Boost.Heap

Tim Blechmann

Table of Contents

Introduction & Motivation

`boost.heap` is an implementation of priority queues. Priority queues are queue data structures, that order their elements by a priority. The STL provides a single template class `std::priority_queue`, which only provides a limited functionality. To overcome these limitations, `boost.heap` implements data structures with more functionality and different performance characteristics. Especially, it deals with additional aspects:

- **Mutability**: The priority of heap elements can be modified.
- **Iterators**: Heaps provide iterators to iterate all elements.
- **Mergable**: While all heaps can be merged, some can be merged efficiently.
- **Stability**: Heaps can be configured to be stable sorted.
- **Comparison**: Heaps can be compared for equivalence.

Concepts & Interface

Basic Priority Queue Interface

Priority queues are queues of objects, that are ordered by their priority. They support the operations of adding nodes to the data structure, accessing the top element (the element with the highest priority), and removing the top element.

Note

`boost.heap` implements priority queues as max-heaps to be consistent with the STL heap functions. This is in contrast to the typical textbook design, which uses min-heaps.

Synopsis

```
template <typename T, class ...Options>
class priority_queue
{
    // types
    typedef T                   value_type;
    typedef unspecified         size_type;
    typedef unspecified         difference_type;

    typedef unspecified         allocator_type;
    typedef unspecified         value_compare;

    typedef unspecified         reference;
    typedef unspecified         const_reference;
    typedef unspecified         pointer;
    typedef unspecified         const_pointer;

    // construct/copy/destruct
    explicit priority_queue(value_compare const & = value_compare());
    priority_queue(priority_queue const &);
    priority_queue& operator=(priority_queue const &);
    priority_queue(priority_queue &&);                  // move semantics (C++11 only)
    priority_queue& operator=(priority_queue &&);       // move semantics (C++11 only)

    // public member functions
    unspecified push(const_reference);                  // push new element to heap
    template<class... Args> void emplace(Args &&...);   // push new element to heap, C++11 only
    const_reference top() const;                        // return top element
    void pop();                                         // remove top element
    void clear();                                       // clear heap
    size_type size() const;                             // number of elements
    bool empty() const;                                 // priority queue is empty
    allocator_type get_allocator(void) const;           // return allocator
    size_type max_size(void) const;                     // maximal possible size
    void reserve(size_type);                            // reserve space, only available if ↵
(has_reserve == true)

    // heap equivalence
    template<typename HeapType> bool operator==(HeapType const &) const;
    template<typename HeapType> bool operator!=(HeapType const &) const;

    // heap comparison
    template<typename HeapType> bool operator<(HeapType const &) const;
    template<typename HeapType> bool operator>(HeapType const &) const;
    template<typename HeapType> bool operator>=(HeapType const &) const;
    template<typename HeapType> bool operator<=(HeapType const &) const;
```

```
    // public data members
    static const bool constant_time_size;              // size() has constant complexity
    static const bool has_ordered_iterators;           // priority queue has ordered iterators
    static const bool is_mergable;                     // priority queue is efficiently mergable
    static const bool is_stable;                       // priority queue has a stable heap order
    static const bool has_reserve;                     // priority queue has a reserve() member
};
```

Example

```
// PriorityQueue is expected to be a max-heap of integer values
template <typename PriorityQueue>
void basic_interface(void)
{
    PriorityQueue pq;

    pq.push(2);
    pq.push(3);
    pq.push(1);

    cout << "Priority Queue: popped elements" << endl;
    cout << pq.top() << " "; // 3
    pq.pop();
    cout << pq.top() << " "; // 2
    pq.pop();
    cout << pq.top() << " "; // 1
    pq.pop();
    cout << endl;
}
```

Priority Queue Iterators

Synopsis

```
class iteratable_heap_interface
{
public:
    // types
    typedef unspecified         iterator;
    typedef unspecified         const_iterator;
    typedef unspecified         ordered_iterator;

    // public member functions
    iterator begin(void) const;
    iterator end(void) const;
    ordered_iterator ordered_begin(void) const;
    ordered_iterator ordered_end(void) const;
};
```

Priority queues provide iterators, that can be used to traverse their elements. All heap iterators are const_iterators, that means they cannot be used to modify the values, because changing the value of a heap node may corrupt the heap order. Details about modifying heap nodes are described in the section about the mutability interface.

Iterators do not visit heap elements in any specific order. Unless otherwise noted, all non-const heap member functions invalidate iterators, while all const member functions preserve the iterator validity.

Note

Some implementations require iterators, that contain a set of elements, that are **discovered**, but not **visited**. Therefore copying iterators can be inefficient and should be avoided.

Example

```
// PriorityQueue is expected to be a max-heap of integer values
template <typename PriorityQueue>
void iterator_interface(void)
{
    PriorityQueue pq;

    pq.push(2);
    pq.push(3);
    pq.push(1);

    typename PriorityQueue::iterator begin = pq.begin();
    typename PriorityQueue::iterator end = pq.end();

    cout << "Priority Queue: iteration" << endl;
    for (typename PriorityQueue::iterator it = begin; it != end; ++it)
        cout << *it << " "; // 1, 2, 3 in unspecified order
    cout << endl;
}
```

Ordered Iterators

Except for `boost::heap::priority_queue` all `boost.heap` data structures support ordered iterators, which visit all elements of the heap in heap-order. The implementation of these `ordered_iterators` requires some internal bookkeeping, so iterating the a heap in heap order has an amortized complexity of O(N*log(N)).

Example

```
// PriorityQueue is expected to be a max-heap of integer values
template <typename PriorityQueue>
void ordered_iterator_interface(void)
{
    PriorityQueue pq;

    pq.push(2);
    pq.push(3);
    pq.push(1);

    typename PriorityQueue::ordered_iterator begin = pq.ordered_begin();
    typename PriorityQueue::ordered_iterator end = pq.ordered_end();

    cout << "Priority Queue: ordered iteration" << endl;
    for (typename PriorityQueue::ordered_iterator it = begin; it != end; ++it)
        cout << *it << " "; // 3, 2, 1 (i.e. 1, 2, 3 in heap order)
    cout << endl;
}
```

Comparing Priority Queues & Equivalence

The data structures of `boost.heap` can be compared with standard comparison operators. The comparison is performed by comparing two heaps element by element using `value_compare`.

Note

Depending on the heap type, this operation can be rather expensive, because both data structures need to be traversed in heap order. On heaps without ordered iterators, the heap needs to be copied internally. The typical complexity is O(n log(n)).

Merging Priority Queues

Mergable Priority Queues

Synopsis

```
class mergable_heap_interface
{
public:
    // public member functions
    void merge(mergable_heap_interface &);
};
```

`boost.heap` has a concept of a Mergable Priority Queue. A mergable priority queue can efficiently be merged with a different instance of the same type.

Example

```
// PriorityQueue is expected to be a max-heap of integer values
template <typename PriorityQueue>
void merge_interface(void)
{
    PriorityQueue pq;

    pq.push(3);
    pq.push(5);
    pq.push(1);

    PriorityQueue pq2;

    pq2.push(2);
    pq2.push(4);
    pq2.push(0);

    pq.merge(pq2);

    cout << "Priority Queue: merge" << endl;
    cout << "first queue: ";
    while (!pq.empty()) {
        cout << pq.top() << " "; // 5 4 3 2 1 0
        pq.pop();
    }
    cout << endl;

    cout << "second queue: ";
    while (!pq2.empty()) {
        cout << pq2.top() << " "; // 4 2 0
        pq2.pop();
    }
    cout << endl;
}
```

Heap Merge Algorithms

`boost.heap` provides a `heap_merge()` algorithm that is can be used to merge different kinds of heaps. Using this algorithm, all `boost.heap` data structures can be merged, although some cannot be merged efficiently.

Example

```
// PriorityQueue is expected to be a max-heap of integer values
template <typename PriorityQueue>
void heap_merge_algorithm(void)
{
    PriorityQueue pq;

    pq.push(3);
    pq.push(5);
    pq.push(1);

    PriorityQueue pq2;

    pq2.push(2);
    pq2.push(4);
    pq2.push(0);

    boost::heap::heap_merge(pq, pq2);

    cout << "Priority Queue: merge" << endl;
    cout << "first queue: ";
    while (!pq.empty()) {
        cout << pq.top() << " "; // 5 4 3 2 1 0
        pq.pop();
    }
    cout << endl;

    cout << "second queue: ";
    while (!pq2.empty()) {
        cout << pq2.top() << " "; // 4 2 0
        pq2.pop();
    }
    cout << endl;
}
```

Mutability

Some priority queues of `boost.heap` are mutable, that means the priority of their elements can be changed. To achieve mutability, `boost.heap` introduces the concept of **handles**, which can be used to access the internal nodes of the priority queue in order to change its value and to restore the heap order.

Synopsis

```
class mutable_heap_interface
{
public:
    typedef unspecified iterator;
    struct handle_type
    {
        value_type & operator*() const;
    };

    static handle_type s_iterator_to_handle(iterator const &);

    // priority queue interface
    handle_type push(T const & v);

    // update element via assignment and fix heap
    void update(handle_type const & handle, value_type const & v);
    void increase(handle_type const & handle, value_type const & v);
    void decrease(handle_type const & handle, value_type const & v);

    // fix heap after element has been changed via the handle
    void update(handle_type const & handle);
    void increase(handle_type const & handle);
    void decrease(handle_type const & handle);
};
```

Warning

Incorrect use of `increase` or `decrease` may corrupt the priority queue data structure. If unsure use `update` can be used at the cost of efficiency.

Example

```
// PriorityQueue is expected to be a max-heap of integer values
template <typename PriorityQueue>
void mutable_interface(void)
{
    PriorityQueue pq;
    typedef typename PriorityQueue::handle_type handle_t;

    handle_t t3 = pq.push(3);
    handle_t t5 = pq.push(5);
    handle_t t1 = pq.push(1);

    pq.update(t3, 4);
    pq.increase(t5, 7);
    pq.decrease(t1, 0);

    cout << "Priority Queue: update" << endl;
    while (!pq.empty()) {
        cout << pq.top() << " "; // 7, 4, 0
        pq.pop();
    }
    cout << endl;
}
```

Note that handles can be stored inside the `value_type`:

```
struct heap_data
{
    fibonacci_heap<heap_data>::handle_type handle;
    int payload;

    heap_data(int i):
        payload(i)
    {}

    bool operator<(heap_data const & rhs) const
    {
        return payload < rhs.payload;
    }
};

void mutable_interface_handle_in_value(void)
{
    fibonacci_heap<heap_data> heap;
    heap_data f(2);

    fibonacci_heap<heap_data>::handle_type handle = heap.push(f);
    (*handle).handle = handle; // store handle in node
}
```

The Fixup Interface

There are two different APIs to support mutability. The first family of functions provides update functionality by changing the current element by assigning a new value. The second family of functions can be used to fix the heap data structure after an element has been changed directly via a handle. While this provides the user with a means to modify the priority of queue elements without the need to change their non-priority part, this needs to be handled with care. The heap needs to be fixed up immediately after the priority of the element has been changed.

