The implementation of skew heaps is left as a (trivial) exercise. Note that because a right path could be long, a recursive implementation could fail because of lack of stack space, even though performance would otherwise be acceptable. Skew heaps have the advantage that no extra space is required to maintain path lengths and no tests are required to determine when to swap children. It is an open problem to determine precisely the expected right path length of both leftist and skew heaps (the latter is undoubtedly more difficult). Such a comparison would make it easier to determine whether the slight loss of balance information is compensated by the lack of testing.

6.8 Binomial Queues

Although both leftist and skew heaps support merging, insertion, and `deleteMin` all effectively in $O(\log N)$ time per operation, there is room for improvement because we know that binary heaps support insertion in *constant average* time per operation. Binomial queues support all three operations in $O(\log N)$ worst-case time per operation, but insertions take constant time on average.

6.8.1 Binomial Queue Structure

No. of elements in $B_k = 2^k$

Binomial queues differ from all the priority queue implementations that we have seen in that a binomial queue is not a heap-ordered tree but rather a *collection* of heap-ordered trees, known as a **forest.** Each of the heap-ordered trees is of a constrained form known as a **binomial tree** (the reason for the name will be obvious later). There is at most one binomial tree of every height. A binomial tree of height 0 is a one-node tree; a binomial tree, B_k, of height k is formed by attaching a binomial tree, B_{k-1}, to the root of another binomial tree, B_{k-1}. Figure 6.34 shows binomial trees B_0, B_1, B_2, B_3, and B_4.

From the diagram we see that a binomial tree, B_k, consists of a root with children $B_0, B_1, \ldots, B_{k-1}$. Binomial trees of height k have exactly 2^k nodes, and the number of nodes at depth d is the binomial coefficient $\binom{k}{d}$. If we impose heap order on the binomial trees and allow at most one binomial tree of any height, we can represent a priority queue of any size by a collection of binomial trees. For instance, a priority queue of size 13 could be represented by the forest B_3, B_2, B_0. We might write this representation as 1101, which not only represents 13 in binary but also represents the fact that B_3, B_2, and B_0 are present in the representation and B_1 is not.

As an example, a priority queue of six elements could be represented as in Figure 6.35.

6.8.2 Binomial Queue Operations

The minimum element can then be found by scanning the roots of all the trees. Since there are at most $\log N$ different trees, the minimum can be found in $O(\log N)$ time. Alternatively, we can maintain knowledge of the minimum and perform the operation in $O(1)$ time if we remember to update the minimum when it changes during other operations.

Merging two binomial queues is a conceptually easy operation, which we will describe by example. Consider the two binomial queues, H_1 and H_2, with six and seven elements, respectively, pictured in Figure 6.36.

