_local_iterator_to(const_reference value);
```

Requires: value must be an lvalue and shall be in a `unordered_set` of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid const_local_iterator belonging to the `unordered_set` that points to the value

Complexity: Constant.

Throws: Nothing.

Note: This static function is available only if the *value traits* is stateless.

3.
```
static size_type suggested_upper_bucket_count(size_type n);
```

Effects: Returns the nearest new bucket count optimized for the container that is bigger than n. This suggestion can be used to create bucket arrays with a size that will usually improve container's performance. If such value does not exist, the higher possible value is returned.

Complexity: Amortized constant time.

Throws: Nothing.

4.
```
static size_type suggested_lower_bucket_count(size_type n);
```

Effects: Returns the nearest new bucket count optimized for the container that is smaller than n. This suggestion can be used to create bucket arrays with a size that will usually improve container's performance. If such value does not exist, the lower possible value is returned.

Complexity: Amortized constant time.

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Struct template make_unordered_set

boost::intrusive::make_unordered_set

Synopsis

```
// In header: <boost/intrusive/unordered_set.hpp>

template<typename T, class... Options>
struct make_unordered_set {
  // types
  typedef implementation_defined type;
};
```

Description

Helper metafunction to define an `unordered_set` that yields to the same type when the same options (either explicitly or implicitly) are used.

Class template unordered_multiset

boost::intrusive::unordered_multiset

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/intrusive/unordered_set.hpp>

template<typename T, class... Options>
class unordered_multiset {
public:
  // types
  typedef implementation_defined::value_type           value_type;
  typedef implementation_defined::value_traits         value_traits;
  typedef implementation_defined::bucket_traits        bucket_traits;
  typedef implementation_defined::pointer              pointer;
  typedef implementation_defined::const_pointer        const_pointer;
  typedef implementation_defined::reference            reference;
  typedef implementation_defined::const_reference      const_reference;
  typedef implementation_defined::difference_type      difference_type;
  typedef implementation_defined::size_type            size_type;
  typedef implementation_defined::key_type             key_type;
  typedef implementation_defined::key_equal            key_equal;
  typedef implementation_defined::hasher               hasher;
  typedef implementation_defined::bucket_type          bucket_type;
  typedef implementation_defined::bucket_ptr           bucket_ptr;
  typedef implementation_defined::iterator             iterator;
  typedef implementation_defined::const_iterator       const_iterator;
  typedef implementation_defined::insert_commit_data   insert_commit_data;
  typedef implementation_defined::local_iterator       local_iterator;
  typedef implementation_defined::const_local_iterator const_local_iterator;
  typedef implementation_defined::node_traits          node_traits;
  typedef implementation_defined::node                 node;
  typedef implementation_defined::node_ptr             node_ptr;
  typedef implementation_defined::const_node_ptr       const_node_ptr;
  typedef implementation_defined::node_algorithms      node_algorithms;

  // construct/copy/destruct
  explicit unordered_multiset(const bucket_traits &,
                              const hasher & = hasher(),
                              const key_equal & = key_equal(),
                              const value_traits & = value_traits());
  template<typename Iterator>
    unordered_multiset(Iterator, Iterator, const bucket_traits &,
                       const hasher & = hasher(),
                       const key_equal & = key_equal(),
                       const value_traits & = value_traits());
  unordered_multiset(BOOST_RV_REF(unordered_multiset));
  unordered_multiset& operator=(BOOST_RV_REF(unordered_multiset));
  ~unordered_multiset();

  // public member functions
  iterator begin();
  const_iterator begin() const;
  const_iterator cbegin() const;
  iterator end();
  const_iterator end() const;
  const_iterator cend() const;
  hasher hash_function() const;
  key_equal key_eq() const;
  bool empty() const;
  size_type size() const;
  void swap(unordered_multiset &);
  template<typename Cloner, typename Disposer>
    void clone_from(const unordered_multiset &, Cloner, Disposer);
  iterator insert(reference);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
  template<typename Iterator> void insert(Iterator, Iterator);
  void erase(const_iterator);
  void erase(const_iterator, const_iterator);
  size_type erase(const_reference);
  template<typename KeyType, typename KeyHasher, typename KeyValueEqual>
    size_type erase(const KeyType &, KeyHasher, KeyValueEqual);
  template<typename Disposer> void erase_and_dispose(const_iterator, Disposer);
  template<typename Disposer>
    void erase_and_dispose(const_iterator, const_iterator, Disposer);
  template<typename Disposer>
    size_type erase_and_dispose(const_reference, Disposer);
  template<typename KeyType, typename KeyHasher, typename KeyValueEqual,
           typename Disposer>
    size_type erase_and_dispose(const KeyType &, KeyHasher, KeyValueEqual,
                                Disposer);
  void clear();
  template<typename Disposer> void clear_and_dispose(Disposer);
  size_type count(const_reference) const;
  template<typename KeyType, typename KeyHasher, typename KeyValueEqual>
    size_type count(const KeyType &, KeyHasher, KeyValueEqual) const;
  iterator find(const_reference);
  template<typename KeyType, typename KeyHasher, typename KeyValueEqual>
    iterator find(const KeyType &, KeyHasher, KeyValueEqual);
  const_iterator find(const_reference) const;
  template<typename KeyType, typename KeyHasher, typename KeyValueEqual>
    const_iterator find(const KeyType &, KeyHasher, KeyValueEqual) const;
  std::pair< iterator, iterator > equal_range(const_reference);
  template<typename KeyType, typename KeyHasher, typename KeyValueEqual>
    std::pair< iterator, iterator >
    equal_range(const KeyType &, KeyHasher, KeyValueEqual);
  std::pair< const_iterator, const_iterator >
  equal_range(const_reference) const;
  template<typename KeyType, typename KeyHasher, typename KeyValueEqual>
    std::pair< const_iterator, const_iterator >