Beside an `update` function, two additional functions `increase` and `decrease` are provided, that are generally more efficient than the generic `update` function. However the user has do ensure, that the priority of an element is changed to the right direction.

Example

```
// PriorityQueue is expected to be a max-heap of integer values
template <typename PriorityQueue>
void mutable_fixup_interface(void)
{
    PriorityQueue pq;
    typedef typename PriorityQueue::handle_type handle_t;

    handle_t t3 = pq.push(3);
    handle_t t5 = pq.push(5);
    handle_t t1 = pq.push(1);

    *t3 = 4;
    pq.update(t3);

    *t5 = 7;
    pq.increase(t5);

    *t1 = 0;
    pq.decrease(t1);

    cout << "Priority Queue: update with fixup" << endl;
    while (!pq.empty()) {
        cout << pq.top() << " "; // 7, 4, 0
        pq.pop();
    }
    cout << endl;
}
```

Iterators can be coverted to handles using the static member function `s_handle_from_iterator`. However most implementations of `update` invalidate all iterators. The most notable exception is the `fibonacci heap`, providing a lazy update function, that just invalidates the iterators, that are related to this handle.

Warning

After changing the priority via a handle, the heap needs to be fixed by calling one of the update functions. Otherwise the priority queue structure may be corrupted!

Stability

A priority queue is `stable', if elements with the same priority are popped from the heap, in the same order as they are inserted. The data structures provided by `boost.heap`, can be configured to be stable at compile time using the `boost::heap::stable` policy. Two notions of stability are supported. If a heap is configured with **no stability**, the order of nodes of the same priority is undefined, if it is configured as **stable**, nodes of the same priority are ordered by their insertion time.

Stability is achieved by associating an integer version count with each value in order to distinguish values with the same node. The type of this version count defaults to `boost::uintmax_t`, which is at least 64bit on most systems. However it can be configured to use a different type using the `boost::heap::stability_counter_type` template argument.

Warning

The stability counter is prone to integer overflows. If an overflow occurs during a `push()` call, the operation will fail and an exception is thrown. Later `push()` call will succeed, but the stable heap order will be compromised. However an integer overflow at 64bit is very unlikely: if an application would issue one `push()` operation per microsecond, the value will overflow in more than 500000 years.

Data Structures

`boost.heap` provides the following data structures:

`boost::heap::priority_queue` The `priority_queue` class is a wrapper to the stl heap functions. It implements a heap as container adaptor ontop of a `std::vector` and is immutable.

`boost::heap::d_ary_heap` D-ary heaps are a generalization of binary heap with each non-leaf node having N children. For a low arity, the height of the heap is larger, but the number of comparisons to find the largest child node is bigger. D-ary heaps are implemented as container adaptors based on a `std::vector`.

The data structure can be configured as mutable. This is achieved by storing the values inside a std::list.

`boost::heap::binomial_heap` Binomial heaps are node-base heaps, that are implemented as a set of binomial trees of piecewise different order. The most important heap operations have a worst-case complexity of O(log n). In contrast to d-ary heaps, binomial heaps can also be merged in O(log n).

`boost::heap::fibonacci_heap` Fibonacci heaps are node-base heaps, that are implemented as a forest of heap-ordered trees. They provide better amortized performance than binomial heaps. Except `pop()` and `erase()`, the most important heap operations have constant amortized complexity.

`boost::heap::pairing_heap` Pairing heaps are self-adjusting node-based heaps. Although design and implementation are rather simple, the complexity analysis is yet unsolved. For details, consult:

Pettie, Seth (2005), "Towards a final analysis of pairing heaps", Proc. 46th Annual IEEE Symposium on Foundations of Computer Science, pp. 174–183

`boost::heap::skew_heap` Skew heaps are self-adjusting node-based heaps. Although there are no constraints for the tree structure, all heap operations can be performed in O(log n).

Table 1. Comparison of amortized complexity

	`top()`	`push()`	`pop()`	`update()`	`increase()`	`decrease()`	`erase()`	`merge()`
`boost::heap::priority_queue`	`O(1)`	O(log(N))	O(log(N))	n/a	n/a	n/a	n/a	O(N+M)
`boost::heap::d_ary_heap`	`O(1)`	O(log(N))	O(log(N))	O(log(N))	O(log(N))	O(log(N))	O(log(N))	O(N+M)
`boost::heap::binomial_heap`	`O(1)`	O(log(N))	O(log(N))	O(log(N))	O(log(N))	O(log(N))	O(log(N))	O(log(N+M))
`boost::heap::fibonacci_heap`	`O(1)`	O(1)	O(log(N))	O(log(N)) [a]	O(1)	O(log(N))	O(log(N))	O(1)
`boost::heap::pairing_heap`	`O(1)`	O(2**2*log(N))	O(log(N))	O(2**2*log(N))	O(2**2*log(N))	O(2**2*log(N))	O(2**2*log(N))	O(2**2*log(N))
`boost::heap::skew_heap`	`O(1)`	O(log(N))	O(log(N))	O(log(N))	O(log(N))	O(log(N))	O(log(N))	O(log(N+M))

[a] The fibonacci a `update_lazy()` method, which has O(log(N)) amortized complexity as well, but does not try to consolidate the internal forest

Data Structure Configuration

The data structures can be configured with Boost.Parameter-style templates.

`boost::heap::compare`	Predicate for defining the heap order, optional (defaults to `boost::heap::compare<std::less<T> >`)
`boost::heap::allocator`	Allocator for internal memory management, optional (defaults to `boost::heap::allocator<std::allocator<T> >`)
`boost::heap::stable`	Configures the heap to use a stable heap order, optional (defaults to `boost::heap::stable<false>`).
`boost::heap::mutable_`	Configures the heap to be mutable. `boost::heap::d_ary_heap` and `boost::heap::skew_heap` have to be configured with this policy to enable the mutability interface.
`boost::heap::stability_counter_type`	Configures the integer type used for the stability counter, optional (defaults to `boost::heap::stability_counter_type<boost::uintmax_t>`). For more details, consult the Stability section.
`boost::heap::constant_time_size`	Specifies, whether `size()` should have linear or constant complexity. This argument is only available for node-based heap data structures and if available, it is optional (defaults to `boost::heap::constant_time_size<true>`)
`boost::heap::arity`	Specifies the arity of a d-ary heap. For details, please consult the class reference of `boost::heap::d_ary_heap`
`boost::heap::store_parent_pointer`	Store the parent pointer in the heap nodes. This policy is only available in the `boost::heap::skew_heap`.

Reference

Header <boost/heap/binomial_heap.hpp>

```
namespace boost {
  namespace heap {
    template<typename T, class... Options> class binomial_heap;
  }
}
```

Class template binomial_heap

boost::heap::binomial_heap — binomial heap

Synopsis

```
// In header: <boost/heap/binomial_heap.hpp>

template<typename T, class... Options>
class binomial_heap {
public:
  // types
  typedef T                                          value_type;
  typedef implementation_defined::size_type          size_type;
  typedef implementation_defined::difference_type    difference_type;
  typedef implementation_defined::value_compare      value_compare;
  typedef implementation_defined::allocator_type     allocator_type;
  typedef implementation_defined::reference          reference;
  typedef implementation_defined::const_reference    const_reference;
  typedef implementation_defined::pointer            pointer;
  typedef implementation_defined::const_pointer      const_pointer;
  typedef implementation_defined::iterator           iterator;
  typedef implementation_defined::const_iterator     const_iterator;
  typedef implementation_defined::ordered_iterator   ordered_iterator;
  typedef implementation_defined::handle_type        handle_type;

  // construct/copy/destruct
  explicit binomial_heap(value_compare const & = value_compare());
  binomial_heap(binomial_heap const &);
  binomial_heap(binomial_heap &&);
  binomial_heap & operator=(binomial_heap const &);
  binomial_heap & operator=(binomial_heap &&);
  ~binomial_heap(void);

  // public member functions
  bool empty(void) const;
  size_type size(void) const;
  size_type max_size(void) const;
  void clear(void);
  allocator_type get_allocator(void) const;
  void swap(binomial_heap &);
  const_reference top(void) const;
  handle_type push(value_type const &);
  template<class... Args> handle_type emplace(Args &&...);
  void pop(void);
  void update(handle_type, const_reference);
  void update(handle_type);
  void increase(handle_type, const_reference);
```

```
  void increase(handle_type);
  void decrease(handle_type, const_reference);
  void decrease(handle_type);
  void merge(binomial_heap &);
  iterator begin(void) const;
  iterator end(void) const;
  ordered_iterator ordered_begin(void) const;
  ordered_iterator ordered_end(void) const;
  void erase(handle_type);
  value_compare const & value_comp(void) const;
  template<typename HeapType> bool operator<(HeapType const &) const;
  template<typename HeapType> bool operator>(HeapType const &) const;
  template<typename HeapType> bool operator>=(HeapType const &) const;
  template<typename HeapType> bool operator<=(HeapType const &) const;
  template<typename HeapType> bool operator==(HeapType const &) const;
  template<typename HeapType> bool operator!=(HeapType const &) const;

  // public static functions
  static handle_type s_handle_from_iterator(iterator const &);

  // public data members
  static const bool constant_time_size;
  static const bool has_ordered_iterators;
  static const bool is_mergable;
  static const bool is_stable;
  static const bool has_reserve;
};
```

Description

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:

- `boost::heap::stable<>`, defaults to `stable<false>`
- `boost::heap::compare<>`, defaults to `compare<std::less<T> >`
- `boost::heap::allocator<>`, defaults to `allocator<std::allocator<T> >`
- `boost::heap::constant_time_size<>`, defaults to `constant_time_size<true>`
- `boost::heap::stability_counter_type<>`, defaults to `stability_counter_type<boost::uintmax_t>`

`binomial_heap` public types

1. typedef implementation_defined::iterator iterator;

   Note: The iterator does not traverse the priority queue in order of the priorities.