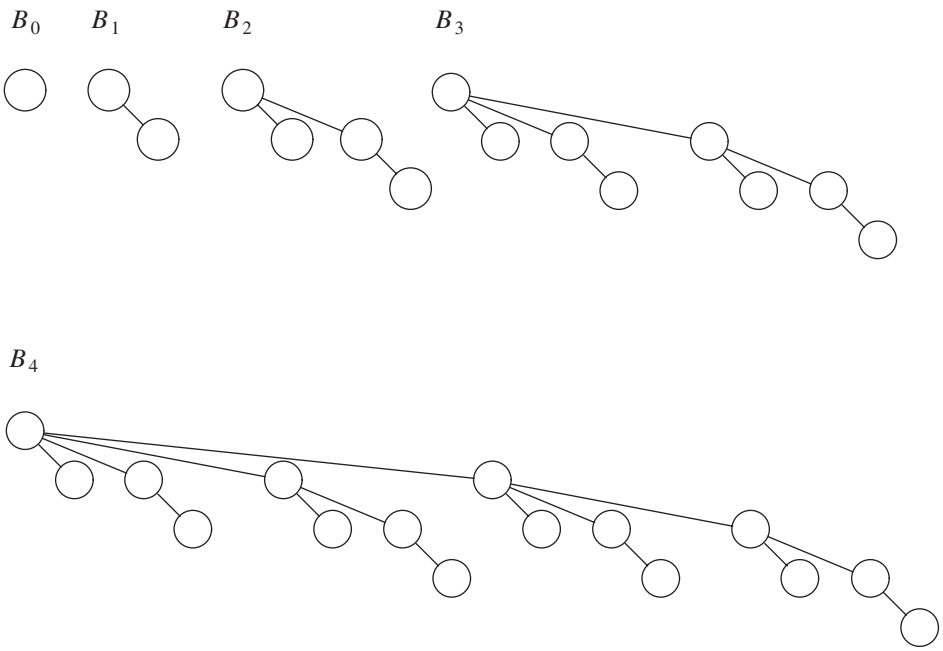

Figure 6.34 Binomial trees B_0, B_1, B_2, B_3, and B_4

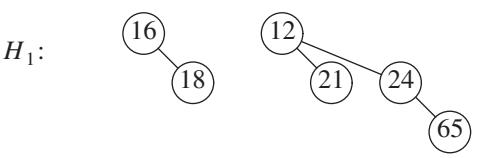

Figure 6.35 Binomial queue H_1 with six elements

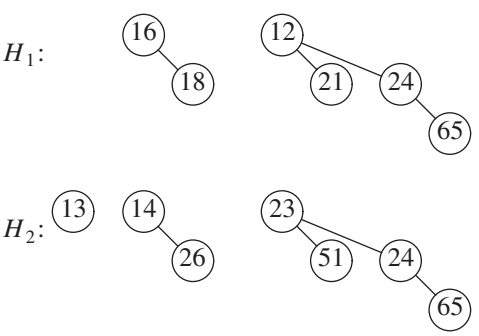

Figure 6.36 Two binomial queues H_1 and H_2

The merge is performed by essentially adding the two queues together. Let H_3 be the new binomial queue. Since H_1 has no binomial tree of height 0 and H_2 does, we can just use the binomial tree of height 0 in H_2 as part of H_3. Next, we add binomial trees of height 1. Since both H_1 and H_2 have binomial trees of height 1, we merge them by making the larger root a subtree of the smaller, creating a binomial tree of height 2, shown in Figure 6.37. Thus, H_3 will not have a binomial tree of height 1. There are now three binomial trees of height 2, namely, the original trees of H_1 and H_2 plus the tree formed by the previous step. We keep one binomial tree of height 2 in H_3 and merge the other two, creating a binomial tree of height 3. Since H_1 and H_2 have no trees of height 3, this tree becomes part of H_3 and we are finished. The resulting binomial queue is shown in Figure 6.38.

Since merging two binomial trees takes constant time with almost any reasonable implementation, and there are $O(\log N)$ binomial trees, the merge takes $O(\log N)$ time in the worst case. To make this operation efficient, we need to keep the trees in the binomial queue sorted by height, which is certainly a simple thing to do.

Insertion is just a special case of merging, since we merely create a one-node tree and perform a merge. The worst-case time of this operation is likewise $O(\log N)$. More precisely, if the priority queue into which the element is being inserted has the property that the smallest nonexistent binomial tree is B_i, the running time is proportional to $i + 1$. For example, H_3 (Fig. 6.38) is missing a binomial tree of height 1, so the insertion will terminate in two steps. Since each tree in a binomial queue is present with probability $\frac{1}{2}$, it follows that we expect an insertion to terminate in two steps, so the average time is constant. Furthermore, an analysis will show that performing N `inserts` on an initially empty binomial queue will take $O(N)$ worst-case time. Indeed, it is possible to do this operation using only $N - 1$ comparisons; we leave this as an exercise.

As an example, we show in Figures 6.39 through 6.45 the binomial queues that are formed by inserting 1 through 7 in order. Inserting 4 shows off a bad case. We merge 4