    equal_range(const KeyType &, KeyHasher, KeyValueEqual) const;
  iterator iterator_to(reference);
  const_iterator iterator_to(const_reference) const;
  local_iterator local_iterator_to(reference);
  const_local_iterator local_iterator_to(const_reference) const;
  size_type bucket_count() const;
  size_type bucket_size(size_type) const;
  size_type bucket(const value_type &) const;
  template<typename KeyType, typename KeyHasher>
    size_type bucket(const KeyType &, const KeyHasher &) const;
  bucket_ptr bucket_pointer() const;
  local_iterator begin(size_type);
  const_local_iterator begin(size_type) const;
  const_local_iterator cbegin(size_type) const;
  local_iterator end(size_type);
  const_local_iterator end(size_type) const;
  const_local_iterator cend(size_type) const;
  void rehash(const bucket_traits &);
  bool incremental_rehash(bool = true);
  bool incremental_rehash(const bucket_traits &);
  size_type split_count() const;

  // public static functions
  static local_iterator s_local_iterator_to(reference);
  static const_local_iterator s_local_iterator_to(const_reference);
  static size_type suggested_upper_bucket_count(size_type);
  static size_type suggested_lower_bucket_count(size_type);
};
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Description

The class template unordered_multiset is an intrusive container, that mimics most of the interface of std::tr1::unordered_multiset as described in the C++ TR1.

unordered_multiset is a semi-intrusive container: each object to be stored in the container must contain a proper hook, but the container also needs additional auxiliary memory to work: unordered_multiset needs a pointer to an array of type `bucket_type` to be passed in the constructor. This bucket array must have at least the same lifetime as the container. This makes the use of unordered_multiset more complicated than purely intrusive containers. `bucket_type` is default-constructible, copyable and assignable

The template parameter `T` is the type to be managed by the container. The user can specify additional options and if no options are provided default options are used.

The container supports the following options: `base_hook<>/member_hook<>/value_traits<>`, `constant_time_size<>`, `size_type<>`, `hash<>` and `equal<>` `bucket_traits<>`, `power_2_buckets<>` and `cache_begin<>`.

unordered_multiset only provides forward iterators but it provides 4 iterator types: iterator and const_iterator to navigate through the whole container and local_iterator and const_local_iterator to navigate through the values stored in a single bucket. Local iterators are faster and smaller.

It's not recommended to use non constant-time size unordered_multisets because several key functions, like "empty()", become non-constant time functions. Non constant-time size unordered_multisets are mainly provided to support auto-unlink hooks.

unordered_multiset, unlike std::unordered_set, does not make automatic rehashings nor offers functions related to a load factor. Rehashing can be explicitly requested and the user must provide a new bucket array that will be used from that moment.

Since no automatic rehashing is done, iterators are never invalidated when inserting or erasing elements. Iterators are only invalidated when rehasing.

`unordered_multiset` public construct/copy/destruct

1.
```
explicit unordered_multiset(const bucket_traits & b_traits,
                            const hasher & hash_func = hasher(),
                            const key_equal & equal_func = key_equal(),
                            const value_traits & v_traits = value_traits());
```

Requires: buckets must not be being used by any other resource.

Effects: Constructs an empty `unordered_multiset`, storing a reference to the bucket array and copies of the hasher and equal functors.

Complexity: Constant.

Throws: If value_traits::node_traits::node constructor throws (this does not happen with predefined Boost.Intrusive hooks) or the copy constructor or invocation of Hash or Equal throws.

Notes: buckets array must be disposed only after *this is disposed.

2.
```
template<typename Iterator>
  unordered_multiset(Iterator b, Iterator e, const bucket_traits & b_traits,
                     const hasher & hash_func = hasher(),
                     const key_equal & equal_func = key_equal(),
                     const value_traits & v_traits = value_traits());
```

Requires: buckets must not be being used by any other resource and Dereferencing iterator must yield an lvalue of type value_type.

Effects: Constructs an empty `unordered_multiset` and inserts elements from [b, e).

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: If N is std::distance(b, e): Average case is O(N) (with a good hash function and with buckets_len >= N),worst case O(N2).

Throws: If value_traits::node_traits::node constructor throws (this does not happen with predefined Boost.Intrusive hooks) or the copy constructor or invocation of hasher or key_equal throws.

Notes: buckets array must be disposed only after *this is disposed.

3.
```
unordered_multiset(BOOST_RV_REF(unordered_multiset) x);
```

Effects: to-do

4.
```
unordered_multiset& operator=(BOOST_RV_REF(unordered_multiset) x);
```

Effects: to-do

5.
```
~unordered_multiset();
```

Effects: Detaches all elements from this. The objects in the `unordered_multiset` are not deleted (i.e. no destructors are called).

Complexity: Linear to the number of elements in the `unordered_multiset`, if it's a safe-mode or auto-unlink value. Otherwise constant.

Throws: Nothing.

`unordered_multiset` public member functions

1.
```
iterator begin();
```

Effects: Returns an iterator pointing to the beginning of the `unordered_multiset`.

Complexity: Constant time if `cache_begin`<> is true. Amortized constant time with worst case (empty `unordered_set`) O(this->bucket_count())

Throws: Nothing.

2.
```
const_iterator begin() const;
```

Effects: Returns a const_iterator pointing to the beginning of the `unordered_multiset`.

Complexity: Constant time if `cache_begin`<> is true. Amortized constant time with worst case (empty `unordered_set`) O(this->bucket_count())

Throws: Nothing.

3.
```
const_iterator cbegin() const;
```

Effects: Returns a const_iterator pointing to the beginning of the `unordered_multiset`.

Complexity: Constant time if `cache_begin`<> is true. Amortized constant time with worst case (empty `unordered_set`) O(this->bucket_count())

Throws: Nothing.

4.
```
iterator end();
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: Returns an iterator pointing to the end of the unordered_multiset.

