Chapter 17. Boost.Heap

Boost.Heap could have also been called Boost.PriorityQueue since the library provides several priority queues. However, the priority queues in Boost.Heap differ from `std::priority_queue` by supporting more functions.

Example 17.1. Using `boost::heap::priority_queue`

```
#include <boost/heap/priority_queue.hpp>
#include <iostream>

using namespace boost::heap;

int main()
{
  priority_queue<int> pq;
  pq.push(2);
  pq.push(3);
  pq.push(1);

  for (int i : pq)
    std::cout << i << '\n';

  priority_queue<int> pq2;
  pq2.push(4);
  std::cout << std::boolalpha << (pq > pq2) << '\n';
}
```

Example 17.1 uses the class `boost::heap::priority_queue`, which is defined in `boost/heap/priority_queue.hpp`. In general this class behaves like `std::priority_queue`, except it allows you to iterate over elements. The order of elements returned in the iteration is random.

Objects of type `boost::heap::priority_queue` can be compared with each other. The comparison in Example 17.1 returns `true` because **pq** has more elements than **pq2**. If both queues had the same number of elements, the elements would be compared in pairs.

Example 17.2. Using `boost::heap::binomial_heap`

```
#include <boost/heap/binomial_heap.hpp>
#include <iostream>

using namespace boost::heap;

int main()
{
  binomial_heap<int> bh;
  bh.push(2);
  bh.push(3);
  bh.push(1);

  binomial_heap<int> bh2;
  bh2.push(4);
  bh.merge(bh2);

  for (auto it = bh.ordered_begin(); it != bh.ordered_end(); ++it)
    std::cout << *it << '\n';
  std::cout << std::boolalpha << bh2.empty() << '\n';
}
```

Example 17.2 introduces the class `boost::heap::binomial_heap`. In addition to allowing you to iterate over elements in priority order, it also lets you merge priority queues. Elements from one queue can be added to another queue.

The example calls `merge()` on the queue **bh**. The queue **bh2** is passed as a parameter. The call to `merge()` moves the number 4 from **bh2** to **bh**. After the call, **bh** contains four numbers, and **bh2** is empty.

The `for` loop calls `ordered_begin()` and `ordered_end()` on **bh**. `ordered_begin()` returns an iterator that iterates from high priority elements to low priority elements. Thus, Example 17.2 writes the numbers 4, 3, 2, and 1 in order to standard output.

Example 17.3. Changing elements in `boost::heap::binomial_heap`

```
#include <boost/heap/binomial_heap.hpp>
#include <iostream>

using namespace boost::heap;

int main()
{
  binomial_heap<int> bh;
  auto handle = bh.push(2);
  bh.push(3);
  bh.push(1);

  bh.update(handle, 4);

  std::cout << bh.top() << '\n';
}
```

`boost::heap::binomial_heap` lets you change elements after they have been added to the queue. Example 17.3 saves a handle returned by `push()`, making it possible to access the number 2 stored in **bh**.

`update()` is a member function of `boost::heap::binomial_heap` that can be called to change an element. Example 17.3 calls the member function to replace 2 with 4. Afterwards, the element with the highest priority, now 4, is fetched with `top()`.

In addition to `update()`, `boost::heap::binomial_heap` provides other member functions to change elements. The member functions `increase()` or `decrease()` can be called if you know in advance whether a change will result in a higher or lower priority. In Example 17.3, the call to `update()` could be replaced with a call to `increase()` since the number is increased from 2 to 4.

Boost.Heap provides additional priority queues whose member functions mainly differ in their runtime complexity. For example, you can use the class `boost::heap::fibonacci_heap` if you want the member function `push()` to have a constant runtime complexity. The documentation on Boost.Heap provides a table with an overview of the runtime complexities of the various classes and functions.