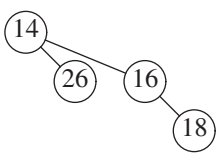

Figure 6.37 Merge of the two B_1 trees in H_1 and H_2

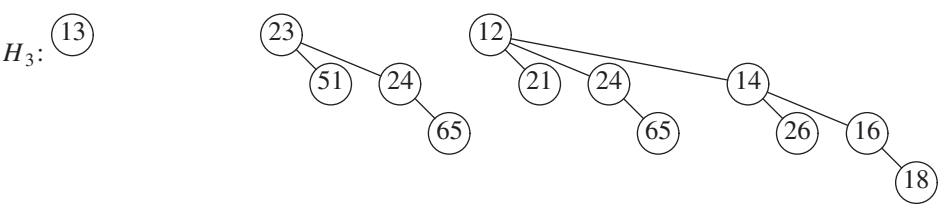

Figure 6.38 Binomial queue H_3: the result of merging H_1 and H_2

Figure 6.39 After 1 is inserted

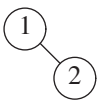

Figure 6.40 After 2 is inserted

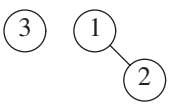

Figure 6.41 After 3 is inserted

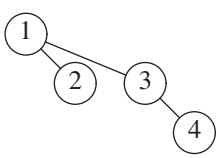

Figure 6.42 After 4 is inserted

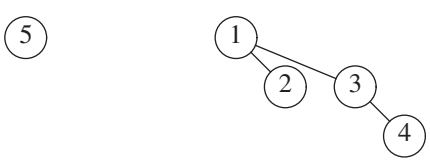

Figure 6.43 After 5 is inserted

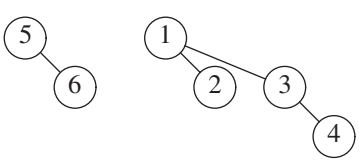

Figure 6.44 After 6 is inserted

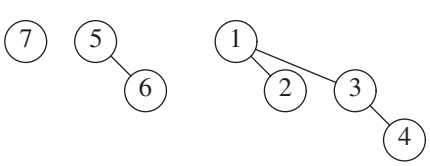

Figure 6.45 After 7 is inserted

with B_0, obtaining a new tree of height 1. We then merge this tree with B_1, obtaining a tree of height 2, which is the new priority queue. We count this as three steps (two tree merges plus the stopping case). The next insertion after 7 is inserted is another bad case and would require three tree merges.

A `deleteMin` can be performed by first finding the binomial tree with the smallest root. Let this tree be B_k, and let the original priority queue be H. We remove the binomial tree B_k from the forest of trees in H, forming the new binomial queue H'. We also remove the root of B_k, creating binomial trees $B_0, B_1, \ldots, B_{k-1}$, which collectively form priority queue H''. We finish the operation by merging H' and H''.

As an example, suppose we perform a `deleteMin` on H_3, which is shown again in Figure 6.46. The minimum root is 12, so we obtain the two priority queues H' and H'' in Figure 6.47 and Figure 6.48. The binomial queue that results from merging H' and H'' is the final answer and is shown in Figure 6.49.

For the analysis, note first that the `deleteMin` operation breaks the original binomial queue into two. It takes $O(\log N)$ time to find the tree containing the minimum element and to create the queues H' and H''. Merging these two queues takes $O(\log N)$ time, so the entire `deleteMin` operation takes $O(\log N)$ time.

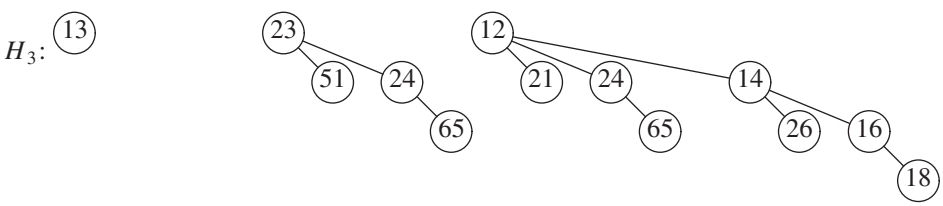

Figure 6.46 Binomial queue H_3

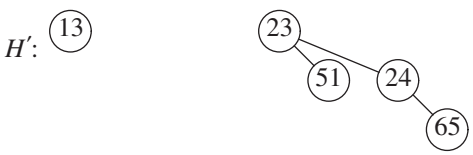

Figure 6.47 Binomial queue H', containing all the binomial trees in H_3 except B_3