Complexity: Constant.

Throws: Nothing.

5.
```
const_iterator end() const;
```

Effects: Returns a const_iterator pointing to the end of the unordered_multiset.

Complexity: Constant.

Throws: Nothing.

6.
```
const_iterator cend() const;
```

Effects: Returns a const_iterator pointing to the end of the unordered_multiset.

Complexity: Constant.

Throws: Nothing.

7.
```
hasher hash_function() const;
```

Effects: Returns the hasher object used by the unordered_set.

Complexity: Constant.

Throws: If hasher copy-constructor throws.

8.
```
key_equal key_eq() const;
```

Effects: Returns the key_equal object used by the unordered_multiset.

Complexity: Constant.

Throws: If key_equal copy-constructor throws.

9.
```
bool empty() const;
```

Effects: Returns true if the container is empty.

Complexity: if constant-time size and cache_last options are disabled, average constant time (worst case, with empty() == true: O(this->bucket_count()). Otherwise constant.

Throws: Nothing.

10.
```
size_type size() const;
```

Effects: Returns the number of elements stored in the unordered_multiset.

Complexity: Linear to elements contained in *this if constant-time size option is disabled. Constant-time otherwise.

Throws: Nothing.

11.
```
void swap(unordered_multiset & other);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Requires: the hasher and the equality function unqualified swap call should not throw.

Effects: Swaps the contents of two unordered_multisets. Swaps also the contained bucket array and equality and hasher functors.

Complexity: Constant.

Throws: If the swap() call for the comparison or hash functors found using ADL throw. Basic guarantee.

12.
```
template<typename Cloner, typename Disposer> 
  void clone_from(const unordered_multiset & src, Cloner cloner, 
                  Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw. Cloner should yield to nodes that compare equal and produce the same hash than the original node.

Effects: Erases all the elements from *this calling Disposer::operator()(pointer), clones all the elements from src calling Cloner::operator()(const_reference) and inserts them on *this. The hash function and the equality predicate are copied from the source.

If `store_hash` option is true, this method does not use the hash function.

If any operation throws, all cloned elements are unlinked and disposed calling Disposer::operator()(pointer).

Complexity: Linear to erased plus inserted elements.

Throws: If cloner or hasher throw or hash or equality predicate copying throws. Basic guarantee.

13.
```
iterator insert(reference value);
```

Requires: value must be an lvalue

Effects: Inserts value into the `unordered_multiset`.

Returns: An iterator to the new inserted value.

Complexity: Average case O(1), worst case O(this->size()).

Throws: If the internal hasher or the equality functor throws. Strong guarantee.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

14.
```
template<typename Iterator> void insert(Iterator b, Iterator e);
```

Requires: Dereferencing iterator must yield an lvalue of type value_type.

Effects: Equivalent to this->insert(t) for each element in [b, e).

Complexity: Average case is O(N), where N is the size of the range.

Throws: If the internal hasher or the equality functor throws. Basic guarantee.

Note: Does not affect the validity of iterators and references. No copy-constructors are called.

15.
```
void erase(const_iterator i);
```

Effects: Erases the element pointed to by i.

Complexity: Average case O(1), worst case O(this->size()).