`binomial_heap` public construct/copy/destruct

1. ```
   explicit binomial_heap(value_compare const & cmp = value_compare());
   ```

   Effects: constructs an empty priority queue.

   Complexity: Constant.

2. ```
   binomial_heap(binomial_heap const & rhs);
   ```

Effects: copy-constructs priority queue from rhs.

Complexity: Linear.

3. ```
   binomial_heap(binomial_heap && rhs);
   ```

   Effects: C++11-style move constructor.

   Complexity: Constant.

   Note: Only available, if BOOST_NO_CXX11_RVALUE_REFERENCES is not defined

4. ```
   binomial_heap & operator=(binomial_heap const & rhs);
   ```

   Effects: Assigns priority queue from rhs.

   Complexity: Linear.

5. ```
   binomial_heap & operator=(binomial_heap && rhs);
   ```

   Effects: C++11-style move assignment.

   Complexity: Constant.

   Note: Only available, if BOOST_NO_CXX11_RVALUE_REFERENCES is not defined

6. ```
   ~binomial_heap(void);
   ```

`binomial_heap` public member functions

1. ```
   bool empty(void) const;
   ```

   Effects: Returns true, if the priority queue contains no elements.

   Complexity: Constant.

2. ```
   size_type size(void) const;
   ```

   Effects: Returns the number of elements contained in the priority queue.

   Complexity: Constant, if configured with constant_time_size<true>, otherwise linear.

3. ```
   size_type max_size(void) const;
   ```

   Effects: Returns the maximum number of elements the priority queue can contain.

   Complexity: Constant.

4. ```
   void clear(void);
   ```

   Effects: Removes all elements from the priority queue.

   Complexity: Linear.

5. ```
   allocator_type get_allocator(void) const;
   ```

   Effects: Returns allocator.

   Complexity: Constant.

6. ```
   void swap(binomial_heap & rhs);
   ```

   Effects: Swaps two priority queues.

   Complexity: Constant.

7. ```
   const_reference top(void) const;
   ```

   Effects: Returns a const_reference to the maximum element.

   Complexity: Constant.

8. ```
   handle_type push(value_type const & v);
   ```

   Effects: Adds a new element to the priority queue. Returns handle to element

   Complexity: Logarithmic.

9. ```
   template<class... Args> handle_type emplace(Args &&... args);
   ```

   Effects: Adds a new element to the priority queue. The element is directly constructed in-place. Returns handle to element.

   Complexity: Logarithmic.

10. ```
    void pop(void);
    ```

    Effects: Removes the top element from the priority queue.

    Complexity: Logarithmic.

11. ```
    void update(handle_type handle, const_reference v);
    ```

    Effects: Assigns `v` to the element handled by `handle` & updates the priority queue.

    Complexity: Logarithmic.

12. ```
    void update(handle_type handle);
    ```

    Effects: Updates the heap after the element handled by `handle` has been changed.

    Complexity: Logarithmic.

    Note: If this is not called, after a handle has been updated, the behavior of the data structure is undefined!

13. ```
    void increase(handle_type handle, const_reference v);
    ```

    Effects: Assigns `v` to the element handled by `handle` & updates the priority queue.
    ```

Complexity: Logarithmic.

Note: The new value is expected to be greater than the current one

14.
```
void increase(handle_type handle);
```

Effects: Updates the heap after the element handled by `handle` has been changed.

Complexity: Logarithmic.

Note: If this is not called, after a handle has been updated, the behavior of the data structure is undefined!

15.
```
void decrease(handle_type handle, const_reference v);
```

Effects: Assigns `v` to the element handled by `handle` & updates the priority queue.

Complexity: Logarithmic.

Note: The new value is expected to be less than the current one

16.
```
void decrease(handle_type handle);
```

Effects: Updates the heap after the element handled by `handle` has been changed.

Complexity: Logarithmic.

Note: The new value is expected to be less than the current one. If this is not called, after a handle has been updated, the behavior of the data structure is undefined!

17.
```
void merge(binomial_heap & rhs);
```

Effects: Merge with priority queue rhs.

Complexity: Logarithmic.

18.
```
iterator begin(void) const;
```

Effects: Returns an iterator to the first element contained in the priority queue.

Complexity: Constant.

19.
```
iterator end(void) const;
```

Effects: Returns an iterator to the end of the priority queue.

Complexity: Constant.

20.
```
ordered_iterator ordered_begin(void) const;
```

Effects: Returns an ordered iterator to the first element contained in the priority queue.

Note: Ordered iterators traverse the priority queue in heap order.

21.
```
ordered_iterator ordered_end(void) const;
```

Effects: Returns an ordered iterator to the first element contained in the priority queue.

Note: Ordered iterators traverse the priority queue in heap order.

22.
```
void erase(handle_type handle);
```

Effects: Removes the element handled by `handle` from the `priority_queue`.

Complexity: Logarithmic.

23.
```
value_compare const & value_comp(void) const;
```

Effect: Returns the value_compare object used by the priority queue

24.
```
template<typename HeapType> bool operator<(HeapType const & rhs) const;
```

Returns: Element-wise comparison of heap data structures

Requirement: the `value_compare` object of both heaps must match.

25.
```
template<typename HeapType> bool operator>(HeapType const & rhs) const;
```

Returns: Element-wise comparison of heap data structures

Requirement: the `value_compare` object of both heaps must match.

26.
```
template<typename HeapType> bool operator>=(HeapType const & rhs) const;
```

Returns: Element-wise comparison of heap data structures

Requirement: the `value_compare` object of both heaps must match.

27.
```
template<typename HeapType> bool operator<=(HeapType const & rhs) const;
```

Returns: Element-wise comparison of heap data structures

Requirement: the `value_compare` object of both heaps must match.

28.
```
template<typename HeapType> bool operator==(HeapType const & rhs) const;
```

Equivalent comparison **Returns:** True, if both heap data structures are equivalent.

Requirement: the `value_compare` object of both heaps must match.

29.
```
template<typename HeapType> bool operator!=(HeapType const & rhs) const;
```

Equivalent comparison **Returns:** True, if both heap data structures are not equivalent.

Requirement: the `value_compare` object of both heaps must match.

`binomial_heap` public static functions

1.
```
static handle_type s_handle_from_iterator(iterator const & it);
```

Header <boost/heap/d_ary_heap.hpp>

```
namespace boost {
  namespace heap {
    template<typename T, class... Options> class d_ary_heap;
  }
}
```

Class template d_ary_heap

boost::heap::d_ary_heap — d-ary heap class

Synopsis

```
// In header: <boost/heap/d_ary_heap.hpp>

template<typename T, class... Options>
class d_ary_heap {
public:
  // types
  typedef T                                          value_type;
  typedef implementation_defined::size_type          size_type;
  typedef implementation_defined::difference_type    difference_type;
  typedef implementation_defined::value_compare      value_compare;
  typedef implementation_defined::allocator_type     allocator_type;
  typedef implementation_defined::reference          reference;
  typedef implementation_defined::const_reference    const_reference;
  typedef implementation_defined::pointer            pointer;
  typedef implementation_defined::const_pointer      const_pointer;
  typedef implementation_defined::iterator           iterator;
  typedef implementation_defined::const_iterator     const_iterator;
  typedef implementation_defined::ordered_iterator   ordered_iterator;
  typedef implementation_defined::handle_type        handle_type;

  // construct/copy/destruct
  explicit d_ary_heap(value_compare const & = value_compare());
  d_ary_heap(d_ary_heap const &);
  d_ary_heap(d_ary_heap &&);
  d_ary_heap & operator=(d_ary_heap &&);
  d_ary_heap & operator=(d_ary_heap const &);

  // public member functions
  bool empty(void) const;
  size_type size(void) const;
  size_type max_size(void) const;
  void clear(void);
  allocator_type get_allocator(void) const;
  value_type const & top(void) const;
  mpl::if_c< is_mutable, handle_type, void >::type push(value_type const &);
  template<class... Args>
    mpl::if_c< is_mutable, handle_type, void >::type emplace(Args &&...);
  template<typename HeapType> bool operator<(HeapType const &) const;
  template<typename HeapType> bool operator>(HeapType const &) const;
  template<typename HeapType> bool operator>=(HeapType const &) const;
  template<typename HeapType> bool operator<=(HeapType const &) const;
  template<typename HeapType> bool operator==(HeapType const &) const;
  template<typename HeapType> bool operator!=(HeapType const &) const;
  void update(handle_type, const_reference);
  void update(handle_type);
```

```
  void increase(handle_type, const_reference);
  void increase(handle_type);
  void decrease(handle_type, const_reference);
  void decrease(handle_type);
  void erase(handle_type);
  void pop(void);
  void swap(d_ary_heap &);
  const_iterator begin(void) const;
  iterator begin(void);
  iterator end(void);
  const_iterator end(void) const;
  ordered_iterator ordered_begin(void) const;
  ordered_iterator ordered_end(void) const;
  void reserve(size_type);
  value_compare const & value_comp(void) const;

  // public static functions
  static handle_type s_handle_from_iterator(iterator const &);

  // public data members
  static const bool constant_time_size;
  static const bool has_ordered_iterators;
  static const bool is_mergable;
  static const bool has_reserve;
  static const bool is_stable;
};
```

Description

This class implements an immutable priority queue. Internally, the d-ary heap is represented as dynamically sized array (std::vector), that directly stores the values.

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:

- `boost::heap::arity<>`, required
- `boost::heap::compare<>`, defaults to `compare<std::less<T> >`
- `boost::heap::stable<>`, defaults to `stable<false>`
- `boost::heap::stability_counter_type<>`, defaults to `stability_counter_type<boost::uintmax_t>`
- `boost::heap::allocator<>`, defaults to `allocator<std::allocator<T> >`
- `boost::heap::mutable_<>`, defaults to `mutable_<false>`

`d_ary_heap` public types

1. typedef implementation_defined::iterator iterator;

   Note: The iterator does not traverse the priority queue in order of the priorities.

`d_ary_heap` public construct/copy/destruct

1. ```
   explicit d_ary_heap(value_compare const & cmp = value_compare());
   ```

   Effects: constructs an empty priority queue.
   ```

Complexity: Constant.

2. ```
   d_ary_heap(d_ary_heap const & rhs);
   ```

   Effects: copy-constructs priority queue from rhs.

   Complexity: Linear.