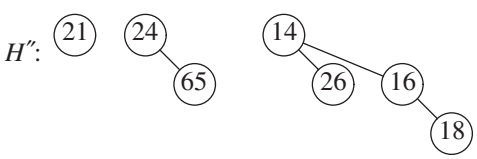

Figure 6.48 Binomial queue H'': B_3 with 12 removed

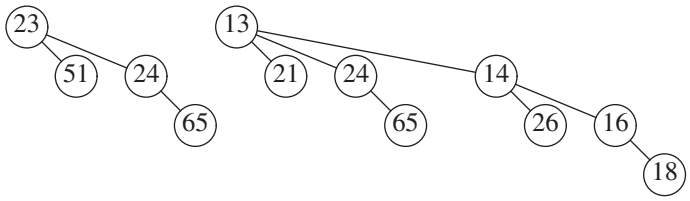

Figure 6.49 Result of applying `deleteMin` to H_3

6.8.3 Implementation of Binomial Queues

The `deleteMin` operation requires the ability to find all the subtrees of the root quickly, so the standard representation of general trees is required: The children of each node are kept in a linked list, and each node has a pointer to its first child (if any). This operation also requires that the children be ordered by the size of their subtrees. We also need to make sure that it is easy to merge two trees. When two trees are merged, one of the trees is added as a child to the other. Since this new tree will be the largest subtree, it makes sense to maintain the subtrees in decreasing sizes. Only then will we be able to merge two binomial trees, and thus two binomial queues, efficiently. The binomial queue will be an array of binomial trees.

To summarize, then, each node in a binomial tree will contain the data, first child, and right sibling. The children in a binomial tree are arranged in decreasing rank.

Figure 6.51 shows how the binomial queue in Figure 6.50 is represented. Figure 6.52 shows the type declarations for a node in the binomial tree and the binomial queue class interface.

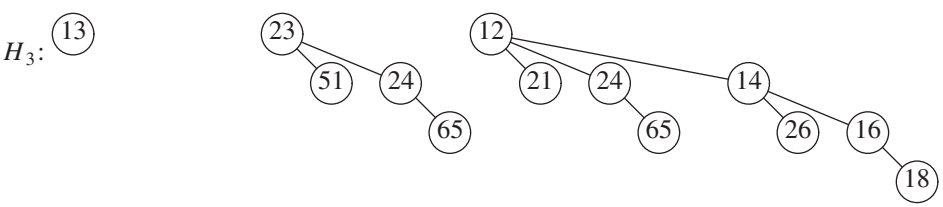

Figure 6.50 Binomial queue H_3 drawn as a forest

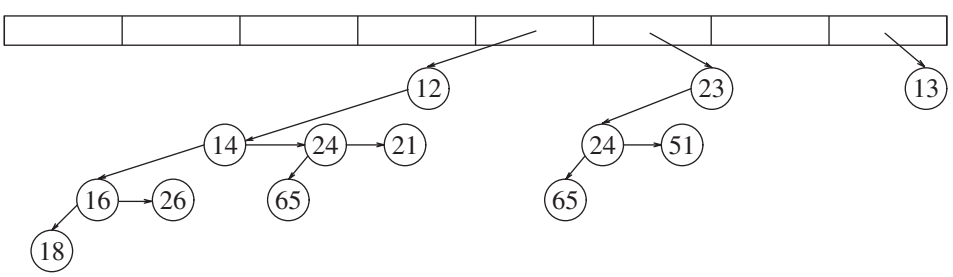

Figure 6.51 Representation of binomial queue H_3

```
template <typename Comparable>
class BinomialQueue
{
  public:
    BinomialQueue( );
    BinomialQueue( const Comparable & item );
    BinomialQueue( const BinomialQueue & rhs );
    BinomialQueue( BinomialQueue && rhs );

    ~BinomialQueue( );

    BinomialQueue & operator=( const BinomialQueue & rhs );
    BinomialQueue & operator=( BinomialQueue && rhs );

    bool isEmpty( ) const;
    const Comparable & findMin( ) const;

    void insert( const Comparable & x );
    void insert( Comparable && x );
    void deleteMin( );
    void deleteMin( Comparable & minItem );

    void makeEmpty( );
    void merge( BinomialQueue & rhs );

  private:
    struct BinomialNode
    {
        Comparable    element;
        BinomialNode *leftChild;
        BinomialNode *nextSibling;

        BinomialNode( const Comparable & e, BinomialNode *lt, BinomialNode *rt )
          : element{ e }, leftChild{ lt }, nextSibling{ rt } { }

        BinomialNode( Comparable && e, BinomialNode *lt, BinomialNode *rt )
          : element{ std::move( e ) }, leftChild{ lt }, nextSibling{ rt } { }
    };

    const static int DEFAULT_TREES = 1;

    vector<BinomialNode *> theTrees;  // An array of tree roots
    int currentSize;                  // Number of items in the priority queue

    int findMinIndex( ) const;
    int capacity( ) const;
    BinomialNode * combineTrees( BinomialNode *t1, BinomialNode *t2 );
    void makeEmpty( BinomialNode * & t );
    BinomialNode * clone( BinomialNode * t ) const;
};
```