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased element. No destructors are called.

16.
```
void erase(const_iterator b, const_iterator e);
```

Effects: Erases the range pointed to by b end e.

Complexity: Average case O(std::distance(b, e)), worst case O(this->size()).

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

17.
```
size_type erase(const_reference value);
```

Effects: Erases all the elements with the given value.

Returns: The number of erased elements.

Complexity: Average case O(this->count(value)). Worst case O(this->size()).

Throws: If the internal hasher or the equality functor throws. Basic guarantee.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

18.
```
template<typename KeyType, typename KeyHasher, typename KeyValueEqual>
  size_type erase(const KeyType & key, KeyHasher hash_func,
                  KeyValueEqual equal_func);
```

Requires: "hash_func" must be a hash function that induces the same hash values as the stored hasher. The difference is that "hash_func" hashes the given key instead of the value_type.

"key_value_equal" must be a equality function that induces the same equality as key_equal. The difference is that "key_value_equal" compares an arbitrary key with the contained values.

Effects: Erases all the elements that have the same hash and compare equal with the given key.

Returns: The number of erased elements.

Complexity: Average case O(this->count(value)). Worst case O(this->size()).

Throws: If the hash_func or the equal_func functors throws. Basic guarantee.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

19.
```
template<typename Disposer>
  void erase_and_dispose(const_iterator i, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases the element pointed to by i. Disposer::operator()(pointer) is called for the removed element.

Complexity: Average case O(1), worst case O(this->size()).

Throws: Nothing.

Note: Invalidates the iterators to the erased elements.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

20.

```
template<typename Disposer>
  void erase_and_dispose(const_iterator b, const_iterator e,
                         Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases the range pointed to by b end e. Disposer::operator()(pointer) is called for the removed elements.

Complexity: Average case O(std::distance(b, e)), worst case O(this->size()).

Throws: Nothing.

Note: Invalidates the iterators to the erased elements.

21.

```
template<typename Disposer>
  size_type erase_and_dispose(const_reference value, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases all the elements with the given value. Disposer::operator()(pointer) is called for the removed elements.

Returns: The number of erased elements.

Complexity: Average case O(this->count(value)). Worst case O(this->size()).

Throws: If the internal hasher or the equality functor throws. Basic guarantee.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

22.

```
template<typename KeyType, typename KeyHasher, typename KeyValueEqual,
         typename Disposer>
  size_type erase_and_dispose(const KeyType & key, KeyHasher hash_func,
                              KeyValueEqual equal_func, Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases all the elements with the given key. according to the comparison functor "equal_func". Disposer::operator()(pointer) is called for the removed elements.

Returns: The number of erased elements.

Complexity: Average case O(this->count(value)). Worst case O(this->size()).

Throws: If hash_func or equal_func throw. Basic guarantee.

Note: Invalidates the iterators to the erased elements.

23.

```
void clear();
```

Effects: Erases all the elements of the container.

Complexity: Linear to the number of elements on the container. if it's a safe-mode or auto-unlink value_type. Constant time otherwise.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

24.
```
template<typename Disposer> void clear_and_dispose(Disposer disposer);
```

Requires: Disposer::operator()(pointer) shouldn't throw.

Effects: Erases all the elements of the container.

Complexity: Linear to the number of elements on the container. Disposer::operator()(pointer) is called for the removed elements.

Throws: Nothing.

Note: Invalidates the iterators (but not the references) to the erased elements. No destructors are called.

25.
```
size_type count(const_reference value) const;
```

Effects: Returns the number of contained elements with the given key

Complexity: Average case O(1), worst case O(this->size()).

Throws: If the internal hasher or the equality functor throws.

26.
```
template<typename KeyType, typename KeyHasher, typename KeyValueEqual> 
  size_type count(const KeyType & key, KeyHasher hash_func, 
                  KeyValueEqual equal_func) const;
```

Requires: "hash_func" must be a hash function that induces the same hash values as the stored hasher. The difference is that "hash_func" hashes the given key instead of the value_type.

"key_value_equal" must be a equality function that induces the same equality as key_equal. The difference is that "key_value_equal" compares an arbitrary key with the contained values.

Effects: Returns the number of contained elements with the given key

Complexity: Average case O(1), worst case O(this->size()).

Throws: If the internal hasher or the equality functor throws.

27.
```
iterator find(const_reference value);
```

Effects: Finds an iterator to the first element whose value is "value" or end() if that element does not exist.

Complexity: Average case O(1), worst case O(this->size()).

Throws: If the internal hasher or the equality functor throws.

28.
```
template<typename KeyType, typename KeyHasher, typename KeyValueEqual> 
  iterator find(const KeyType & key, KeyHasher hash_func, 
                KeyValueEqual equal_func);
```

Requires: "hash_func" must be a hash function that induces the same hash values as the stored hasher. The difference is that "hash_func" hashes the given key instead of the value_type.

"key_value_equal" must be a equality function that induces the same equality as key_equal. The difference is that "key_value_equal" compares an arbitrary key with the contained values.