3. ```
   d_ary_heap(d_ary_heap && rhs);
   ```

   Effects: C++11-style move constructor.

   Complexity: Constant.

   Note: Only available, if BOOST_NO_CXX11_RVALUE_REFERENCES is not defined

4. ```
   d_ary_heap & operator=(d_ary_heap && rhs);
   ```

   Effects: C++11-style move assignment.

   Complexity: Constant.

   Note: Only available, if BOOST_NO_CXX11_RVALUE_REFERENCES is not defined

5. ```
   d_ary_heap & operator=(d_ary_heap const & rhs);
   ```

   Effects: Assigns priority queue from rhs.

   Complexity: Linear.

`d_ary_heap` public member functions

1. ```
   bool empty(void) const;
   ```

   Effects: Returns true, if the priority queue contains no elements.

   Complexity: Constant.

2. ```
   size_type size(void) const;
   ```

   Effects: Returns the number of elements contained in the priority queue.

   Complexity: Constant.

3. ```
   size_type max_size(void) const;
   ```

   Effects: Returns the maximum number of elements the priority queue can contain.

   Complexity: Constant.

4. ```
   void clear(void);
   ```

   Effects: Removes all elements from the priority queue.

   Complexity: Linear.

5. ```
   allocator_type get_allocator(void) const;
   ```

   Effects: Returns allocator.

   Complexity: Constant.

6. ```
   value_type const & top(void) const;
   ```

   Effects: Returns a const_reference to the maximum element.

   Complexity: Constant.

7. ```
   mpl::if_c< is_mutable, handle_type, void >::type push(value_type const & v);
   ```

   Effects: Adds a new element to the priority queue.

   Complexity: Logarithmic (amortized). Linear (worst case).

8. ```
   template<class... Args>
     mpl::if_c< is_mutable, handle_type, void >::type emplace(Args &&... args);
   ```

   Effects: Adds a new element to the priority queue. The element is directly constructed in-place.

   Complexity: Logarithmic (amortized). Linear (worst case).

9. ```
   template<typename HeapType> bool operator<(HeapType const & rhs) const;
   ```

   Returns: Element-wise comparison of heap data structures

   Requirement: the `value_compare` object of both heaps must match.

10. ```
    template<typename HeapType> bool operator>(HeapType const & rhs) const;
    ```

    Returns: Element-wise comparison of heap data structures

    Requirement: the `value_compare` object of both heaps must match.

11. ```
    template<typename HeapType> bool operator>=(HeapType const & rhs) const;
    ```

    Returns: Element-wise comparison of heap data structures

    Requirement: the `value_compare` object of both heaps must match.

12. ```
    template<typename HeapType> bool operator<=(HeapType const & rhs) const;
    ```

    Returns: Element-wise comparison of heap data structures

    Requirement: the `value_compare` object of both heaps must match.

13. ```
    template<typename HeapType> bool operator==(HeapType const & rhs) const;
    ```

    Equivalent comparison **Returns:** True, if both heap data structures are equivalent.

    Requirement: the `value_compare` object of both heaps must match.
    ```

14. ```
template<typename HeapType> bool operator!=(HeapType const & rhs) const;
```

Equivalent comparison **Returns:** True, if both heap data structures are not equivalent.

**Requirement:** the `value_compare` object of both heaps must match.

15. ```
void update(handle_type handle, const_reference v);
```

Effects: Assigns `v` to the element handled by `handle` & updates the priority queue.

Complexity: Logarithmic.

Requirement: data structure must be configured as mutable

16. ```
void update(handle_type handle);
```

**Effects:** Updates the heap after the element handled by `handle` has been changed.

**Complexity:** Logarithmic.

**Note:** If this is not called, after a handle has been updated, the behavior of the data structure is undefined!

**Requirement:** data structure must be configured as mutable

17. ```
void increase(handle_type handle, const_reference v);
```

Effects: Assigns `v` to the element handled by `handle` & updates the priority queue.

Complexity: Logarithmic.

Note: The new value is expected to be greater than the current one

Requirement: data structure must be configured as mutable

18. ```
void increase(handle_type handle);
```

**Effects:** Updates the heap after the element handled by `handle` has been changed.

**Complexity:** Logarithmic.

**Note:** The new value is expected to be greater than the current one. If this is not called, after a handle has been updated, the behavior of the data structure is undefined!

**Requirement:** data structure must be configured as mutable

19. ```
void decrease(handle_type handle, const_reference v);
```

Effects: Assigns `v` to the element handled by `handle` & updates the priority queue.

Complexity: Logarithmic.

Note: The new value is expected to be less than the current one

Requirement: data structure must be configured as mutable

20. ```
void decrease(handle_type handle);
```
```

Effects: Updates the heap after the element handled by `handle` has been changed.

Complexity: Logarithmic.

Note: The new value is expected to be less than the current one. If this is not called, after a handle has been updated, the behavior of the data structure is undefined!

Requirement: data structure must be configured as mutable

21.
```
void erase(handle_type handle);
```

Effects: Removes the element handled by `handle` from the `priority_queue`.

Complexity: Logarithmic.

Requirement: data structure must be configured as mutable

22.
```
void pop(void);
```

Effects: Removes the top element from the priority queue.

Complexity: Logarithmic (amortized). Linear (worst case).

23.
```
void swap(d_ary_heap & rhs);
```

Effects: Swaps two priority queues.

Complexity: Constant.

24.
```
const_iterator begin(void) const;
```

Effects: Returns an iterator to the first element contained in the priority queue.

Complexity: Constant.

25.
```
iterator begin(void);
```

Effects: Returns an iterator to the first element contained in the priority queue.

Complexity: Constant.

26.
```
iterator end(void);
```

Effects: Returns an iterator to the end of the priority queue.

Complexity: Constant.

27.
```
const_iterator end(void) const;
```

Effects: Returns an iterator to the end of the priority queue.

Complexity: Constant.

28.
```
ordered_iterator ordered_begin(void) const;
```

Effects: Returns an ordered iterator to the first element contained in the priority queue.

Note: Ordered iterators traverse the priority queue in heap order.

29.
```
ordered_iterator ordered_end(void) const;
```

Effects: Returns an ordered iterator to the first element contained in the priority queue.

Note: Ordered iterators traverse the priority queue in heap order.

30.
```
void reserve(size_type element_count);
```

Effects: Reserves memory for element_count elements

Complexity: Linear.

Node: Invalidates iterators

31.
```
value_compare const & value_comp(void) const;
```

Effect: Returns the value_compare object used by the priority queue

`d_ary_heap` public static functions

1.
```
static handle_type s_handle_from_iterator(iterator const & it);
```

Effects: Casts an iterator to a node handle.

Complexity: Constant.

Requirement: data structure must be configured as mutable

Header <boost/heap/fibonacci_heap.hpp>

```
namespace boost {
  namespace heap {
    template<typename T, class... Options> class fibonacci_heap;
  }
}
```

Class template fibonacci_heap

boost::heap::fibonacci_heap — fibonacci heap

Synopsis

```
// In header: <boost/heap/fibonacci_heap.hpp>

template<typename T, class... Options>
class fibonacci_heap {
public:
  // types
  typedef T                                          value_type;
  typedef implementation_defined::size_type          size_type;
  typedef implementation_defined::difference_type    difference_type;
  typedef implementation_defined::value_compare      value_compare;
  typedef implementation_defined::allocator_type     allocator_type;
  typedef implementation_defined::reference          reference;
  typedef implementation_defined::const_reference    const_reference;
  typedef implementation_defined::pointer            pointer;
  typedef implementation_defined::const_pointer      const_pointer;
  typedef implementation_defined::iterator           iterator;
  typedef implementation_defined::const_iterator     const_iterator;
  typedef implementation_defined::ordered_iterator   ordered_iterator;
  typedef implementation_defined::handle_type        handle_type;

  // construct/copy/destruct
  explicit fibonacci_heap(value_compare const & = value_compare());
  fibonacci_heap(fibonacci_heap const &);
  fibonacci_heap(fibonacci_heap &&);
  fibonacci_heap(fibonacci_heap &);
  fibonacci_heap & operator=(fibonacci_heap &&);
  fibonacci_heap & operator=(fibonacci_heap const &);
  ~fibonacci_heap(void);

  // public member functions
  bool empty(void) const;
  size_type size(void) const;
  size_type max_size(void) const;
  void clear(void);
  allocator_type get_allocator(void) const;
  void swap(fibonacci_heap &);
  value_type const & top(void) const;
  handle_type push(value_type const &);
  template<class... Args> handle_type emplace(Args &&...);
  void pop(void);
  void update(handle_type, const_reference);
  void update_lazy(handle_type, const_reference);
  void update(handle_type);
  void update_lazy(handle_type);
  void increase(handle_type, const_reference);
  void increase(handle_type);
  void decrease(handle_type, const_reference);
  void decrease(handle_type);
  void erase(handle_type const &);
  iterator begin(void) const;
  iterator end(void) const;
  ordered_iterator ordered_begin(void) const;
  ordered_iterator ordered_end(void) const;
  void merge(fibonacci_heap &);
  value_compare const & value_comp(void) const;
  template<typename HeapType> bool operator<(HeapType const &) const;
  template<typename HeapType> bool operator>(HeapType const &) const;
  template<typename HeapType> bool operator>=(HeapType const &) const;
  template<typename HeapType> bool operator<=(HeapType const &) const;
  template<typename HeapType> bool operator==(HeapType const &) const;
```

```
  template<typename HeapType> bool operator!=(HeapType const &) const;

  // public static functions
  static handle_type s_handle_from_iterator(iterator const &);

  // public data members
  static const bool constant_time_size;
  static const bool has_ordered_iterators;
  static const bool is_mergable;
  static const bool is_stable;
  static const bool has_reserve;
};
```

Description

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:

- `boost::heap::stable<>`, defaults to `stable<false>`
- `boost::heap::compare<>`, defaults to `compare<std::less<T> >`
- `boost::heap::allocator<>`, defaults to `allocator<std::allocator<T> >`
- `boost::heap::constant_time_size<>`, defaults to `constant_time_size<true>`
- `boost::heap::stability_counter_type<>`, defaults to `stability_counter_type<boost::uintmax_t>`

`fibonacci_heap` public types

1. typedef implementation_defined::iterator iterator;

   Note: The iterator does not traverse the priority queue in order of the priorities.

`fibonacci_heap` public construct/copy/destruct

1. ```
   explicit fibonacci_heap(value_compare const & cmp = value_compare());
   ```

   Effects: constructs an empty priority queue.

   Complexity: Constant.

2. ```
   fibonacci_heap(fibonacci_heap const & rhs);
   ```

   Effects: copy-constructs priority queue from rhs.