Figure 6.52 Binomial queue class interface and node definition

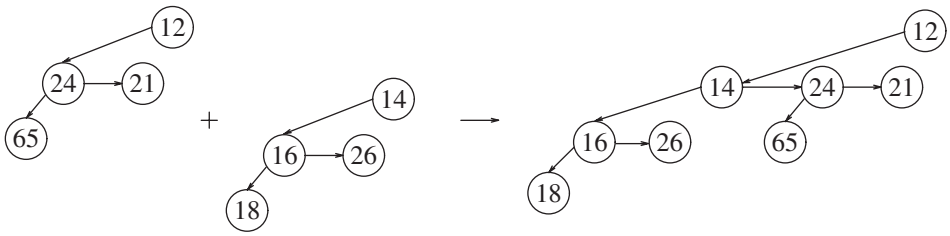

Figure 6.53 Merging two binomial trees

In order to merge two binomial queues, we need a routine to merge two binomial trees of the same size. Figure 6.53 shows how the links change when two binomial trees are merged. The code to do this is simple and is shown in Figure 6.54.

We provide a simple implementation of the `merge` routine. H_1 is represented by the current object and H_2 is represented by `rhs`. The routine combines H_1 and H_2, placing the result in H_1 and making H_2 empty. At any point we are dealing with trees of rank i. `t1` and `t2` are the trees in H_1 and H_2, respectively, and `carry` is the tree carried from a previous step (it might be `nullptr`). Depending on each of the eight possible cases, the tree that results for rank i and the `carry` tree of rank $i + 1$ is formed. This process proceeds from rank 0 to the last rank in the resulting binomial queue. The code is shown in Figure 6.55. Improvements to the code are suggested in Exercise 6.35.

The `deleteMin` routine for binomial queues is given in Figure 6.56 (on pages 280–281).

We can extend binomial queues to support some of the nonstandard operations that binary heaps allow, such as `decreaseKey` and `remove`, when the position of the affected element is known. A `decreaseKey` is a `percolateUp`, which can be performed in $O(\log N)$ time if we add a data member to each node that stores a parent link. An arbitrary `remove` can be performed by a combination of `decreaseKey` and `deleteMin` in $O(\log N)$ time.

```
1       /**
2        * Return the result of merging equal-sized t1 and t2.
3        */
4       BinomialNode * combineTrees( BinomialNode *t1, BinomialNode *t2 )
5       {
6           if( t2->element < t1->element )
7               return combineTrees( t2, t1 );
8           t2->nextSibling = t1->leftChild;
9           t1->leftChild = t2;
10          return t1;
11      }
```

Figure 6.54 Routine to merge two equal-sized binomial trees

```
/**
 * Merge rhs into the priority queue.
 * rhs becomes empty. rhs must be different from this.
 * Exercise 6.35 needed to make this operation more efficient.
 */
void merge( BinomialQueue & rhs )
{
    if( this == &rhs )    // Avoid aliasing problems
        return;

    currentSize += rhs.currentSize;

    if( currentSize > capacity( ) )
    {
        int oldNumTrees = theTrees.size( );
        int newNumTrees = max( theTrees.size( ), rhs.theTrees.size( ) ) + 1;
        theTrees.resize( newNumTrees );
        for( int i = oldNumTrees; i < newNumTrees; ++i )
            theTrees[ i ] = nullptr;
    }

    BinomialNode *carry = nullptr;
    for( int i = 0, j = 1; j <= currentSize; ++i, j *= 2 )
    {
        BinomialNode *t1 = theTrees[ i ];
        BinomialNode *t2 = i < rhs.theTrees.size( ) ? rhs.theTrees[ i ]
                                                    : nullptr;
        int whichCase = t1 == nullptr ? 0 : 1;
        whichCase += t2 == nullptr ? 0 : 2;
        whichCase += carry == nullptr ? 0 : 4;

        switch( whichCase )
        {
          case 0: /* No trees */
          case 1: /* Only this */
            break;
          case 2: /* Only rhs */
            theTrees[ i ] = t2;
            rhs.theTrees[ i ] = nullptr;
            break;
          case 4: /* Only carry */
            theTrees[ i ] = carry;
            carry = nullptr;
            break;
```