Effects: Finds an iterator to the first element whose key is "key" according to the given hasher and equality functor or end() if that element does not exist.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Average case O(1), worst case O(this->size()).

Throws: If the internal hasher or the equality functor throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

29.
```
const_iterator find(const_reference value) const;
```

Effects: Finds a const_iterator to the first element whose key is "key" or end() if that element does not exist.

Complexity: Average case O(1), worst case O(this->size()).

Throws: If the internal hasher or the equality functor throws.

30.
```
template<typename KeyType, typename KeyHasher, typename KeyValueEqual>
  const_iterator
  find(const KeyType & key, KeyHasher hash_func, KeyValueEqual equal_func) const;
```

Requires: "hash_func" must be a hash function that induces the same hash values as the stored hasher. The difference is that "hash_func" hashes the given key instead of the value_type.

"key_value_equal" must be a equality function that induces the same equality as key_equal. The difference is that "key_value_equal" compares an arbitrary key with the contained values.

Effects: Finds an iterator to the first element whose key is "key" according to the given hasher and equality functor or end() if that element does not exist.

Complexity: Average case O(1), worst case O(this->size()).

Throws: If the internal hasher or the equality functor throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

31.
```
std::pair< iterator, iterator > equal_range(const_reference value);
```

Effects: Returns a range containing all elements with values equivalent to value. Returns std::make_pair(this->end(), this->end()) if no such elements exist.

Complexity: Average case O(this->count(value)). Worst case O(this->size()).

Throws: If the internal hasher or the equality functor throws.

32.
```
template<typename KeyType, typename KeyHasher, typename KeyValueEqual>
  std::pair< iterator, iterator >
  equal_range(const KeyType & key, KeyHasher hash_func,
              KeyValueEqual equal_func);
```

Requires: "hash_func" must be a hash function that induces the same hash values as the stored hasher. The difference is that "hash_func" hashes the given key instead of the value_type.

"key_value_equal" must be a equality function that induces the same equality as key_equal. The difference is that "key_value_equal" compares an arbitrary key with the contained values.

Effects: Returns a range containing all elements with equivalent keys. Returns std::make_pair(this->end(), this->end()) if no such elements exist.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complexity: Average case O(this->count(key, hash_func, equal_func)). Worst case O(this->size()).

Throws: If the internal hasher or the equality functor throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

33.
```
std::pair< const_iterator, const_iterator >
equal_range(const_reference value) const;
```

Effects: Returns a range containing all elements with values equivalent to value. Returns std::make_pair(this->end(), this->end()) if no such elements exist.

Complexity: Average case O(this->count(value)). Worst case O(this->size()).

Throws: If the internal hasher or the equality functor throws.

34.
```
template<typename KeyType, typename KeyHasher, typename KeyValueEqual>
  std::pair< const_iterator, const_iterator >
  equal_range(const KeyType & key, KeyHasher hash_func,
              KeyValueEqual equal_func) const;
```

Requires: "hash_func" must be a hash function that induces the same hash values as the stored hasher. The difference is that "hash_func" hashes the given key instead of the value_type.

"key_value_equal" must be a equality function that induces the same equality as key_equal. The difference is that "key_value_equal" compares an arbitrary key with the contained values.

Effects: Returns a range containing all elements with equivalent keys. Returns std::make_pair(this->end(), this->end()) if no such elements exist.

Complexity: Average case O(this->count(key, hash_func, equal_func)). Worst case O(this->size()).

Throws: If the internal hasher or the equality functor throws.

Note: This function is used when constructing a value_type is expensive and the value_type can be compared with a cheaper key type. Usually this key is part of the value_type.

35.
```
iterator iterator_to(reference value);
```

Requires: value must be an lvalue and shall be in a `unordered_multiset` of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid iterator belonging to the `unordered_multiset` that points to the value

Complexity: Constant.

Throws: If the hash function throws.

36.
```
const_iterator iterator_to(const_reference value) const;
```

Requires: value must be an lvalue and shall be in a `unordered_multiset` of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid const_iterator belonging to the `unordered_multiset` that points to the value

Complexity: Constant.

Throws: If the hash function throws.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

37.
```
local_iterator local_iterator_to(reference value);
```

Requires: value must be an lvalue and shall be in a `unordered_set` of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid local_iterator belonging to the `unordered_set` that points to the value

Complexity: Constant.

Throws: Nothing.

38.
```
const_local_iterator local_iterator_to(const_reference value) const;
```

Requires: value must be an lvalue and shall be in a `unordered_set` of appropriate type. Otherwise the behavior is undefined.

Effects: Returns: a valid const_local_iterator belonging to the `unordered_set` that points to the value

Complexity: Constant.

Throws: Nothing.

39.
```
size_type bucket_count() const;
```

Effects: Returns the number of buckets passed in the constructor or the last rehash function.

Complexity: Constant.

Throws: Nothing.

40.
```
size_type bucket_size(size_type n) const;
```

Requires: n is in the range [0, this->bucket_count()).

Effects: Returns the number of elements in the nth bucket.

Complexity: Constant.

Throws: Nothing.

41.
```
size_type bucket(const value_type & k) const;
```

Effects: Returns the index of the bucket in which elements with keys equivalent to k would be found, if any such element existed.

Complexity: Constant.

Throws: If the hash functor throws.

Note: the return value is in the range [0, this->bucket_count()).

42.
```
template<typename KeyType, typename KeyHasher>
  size_type bucket(const KeyType & k, const KeyHasher & hash_func) const;
```

Requires: "hash_func" must be a hash function that induces the same hash values as the stored hasher. The difference is that "hash_func" hashes the given key instead of the value_type.

Effects: Returns the index of the bucket in which elements with keys equivalent to k would be found, if any such element existed.

Complexity: Constant.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: If the hash functor throws.

Note: the return value is in the range [0, this->bucket_count()).

43.
```
bucket_ptr bucket_pointer() const;
```

Effects: Returns the bucket array pointer passed in the constructor or the last rehash function.

Complexity: Constant.

Throws: Nothing.

44.
```
local_iterator begin(size_type n);
```

Requires: n is in the range [0, this->bucket_count()).

Effects: Returns a local_iterator pointing to the beginning of the sequence stored in the bucket n.

Complexity: Constant.

Throws: Nothing.

Note: [this->begin(n), this->end(n)) is a valid range containing all of the elements in the nth bucket.

45.
```
const_local_iterator begin(size_type n) const;
```

Requires: n is in the range [0, this->bucket_count()).

Effects: Returns a const_local_iterator pointing to the beginning of the sequence stored in the bucket n.

Complexity: Constant.

Throws: Nothing.

Note: [this->begin(n), this->end(n)) is a valid range containing all of the elements in the nth bucket.

46.
```
const_local_iterator cbegin(size_type n) const;
```

Requires: n is in the range [0, this->bucket_count()).

Effects: Returns a const_local_iterator pointing to the beginning of the sequence stored in the bucket n.

Complexity: Constant.

Throws: Nothing.

Note: [this->begin(n), this->end(n)) is a valid range containing all of the elements in the nth bucket.

47.
```
local_iterator end(size_type n);
```

Requires: n is in the range [0, this->bucket_count()).

Effects: Returns a local_iterator pointing to the end of the sequence stored in the bucket n.

Complexity: Constant.

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Note: [this->begin(n), this->end(n)) is a valid range containing all of the elements in the nth bucket.

48.
```
const_local_iterator end(size_type n) const;
```

Requires: n is in the range [0, this->bucket_count()).

Effects: Returns a const_local_iterator pointing to the end of the sequence stored in the bucket n.

Complexity: Constant.

Throws: Nothing.

Note: [this->begin(n), this->end(n)) is a valid range containing all of the elements in the nth bucket.

49.
```
const_local_iterator cend(size_type n) const;
```

Requires: n is in the range [0, this->bucket_count()).

Effects: Returns a const_local_iterator pointing to the end of the sequence stored in the bucket n.

Complexity: Constant.

Throws: Nothing.

Note: [this->begin(n), this->end(n)) is a valid range containing all of the elements in the nth bucket.

50.
```
void rehash(const bucket_traits & new_bucket_traits);
```

Requires: new_buckets must be a pointer to a new bucket array or the same as the old bucket array. new_size is the length of the the array pointed by new_buckets. If new_buckets == this->bucket_pointer() n can be bigger or smaller than this->bucket_count().

Effects: Updates the internal reference with the new bucket erases the values from the old bucket and inserts then in the new one.

If `store_hash` option is true, this method does not use the hash function.

Complexity: Average case linear in this->size(), worst case quadratic.

Throws: If the hasher functor throws.

51.
```
bool incremental_rehash(bool grow = true);
```

Requires:

Effects:

Complexity:

Throws:

Note: this method is only available if incremental<true> option is activated.

52.
```
bool incremental_rehash(const bucket_traits & new_bucket_traits);
```

Note: this method is only available if incremental<true> option is activated.

53.
```
size_type split_count() const;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Requires:

Effects:

Complexity:

Throws:

`unordered_multiset` public static functions

1.
   ```
   static local_iterator s_local_iterator_to(reference value);
   ```

   Requires: value must be an lvalue and shall be in a `unordered_set` of appropriate type. Otherwise the behavior is undefined.

   Effects: Returns: a valid local_iterator belonging to the `unordered_set` that points to the value

   Complexity: Constant.

   Throws: Nothing.

   Note: This static function is available only if the *value traits* is stateless.

2.
   ```
   static const_local_iterator s_local_iterator_to(const_reference value);
   ```

   Requires: value must be an lvalue and shall be in a `unordered_set` of appropriate type. Otherwise the behavior is undefined.

   Effects: Returns: a valid const_local_iterator belonging to the `unordered_set` that points to the value

   Complexity: Constant.

   Throws: Nothing.

   Note: This static function is available only if the *value traits* is stateless.

3.
   ```
   static size_type suggested_upper_bucket_count(size_type n);
   ```

   Effects: Returns the nearest new bucket count optimized for the container that is bigger than n. This suggestion can be used to create bucket arrays with a size that will usually improve container's performance. If such value does not exist, the higher possible value is returned.

   Complexity: Amortized constant time.

   Throws: Nothing.

4.
   ```
   static size_type suggested_lower_bucket_count(size_type n);
   ```

   Effects: Returns the nearest new bucket count optimized for the container that is smaller than n. This suggestion can be used to create bucket arrays with a size that will usually improve container's performance. If such value does not exist, the lower possible value is returned.

   Complexity: Amortized constant time.

   Throws: Nothing.

Struct template make_unordered_multiset

boost::intrusive::make_unordered_multiset

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/intrusive/unordered_set.hpp>

template<typename T, class... Options>
struct make_unordered_multiset {
  // types
  typedef implementation_defined type;
};
```

Description

Helper metafunction to define an `unordered_multiset` that yields to the same type when the same options (either explicitly or implicitly) are used.

Header <boost/intrusive/unordered_set_hook.hpp>

```
namespace boost {
  namespace intrusive {
    template<class... Options> struct make_unordered_set_base_hook;

    template<class... Options> class unordered_set_base_hook;

    template<class... Options> struct make_unordered_set_member_hook;

    template<class... Options> class unordered_set_member_hook;
  }
}
```

Struct template make_unordered_set_base_hook

boost::intrusive::make_unordered_set_base_hook

Synopsis

```
// In header: <boost/intrusive/unordered_set_hook.hpp>

template<class... Options>
struct make_unordered_set_base_hook {
  // types
  typedef implementation_defined type;
};
```

Description

Helper metafunction to define a `unordered_set_base_hook` that yields to the same type when the same options (either explicitly or implicitly) are used.

Class template unordered_set_base_hook

boost::intrusive::unordered_set_base_hook

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/intrusive/unordered_set_hook.hpp>

template<class... Options>
class unordered_set_base_hook :
  public make_unordered_set_base_hook::type< O1, O2, O3, O4 >
{
public:
  // construct/copy/destruct
  unordered_set_base_hook();
  unordered_set_base_hook(const unordered_set_base_hook &);
  unordered_set_base_hook& operator=(const unordered_set_base_hook &);
  ~unordered_set_base_hook();

  // public member functions
  void swap_nodes(unordered_set_base_hook &);
  bool is_linked() const;
  void unlink();
};
```

Description

Derive a class from unordered_set_base_hook in order to store objects in in an unordered_set/unordered_multi_set. unordered_set_base_hook holds the data necessary to maintain the unordered_set/unordered_multi_set and provides an appropriate value_traits class for unordered_set/unordered_multi_set.

The hook admits the following options: `tag<>`, `void_pointer<>`, `link_mode<>`, `store_hash<>` and `optimize_multikey<>`.

`tag<>` defines a tag to identify the node. The same tag value can be used in different classes, but if a class is derived from more than one `list_base_hook`, then each `list_base_hook` needs its unique tag.

`void_pointer<>` is the pointer type that will be used internally in the hook and the the container configured to use this hook.

`link_mode<>` will specify the linking mode of the hook (`normal_link`, `auto_unlink` or `safe_link`).

`store_hash<>` will tell the hook to store the hash of the value to speed up rehashings.

`optimize_multikey<>` will tell the hook to store a link to form a group with other value with the same value to speed up searches and insertions in unordered_multisets with a great number of with equivalent keys.

`unordered_set_base_hook` public construct/copy/destruct

1.
   ```
   unordered_set_base_hook();
   ```

   Effects: If `link_mode` is `auto_unlink` or `safe_link` initializes the node to an unlinked state.

   Throws: Nothing.

2.
   ```
   unordered_set_base_hook(const unordered_set_base_hook &);
   ```

   Effects: If `link_mode` is `auto_unlink` or `safe_link` initializes the node to an unlinked state. The argument is ignored.

   Throws: Nothing.

   Rationale: Providing a copy-constructor makes classes using the hook STL-compliant without forcing the user to do some additional work. `swap` can be used to emulate move-semantics.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

3.
```
unordered_set_base_hook& operator=(const unordered_set_base_hook &);
```

Effects: Empty function. The argument is ignored.

Throws: Nothing.

Rationale: Providing an assignment operator makes classes using the hook STL-compliant without forcing the user to do some additional work. `swap` can be used to emulate move-semantics.

4.
```
~unordered_set_base_hook();
```

Effects: If `link_mode` is `normal_link`, the destructor does nothing (ie. no code is generated). If `link_mode` is `safe_link` and the object is stored in an `unordered_set` an assertion is raised. If `link_mode` is `auto_unlink` and `is_linked()` is true, the node is unlinked.

Throws: Nothing.

`unordered_set_base_hook` public member functions

1.
```
void swap_nodes(unordered_set_base_hook & other);
```

Effects: Swapping two nodes swaps the position of the elements related to those nodes in one or two containers. That is, if the node this is part of the element e1, the node x is part of the element e2 and both elements are included in the containers s1 and s2, then after the swap-operation e1 is in s2 at the position of e2 and e2 is in s1 at the position of e1. If one element is not in a container, then after the swap-operation the other element is not in a container. Iterators to e1 and e2 related to those nodes are invalidated.

Complexity: Constant

Throws: Nothing.

2.
```
bool is_linked() const;
```

Precondition: `link_mode` must be `safe_link` or `auto_unlink`.

Returns: true, if the node belongs to a container, false otherwise. This function can be used to test whether `unordered_set::iterator_to` will return a valid iterator.

Complexity: Constant

3.
```
void unlink();
```

Effects: Removes the node if it's inserted in a container. This function is only allowed if `link_mode` is `auto_unlink`.

Throws: Nothing.

Struct template make_unordered_set_member_hook

boost::intrusive::make_unordered_set_member_hook

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/intrusive/unordered_set_hook.hpp>

template<class... Options>
struct make_unordered_set_member_hook {
  // types
  typedef implementation_defined type;
};
```

Description

Helper metafunction to define a `unordered_set_member_hook` that yields to the same type when the same options (either explicitly or implicitly) are used.

Class template unordered_set_member_hook

boost::intrusive::unordered_set_member_hook

Synopsis

```
// In header: <boost/intrusive/unordered_set_hook.hpp>

template<class... Options>
class unordered_set_member_hook :
  public make_unordered_set_member_hook::type< O1, O2, O3, O4 >
{
public:
  // construct/copy/destruct
  unordered_set_member_hook();
  unordered_set_member_hook(const unordered_set_member_hook &);
  unordered_set_member_hook& operator=(const unordered_set_member_hook &);
  ~unordered_set_member_hook();

  // public member functions
  void swap_nodes(unordered_set_member_hook &);
  bool is_linked() const;
  void unlink();
};
```

Description

Put a public data member unordered_set_member_hook in order to store objects of this class in an unordered_set/unordered_multi_set. unordered_set_member_hook holds the data necessary for maintaining the unordered_set/unordered_multi_set and provides an appropriate value_traits class for unordered_set/unordered_multi_set.

The hook admits the following options: `void_pointer<>`, `link_mode<>` and `store_hash<>`.

`void_pointer<>` is the pointer type that will be used internally in the hook and the the container configured to use this hook.

`link_mode<>` will specify the linking mode of the hook (`normal_link`, `auto_unlink` or `safe_link`).

`store_hash<>` will tell the hook to store the hash of the value to speed up rehashings.

`unordered_set_member_hook` public construct/copy/destruct

1.
```
unordered_set_member_hook();
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Effects: If `link_mode` is `auto_unlink` or `safe_link` initializes the node to an unlinked state.

Throws: Nothing.

2.
```
unordered_set_member_hook(const unordered_set_member_hook &);
```

Effects: If `link_mode` is `auto_unlink` or `safe_link` initializes the node to an unlinked state. The argument is ignored.

Throws: Nothing.

Rationale: Providing a copy-constructor makes classes using the hook STL-compliant without forcing the user to do some additional work. `swap` can be used to emulate move-semantics.

3.
```
unordered_set_member_hook& operator=(const unordered_set_member_hook &);
```

Effects: Empty function. The argument is ignored.

Throws: Nothing.

Rationale: Providing an assignment operator makes classes using the hook STL-compliant without forcing the user to do some additional work. `swap` can be used to emulate move-semantics.

4.
```
~unordered_set_member_hook();
```

Effects: If `link_mode` is `normal_link`, the destructor does nothing (ie. no code is generated). If `link_mode` is `safe_link` and the object is stored in an `unordered_set` an assertion is raised. If `link_mode` is `auto_unlink` and `is_linked()` is true, the node is unlinked.

Throws: Nothing.

`unordered_set_member_hook` public member functions

1.
```
void swap_nodes(unordered_set_member_hook & other);
```

Effects: Swapping two nodes swaps the position of the elements related to those nodes in one or two containers. That is, if the node this is part of the element e1, the node x is part of the element e2 and both elements are included in the containers s1 and s2, then after the swap-operation e1 is in s2 at the position of e2 and e2 is in s1 at the position of e1. If one element is not in a container, then after the swap-operation the other element is not in a container. Iterators to e1 and e2 related to those nodes are invalidated.

Complexity: Constant

Throws: Nothing.

2.
```
bool is_linked() const;
```

Precondition: `link_mode` must be `safe_link` or `auto_unlink`.

Returns: true, if the node belongs to a container, false otherwise. This function can be used to test whether `unordered_set::iterator_to` will return a valid iterator.

Complexity: Constant

3.
```
void unlink();
```

Effects: Removes the node if it's inserted in a container. This function is only allowed if `link_mode` is `auto_unlink`.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/