   Complexity: Linear.

3. ```
   fibonacci_heap(fibonacci_heap && rhs);
   ```

   Effects: C++11-style move constructor.

   Complexity: Constant.

   Note: Only available, if BOOST_NO_CXX11_RVALUE_REFERENCES is not defined
   ```

4. ```
   fibonacci_heap(fibonacci_heap & rhs);
   ```

5. ```
   fibonacci_heap & operator=(fibonacci_heap && rhs);
   ```

   Effects: C++11-style move assignment.

   Complexity: Constant.

   Note: Only available, if BOOST_NO_CXX11_RVALUE_REFERENCES is not defined

6. ```
   fibonacci_heap & operator=(fibonacci_heap const & rhs);
   ```

   Effects: Assigns priority queue from rhs.

   Complexity: Linear.

7. ```
   ~fibonacci_heap(void);
   ```

`fibonacci_heap` public member functions

1. ```
   bool empty(void) const;
   ```

   Effects: Returns true, if the priority queue contains no elements.

   Complexity: Constant.

2. ```
   size_type size(void) const;
   ```

   Effects: Returns the number of elements contained in the priority queue.

   Complexity: Constant.

3. ```
   size_type max_size(void) const;
   ```

   Effects: Returns the maximum number of elements the priority queue can contain.

   Complexity: Constant.

4. ```
   void clear(void);
   ```

   Effects: Removes all elements from the priority queue.

   Complexity: Linear.

5. ```
   allocator_type get_allocator(void) const;
   ```

   Effects: Returns allocator.

   Complexity: Constant.

6. ```
   void swap(fibonacci_heap & rhs);
   ```

Effects: Swaps two priority queues.

Complexity: Constant.

7.
```
value_type const & top(void) const;
```

Effects: Returns a const_reference to the maximum element.

Complexity: Constant.

8.
```
handle_type push(value_type const & v);
```

Effects: Adds a new element to the priority queue. Returns handle to element

Complexity: Constant.

Note: Does not invalidate iterators.

9.
```
template<class... Args> handle_type emplace(Args &&... args);
```

Effects: Adds a new element to the priority queue. The element is directly constructed in-place. Returns handle to element.

Complexity: Constant.

Note: Does not invalidate iterators.

10.
```
void pop(void);
```

Effects: Removes the top element from the priority queue.

Complexity: Logarithmic (amortized). Linear (worst case).

11.
```
void update(handle_type handle, const_reference v);
```

Effects: Assigns `v` to the element handled by `handle` & updates the priority queue.

Complexity: Logarithmic if current value < v, Constant otherwise.

12.
```
void update_lazy(handle_type handle, const_reference v);
```

Effects: Assigns `v` to the element handled by `handle` & updates the priority queue.

Complexity: Logarithmic if current value < v, Constant otherwise.

Rationale: The lazy update function is a modification of the traditional update, that just invalidates the iterator to the object referred to by the handle.

13.
```
void update(handle_type handle);
```

Effects: Updates the heap after the element handled by `handle` has been changed.

Complexity: Logarithmic.

Note: If this is not called, after a handle has been updated, the behavior of the data structure is undefined!

14. ```
    void update_lazy(handle_type handle);
    ```

    (handle_type handle)

    Effects: Assigns `v` to the element handled by `handle` & updates the priority queue.

    Complexity: Logarithmic if current value < v, Constant otherwise. (handle_type handle)

    Rationale: The lazy update function is a modification of the traditional update, that just invalidates the iterator to the object referred to by the handle.

15. ```
    void increase(handle_type handle, const_reference v);
    ```

    Effects: Assigns `v` to the element handled by `handle` & updates the priority queue.

    Complexity: Constant.

    Note: The new value is expected to be greater than the current one

16. ```
    void increase(handle_type handle);
    ```

    Effects: Updates the heap after the element handled by `handle` has been changed.

    Complexity: Constant.

    Note: If this is not called, after a handle has been updated, the behavior of the data structure is undefined!

17. ```
    void decrease(handle_type handle, const_reference v);
    ```

    Effects: Assigns `v` to the element handled by `handle` & updates the priority queue.

    Complexity: Logarithmic.

    Note: The new value is expected to be less than the current one

18. ```
    void decrease(handle_type handle);
    ```

    Effects: Updates the heap after the element handled by `handle` has been changed.

    Complexity: Logarithmic.

    Note: The new value is expected to be less than the current one. If this is not called, after a handle has been updated, the behavior of the data structure is undefined!

19. ```
    void erase(handle_type const & handle);
    ```

    Effects: Removes the element handled by `handle` from the `priority_queue`.

    Complexity: Logarithmic.

20. ```
    iterator begin(void) const;
    ```

    Effects: Returns an iterator to the first element contained in the priority queue.

    Complexity: Constant.
    ```

21. ```
iterator end(void) const;
```

**Effects:** Returns an iterator to the end of the priority queue.

**Complexity:** Constant.

22. ```
ordered_iterator ordered_begin(void) const;
```

Effects: Returns an ordered iterator to the first element contained in the priority queue.

Note: Ordered iterators traverse the priority queue in heap order.

23. ```
ordered_iterator ordered_end(void) const;
```

**Effects:** Returns an ordered iterator to the first element contained in the priority queue.

**Note:** Ordered iterators traverse the priority queue in heap order.

24. ```
void merge(fibonacci_heap & rhs);
```

Effects: Merge with priority queue rhs.

Complexity: Constant.

25. ```
value_compare const & value_comp(void) const;
```

**Effect:** Returns the value_compare object used by the priority queue

26. ```
template<typename HeapType> bool operator<(HeapType const & rhs) const;
```

Returns: Element-wise comparison of heap data structures

Requirement: the `value_compare` object of both heaps must match.

27. ```
template<typename HeapType> bool operator>(HeapType const & rhs) const;
```

**Returns:** Element-wise comparison of heap data structures

**Requirement:** the `value_compare` object of both heaps must match.

28. ```
template<typename HeapType> bool operator>=(HeapType const & rhs) const;
```

Returns: Element-wise comparison of heap data structures

Requirement: the `value_compare` object of both heaps must match.

29. ```
template<typename HeapType> bool operator<=(HeapType const & rhs) const;
```

**Returns:** Element-wise comparison of heap data structures

**Requirement:** the `value_compare` object of both heaps must match.

30. ```
template<typename HeapType> bool operator==(HeapType const & rhs) const;
```

Equivalent comparison **Returns:** True, if both heap data structures are equivalent.

Requirement: the `value_compare` object of both heaps must match.

31.
```
template<typename HeapType> bool operator!=(HeapType const & rhs) const;
```

Equivalent comparison **Returns:** True, if both heap data structures are not equivalent.

Requirement: the `value_compare` object of both heaps must match.

`fibonacci_heap` public static functions

1.
```
static handle_type s_handle_from_iterator(iterator const & it);
```

Header <boost/heap/heap_concepts.hpp>

```
namespace boost {
  namespace heap {
    template<typename C> struct MergablePriorityQueue;
    template<typename C> struct MutablePriorityQueue;
    template<typename C> struct PriorityQueue;
  }
}
```

Struct template MergablePriorityQueue

boost::heap::MergablePriorityQueue

Synopsis

```
// In header: <boost/heap/heap_concepts.hpp>

template<typename C>
struct MergablePriorityQueue : public boost::heap::PriorityQueue< C > {
  // types
  typedef C::iterator        iterator;
  typedef C::const_iterator  const_iterator;
  typedef C::allocator_type  allocator_type;
  typedef C::value_compare   value_compare;
  typedef C::value_type      value_type;
  typedef C::const_reference const_reference;

  // public member functions
   BOOST_CONCEPT_USAGE(MergablePriorityQueue);
   BOOST_CONCEPT_USAGE(PriorityQueue);
};
```

Description

`MergablePriorityQueue` public member functions

1.
```
 BOOST_CONCEPT_USAGE(MergablePriorityQueue);
```

2.
```
BOOST_CONCEPT_USAGE(PriorityQueue);
```

Struct template MutablePriorityQueue

boost::heap::MutablePriorityQueue

Synopsis

```
// In header: <boost/heap/heap_concepts.hpp>

template<typename C>
struct MutablePriorityQueue : public boost::heap::PriorityQueue< C > {
  // types
  typedef C::handle_type     handle_type;
  typedef C::iterator        iterator;
  typedef C::const_iterator  const_iterator;
  typedef C::allocator_type  allocator_type;
  typedef C::value_compare   value_compare;
  typedef C::value_type      value_type;
  typedef C::const_reference const_reference;

  // public member functions
   BOOST_CONCEPT_USAGE(MutablePriorityQueue);
   BOOST_CONCEPT_USAGE(PriorityQueue);

  // public data members
  C c;
  bool equal;
  bool not_equal;
};
```

Description

`MutablePriorityQueue` public member functions

1.
```
BOOST_CONCEPT_USAGE(MutablePriorityQueue);
```

2.
```
BOOST_CONCEPT_USAGE(PriorityQueue);
```

Struct template PriorityQueue

boost::heap::PriorityQueue

Synopsis

```
// In header: <boost/heap/heap_concepts.hpp>

template<typename C>
struct PriorityQueue : public boost::ForwardContainer< C > {
  // types
  typedef C::iterator        iterator;
  typedef C::const_iterator  const_iterator;
  typedef C::allocator_type  allocator_type;
  typedef C::value_compare   value_compare;
  typedef C::value_type      value_type;
  typedef C::const_reference const_reference;

  // public member functions
   BOOST_CONCEPT_USAGE(PriorityQueue);
};
```

Description

`PriorityQueue` public member functions

1. ```
   BOOST_CONCEPT_USAGE(PriorityQueue);
   ```

Header <boost/heap/heap_merge.hpp>

```
namespace boost {
  namespace heap {
    template<typename Heap1, typename Heap2> void heap_merge(Heap1 &, Heap2 &);
  }
}
```

Function template heap_merge

boost::heap::heap_merge

Synopsis

```
// In header: <boost/heap/heap_merge.hpp>


template<typename Heap1, typename Heap2>
  void heap_merge(Heap1 & lhs, Heap2 & rhs);
```

Description

merge rhs into lhs

Effect: lhs contains all elements that have been part of rhs, rhs is empty.
```

Header <boost/heap/pairing_heap.hpp>

```
namespace boost {
  namespace heap {
    template<typename T, class... Options> class pairing_heap;
  }
}
```

Class template pairing_heap

boost::heap::pairing_heap — pairing heap

Synopsis

```
// In header: <boost/heap/pairing_heap.hpp>

template<typename T, class... Options>
class pairing_heap {
public:
  // types
  typedef T                                          value_type;
  typedef implementation_defined::size_type          size_type;
  typedef implementation_defined::difference_type    difference_type;
  typedef implementation_defined::value_compare      value_compare;
  typedef implementation_defined::allocator_type     allocator_type;
  typedef implementation_defined::reference          reference;
  typedef implementation_defined::const_reference    const_reference;
  typedef implementation_defined::pointer            pointer;
  typedef implementation_defined::const_pointer      const_pointer;
  typedef implementation_defined::iterator           iterator;
  typedef implementation_defined::const_iterator     const_iterator;
  typedef implementation_defined::ordered_iterator   ordered_iterator;
  typedef implementation_defined::handle_type        handle_type;

  // construct/copy/destruct
  explicit pairing_heap(value_compare const & = value_compare());
  pairing_heap(pairing_heap const &);
  pairing_heap(pairing_heap &&);
  pairing_heap & operator=(pairing_heap &&);
  pairing_heap & operator=(pairing_heap const &);
  ~pairing_heap(void);

  // public member functions
  bool empty(void) const;
  size_type size(void) const;
  size_type max_size(void) const;
  void clear(void);
  allocator_type get_allocator(void) const;
  void swap(pairing_heap &);
  const_reference top(void) const;
  handle_type push(value_type const &);
  template<class... Args> handle_type emplace(Args &&...);
  void pop(void);
  void update(handle_type, const_reference);
  void update(handle_type);
  void increase(handle_type, const_reference);
  void increase(handle_type);
  void decrease(handle_type, const_reference);
  void decrease(handle_type);
```

```
  void erase(handle_type);
  iterator begin(void) const;
  iterator end(void) const;
  ordered_iterator ordered_begin(void) const;
  ordered_iterator ordered_end(void) const;
  void merge(pairing_heap &);
  value_compare const & value_comp(void) const;
  template<typename HeapType> bool operator<(HeapType const &) const;
  template<typename HeapType> bool operator>(HeapType const &) const;
  template<typename HeapType> bool operator>=(HeapType const &) const;
  template<typename HeapType> bool operator<=(HeapType const &) const;
  template<typename HeapType> bool operator==(HeapType const &) const;
  template<typename HeapType> bool operator!=(HeapType const &) const;

  // public static functions
  static handle_type s_handle_from_iterator(iterator const &);

  // public data members
  static const bool constant_time_size;
  static const bool has_ordered_iterators;
  static const bool is_mergable;
  static const bool is_stable;
  static const bool has_reserve;
};
```

Description

Pairing heaps are self-adjusting binary heaps. Although design and implementation are rather simple, the complexity analysis is yet unsolved. For details, consult:

Pettie, Seth (2005), "Towards a final analysis of pairing heaps", Proc. 46th Annual IEEE Symposium on Foundations of Computer Science, pp. 174-183

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:

- `boost::heap::compare<>`, defaults to `compare<std::less<T> >`
- `boost::heap::stable<>`, defaults to `stable<false>`
- `boost::heap::stability_counter_type<>`, defaults to `stability_counter_type<boost::uintmax_t>`
- `boost::heap::allocator<>`, defaults to `allocator<std::allocator<T> >`
- `boost::heap::constant_time_size<>`, defaults to `constant_time_size<true>`

`pairing_heap` public types

1. typedef implementation_defined::iterator iterator;

   Note: The iterator does not traverse the priority queue in order of the priorities.

`pairing_heap` public construct/copy/destruct

1. ```
   explicit pairing_heap(value_compare const & cmp = value_compare());
   ```

   Effects: constructs an empty priority queue.

   Complexity: Constant.
   ```

2. ```
   pairing_heap(pairing_heap const & rhs);
   ```

   Effects: copy-constructs priority queue from rhs.

   Complexity: Linear.

3. ```
   pairing_heap(pairing_heap && rhs);
   ```

   Effects: C++11-style move constructor.

   Complexity: Constant.

   Note: Only available, if BOOST_NO_CXX11_RVALUE_REFERENCES is not defined

4. ```
   pairing_heap & operator=(pairing_heap && rhs);
   ```

   Effects: C++11-style move assignment.

   Complexity: Constant.

   Note: Only available, if BOOST_NO_CXX11_RVALUE_REFERENCES is not defined

5. ```
   pairing_heap & operator=(pairing_heap const & rhs);
   ```

   Effects: Assigns priority queue from rhs.

   Complexity: Linear.

6. ```
   ~pairing_heap(void);
   ```

`pairing_heap` public member functions

1. ```
   bool empty(void) const;
   ```

   Effects: Returns true, if the priority queue contains no elements.

   Complexity: Constant.

2. ```
   size_type size(void) const;
   ```

   Effects: Returns the number of elements contained in the priority queue.

   Complexity: Constant, if configured with constant_time_size<true>, otherwise linear.

3. ```
   size_type max_size(void) const;
   ```

   Effects: Returns the maximum number of elements the priority queue can contain.

   Complexity: Constant.

4. ```
   void clear(void);
   ```

   Effects: Removes all elements from the priority queue.
   ```

Complexity: Linear.

5.
```
allocator_type get_allocator(void) const;
```

Effects: Returns allocator.

Complexity: Constant.

6.
```
void swap(pairing_heap & rhs);
```

Effects: Swaps two priority queues.

Complexity: Constant.

7.
```
const_reference top(void) const;
```

Effects: Returns a const_reference to the maximum element.

Complexity: Constant.

8.
```
handle_type push(value_type const & v);
```

Effects: Adds a new element to the priority queue. Returns handle to element

Complexity: 2**2*log(log(N)) (amortized).

9.
```
template<class... Args> handle_type emplace(Args &&... args);
```

Effects: Adds a new element to the priority queue. The element is directly constructed in-place. Returns handle to element.

Complexity: 2**2*log(log(N)) (amortized).

10.
```
void pop(void);
```

Effects: Removes the top element from the priority queue.

Complexity: Logarithmic (amortized).

11.
```
void update(handle_type handle, const_reference v);
```

Effects: Assigns `v` to the element handled by `handle` & updates the priority queue.

Complexity: 2**2*log(log(N)) (amortized).

12.
```
void update(handle_type handle);
```

Effects: Updates the heap after the element handled by `handle` has been changed.

Complexity: 2**2*log(log(N)) (amortized).

Note: If this is not called, after a handle has been updated, the behavior of the data structure is undefined!

13.
```
void increase(handle_type handle, const_reference v);
```

Effects: Assigns `v` to the element handled by `handle` & updates the priority queue.

Complexity: 2**2*log(log(N)) (amortized).

Note: The new value is expected to be greater than the current one

14.
```
void increase(handle_type handle);
```

Effects: Updates the heap after the element handled by `handle` has been changed.

Complexity: 2**2*log(log(N)) (amortized).

Note: If this is not called, after a handle has been updated, the behavior of the data structure is undefined!

15.
```
void decrease(handle_type handle, const_reference v);
```

Effects: Assigns `v` to the element handled by `handle` & updates the priority queue.

Complexity: 2**2*log(log(N)) (amortized).

Note: The new value is expected to be less than the current one

16.
```
void decrease(handle_type handle);
```

Effects: Updates the heap after the element handled by `handle` has been changed.

Complexity: 2**2*log(log(N)) (amortized).

Note: The new value is expected to be less than the current one. If this is not called, after a handle has been updated, the behavior of the data structure is undefined!

17.
```
void erase(handle_type handle);
```

Effects: Removes the element handled by `handle` from the `priority_queue`.

Complexity: 2**2*log(log(N)) (amortized).

18.
```
iterator begin(void) const;
```

Effects: Returns an iterator to the first element contained in the priority queue.

Complexity: Constant.

19.
```
iterator end(void) const;
```

Effects: Returns an iterator to the end of the priority queue.

Complexity: Constant.

20.
```
ordered_iterator ordered_begin(void) const;
```

Effects: Returns an ordered iterator to the first element contained in the priority queue.

Note: Ordered iterators traverse the priority queue in heap order.

21. ```
ordered_iterator ordered_end(void) const;
```

**Effects:** Returns an ordered iterator to the first element contained in the priority queue.

**Note:** Ordered iterators traverse the priority queue in heap order.

22. ```
void merge(pairing_heap & rhs);
```

Effects: Merge all elements from rhs into this

Complexity: 2**2*log(log(N)) (amortized).

23. ```
value_compare const & value_comp(void) const;
```

**Effect:** Returns the value_compare object used by the priority queue

24. ```
template<typename HeapType> bool operator<(HeapType const & rhs) const;
```

Returns: Element-wise comparison of heap data structures

Requirement: the `value_compare` object of both heaps must match.

25. ```
template<typename HeapType> bool operator>(HeapType const & rhs) const;
```

**Returns:** Element-wise comparison of heap data structures

**Requirement:** the `value_compare` object of both heaps must match.

26. ```
template<typename HeapType> bool operator>=(HeapType const & rhs) const;
```

Returns: Element-wise comparison of heap data structures

Requirement: the `value_compare` object of both heaps must match.

27. ```
template<typename HeapType> bool operator<=(HeapType const & rhs) const;
```

**Returns:** Element-wise comparison of heap data structures

**Requirement:** the `value_compare` object of both heaps must match.

28. ```
template<typename HeapType> bool operator==(HeapType const & rhs) const;
```

Equivalent comparison **Returns:** True, if both heap data structures are equivalent.

Requirement: the `value_compare` object of both heaps must match.

29. ```
template<typename HeapType> bool operator!=(HeapType const & rhs) const;
```

Equivalent comparison **Returns:** True, if both heap data structures are not equivalent.

**Requirement:** the `value_compare` object of both heaps must match.
```

`pairing_heap` public static functions

1. ```
   static handle_type s_handle_from_iterator(iterator const & it);
   ```

Header <boost/heap/policies.hpp>

```
namespace boost {
  namespace heap {
    template<typename T> struct allocator;
    template<unsigned int T> struct arity;
    template<typename T> struct compare;
    template<bool T> struct constant_time_size;
    template<bool T> struct mutable_;
    template<typename IntType> struct stability_counter_type;
    template<bool T> struct stable;
    template<bool T> struct store_parent_pointer;
  }
}
```

Struct template allocator

boost::heap::allocator — Specifies allocator for the internal memory management.

Synopsis

```
// In header: <boost/heap/policies.hpp>

template<typename T>
struct allocator {
};
```

Struct template arity

boost::heap::arity — Specify arity.

Synopsis

```
// In header: <boost/heap/policies.hpp>

template<unsigned int T>
struct arity {
};
```

Description

Specifies the arity of a D-ary heap

Struct template compare

boost::heap::compare — Specifies the predicate for the heap order.
```

Synopsis

```
// In header: <boost/heap/policies.hpp>

template<typename T>
struct compare {
};
```

Struct template constant_time_size

boost::heap::constant_time_size — Configures complexity of `size()`

Synopsis

```
// In header: <boost/heap/policies.hpp>

template<bool T>
struct constant_time_size {
};
```

Description

Specifies, whether size() should have linear or constant complexity.

Struct template mutable_

boost::heap::mutable_ — Configure heap as mutable.

Synopsis

```
// In header: <boost/heap/policies.hpp>

template<bool T>
struct mutable_ {
};
```

Description

Certain heaps need to be configured specifically do be mutable.

Struct template stability_counter_type

boost::heap::stability_counter_type — Specifies the type for stability counter.

Synopsis

```
// In header: <boost/heap/policies.hpp>

template<typename IntType>
struct stability_counter_type {
};
```

Struct template stable

boost::heap::stable — Configure a heap as **stable**.

Synopsis

```
// In header: <boost/heap/policies.hpp>

template<bool T>
struct stable {
};
```

Description

A priority queue is stable, if elements with the same priority are popped from the heap, in the same order as they are inserted.

Struct template store_parent_pointer

boost::heap::store_parent_pointer — Store parent pointer in heap node.

Synopsis

```
// In header: <boost/heap/policies.hpp>

template<bool T>
struct store_parent_pointer {
};
```

Description

Maintaining a parent pointer adds some maintenance and size overhead, but iterating a heap is more efficient.

Header <boost/heap/priority_queue.hpp>

```
namespace boost {
  namespace heap {
    template<typename T, class... Options> class priority_queue;
  }
}
```

Class template priority_queue

boost::heap::priority_queue — priority queue, based on stl heap functions

Synopsis

```
// In header: <boost/heap/priority_queue.hpp>

template<typename T, class... Options>
class priority_queue {
public:
  // types
  typedef T                                        value_type;
  typedef implementation_defined::size_type        size_type;
  typedef implementation_defined::difference_type  difference_type;
  typedef implementation_defined::value_compare    value_compare;
  typedef implementation_defined::allocator_type   allocator_type;
  typedef implementation_defined::reference        reference;
  typedef implementation_defined::const_reference  const_reference;
  typedef implementation_defined::pointer          pointer;
  typedef implementation_defined::const_pointer    const_pointer;
  typedef implementation_defined::iterator         iterator;
  typedef implementation_defined::const_iterator   const_iterator;

  // construct/copy/destruct
  explicit priority_queue(value_compare const & = value_compare());
  priority_queue(priority_queue const &);
  priority_queue(priority_queue &&);
  priority_queue & operator=(priority_queue &&);
  priority_queue & operator=(priority_queue const &);

  // public member functions
  bool empty(void) const;
  size_type size(void) const;
  size_type max_size(void) const;
  void clear(void);
  allocator_type get_allocator(void) const;
  const_reference top(void) const;
  void push(value_type const &);
  template<class... Args> void emplace(Args &&...);
  void pop(void);
  void swap(priority_queue &);
  iterator begin(void) const;
  iterator end(void) const;
  void reserve(size_type);
  value_compare const & value_comp(void) const;
  template<typename HeapType> bool operator<(HeapType const &) const;
  template<typename HeapType> bool operator>(HeapType const &) const;
  template<typename HeapType> bool operator>=(HeapType const &) const;
  template<typename HeapType> bool operator<=(HeapType const &) const;
  template<typename HeapType> bool operator==(HeapType const &) const;
  template<typename HeapType> bool operator!=(HeapType const &) const;

  // public data members
  static const bool constant_time_size;
  static const bool has_ordered_iterators;
  static const bool is_mergable;
  static const bool is_stable;
  static const bool has_reserve;
};
```

Description

The priority_queue class is a wrapper for the stl heap functions.
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:

- `boost::heap::compare<>`, defaults to `compare<std::less<T> >`
- `boost::heap::stable<>`, defaults to `stable<false>`
- `boost::heap::stability_counter_type<>`, defaults to `stability_counter_type<boost::uintmax_t>`
- `boost::heap::allocator<>`, defaults to `allocator<std::allocator<T> >`

`priority_queue` public types

1. typedef implementation_defined::iterator iterator;

   Note: The iterator does not traverse the priority queue in order of the priorities.

`priority_queue` public construct/copy/destruct

1. ```
   explicit priority_queue(value_compare const & cmp = value_compare());
   ```

   Effects: constructs an empty priority queue.

   Complexity: Constant.

2. ```
   priority_queue(priority_queue const & rhs);
   ```

   Effects: copy-constructs priority queue from rhs.

   Complexity: Linear.

3. ```
   priority_queue(priority_queue && rhs);
   ```

   Effects: C++11-style move constructor.

   Complexity: Constant.

   Note: Only available, if BOOST_NO_CXX11_RVALUE_REFERENCES is not defined

4. ```
   priority_queue & operator=(priority_queue && rhs);
   ```

   Effects: C++11-style move assignment.

   Complexity: Constant.

   Note: Only available, if BOOST_NO_CXX11_RVALUE_REFERENCES is not defined

5. ```
   priority_queue & operator=(priority_queue const & rhs);
   ```

   Effects: Assigns priority queue from rhs.

   Complexity: Linear.

`priority_queue` public member functions

1. ```
   bool empty(void) const;
   ```

   Effects: Returns true, if the priority queue contains no elements.

Complexity: Constant.

2. ```
   size_type size(void) const;
   ```

   Effects: Returns the number of elements contained in the priority queue.

   Complexity: Constant.

3. ```
   size_type max_size(void) const;
   ```

   Effects: Returns the maximum number of elements the priority queue can contain.

   Complexity: Constant.

4. ```
   void clear(void);
   ```

   Effects: Removes all elements from the priority queue.

   Complexity: Linear.

5. ```
   allocator_type get_allocator(void) const;
   ```

   Effects: Returns allocator.

   Complexity: Constant.

6. ```
   const_reference top(void) const;
   ```

   Effects: Returns a const_reference to the maximum element.

   Complexity: Constant.

7. ```
   void push(value_type const & v);
   ```

   Effects: Adds a new element to the priority queue.

   Complexity: Logarithmic (amortized). Linear (worst case).

8. ```
   template<class... Args> void emplace(Args &&... args);
   ```

   Effects: Adds a new element to the priority queue. The element is directly constructed in-place.

   Complexity: Logarithmic (amortized). Linear (worst case).

9. ```
   void pop(void);
   ```

   Effects: Removes the top element from the priority queue.

   Complexity: Logarithmic (amortized). Linear (worst case).

10. ```
    void swap(priority_queue & rhs);
    ```

    Effects: Swaps two priority queues.
    ```

Complexity: Constant.

11. ```
iterator begin(void) const;
```

**Effects:** Returns an iterator to the first element contained in the priority queue.

**Complexity:** Constant.

12. ```
iterator end(void) const;
```

Effects: Returns an iterator to the end of the priority queue.

Complexity: Constant.

13. ```
void reserve(size_type element_count);
```

**Effects:** Reserves memory for element_count elements

**Complexity:** Linear.

**Node:** Invalidates iterators

14. ```
value_compare const & value_comp(void) const;
```

Effect: Returns the value_compare object used by the priority queue

15. ```
template<typename HeapType> bool operator<(HeapType const & rhs) const;
```

**Returns:** Element-wise comparison of heap data structures

**Requirement:** the `value_compare` object of both heaps must match.

16. ```
template<typename HeapType> bool operator>(HeapType const & rhs) const;
```

Returns: Element-wise comparison of heap data structures

Requirement: the `value_compare` object of both heaps must match.

17. ```
template<typename HeapType> bool operator>=(HeapType const & rhs) const;
```

**Returns:** Element-wise comparison of heap data structures

**Requirement:** the `value_compare` object of both heaps must match.

18. ```
template<typename HeapType> bool operator<=(HeapType const & rhs) const;
```

Returns: Element-wise comparison of heap data structures

Requirement: the `value_compare` object of both heaps must match.

19. ```
template<typename HeapType> bool operator==(HeapType const & rhs) const;
```

Equivalent comparison **Returns:** True, if both heap data structures are equivalent.
```

Requirement: the `value_compare` object of both heaps must match.

20.
```
template<typename HeapType> bool operator!=(HeapType const & rhs) const;
```

Equivalent comparison **Returns:** True, if both heap data structures are not equivalent.

Requirement: the `value_compare` object of both heaps must match.

Header <boost/heap/skew_heap.hpp>

```
namespace boost {
  namespace heap {
    template<typename T, class... Options> class skew_heap;
  }
}
```

Class template skew_heap

boost::heap::skew_heap — skew heap

Synopsis

```
// In header: <boost/heap/skew_heap.hpp>

template<typename T, class... Options>
class skew_heap {
public:
  // types
  typedef T                                                                  ↵
 value_type;
  typedef implementation_defined::size_type                                  ↵
 size_type;
  typedef implementation_defined::difference_type                            ↵
 difference_type;
  typedef implementation_defined::value_compare                              ↵
 value_compare;
  typedef implementation_defined::allocator_type                             al↵
locator_type;
  typedef implementation_defined::reference                                  ↵
 reference;
  typedef implementation_defined::const_reference                            ↵
 const_reference;
  typedef implementation_defined::pointer                                    ↵
 pointer;
  typedef implementation_defined::const_pointer                              ↵
 const_pointer;
  typedef implementation_defined::iterator                                   ↵
 iterator;
  typedef implementation_defined::const_iterator                             ↵
 const_iterator;
  typedef implementation_defined::ordered_iterator                           ↵
 ordered_iterator;
  typedef mpl::if_c< is_mutable, typename implementa↵
tion_defined::handle_type, void * >::type handle_type;

  // member classes/structs/unions

  struct implementation_defined {
    // types
    typedef T                                value_type;
    typedef base_maker::compare_argument     value_compare;
    typedef base_maker::allocator_type       allocator_type;
    typedef base_maker::node_type            node;
    typedef allocator_type::pointer          node_pointer;
    typedef allocator_type::const_pointer    const_node_pointer;
    typedef unspecified                      value_extractor;
    typedef boost::array< node_pointer, 2 > child_list_type;
    typedef child_list_type::iterator        child_list_iterator;
    typedef unspecified                      iterator;
    typedef iterator                         const_iterator;
    typedef unspecified                      ordered_iterator;
    typedef unspecified                      reference;
    typedef unspecified                      handle_type;
  };

  // construct/copy/destruct
  explicit skew_heap(value_compare const & = value_compare());
  skew_heap(skew_heap const &);
  skew_heap(skew_heap &&);
  skew_heap & operator=(skew_heap const &);
  skew_heap & operator=(skew_heap &&);
  ~skew_heap(void);
```

```
  // public member functions
  mpl::if_c< is_mutable, handle_type, void >::type push(value_type const &);
  template<typename... Args>
    mpl::if_c< is_mutable, handle_type, void >::type emplace(Args &&...);
  bool empty(void) const;
  size_type size(void) const;
  size_type max_size(void) const;
  void clear(void);
  allocator_type get_allocator(void) const;
  void swap(skew_heap &);
  const_reference top(void) const;
  void pop(void);
  iterator begin(void) const;
  iterator end(void) const;
  ordered_iterator ordered_begin(void) const;
  ordered_iterator ordered_end(void) const;
  void merge(skew_heap &);
  value_compare const & value_comp(void) const;
  template<typename HeapType> bool operator<(HeapType const &) const;
  template<typename HeapType> bool operator>(HeapType const &) const;
  template<typename HeapType> bool operator>=(HeapType const &) const;
  template<typename HeapType> bool operator<=(HeapType const &) const;
  template<typename HeapType> bool operator==(HeapType const &) const;
  template<typename HeapType> bool operator!=(HeapType const &) const;
  void erase(handle_type);
  void update(handle_type, const_reference);
  void update(handle_type);
  void increase(handle_type, const_reference);
  void increase(handle_type);
  void decrease(handle_type, const_reference);
  void decrease(handle_type);

  // public static functions
  static handle_type s_handle_from_iterator(iterator const &);

  // public data members
  static const bool constant_time_size;
  static const bool has_ordered_iterators;
  static const bool is_mergable;
  static const bool is_stable;
  static const bool has_reserve;
  static const bool is_mutable;
};
```

Description

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:

- `boost::heap::compare<>`, defaults to `compare<std::less<T> >`
- `boost::heap::stable<>`, defaults to `stable<false>`
- `boost::heap::stability_counter_type<>`, defaults to `stability_counter_type<boost::uintmax_t>`
- `boost::heap::allocator<>`, defaults to `allocator<std::allocator<T> >`
- `boost::heap::constant_time_size<>`, defaults to `constant_time_size<true>`

- `boost::heap::store_parent_pointer<>`, defaults to `store_parent_pointer<true>`. Maintaining a parent pointer adds some maintenance and size overhead, but iterating a heap is more efficient.
- `boost::heap::mutable<>`, defaults to `mutable<false>`.

skew_heap public types

1. typedef implementation_defined::iterator iterator;

   Note: The iterator does not traverse the priority queue in order of the priorities.

skew_heap public construct/copy/destruct

1. ```
   explicit skew_heap(value_compare const & cmp = value_compare());
   ```

   Effects: constructs an empty priority queue.

   Complexity: Constant.

2. ```
   skew_heap(skew_heap const & rhs);
   ```

   Effects: copy-constructs priority queue from rhs.

   Complexity: Linear.

3. ```
   skew_heap(skew_heap && rhs);
   ```

   Effects: C++11-style move constructor.

   Complexity: Constant.

   Note: Only available, if BOOST_NO_CXX11_RVALUE_REFERENCES is not defined

4. ```
   skew_heap & operator=(skew_heap const & rhs);
   ```

   Effects: Assigns priority queue from rhs.

   Complexity: Linear.

5. ```
   skew_heap & operator=(skew_heap && rhs);
   ```

   Effects: C++11-style move assignment.

   Complexity: Constant.

   Note: Only available, if BOOST_NO_CXX11_RVALUE_REFERENCES is not defined

6. ```
   ~skew_heap(void);
   ```

skew_heap public member functions

1. ```
   mpl::if_c< is_mutable, handle_type, void >::type push(value_type const & v);
   ```

   Effects: Adds a new element to the priority queue.
   ```

Complexity: Logarithmic (amortized).

2.
```
template<typename... Args>
  mpl::if_c< is_mutable, handle_type, void >::type emplace(Args &&... args);
```

Effects: Adds a new element to the priority queue. The element is directly constructed in-place.

Complexity: Logarithmic (amortized).

3.
```
bool empty(void) const;
```

Effects: Returns true, if the priority queue contains no elements.

Complexity: Constant.

4.
```
size_type size(void) const;
```

Effects: Returns the number of elements contained in the priority queue.

Complexity: Constant, if configured with constant_time_size<true>, otherwise linear.

5.
```
size_type max_size(void) const;
```

Effects: Returns the maximum number of elements the priority queue can contain.

Complexity: Constant.

6.
```
void clear(void);
```

Effects: Removes all elements from the priority queue.

Complexity: Linear.

7.
```
allocator_type get_allocator(void) const;
```

Effects: Returns allocator.

Complexity: Constant.

8.
```
void swap(skew_heap & rhs);
```

Effects: Swaps two priority queues.

Complexity: Constant.

9.
```
const_reference top(void) const;
```

Effects: Returns a const_reference to the maximum element.

Complexity: Constant.

10.
```
void pop(void);
```

Effects: Removes the top element from the priority queue.

Complexity: Logarithmic (amortized).

11.
```
iterator begin(void) const;
```

Effects: Returns an iterator to the first element contained in the priority queue.

Complexity: Constant.

12.
```
iterator end(void) const;
```

Effects: Returns an iterator to the end of the priority queue.

Complexity: Constant.

13.
```
ordered_iterator ordered_begin(void) const;
```

Effects: Returns an ordered iterator to the first element contained in the priority queue.

Note: Ordered iterators traverse the priority queue in heap order.

14.
```
ordered_iterator ordered_end(void) const;
```

Effects: Returns an ordered iterator to the first element contained in the priority queue.

Note: Ordered iterators traverse the priority queue in heap order.

15.
```
void merge(skew_heap & rhs);
```

Effects: Merge all elements from rhs into this

Complexity: Logarithmic (amortized).

16.
```
value_compare const & value_comp(void) const;
```

Effect: Returns the value_compare object used by the priority queue

17.
```
template<typename HeapType> bool operator<(HeapType const & rhs) const;
```

Returns: Element-wise comparison of heap data structures

Requirement: the `value_compare` object of both heaps must match.

18.
```
template<typename HeapType> bool operator>(HeapType const & rhs) const;
```

Returns: Element-wise comparison of heap data structures

Requirement: the `value_compare` object of both heaps must match.

19.
```
template<typename HeapType> bool operator>=(HeapType const & rhs) const;
```

Returns: Element-wise comparison of heap data structures

Requirement: the `value_compare` object of both heaps must match.

20. ```
template<typename HeapType> bool operator<=(HeapType const & rhs) const;
```

**Returns:** Element-wise comparison of heap data structures

**Requirement:** the `value_compare` object of both heaps must match.

21. ```
template<typename HeapType> bool operator==(HeapType const & rhs) const;
```

Equivalent comparison **Returns:** True, if both heap data structures are equivalent.

Requirement: the `value_compare` object of both heaps must match.

22. ```
template<typename HeapType> bool operator!=(HeapType const & rhs) const;
```

Equivalent comparison **Returns:** True, if both heap data structures are not equivalent.

**Requirement:** the `value_compare` object of both heaps must match.

23. ```
void erase(handle_type object);
```

Effects: Removes the element handled by `handle` from the `priority_queue`.

Complexity: Logarithmic (amortized).

24. ```
void update(handle_type handle, const_reference v);
```

**Effects:** Assigns `v` to the element handled by `handle` & updates the priority queue.

**Complexity:** Logarithmic (amortized).

25. ```
void update(handle_type handle);
```

Effects: Updates the heap after the element handled by `handle` has been changed.

Complexity: Logarithmic (amortized).

Note: If this is not called, after a handle has been updated, the behavior of the data structure is undefined!

26. ```
void increase(handle_type handle, const_reference v);
```

**Effects:** Assigns `v` to the element handled by `handle` & updates the priority queue.

**Complexity:** Logarithmic (amortized).

**Note:** The new value is expected to be greater than the current one

27. ```
void increase(handle_type handle);
```

Effects: Updates the heap after the element handled by `handle` has been changed.

Complexity: Logarithmic (amortized).

Note: If this is not called, after a handle has been updated, the behavior of the data structure is undefined!

28. ```
    void decrease(handle_type handle, const_reference v);
    ```

    Effects: Assigns `v` to the element handled by `handle` & updates the priority queue.

    Complexity: Logarithmic (amortized).

    Note: The new value is expected to be less than the current one

29. ```
    void decrease(handle_type handle);
    ```

    Effects: Updates the heap after the element handled by `handle` has been changed.

    Complexity: Logarithmic (amortized).

    Note: The new value is expected to be less than the current one. If this is not called, after a handle has been updated, the behavior of the data structure is undefined!

`skew_heap` public static functions

1. ```
    static handle_type s_handle_from_iterator(iterator const & it);
    ```

    Effects: Casts an iterator to a node handle.

    Complexity: Constant.

    Requirement: data structure must be configured as mutable

Struct implementation_defined

boost::heap::skew_heap::implementation_defined

Synopsis

```
// In header: <boost/heap/skew_heap.hpp>


struct implementation_defined {
  // types
  typedef T                                value_type;
  typedef base_maker::compare_argument     value_compare;
  typedef base_maker::allocator_type       allocator_type;
  typedef base_maker::node_type            node;
  typedef allocator_type::pointer          node_pointer;
  typedef allocator_type::const_pointer    const_node_pointer;
  typedef unspecified                      value_extractor;
  typedef boost::array< node_pointer, 2 > child_list_type;
  typedef child_list_type::iterator        child_list_iterator;
  typedef unspecified                      iterator;
  typedef iterator                         const_iterator;
  typedef unspecified                      ordered_iterator;
  typedef unspecified                      reference;
  typedef unspecified                      handle_type;
};
```
```

Acknowledgements

Google Inc.	For sponsoring the development of this library during the Summer of Code 2010
Hartmut Kaiser	For mentoring the Summer of Code project