Figure 6.55 Routine to merge two priority queues

```
45                case 3: /* this and rhs */
46                  carry = combineTrees( t1, t2 );
47                  theTrees[ i ] = rhs.theTrees[ i ] = nullptr;
48                  break;
49                case 5: /* this and carry */
50                  carry = combineTrees( t1, carry );
51                  theTrees[ i ] = nullptr;
52                  break;
53                case 6: /* rhs and carry */
54                  carry = combineTrees( t2, carry );
55                  rhs.theTrees[ i ] = nullptr;
56                  break;
57                case 7: /* All three */
58                  theTrees[ i ] = carry;
59                  carry = combineTrees( t1, t2 );
60                  rhs.theTrees[ i ] = nullptr;
61                  break;
62              }
63          }
64
65          for( auto & root : rhs.theTrees )
66              root = nullptr;
67          rhs.currentSize = 0;
68      }
```

Figure 6.55 *(continued)*

```
1       /**
2        * Remove the minimum item and place it in minItem.
3        * Throws UnderflowException if empty.
4        */
5       void deleteMin( Comparable & minItem )
6       {
7           if( isEmpty( ) )
8               throw UnderflowException{ };
9
10          int minIndex = findMinIndex( );
11          minItem = theTrees[ minIndex ]->element;
12
```

Figure 6.56 `deleteMin` for binomial queues

```
13          BinomialNode *oldRoot = theTrees[ minIndex ];
14          BinomialNode *deletedTree = oldRoot->leftChild;
15          delete oldRoot;
16
17          // Construct H''
18          BinomialQueue deletedQueue;
19          deletedQueue.theTrees.resize( minIndex + 1 );
20          deletedQueue.currentSize = ( 1 << minIndex ) - 1;
21          for( int j = minIndex - 1; j >= 0; --j )
22          {
23              deletedQueue.theTrees[ j ] = deletedTree;
24              deletedTree = deletedTree->nextSibling;
25              deletedQueue.theTrees[ j ]->nextSibling = nullptr;
26          }
27
28          // Construct H'
29          theTrees[ minIndex ] = nullptr;
30          currentSize -= deletedQueue.currentSize + 1;
31
32          merge( deletedQueue );
33      }
34
35      /**
36       * Find index of tree containing the smallest item in the priority queue.
37       * The priority queue must not be empty.
38       * Return the index of tree containing the smallest item.
39       */
40      int findMinIndex( ) const
41      {
42          int i;
43          int minIndex;
44
45          for( i = 0; theTrees[ i ] == nullptr; ++i )
46              ;
47
48          for( minIndex = i; i < theTrees.size( ); ++i )
49              if( theTrees[ i ] != nullptr &&
50                  theTrees[ i ]->element < theTrees[ minIndex ]->element )
51                  minIndex = i;
52
53          return minIndex;
54      }
```

Figure 6.56 *(continued)*

```
#include <iostream>
#include <vector>
#include <queue>
#include <functional>
#include <string>
using namespace std;

// Empty the priority queue and print its contents.
template <typename PriorityQueue>
void dumpContents( const string & msg, PriorityQueue & pq )
{
    cout << msg << ":" << endl;
    while( !pq.empty( ) )
    {
        cout << pq.top( ) << endl;
        pq.pop( );
    }
}

// Do some inserts and removes (done in dumpContents).
int main( )
{
    priority_queue<int>                          maxPQ;
    priority_queue<int,vector<int>,greater<int>>  minPQ;

    minPQ.push( 4 ); minPQ.push( 3 ); minPQ.push( 5 );
    maxPQ.push( 4 ); maxPQ.push( 3 ); maxPQ.push( 5 );

    dumpContents( "minPQ", minPQ );     // 3 4 5
    dumpContents( "maxPQ", maxPQ );     // 5 4 3

    return 0;
}
```

Figure 6.57 Routine that demonstrates the STL `priority_queue`; the comment shows the expected order of output

6.9 Priority Queues in the Standard Library

The binary heap is implemented in the STL by the class template named `priority_queue` found in the standard header file `queue`. The STL implements a max-heap rather than a min-heap so the largest rather than smallest item is the one that is accessed. The key member functions are: