CHAPTER 6

Priority Queues (Heaps)

Although jobs sent to a printer are generally placed on a queue, this might not always be the best thing to do. For instance, one job might be particularly important, so it might be desirable to allow that job to be run as soon as the printer is available. Conversely, if, when the printer becomes available, there are several 1-page jobs and one 100-page job, it might be reasonable to make the long job go last, even if it is not the last job submitted. (Unfortunately, most systems do not do this, which can be particularly annoying at times.)

Similarly, in a multiuser environment, the operating system scheduler must decide which of several processes to run. Generally, a process is allowed to run only for a fixed period of time. One algorithm uses a queue. Jobs are initially placed at the end of the queue. The scheduler will repeatedly take the first job on the queue, run it until either it finishes or its time limit is up, and place it at the end of the queue if it does not finish. This strategy is generally not appropriate, because very short jobs will seem to take a long time because of the wait involved to run. Generally, it is important that short jobs finish as fast as possible, so these jobs should have precedence over jobs that have already been running. Furthermore, some jobs that are not short are still very important and should also have precedence.

This particular application seems to require a special kind of queue, known as a **priority queue.** In this chapter, we will discuss . . .

- Efficient implementation of the priority queue ADT.
- Uses of priority queues.
- Advanced implementations of priority queues.

The data structures we will see are among the most elegant in computer science.

6.1 Model

A priority queue is a data structure that allows at least the following two operations: `insert`, which does the obvious thing; and `deleteMin`, which finds, returns, and removes the minimum element in the priority queue.[1] The `insert` operation is the equivalent of `enqueue`, and `deleteMin` is the priority queue equivalent of the queue's `dequeue` operation.

[1] The C++ code provides two versions of `deleteMin`. One removes the minimum; the other removes the minimum and stores the removed value in an object passed by reference.

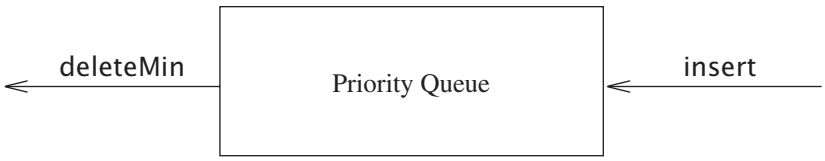

Figure 6.1 Basic model of a priority queue

As with most data structures, it is sometimes possible to add other operations, but these are extensions and not part of the basic model depicted in Figure 6.1.

Priority queues have many applications besides operating systems. In Chapter 7, we will see how priority queues are used for external sorting. Priority queues are also important in the implementation of **greedy algorithms**, which operate by repeatedly finding a minimum; we will see specific examples in Chapters 9 and 10. In this chapter, we will see a use of priority queues in discrete event simulation.

6.2 Simple Implementations

There are several obvious ways to implement a priority queue. We could use a simple linked list, performing insertions at the front in $O(1)$ and traversing the list, which requires $O(N)$ time, to delete the minimum. Alternatively, we could insist that the list be kept always sorted; this makes insertions expensive ($O(N)$) and `deleteMin`s cheap ($O(1)$). The former is probably the better idea of the two, based on the fact that there are never more `deleteMin`s than insertions.

Another way of implementing priority queues would be to use a binary search tree. This gives an $O(\log N)$ average running time for both operations. This is true in spite of the fact that although the insertions are random, the deletions are not. Recall that the only element we ever delete is the minimum. Repeatedly removing a node that is in the left subtree would seem to hurt the balance of the tree by making the right subtree heavy. However, the right subtree is random. In the worst case, where the `deleteMin`s have depleted the left subtree, the right subtree would have at most twice as many elements as it should. This adds only a small constant to its expected depth. Notice that the bound can be made into a worst-case bound by using a balanced tree; this protects one against bad insertion sequences.

Using a search tree could be overkill because it supports a host of operations that are not required. The basic data structure we will use will not require links and will support both operations in $O(\log N)$ worst-case time. Insertion will actually take constant time on average, and our implementation will allow building a priority queue of N items in linear time, if no deletions intervene. We will then discuss how to implement priority queues to support efficient merging. This additional operation seems to complicate matters a bit and apparently requires the use of a linked structure.

6.3 Binary Heap

The implementation we will use is known as a **binary heap**. Its use is so common for priority queue implementations that, in the context of priority queues, when the word *heap* is used without a qualifier, it is generally assumed to be referring to this implementation of the data structure. In this section, we will refer to binary heaps merely as *heaps*. Like binary search trees, heaps have two properties, namely, a structure property and a heap-order property. As with AVL trees, an operation on a heap can destroy one of the properties, so a heap operation must not terminate until all heap properties are in order. This turns out to be simple to do.

6.3.1 Structure Property

A heap is a binary tree that is completely filled, with the possible exception of the bottom level, which is filled from left to right. Such a tree is known as a **complete binary tree**. Figure 6.2 shows an example.

It is easy to show that a complete binary tree of height h has between 2^h and $2^{h+1} - 1$ nodes. This implies that the height of a complete binary tree is $\lfloor \log N \rfloor$, which is clearly $O(\log N)$.

An important observation is that because a complete binary tree is so regular, it can be represented in an array and no links are necessary. The array in Figure 6.3 corresponds to the heap in Figure 6.2.

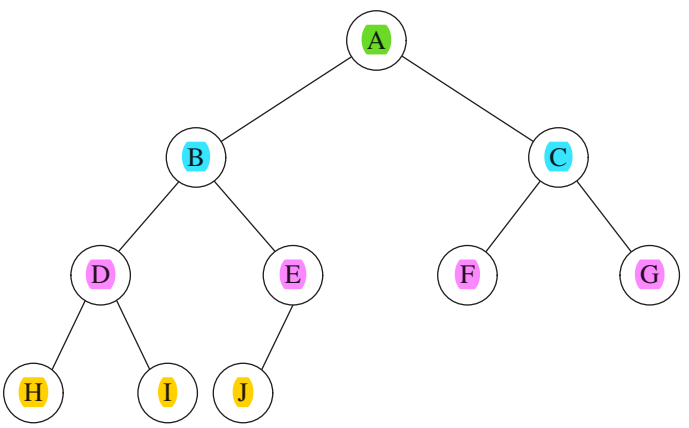

Figure 6.2 A complete binary tree

	A	B	C	D	E	F	G	H	I	J			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Figure 6.3 Array implementation of complete binary tree

```
1  template <typename Comparable>
2  class BinaryHeap
3  {
4    public:
5      explicit BinaryHeap( int capacity = 100 );
6      explicit BinaryHeap( const vector<Comparable> & items );
7
8      bool isEmpty( ) const;
9      const Comparable & findMin( ) const;
10
11     void insert( const Comparable & x );
12     void insert( Comparable && x );
13     void deleteMin( );
14     void deleteMin( Comparable & minItem );
15     void makeEmpty( );
16
17   private:
18     int                currentSize;  // Number of elements in heap
19     vector<Comparable> array;        // The heap array
20
21     void buildHeap( );
22     void percolateDown( int hole );
23 };
```

Figure 6.4 Class interface for priority queue

For any element in array position i, the left child is in position $2i$, the right child is in the cell after the left child ($2i + 1$), and the parent is in position $\lfloor i/2 \rfloor$. Thus, not only are links not required, but the operations required to traverse the tree are extremely simple and likely to be very fast on most computers. The only problem with this implementation is that an estimate of the maximum heap size is required in advance, but typically this is not a problem (and we can resize if needed). In Figure 6.3 the limit on the heap size is 13 elements. The array has a position 0; more on this later.

A heap data structure will, then, consist of an array (of `Comparable` objects) and an integer representing the current heap size. Figure 6.4 shows a priority queue interface.

Throughout this chapter, we shall draw the heaps as trees, with the implication that an actual implementation will use simple arrays.

6.3.2 Heap-Order Property

The property that allows operations to be performed quickly is the **heap-order property**. Since we want to be able to find the minimum quickly, it makes sense that the smallest element should be at the root. If we consider that any subtree should also be a heap, then any node should be smaller than all of its descendants.

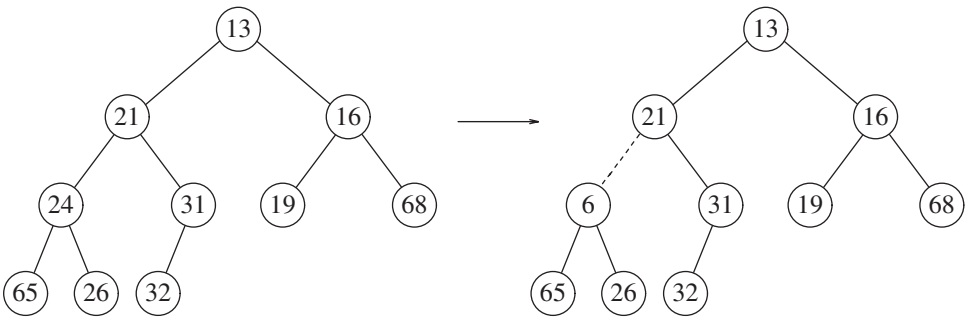

Figure 6.5 Two complete trees (only the left tree is a heap)

Applying this logic, we arrive at the heap-order property. In a heap, for every node X, the key in the parent of X is smaller than (or equal to) the key in X, with the exception of the root (which has no parent).[2] In Figure 6.5 the tree on the left is a heap, but the tree on the right is not (the dashed line shows the violation of heap order).

By the heap-order property, the minimum element can always be found at the root. Thus, we get the extra operation, `findMin`, in constant time.

6.3.3 Basic Heap Operations

It is easy (both conceptually and practically) to perform the two required operations. All the work involves ensuring that the heap-order property is maintained.

`insert`

To insert an element X into the heap, we create a hole in the next available location, since otherwise, the tree will not be complete. If X can be placed in the hole without violating heap order, then we do so and are done. Otherwise, we slide the element that is in the hole's parent node into the hole, thus bubbling the hole up toward the root. We continue this process until X can be placed in the hole. Figure 6.6 shows that to insert 14, we create a hole in the next available heap location. Inserting 14 in the hole would violate the heap-order property, so 31 is slid down into the hole. This strategy is continued in Figure 6.7 until the correct location for 14 is found.

This general strategy is known as a **percolate up;** the new element is percolated up the heap until the correct location is found. Insertion is easily implemented with the code shown in Figure 6.8.

[2] Analogously we can declare a (*max*) heap, which enables us to efficiently find and remove the maximum element by changing the heap-order property. Thus, a priority queue can be used to find *either* a minimum or a maximum, but this needs to be decided ahead of time.

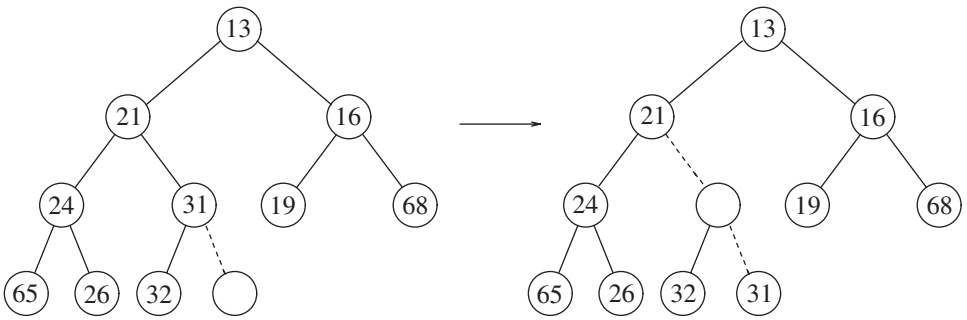

Figure 6.6 Attempt to insert 14: creating the hole, and bubbling the hole up

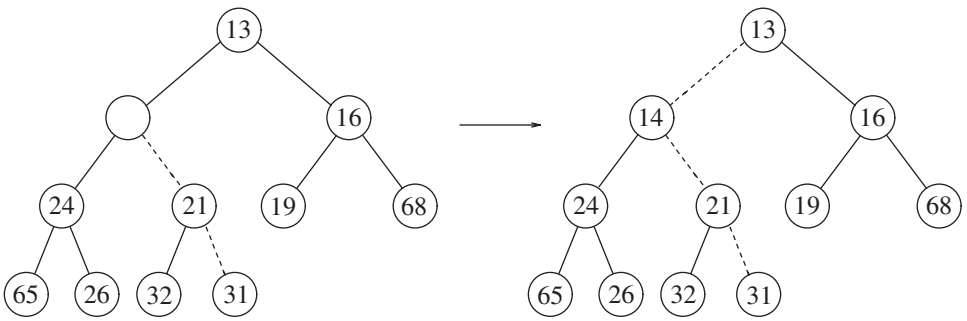

Figure 6.7 The remaining two steps to insert 14 in previous heap

```
1       /**
2        * Insert item x, allowing duplicates.
3        */
4       void insert( const Comparable & x )
5       {
6           if( currentSize == array.size( ) - 1 )
7               array.resize( array.size( ) * 2 );
8
9               // Percolate up
10          int hole = ++currentSize;
11          Comparable copy = x;
12
13          array[ 0 ] = std::move( copy );
14          for( ; x < array[ hole / 2 ]; hole /= 2 )
15              array[ hole ] = std::move( array[ hole / 2 ] );
16          array[ hole ] = std::move( array[ 0 ] );
17      }
```

Figure 6.8 Procedure to insert into a binary heap

We could have implemented the percolation in the `insert` routine by performing repeated swaps until the correct order was established, but a swap requires three assignment statements. If an element is percolated up d levels, the number of assignments performed by the swaps would be $3d$. Our method uses $d + 1$ assignments.

If the element to be inserted is the new minimum, it will be pushed all the way to the top. At some point, `hole` will be 1 and we will want to break out of the loop. We could do this with an explicit test, or we can put a copy of the inserted item in position 0 in order to make the loop terminate. We elect to place X into position 0.

The time to do the insertion could be as much as $O(\log N)$, if the element to be inserted is the new minimum and is percolated all the way to the root. On average, the percolation terminates early; it has been shown that 2.607 comparisons are required on average to perform an insert, so the average `insert` moves an element up 1.607 levels.

`deleteMin`

`deleteMin`s are handled in a similar manner as insertions. Finding the minimum is easy; the hard part is removing it. When the minimum is removed, a hole is created at the root. Since the heap now becomes one smaller, it follows that the last element X in the heap must move somewhere in the heap. If X can be placed in the hole, then we are done. This is unlikely, so we slide the smaller of the hole's children into the hole, thus pushing the hole down one level. We repeat this step until X can be placed in the hole. Thus, our action is to place X in its correct spot along a path from the root containing *minimum* children.

In Figure 6.9 the left figure shows a heap prior to the `deleteMin`. After 13 is removed, we must now try to place 31 in the heap. The value 31 cannot be placed in the hole, because this would violate heap order. Thus, we place the smaller child (14) in the hole, sliding the hole down one level (see Fig. 6.10). We repeat this again, and since 31 is larger than 19, we place 19 into the hole and create a new hole one level deeper. We then place 26 in the hole and create a new hole on the bottom level since, once again, 31 is too large. Finally, we are able to place 31 in the hole (Fig. 6.11). This general strategy is known as a **percolate down.** We use the same technique as in the `insert` routine to avoid the use of swaps in this routine.

A frequent implementation error in heaps occurs when there are an even number of elements in the heap, and the one node that has only one child is encountered. You must

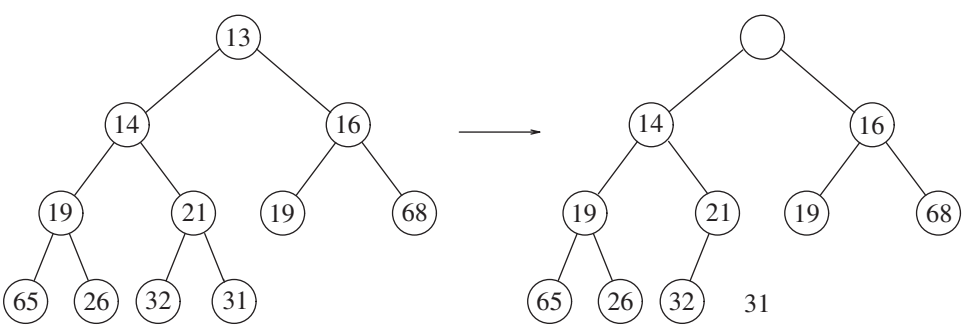

Figure 6.9 Creation of the hole at the root

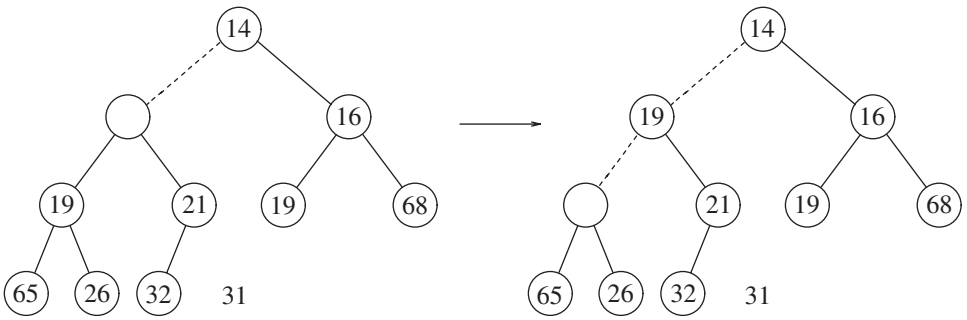

Figure 6.10 Next two steps in `deleteMin`

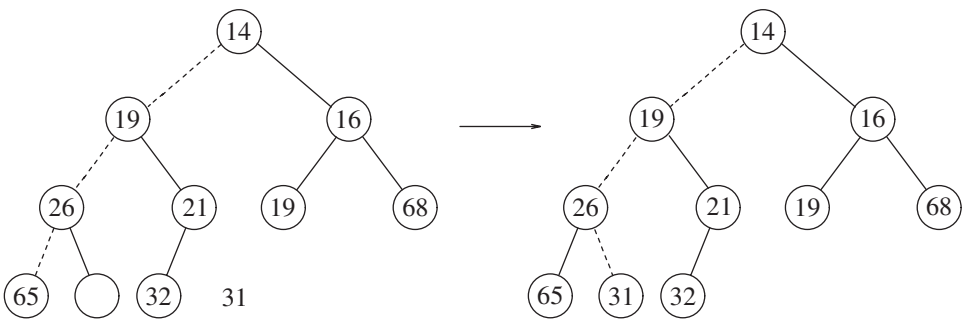

Figure 6.11 Last two steps in `deleteMin`

make sure not to assume that there are always two children, so this usually involves an extra test. In the code depicted in Figure 6.12, we've done this test at line 40. One extremely tricky solution is always to ensure that your algorithm *thinks* every node has two children. Do this by placing a sentinel, of value higher than any in the heap, at the spot after the heap ends, at the start of each *percolate down* when the heap size is even. You should think very carefully before attempting this, and you must put in a prominent comment if you do use this technique. Although this eliminates the need to test for the presence of a right child, you cannot eliminate the requirement that you test when you reach the bottom, because this would require a sentinel for every leaf.

The worst-case running time for this operation is $O(\log N)$. On average, the element that is placed at the root is percolated almost to the bottom of the heap (which is the level it came from), so the average running time is $O(\log N)$.

6.3.4 Other Heap Operations

Notice that although finding the minimum can be performed in constant time, a heap designed to find the minimum element (also known as a (*min*)heap) is of no help whatsoever in finding the maximum element. In fact, a heap has very little ordering information,

```
1       /**
2        * Remove the minimum item.
3        * Throws UnderflowException if empty.
4        */
5       void deleteMin( )
6       {
7           if( isEmpty( ) )
8               throw UnderflowException{ };
9
10          array[ 1 ] = std::move( array[ currentSize-- ] );
11          percolateDown( 1 );
12      }
13
14      /**
15       * Remove the minimum item and place it in minItem.
16       * Throws UnderflowException if empty.
17       */
18      void deleteMin( Comparable & minItem )
19      {
20          if( isEmpty( ) )
21              throw UnderflowException{ };
22
23          minItem = std::move( array[ 1 ] );
24          array[ 1 ] = std::move( array[ currentSize-- ] );
25          percolateDown( 1 );
26      }
27
28      /**
29       * Internal method to percolate down in the heap.
30       * hole is the index at which the percolate begins.
31       */
32      void percolateDown( int hole )
33      {
34          int child;
35          Comparable tmp = std::move( array[ hole ] );
36
37          for( ; hole * 2 <= currentSize; hole = child )
38          {
39              child = hole * 2;
40              if( child != currentSize && array[ child + 1 ] < array[ child ] )
41                  ++child;
42              if( array[ child ] < tmp )
43                  array[ hole ] = std::move( array[ child ] );
44              else
45                  break;
46          }
47          array[ hole ] = std::move( tmp );
48      }
```

Figure 6.12 Method to perform `deleteMin` in a binary heap

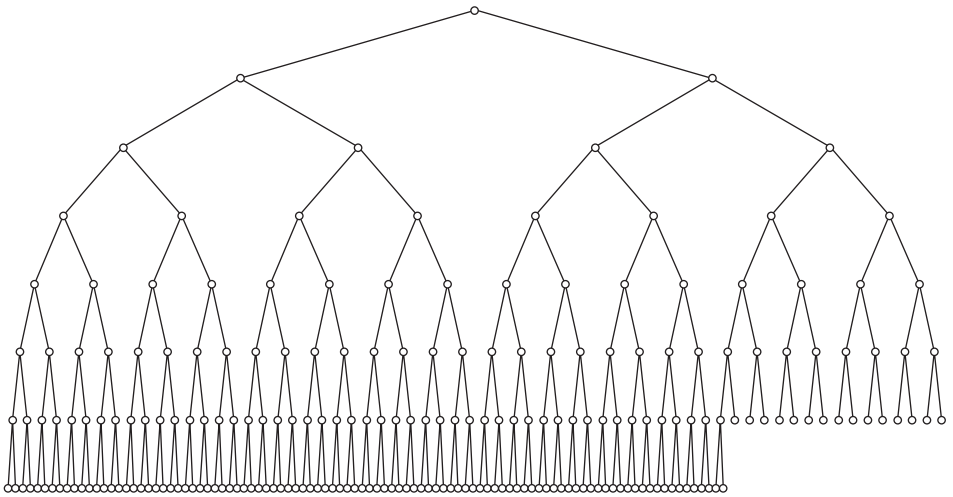

Figure 6.13 A very large complete binary tree

so there is no way to find any particular element without a linear scan through the entire heap. To see this, consider the large heap structure (the elements are not shown) in Figure 6.13, where we see that the only information known about the maximum element is that it is at one of the leaves. Half the elements, though, are contained in leaves, so this is practically useless information. For this reason, if it is important to know where elements are, some other data structure, such as a hash table, must be used in addition to the heap. (Recall that the model does not allow looking inside the heap.)

If we assume that the position of every element is known by some other method, then several other operations become cheap. The first three operations below all run in logarithmic worst-case time.

decreaseKey

The `decreaseKey(p,`Δ`)` operation lowers the value of the item at position `p` by a positive amount Δ. Since this might violate the heap order, it must be fixed by a *percolate up*. This operation could be useful to system administrators: They can make their programs run with highest priority.

increaseKey

The `increaseKey(p,`Δ`)` operation increases the value of the item at position `p` by a positive amount Δ. This is done with a *percolate down*. Many schedulers automatically drop the priority of a process that is consuming excessive CPU time.

remove

The `remove(p)` operation removes the node at position `p` from the heap. This is done by first performing `decreaseKey(p,`∞`)` and then performing `deleteMin()`. When a process

is terminated by a user (instead of finishing normally), it must be removed from the priority queue.

buildHeap

The binary heap is sometimes constructed from an initial collection of items. This constructor takes as input N items and places them into a heap. Obviously, this can be done with N successive `insert`s. Since each `insert` will take $O(1)$ average and $O(\log N)$ worst-case time, the total running time of this algorithm would be $O(N)$ average but $O(N \log N)$ worst-case. Since this is a special instruction and there are no other operations intervening, and we already know that the instruction can be performed in linear average time, it is reasonable to expect that with reasonable care a linear time bound can be guaranteed.

The general algorithm is to place the N items into the tree in any order, maintaining the structure property. Then, if `percolateDown(i)` percolates down from node `i`, the `buildHeap` routine in Figure 6.14 can be used by the constructor to create a heap-ordered tree.

The first tree in Figure 6.15 is the unordered tree. The seven remaining trees in Figures 6.15 through 6.18 show the result of each of the seven `percolateDown`s. Each dashed line corresponds to two comparisons: one to find the smaller child and one to compare the smaller child with the node. Notice that there are only 10 dashed lines in the entire algorithm (there could have been an 11th—where?) corresponding to 20 comparisons.

To bound the running time of `buildHeap`, we must bound the number of dashed lines. This can be done by computing the sum of the heights of all the nodes in the heap, which is the maximum number of dashed lines. What we would like to show is that this sum is $O(N)$.

```
1       explicit BinaryHeap( const vector<Comparable> & items )
2         : array( items.size( ) + 10 ), currentSize{ items.size( ) }
3       {
4           for( int i = 0; i < items.size( ); ++i )
5               array[ i + 1 ] = items[ i ];
6           buildHeap( );
7       }
8
9       /**
10       * Establish heap order property from an arbitrary
11       * arrangement of items. Runs in linear time.
12       */
13      void buildHeap( )
14      {
15          for( int i = currentSize / 2; i > 0; --i )
16              percolateDown( i );
17      }
```

Figure 6.14 `buildHeap` and constructor

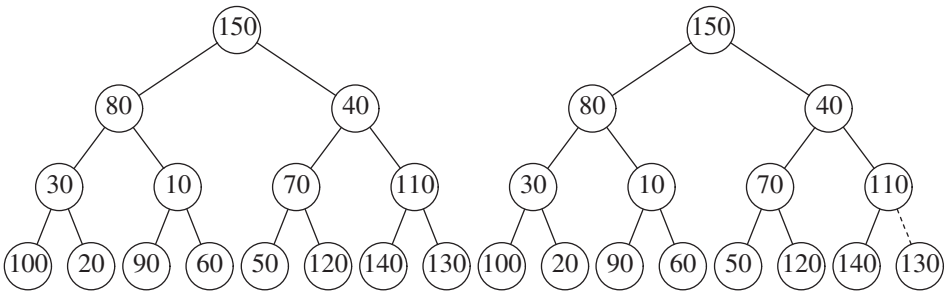

Figure 6.15 Left: initial heap; right: after `percolateDown(7)`

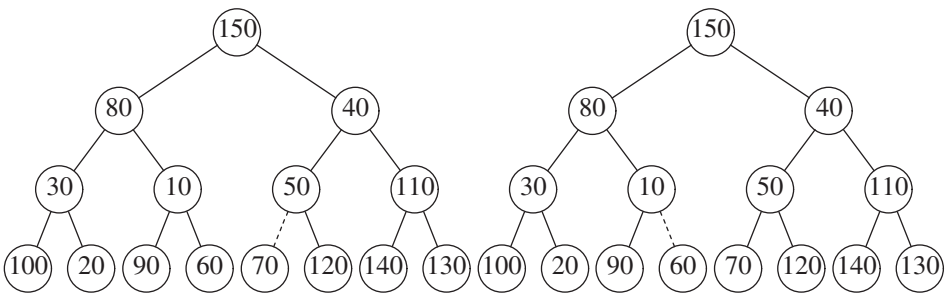

Figure 6.16 Left: after `percolateDown(6)`; right: after `percolateDown(5)`

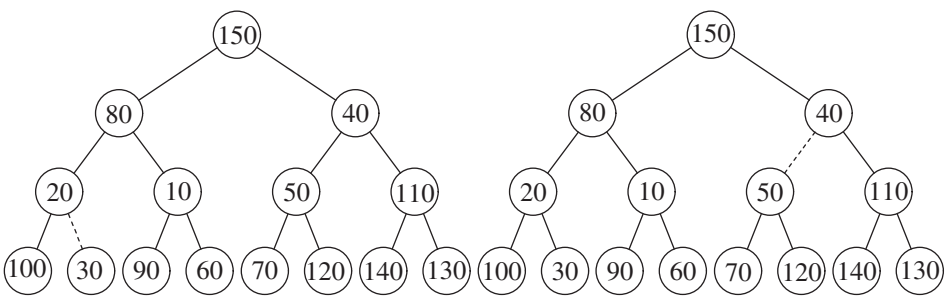

Figure 6.17 Left: after `percolateDown(4)`; right: after `percolateDown(3)`

Theorem 6.1

For the perfect binary tree of height h containing $2^{h+1}-1$ nodes, the sum of the heights of the nodes is $2^{h+1} - 1 - (h + 1)$.

Proof

It is easy to see that this tree consists of 1 node at height h, 2 nodes at height $h - 1$, 2^2 nodes at height $h - 2$, and in general 2^i nodes at height $h - i$. The sum of the heights of all the nodes is then

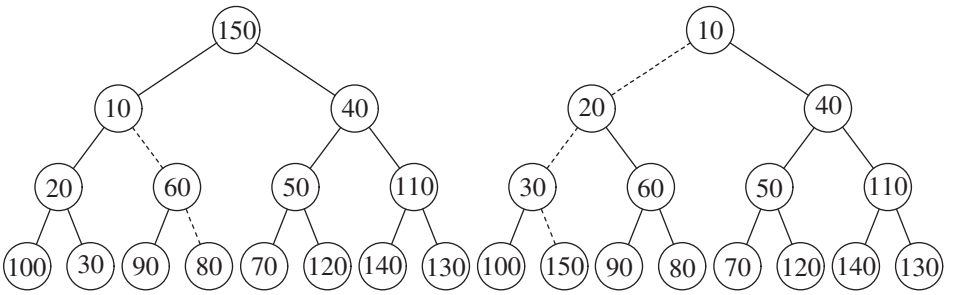

Figure 6.18 Left: after `percolateDown(2)`; right: after `percolateDown(1)`

$$S = \sum_{i=0}^{h} 2^i(h-i)$$
$$= h + 2(h-1) + 4(h-2) + 8(h-3) + 16(h-4) + \cdots + 2^{h-1}(1) \quad (6.1)$$

Multiplying by 2 gives the equation

$$2S = 2h + 4(h-1) + 8(h-2) + 16(h-3) + \cdots + 2^h(1) \quad (6.2)$$

We subtract these two equations and obtain Equation (6.3). We find that certain terms almost cancel. For instance, we have $2h - 2(h-1) = 2$, $4(h-1) - 4(h-2) = 4$, and so on. The last term in Equation (6.2), 2^h, does not appear in Equation (6.1); thus, it appears in Equation (6.3). The first term in Equation (6.1), h, does not appear in Equation (6.2); thus, $-h$ appears in Equation (6.3). We obtain

$$S = -h + 2 + 4 + 8 + \cdots + 2^{h-1} + 2^h = (2^{h+1} - 1) - (h+1) \quad (6.3)$$

which proves the theorem.

A complete tree is not a perfect binary tree, but the result we have obtained is an upper bound on the sum of the heights of the nodes in a complete tree. Since a complete tree has between 2^h and 2^{h+1} nodes, this theorem implies that this sum is $O(N)$, where N is the number of nodes.

Although the result we have obtained is sufficient to show that `buildHeap` is linear, the bound on the sum of the heights is not as strong as possible. For a complete tree with $N = 2^h$ nodes, the bound we have obtained is roughly $2N$. The sum of the heights can be shown by induction to be $N - b(N)$, where $b(N)$ is the number of 1s in the binary representation of N.

6.4 Applications of Priority Queues

We have already mentioned how priority queues are used in operating systems design. In Chapter 9, we will see how priority queues are used to implement several graph algorithms efficiently. Here we will show how to use priority queues to obtain solutions to two problems.

6.4.1 The Selection Problem

The first problem we will examine is the *selection problem* from Section 1.1. Recall that the input is a list of N elements, which can be totally ordered, and an integer k. The selection problem is to find the kth largest element.

Two algorithms were given in Chapter 1, but neither is very efficient. The first algorithm, which we shall call algorithm 1A, is to read the elements into an array and sort them, returning the appropriate element. Assuming a simple sorting algorithm, the running time is $O(N^2)$. The alternative algorithm, 1B, is to read k elements into an array and sort them. The smallest of these is in the kth position. We process the remaining elements one by one. As an element arrives, it is compared with the kth element in the array. If it is larger, then the kth element is removed, and the new element is placed in the correct place among the remaining $k-1$ elements. When the algorithm ends, the element in the kth position is the answer. The running time is $O(N \cdot k)$ (why?). If $k = \lceil N/2 \rceil$, then both algorithms are $O(N^2)$. Notice that for any k, we can solve the symmetric problem of finding the $(N-k+1)$th smallest element, so $k = \lceil N/2 \rceil$ is really the hardest case for these algorithms. This also happens to be the most interesting case, since this value of k is known as the *median*.

We give two algorithms here, both of which run in $O(N \log N)$ in the extreme case of $k = \lceil N/2 \rceil$, which is a distinct improvement.

Algorithm 6A

For simplicity, we assume that we are interested in finding the kth *smallest* element. The algorithm is simple. We read the N elements into an array. We then apply the `buildHeap` algorithm to this array. Finally, we perform k `deleteMin` operations. The last element extracted from the heap is our answer. It should be clear that by changing the heap-order property, we could solve the original problem of finding the kth *largest* element.

The correctness of the algorithm should be clear. The worst-case timing is $O(N)$ to construct the heap, if `buildHeap` is used, and $O(\log N)$ for each `deleteMin`. Since there are k `deleteMin`s, we obtain a total running time of $O(N + k \log N)$. If $k = O(N/\log N)$, then the running time is dominated by the `buildHeap` operation and is $O(N)$. For larger values of k, the running time is $O(k \log N)$. If $k = \lceil N/2 \rceil$, then the running time is $\Theta(N \log N)$.

Notice that if we run this program for $k = N$ and record the values as they leave the heap, we will have essentially sorted the input file in $O(N \log N)$ time. In Chapter 7, we will refine this idea to obtain a fast sorting algorithm known as *heapsort*.

Algorithm 6B

For the second algorithm, we return to the original problem and find the kth *largest* element. We use the idea from algorithm 1B. At any point in time we will maintain a set S of the k largest elements. After the first k elements are read, when a new element is read it is compared with the kth largest element, which we denote by S_k. Notice that S_k is the smallest element in S. If the new element is larger, then it replaces S_k in S. S will then have a new smallest element, which may or may not be the newly added element. At the end of the input, we find the smallest element in S and return it as the answer.

This is essentially the same algorithm described in Chapter 1. Here, however, we will use a heap to implement S. The first k elements are placed into the heap in total time $O(k)$ with a call to `buildHeap`. The time to process each of the remaining elements is $O(1)$, to test

if the element goes into S, plus $O(\log k)$, to delete S_k and insert the new element if this is necessary. Thus, the total time is $O(k + (N - k)\log k) = O(N \log k)$. This algorithm also gives a bound of $\Theta(N \log N)$ for finding the median.

In Chapter 7, we will see how to solve this problem in $O(N)$ average time. In Chapter 10, we will see an elegant, albeit impractical, algorithm to solve this problem in $O(N)$ worst-case time.

6.4.2 Event Simulation

In Section 3.7.3, we described an important queuing problem. Recall that we have a system, such as a bank, where customers arrive and wait in a line until one of k tellers is available. Customer arrival is governed by a probability distribution function, as is the service time (the amount of time to be served once a teller is available). We are interested in statistics such as how long on average a customer has to wait or how long the line might be.

With certain probability distributions and values of k, these answers can be computed exactly. However, as k gets larger, the analysis becomes considerably more difficult, so it is appealing to use a computer to simulate the operation of the bank. In this way, the bank officers can determine how many tellers are needed to ensure reasonably smooth service.

A simulation consists of processing events. The two events here are (a) a customer arriving and (b) a customer departing, thus freeing up a teller.

We can use the probability functions to generate an input stream consisting of ordered pairs of arrival time and service time for each customer, sorted by arrival time. We do not need to use the exact time of day. Rather, we can use a quantum unit, which we will refer to as a *tick*.

One way to do this simulation is to start a simulation clock at zero ticks. We then advance the clock one tick at a time, checking to see if there is an event. If there is, then we process the event(s) and compile statistics. When there are no customers left in the input stream and all the tellers are free, then the simulation is over.

The problem with this simulation strategy is that its running time does not depend on the number of customers or events (there are two events per customer), but instead depends on the number of ticks, which is not really part of the input. To see why this is important, suppose we changed the clock units to milliticks and multiplied all the times in the input by 1,000. The result would be that the simulation would take 1,000 times longer!

The key to avoiding this problem is to advance the clock to the next event time at each stage. This is conceptually easy to do. At any point, the next event that can occur is either (a) the next customer in the input file arrives or (b) one of the customers at a teller leaves. Since all the times when the events will happen are available, we just need to find the event that happens nearest in the future and process that event.

If the event is a departure, processing includes gathering statistics for the departing customer and checking the line (queue) to see whether there is another customer waiting. If so, we add that customer, process whatever statistics are required, compute the time when that customer will leave, and add that departure to the set of events waiting to happen.

If the event is an arrival, we check for an available teller. If there is none, we place the arrival on the line (queue); otherwise we give the customer a teller, compute the customer's departure time, and add the departure to the set of events waiting to happen.

The waiting line for customers can be implemented as a queue. Since we need to find the event *nearest* in the future, it is appropriate that the set of departures waiting to happen be organized in a priority queue. The next event is thus the next arrival or next departure (whichever is sooner); both are easily available.

It is then straightforward, although possibly time-consuming, to write the simulation routines. If there are C customers (and thus $2C$ events) and k tellers, then the running time *of the simulation* would be $O(C \log(k+1))$ because computing and processing each event takes $O(\log H)$, where $H = k + 1$ is the size of the heap.[3]

6.5 *d*-Heaps

Binary heaps are so simple that they are almost always used when priority queues are needed. A simple generalization is a ***d*-heap**, which is exactly like a binary heap except that all nodes have d children (thus, a binary heap is a 2-heap).

Figure 6.19 shows a 3-heap. Notice that a d-heap is much shallower than a binary heap, improving the running time of `inserts` to $O(\log_d N)$. However, for large d, the `deleteMin` operation is more expensive, because even though the tree is shallower, the minimum of d children must be found, which takes $d - 1$ comparisons using a standard algorithm. This raises the time for this operation to $O(d \log_d N)$. If d is a constant, both running times are, of course, $O(\log N)$. Although an array can still be used, the multiplications and divisions to find children and parents are now by d, which, unless d is a power of 2, seriously increases the running time, because we can no longer implement division by a bit shift. d-heaps are interesting in theory, because there are many algorithms where the number of insertions is much greater than the number of `deleteMin`s (and thus a theoretical speedup is possible). They are also of interest when the priority queue is too large to fit entirely in main memory.

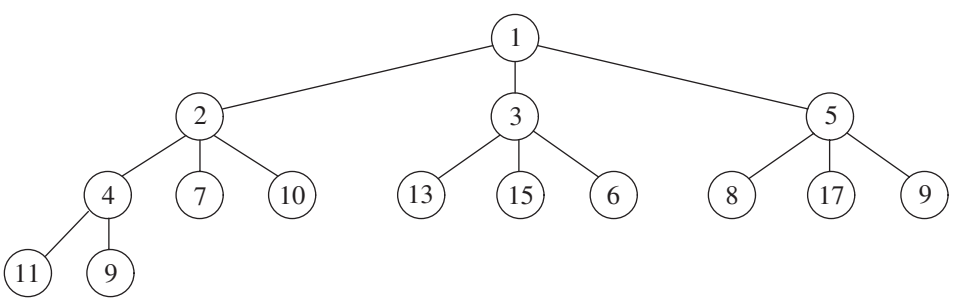

Figure 6.19 A d-heap ($d = 3$)

[3] We use $O(C \log(k+1))$ instead of $O(C \log k)$ to avoid confusion for the $k = 1$ case.

In this case, a *d*-heap can be advantageous in much the same way as B-trees. Finally, there is evidence suggesting that 4-heaps may outperform binary heaps in practice.

The most glaring weakness of the heap implementation, aside from the inability to perform `finds`, is that combining two heaps into one is a hard operation. This extra operation is known as a `merge`. There are quite a few ways to implement heaps so that the running time of a `merge` is $O(\log N)$. We will now discuss three data structures, of various complexity, that support the `merge` operation efficiently. We will defer any complicated analysis until Chapter 11.

6.6 Leftist Heaps

It seems difficult to design a data structure that efficiently supports merging (that is, processes a `merge` in $o(N)$ time) and uses only an array, as in a binary heap. The reason for this is that merging would seem to require copying one array into another, which would take $\Theta(N)$ time for equal-sized heaps. For this reason, all the advanced data structures that support efficient merging require the use of a linked data structure. In practice, we can expect that this will make all the other operations slower.

Like a binary heap, a **leftist heap** has both a structural property and an ordering property. Indeed, a leftist heap, like virtually all heaps used, has the same heap-order property we have already seen. Furthermore, a leftist heap is also a binary tree. The only difference between a leftist heap and a binary heap is that leftist heaps are not perfectly balanced, but actually attempt to be very unbalanced.

6.6.1 Leftist Heap Property

We define the **null path length**, $npl(X)$, of any node X to be the length of the shortest path from X to a node without two children. Thus, the npl of a node with zero or one child is 0, while $npl(\texttt{nullptr}) = -1$. In the tree in Figure 6.20, the null path lengths are indicated inside the tree nodes.

Notice that the null path length of any node is 1 more than the minimum of the null path lengths of its children. This applies to nodes with less than two children because the null path length of `nullptr` is -1.

The leftist heap property is that for every node X in the heap, the null path length of the left child is at least as large as that of the right child. This property is satisfied by only one of the trees in Figure 6.20, namely, the tree on the left. This property actually goes out of its way to ensure that the tree is unbalanced, because it clearly biases the tree to get deep toward the left. Indeed, a tree consisting of a long path of left nodes is possible (and actually preferable to facilitate merging)—hence the name *leftist heap*.

Because leftist heaps tend to have deep left paths, it follows that the right path ought to be short. Indeed, the right path down a leftist heap is as short as any in the heap. Otherwise, there would be a path that goes through some node X and takes the left child. Then X would violate the leftist property.

Theorem 6.2

A leftist tree with r nodes on the right path must have at least $2^r - 1$ nodes.

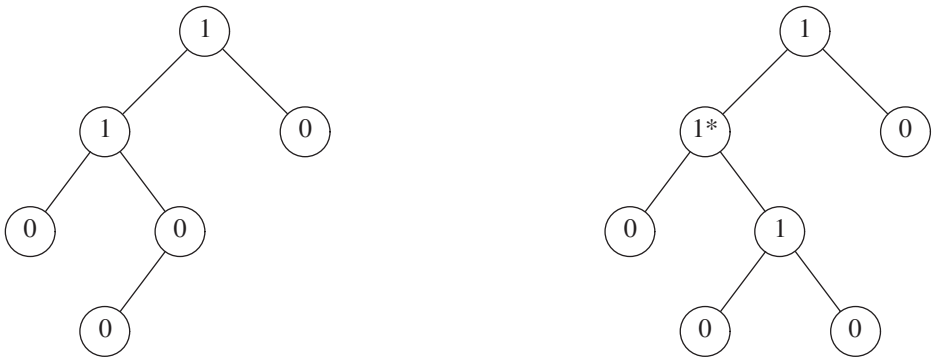

Figure 6.20 Null path lengths for two trees; only the left tree is leftist

Proof

The proof is by induction. If $r = 1$, there must be at least one tree node. Otherwise, suppose that the theorem is true for $1, 2, \ldots, r$. Consider a leftist tree with $r + 1$ nodes on the right path. Then the root has a right subtree with r nodes on the right path, and a left subtree with at least r nodes on the right path (otherwise it would not be leftist). Applying the inductive hypothesis to these subtrees yields a minimum of $2^r - 1$ nodes in each subtree. This plus the root gives at least $2^{r+1} - 1$ nodes in the tree, proving the theorem.

From this theorem, it follows immediately that a leftist tree of N nodes has a right path containing at most $\lfloor \log(N + 1) \rfloor$ nodes. The general idea for the leftist heap operations is to perform all the work on the right path, which is guaranteed to be short. The only tricky part is that performing `insert`s and `merge`s on the right path could destroy the leftist heap property. It turns out to be extremely easy to restore the property.

6.6.2 Leftist Heap Operations

The fundamental operation on leftist heaps is merging. Notice that insertion is merely a special case of merging, since we may view an insertion as a `merge` of a one-node heap with a larger heap. We will first give a simple recursive solution and then show how this might be done nonrecursively. Our input is the two leftist heaps, H_1 and H_2, in Figure 6.21. You should check that these heaps really are leftist. Notice that the smallest elements are at the roots. In addition to space for the data and left and right pointers, each node will have an entry that indicates the null path length.

If either of the two heaps is empty, then we can return the other heap. Otherwise, to merge the two heaps, we compare their roots. First, we recursively merge the heap with the larger root with the right subheap of the heap with the smaller root. In our example, this means we recursively merge H_2 with the subheap of H_1 rooted at 8, obtaining the heap in Figure 6.22.

Since this tree is formed recursively, and we have not yet finished the description of the algorithm, we cannot at this point show how this heap was obtained. However, it is

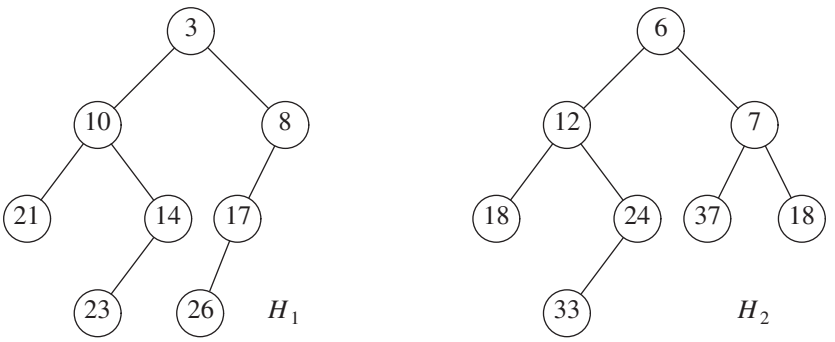

Figure 6.21 Two leftist heaps H_1 and H_2

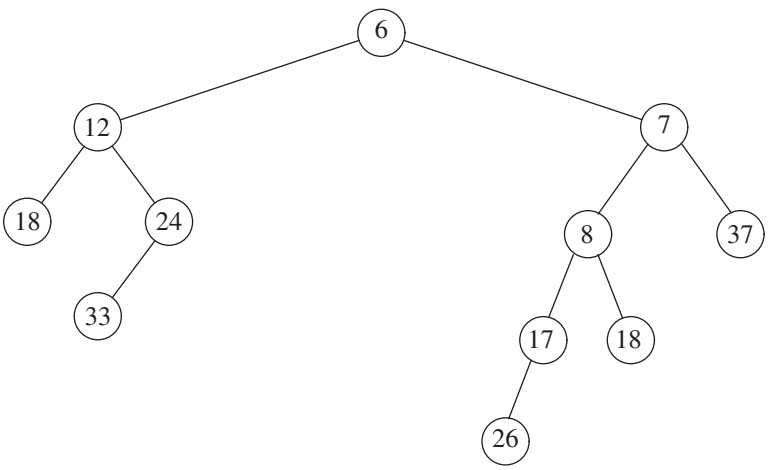

Figure 6.22 Result of merging H_2 with H_1's right subheap

reasonable to assume that the resulting tree is a leftist heap, because it was obtained via a recursive step. This is much like the inductive hypothesis in a proof by induction. Since we can handle the base case (which occurs when one tree is empty), we can assume that the recursive step works as long as we can finish the merge; this is rule 3 of recursion, which we discussed in Chapter 1. We now make this new heap the right child of the root of H_1 (see Fig. 6.23).

Although the resulting heap satisfies the heap-order property, it is not leftist because the left subtree of the root has a null path length of 1 whereas the right subtree has a null path length of 2. Thus, the leftist property is violated at the root. However, it is easy to see that the remainder of the tree must be leftist. The right subtree of the root is leftist because of the recursive step. The left subtree of the root has not been changed, so it too must still be leftist. Thus, we need only to fix the root. We can make the entire tree leftist by merely swapping the root's left and right children (Fig. 6.24) and updating the null path length—the new null path length is 1 plus the null path length of the new right child—completing

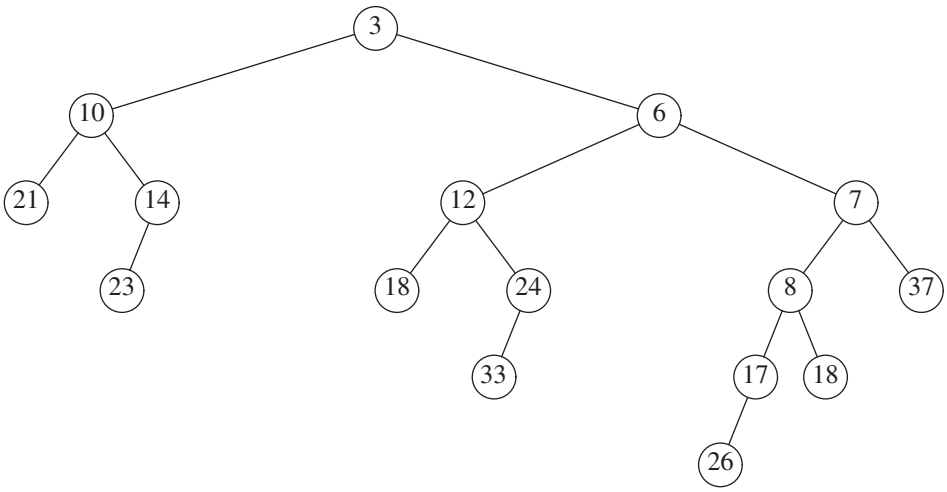

Figure 6.23 Result of attaching leftist heap of previous figure as H_1's right child

the `merge`. Notice that if the null path length is not updated, then all null path lengths will be 0, and the heap will not be leftist but merely random. In this case, the algorithm will work, but the time bound we will claim will no longer be valid.

The description of the algorithm translates directly into code. The node class (Fig. 6.25) is the same as the binary tree, except that it is augmented with the `npl` (null path length) data member. The leftist heap stores a pointer to the root as its data member. We have seen in Chapter 4 that when an element is inserted into an empty binary tree,

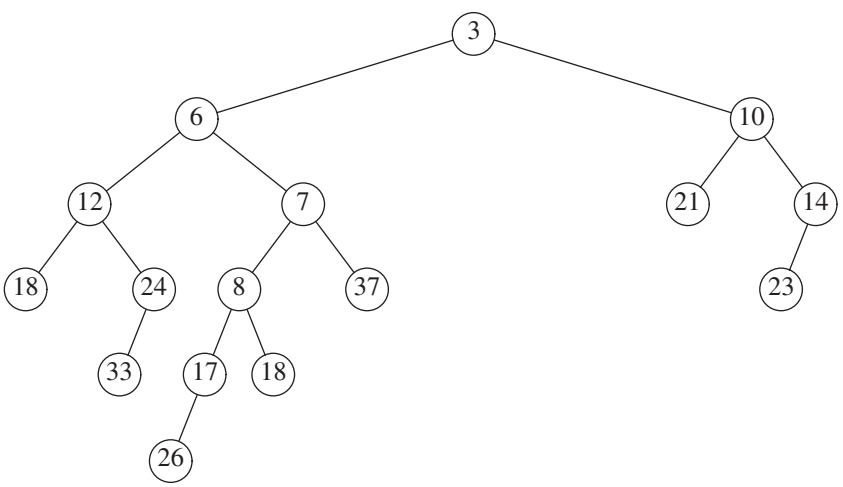

Figure 6.24 Result of swapping children of H_1's root

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

    ~LeftistHeap( );

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

  private:
    struct LeftistNode
    {
        Comparable   element;
        LeftistNode *left;
        LeftistNode *right;
        int          npl;

        LeftistNode( const Comparable & e, LeftistNode *lt = nullptr,
                     LeftistNode *rt = nullptr, int np = 0 )
          : element{ e }, left{ lt }, right{ rt }, npl{ np } { }

        LeftistNode( Comparable && e, LeftistNode *lt = nullptr,
                     LeftistNode *rt = nullptr, int np = 0 )
          : element{ std::move( e ) }, left{ lt }, right{ rt }, npl{ np } { }
    };

    LeftistNode *root;

    LeftistNode * merge( LeftistNode *h1, LeftistNode *h2 );
    LeftistNode * merge1( LeftistNode *h1, LeftistNode *h2 );

    void swapChildren( LeftistNode *t );
    void reclaimMemory( LeftistNode *t );
    LeftistNode * clone( LeftistNode *t ) const;
};
```

Figure 6.25 Leftist heap type declarations

the node referenced by the root will need to change. We use the usual technique of implementing `private` recursive methods to do the merging. The class skeleton is also shown in Figure 6.25.

The two `merge` routines (Fig. 6.26) are drivers designed to remove special cases and ensure that H_1 has the smaller root. The actual merging is performed in `merge1` (Fig. 6.27). The public `merge` method merges `rhs` into the controlling heap. `rhs` becomes empty. The alias test in the public method disallows `h.merge(h)`.

The time to perform the merge is proportional to the sum of the length of the right paths, because constant work is performed at each node visited during the recursive calls. Thus we obtain an $O(\log N)$ time bound to merge two leftist heaps. We can also perform this operation nonrecursively by essentially performing two passes. In the first pass, we create a new tree by merging the right paths of both heaps. To do this, we arrange the nodes on the right paths of H_1 and H_2 in sorted order, keeping their respective left children. In our example, the new right path is 3, 6, 7, 8, 18 and the resulting tree is shown in Figure 6.28.

```
1       /**
2        * Merge rhs into the priority queue.
3        * rhs becomes empty. rhs must be different from this.
4        */
5       void merge( LeftistHeap & rhs )
6       {
7           if( this == &rhs )    // Avoid aliasing problems
8               return;
9
10          root = merge( root, rhs.root );
11          rhs.root = nullptr;
12      }
13
14      /**
15       * Internal method to merge two roots.
16       * Deals with deviant cases and calls recursive merge1.
17       */
18      LeftistNode * merge( LeftistNode *h1, LeftistNode *h2 )
19      {
20          if( h1 == nullptr )
21              return h2;
22          if( h2 == nullptr )
23              return h1;
24          if( h1->element < h2->element )
25              return merge1( h1, h2 );
26          else
27              return merge1( h2, h1 );
28      }
```

Figure 6.26 Driving routines for merging leftist heaps

```
1       /**
2        * Internal method to merge two roots.
3        * Assumes trees are not empty, and h1's root contains smallest item.
4        */
5       LeftistNode * merge1( LeftistNode *h1, LeftistNode *h2 )
6       {
7           if( h1->left == nullptr )   // Single node
8               h1->left = h2;          // Other fields in h1 already accurate
9           else
10          {
11              h1->right = merge( h1->right, h2 );
12              if( h1->left->npl < h1->right->npl )
13                  swapChildren( h1 );
14              h1->npl = h1->right->npl + 1;
15          }
16          return h1;
17      }
```

Figure 6.27 Actual routine to merge leftist heaps

A second pass is made up the heap, and child swaps are performed at nodes that violate the leftist heap property. In Figure 6.28, there is a swap at nodes 7 and 3, and the same tree as before is obtained. The nonrecursive version is simpler to visualize but harder to code. We leave it to the reader to show that the recursive and nonrecursive procedures do the same thing.

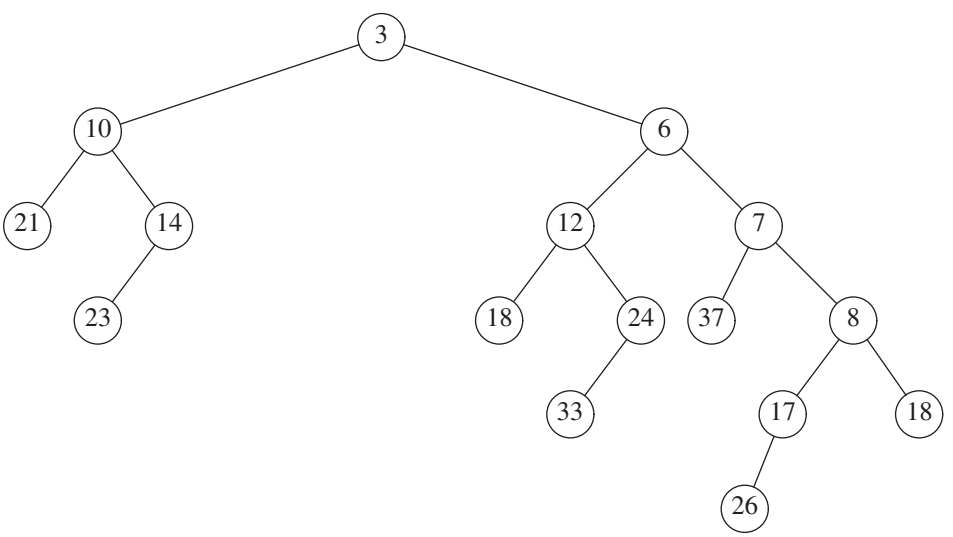

Figure 6.28 Result of merging right paths of H_1 and H_2

```
1       /**
2        * Inserts x; duplicates allowed.
3        */
4       void insert( const Comparable & x )
5       {
6           root = merge( new LeftistNode{ x }, root );
7       }
```

Figure 6.29 Insertion routine for leftist heaps

As mentioned above, we can carry out insertions by making the item to be inserted a one-node heap and performing a `merge`. To perform a `deleteMin`, we merely destroy the root, creating two heaps, which can then be merged. Thus, the time to perform a `deleteMin` is $O(\log N)$. These two routines are coded in Figure 6.29 and Figure 6.30.

Finally, we can build a leftist heap in $O(N)$ time by building a binary heap (obviously using a linked implementation). Although a binary heap is clearly leftist, this is not necessarily the best solution, because the heap we obtain is the worst possible leftist heap. Furthermore, traversing the tree in reverse-level order is not as easy with links. The `buildHeap` effect can be obtained by recursively building the left and right subtrees and then percolating the root down. The exercises contain an alternative solution.

```
1       /**
2        * Remove the minimum item.
3        * Throws UnderflowException if empty.
4        */
5       void deleteMin( )
6       {
7           if( isEmpty( ) )
8               throw UnderflowException{ };
9
10          LeftistNode *oldRoot = root;
11          root = merge( root->left, root->right );
12          delete oldRoot;
13      }
14
15      /**
16       * Remove the minimum item and place it in minItem.
17       * Throws UnderflowException if empty.
18       */
19      void deleteMin( Comparable & minItem )
20      {
21          minItem = findMin( );
22          deleteMin( );
23      }
```

Figure 6.30 `deleteMin` routine for leftist heaps

6.7 Skew Heaps

A **skew heap** is a self-adjusting version of a leftist heap that is incredibly simple to implement. The relationship of skew heaps to leftist heaps is analogous to the relation between splay trees and AVL trees. Skew heaps are binary trees with heap order, but there is no structural constraint on these trees. Unlike leftist heaps, no information is maintained about the null path length of any node. The right path of a skew heap can be arbitrarily long at any time, so the worst-case running time of all operations is $O(N)$. However, as with splay trees, it can be shown (see Chapter 11) that for any M consecutive operations, the total worst-case running time is $O(M \log N)$. Thus, skew heaps have $O(\log N)$ amortized cost per operation.

As with leftist heaps, the fundamental operation on skew heaps is merging. The `merge` routine is once again recursive, and we perform the exact same operations as before, with one exception. The difference is that for leftist heaps, we check to see whether the left and right children satisfy the leftist heap structure property and swap them if they do not. For skew heaps, the swap is unconditional; we *always* do it, with the one exception that the largest of all the nodes on the right paths does not have its children swapped. This one exception is what happens in the natural recursive implementation, so it is not really a special case at all. Furthermore, it is not necessary to prove the bounds, but since this node is guaranteed not to have a right child, it would be silly to perform the swap and give it one. (In our example, there are no children of this node, so we do not worry about it.) Again, suppose our input is the same two heaps as before, Figure 6.31.

If we recursively merge H_2 with the subheap of H_1 rooted at 8, we will get the heap in Figure 6.32.

Again, this is done recursively, so by the third rule of recursion (Section 1.3) we need not worry about how it was obtained. This heap happens to be leftist, but there is no guarantee that this is always the case. We make this heap the new left child of H_1, and the old left child of H_1 becomes the new right child (see Fig. 6.33).

The entire tree is leftist, but it is easy to see that that is not always true: Inserting 15 into this new heap would destroy the leftist property.

We can perform all operations nonrecursively, as with leftist heaps, by merging the right paths and swapping left and right children for every node on the right path, with

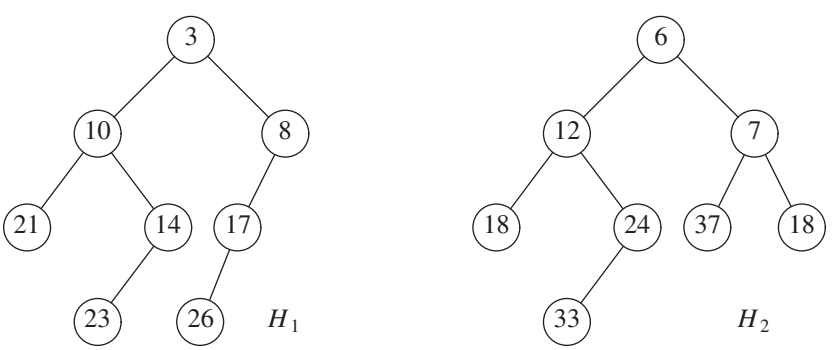

Figure 6.31 Two skew heaps H_1 and H_2

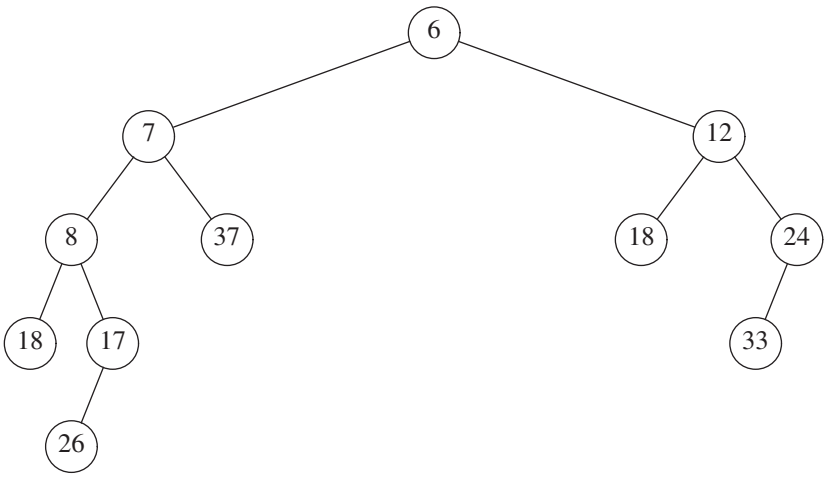

Figure 6.32 Result of merging H_2 with H_1's right subheap

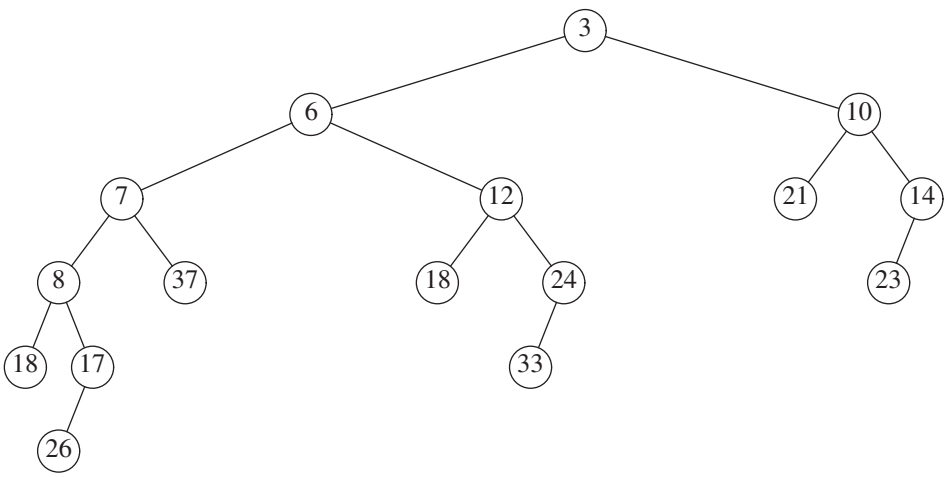

Figure 6.33 Result of merging skew heaps H_1 and H_2

the exception of the last. After a few examples, it becomes clear that since all but the last node on the right path have their children swapped, the net effect is that this becomes the new left path (see the preceding example to convince yourself). This makes it very easy to merge two skew heaps visually.[4]

[4] This is not exactly the same as the recursive implementation (but yields the same time bounds). If we only swap children for nodes on the right path that are above the point where the merging of right paths terminated due to exhaustion of one heap's right path, we get the same result as the recursive version.

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

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.

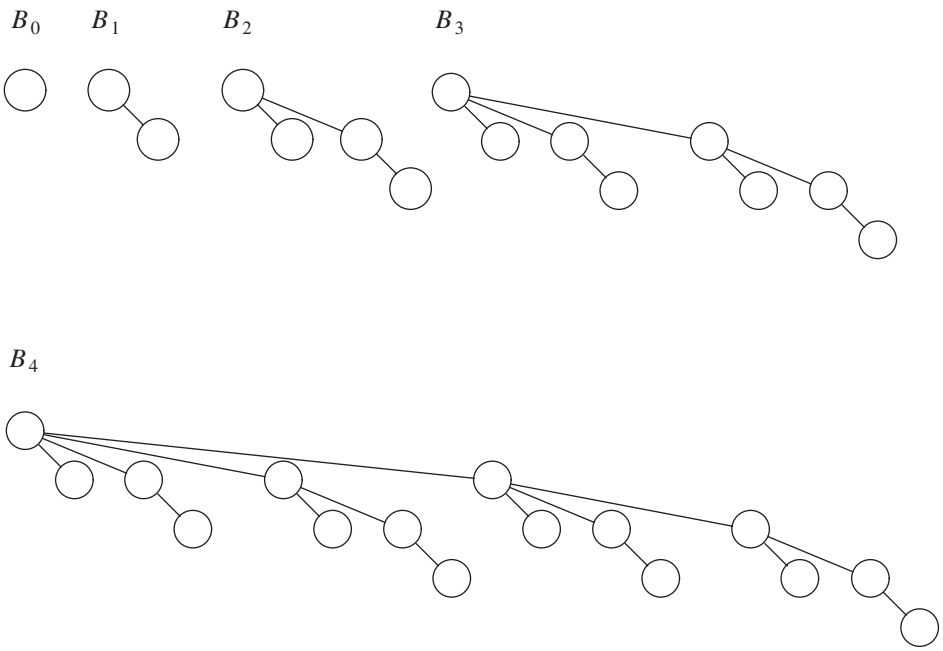

Figure 6.34 Binomial trees B_0, B_1, B_2, B_3, and B_4

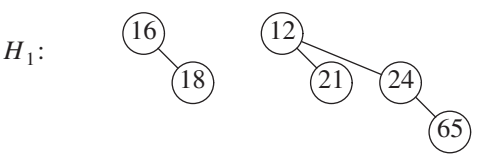

Figure 6.35 Binomial queue H_1 with six elements

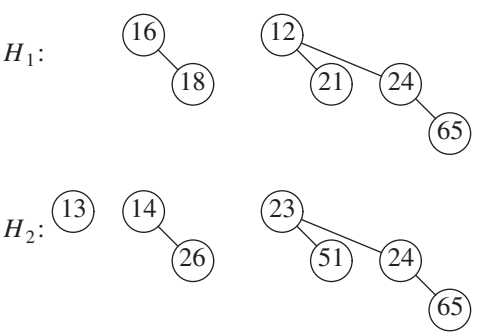

Figure 6.36 Two binomial queues H_1 and H_2

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

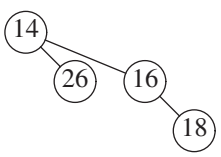

Figure 6.37 Merge of the two B_1 trees in H_1 and H_2

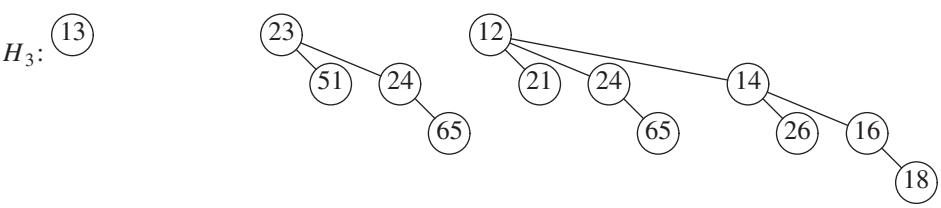

Figure 6.38 Binomial queue H_3: the result of merging H_1 and H_2

Figure 6.39 After 1 is inserted

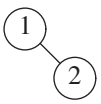

Figure 6.40 After 2 is inserted

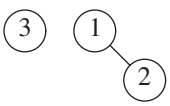

Figure 6.41 After 3 is inserted

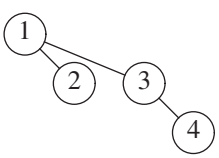

Figure 6.42 After 4 is inserted

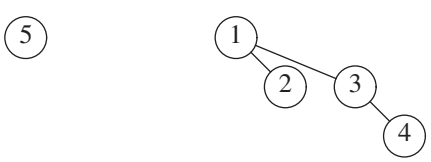

Figure 6.43 After 5 is inserted

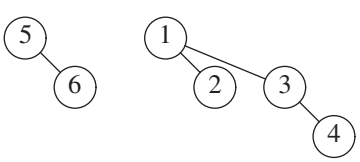

Figure 6.44 After 6 is inserted

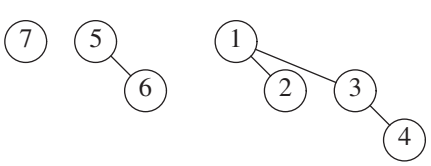

Figure 6.45 After 7 is inserted

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.

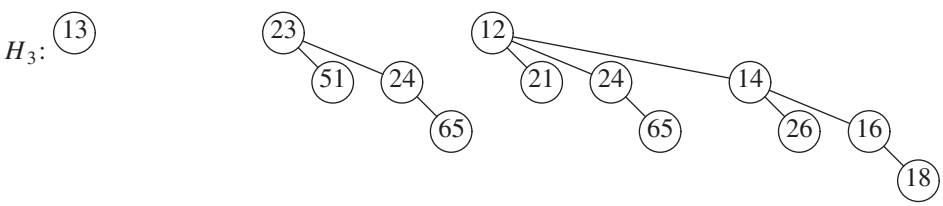

Figure 6.46 Binomial queue H_3

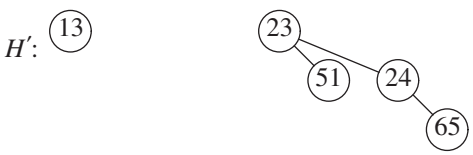

Figure 6.47 Binomial queue H', containing all the binomial trees in H_3 except B_3

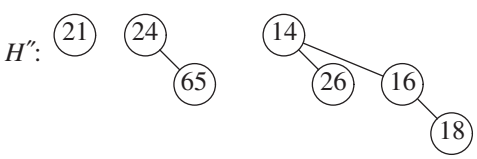

Figure 6.48 Binomial queue H'': B_3 with 12 removed

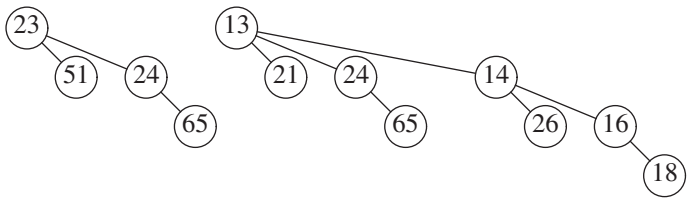

Figure 6.49 Result of applying `deleteMin` to H_3

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.

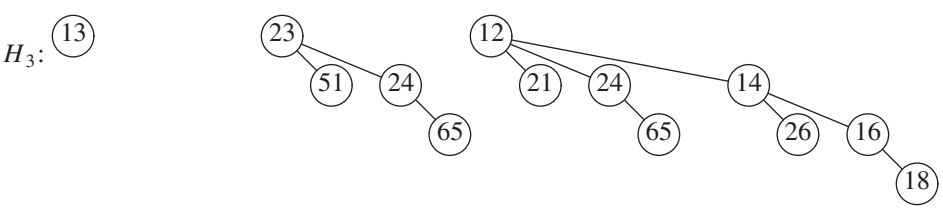

Figure 6.50 Binomial queue H_3 drawn as a forest

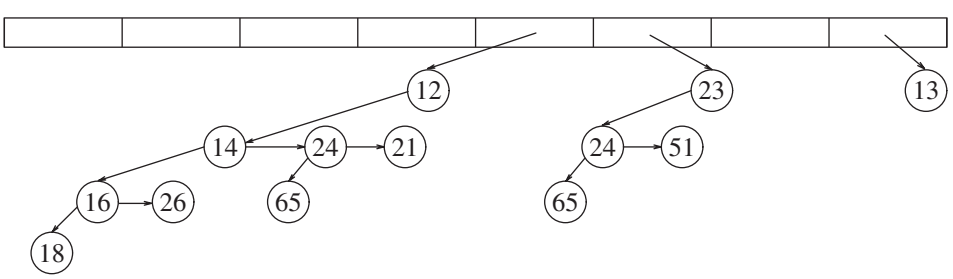

Figure 6.51 Representation of binomial queue H_3

```
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

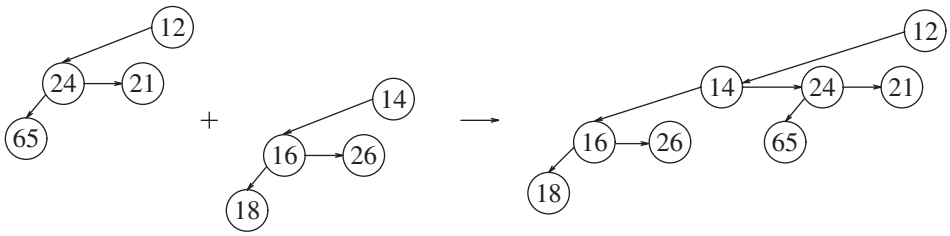

Figure 6.53 Merging two binomial trees

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:

```
void push( const Object & x );
const Object & top( ) const;
void pop( );
bool empty( );
void clear( );
```

`push` adds `x` to the priority queue, `top` returns the largest element in the priority queue, and `pop` removes the largest element from the priority queue. Duplicates are allowed; if there are several largest elements, only one of them is removed.

The priority queue template is instantiated with an item type, the container type (almost always you would want to use a `vector` that stores the items), and the comparator; defaults are allowed for the last two parameters, and the defaults yield a max-heap. Using a `greater` function object as the comparator yields a min-heap.

Figure 6.57 shows a test program that illustrates how the `priority_queue` class template can be used as both the default max-heap and a min-heap.

Summary

In this chapter, we have seen various implementations and uses of the priority queue ADT. The standard binary heap implementation is elegant because of its simplicity and speed. It requires no links and only a constant amount of extra space, yet supports the priority queue operations efficiently.

We considered the additional `merge` operation and developed three implementations, each of which is unique in its own way. The leftist heap is a wonderful example of the power of recursion. The skew heap represents a remarkable data structure because of the lack of balance criteria. Its analysis, which we will perform in Chapter 11, is interesting in its own right. The binomial queue shows how a simple idea can be used to achieve a good time bound.

We have also seen several uses of priority queues, ranging from operating systems scheduling to simulation. We will see their use again in Chapters 7, 9, and 10.

Exercises

6.1 Can both `insert` and `findMin` be implemented in constant time?

6.2 a. Show the result of inserting 10, 12, 1, 14, 6, 5, 8, 15, 3, 9, 7, 4, 11, 13, and 2, one at a time, into an initially empty binary heap.
b. Show the result of using the linear-time algorithm to build a binary heap using the same input.

6.3 Show the result of performing three `deleteMin` operations in the heap of the previous exercise.

6.4 A complete binary tree of N elements uses array positions 1 to N. Suppose we try to use an array representation of a binary tree that is not complete. Determine how large the array must be for the following:

a. a binary tree that has two extra levels (that is, it is very slightly unbalanced)
b. a binary tree that has a deepest node at depth $2 \log N$
c. a binary tree that has a deepest node at depth $4.1 \log N$
d. the worst-case binary tree

6.5 Rewrite the `BinaryHeap insert` routine by placing a copy of the inserted item in position 0.

6.6 How many nodes are in the large heap in Figure 6.13?

6.7 a. Prove that for binary heaps, `buildHeap` does at most $2N-2$ comparisons between elements.
b. Show that a heap of eight elements can be constructed in eight comparisons between heap elements.
** c. Give an algorithm to build a binary heap in $\frac{13}{8}N + O(\log N)$ element comparisons.

6.8 Show the following regarding the maximum item in the heap:
a. It must be at one of the leaves.
b. There are exactly $\lceil N/2 \rceil$ leaves.
c. Every leaf must be examined to find it.

** **6.9** Show that the expected depth of the kth smallest element in a large complete heap (you may assume $N = 2^k - 1$) is bounded by $\log k$.

6.10 * a. Give an algorithm to find all nodes less than some value, X, in a binary heap. Your algorithm should run in $O(K)$, where K is the number of nodes output.
b. Does your algorithm extend to any of the other heap structures discussed in this chapter?
* c. Give an algorithm that finds an arbitrary item X in a binary heap using at most roughly $3N/4$ comparisons.

** **6.11** Propose an algorithm to insert M nodes into a binary heap on N elements in $O(M + \log N \log \log N)$ time. Prove your time bound.

6.12 Write a program to take N elements and do the following:
a. Insert them into a heap one by one.
b. Build a heap in linear time.
Compare the running time of both algorithms for sorted, reverse-ordered, and random inputs.

6.13 Each `deleteMin` operation uses $2 \log N$ comparisons in the worst case.
* a. Propose a scheme so that the `deleteMin` operation uses only $\log N + \log \log N + O(1)$ comparisons between elements. This need not imply less data movement.
** b. Extend your scheme in part (a) so that only $\log N + \log \log \log N + O(1)$ comparisons are performed.
** c. How far can you take this idea?
d. Do the savings in comparisons compensate for the increased complexity of your algorithm?

6.14 If a d-heap is stored as an array, for an entry located in position i, where are the parents and children?

6.15 Suppose we need to perform M `percolateUp`s and N `deleteMin`s on a d-heap that initially has N elements.
a. What is the total running time of all operations in terms of M, N, and d?
b. If $d = 2$, what is the running time of all heap operations?
c. If $d = \Theta(N)$, what is the total running time?
★ d. What choice of d minimizes the total running time?

6.16 Suppose that binary heaps are represented using explicit links. Give a simple algorithm to find the tree node that is at implicit position i.

6.17 Suppose that binary heaps are represented using explicit links. Consider the problem of merging binary heap `lhs` with `rhs`. Assume both heaps are perfect binary trees, containing $2^l - 1$ and $2^r - 1$ nodes, respectively.
a. Give an $O(\log N)$ algorithm to merge the two heaps if $l = r$.
b. Give an $O(\log N)$ algorithm to merge the two heaps if $|l - r| = 1$.
★ c. Give an $O(\log^2 N)$ algorithm to merge the two heaps regardless of l and r.

6.18 A *min-max heap* is a data structure that supports both `deleteMin` and `deleteMax` in $O(\log N)$ per operation. The structure is identical to a binary heap, but the heap-order property is that for any node, X, at even depth, the element stored at X is smaller than the parent but larger than the grandparent (where this makes sense), and for any node, X, at odd depth, the element stored at X is larger than the parent but smaller than the grandparent. See Figure 6.58.
a. How do we find the minimum and maximum elements?
★ b. Give an algorithm to insert a new node into the min-max heap.
★ c. Give an algorithm to perform `deleteMin` and `deleteMax`.
★ d. Can you build a min-max heap in linear time?
★★ e. Suppose we would like to support `deleteMin`, `deleteMax`, and `merge`. Propose a data structure to support all operations in $O(\log N)$ time.

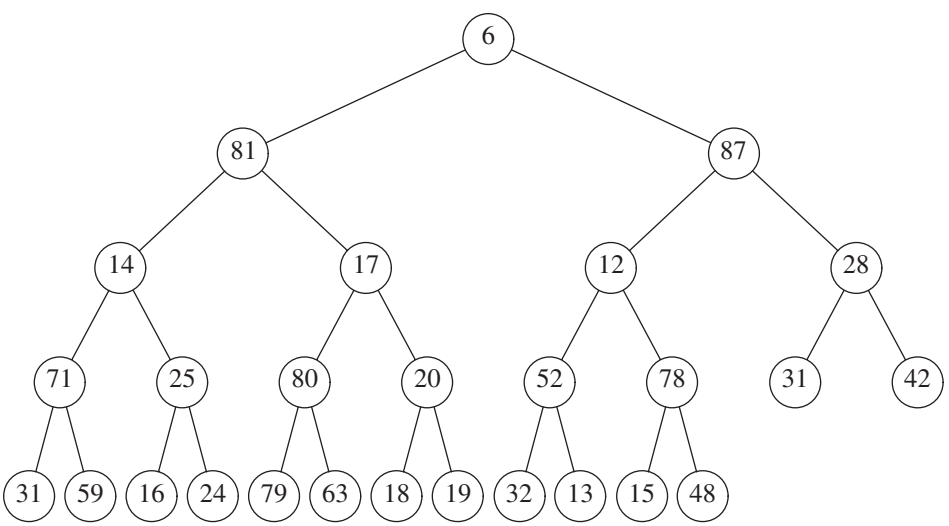

Figure 6.58 Min-max heap

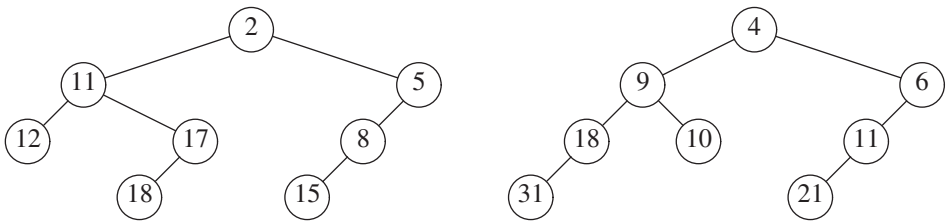

Figure 6.59 Input for Exercises 6.19 and 6.26

6.19 Merge the two leftist heaps in Figure 6.59.

6.20 Show the result of inserting keys 1 to 15 in order into an initially empty leftist heap.

6.21 Prove or disprove: A perfectly balanced tree forms if keys 1 to $2^k - 1$ are inserted in order into an initially empty leftist heap.

6.22 Give an example of input that generates the best leftist heap.

6.23 a. Can leftist heaps efficiently support `decreaseKey`?
b. What changes, if any (if possible), are required to do this?

6.24 One way to delete nodes from a known position in a leftist heap is to use a lazy strategy. To delete a node, merely mark it deleted. When a `findMin` or `deleteMin` is performed, there is a potential problem if the root is marked deleted, since then the node has to be actually deleted and the real minimum needs to be found, which may involve deleting other marked nodes. In this strategy, `remove`s cost one unit, but the cost of a `deleteMin` or `findMin` depends on the number of nodes that are marked deleted. Suppose that after a `deleteMin` or `findMin` there are k fewer marked nodes than before the operation.

* a. Show how to perform the `deleteMin` in $O(k \log N)$ time.

** b. Propose an implementation, with an analysis to show that the time to perform the `deleteMin` is $O(k \log(2N/k))$.

6.25 We can perform `buildHeap` in linear time for leftist heaps by considering each element as a one-node leftist heap, placing all these heaps on a queue, and performing the following step: Until only one heap is on the queue, dequeue two heaps, merge them, and enqueue the result.
a. Prove that this algorithm is $O(N)$ in the worst case.
b. Why might this algorithm be preferable to the algorithm described in the text?

6.26 Merge the two skew heaps in Figure 6.59.

6.27 Show the result of inserting keys 1 to 15 in order into a skew heap.

6.28 Prove or disprove: A perfectly balanced tree forms if the keys 1 to $2^k - 1$ are inserted in order into an initially empty skew heap.

6.29 A skew heap of N elements can be built using the standard binary heap algorithm. Can we use the same merging strategy described in Exercise 6.25 for skew heaps to get an $O(N)$ running time?

6.30 Prove that a binomial tree, B_k, has binomial trees $B_0, B_1, \ldots, B_{k-1}$ as children of the root.

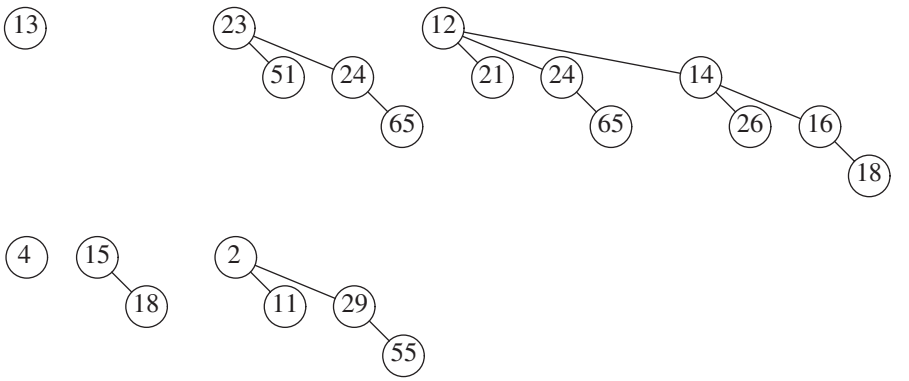

Figure 6.60 Input for Exercise 6.32

6.31 Prove that a binomial tree of height k has $\binom{k}{d}$ nodes at depth d.

6.32 Merge the two binomial queues in Figure 6.60.

6.33 a. Show that N `insert`s into an initially empty binomial queue take $O(N)$ time in the worst case.

b. Give an algorithm to build a binomial queue of N elements, using at most $N - 1$ comparisons between elements.

* c. Propose an algorithm to insert M nodes into a binomial queue of N elements in $O(M + \log N)$ worst-case time. Prove your bound.

6.34 Write an efficient routine to perform `insert` using binomial queues. Do not call `merge`.

6.35 For the binomial queue

a. Modify the `merge` routine to terminate merging if there are no trees left in H_2 and the `carry` tree is `nullptr`.

b. Modify the `merge` so that the smaller tree is always merged into the larger.

** **6.36** Suppose we extend binomial queues to allow at most two trees of the same height per structure. Can we obtain $O(1)$ worst-case time for insertion while retaining $O(\log N)$ for the other operations?

6.37 Suppose you have a number of boxes, each of which can hold total weight C and items $i_1, i_2, i_3, \ldots, i_N$, which weigh $w_1, w_2, w_3, \ldots, w_N$, respectively. The object is to pack all the items without placing more weight in any box than its capacity and using as few boxes as possible. For instance, if $C = 5$, and the items have weights 2, 2, 3, 3, then we can solve the problem with two boxes.

In general, this problem is very hard, and no efficient solution is known. Write programs to implement efficiently the following approximation strategies:

* a. Place the weight in the first box for which it fits (creating a new box if there is no box with enough room). (This strategy and all that follow would give three boxes, which is suboptimal.)

b. Place the weight in the box with the most room for it.

* c. Place the weight in the most filled box that can accept it without overflowing.

** d. Are any of these strategies enhanced by presorting the items by weight?

6.38 Suppose we want to add the `decreaseAllKeys(Δ)` operation to the heap repertoire. The result of this operation is that all keys in the heap have their value decreased by an amount Δ. For the heap implementation of your choice, explain the necessary modifications so that all other operations retain their running times and `decreaseAllKeys` runs in $O(1)$.

6.39 Which of the two selection algorithms has the better time bound?

6.40 The standard copy constructor and `makeEmpty` for leftist heaps can fail because of too many recursive calls. Although this was true for binary search trees, it is more problematic for leftist heaps, because a leftist heap can be very deep, even while it has good worst-case performance for basic operations. Thus the copy constructor and `makeEmpty` need to be reimplemented to avoid deep recursion in leftist heaps. Do this as follows:

a. Reorder the recursive routines so that the recursive call to `t->left` follows the recursive call to `t->right`.

b. Rewrite the routines so that the last statement is a recursive call on the left subtree.

c. Eliminate the tail recursion.

d. These functions are still recursive. Give a precise bound on the depth of the remaining recursion.

* e. Explain how to rewrite the copy constructor and `makeEmpty` for skew heaps.

References

The binary heap was first described in [28]. The linear-time algorithm for its construction is from [14].

The first description of *d*-heaps was in [19]. Recent results suggest that 4-heaps may improve binary heaps in some circumstances [22]. Leftist heaps were invented by Crane [11] and described in Knuth [21]. Skew heaps were developed by Sleator and Tarjan [24]. Binomial queues were invented by Vuillemin [27]; Brown provided a detailed analysis and empirical study showing that they perform well in practice [4], if carefully implemented.

Exercise 6.7(b–c) is taken from [17]. Exercise 6.10(c) is from [6]. A method for constructing binary heaps that uses about $1.52N$ comparisons on average is described in [23]. Lazy deletion in leftist heaps (Exercise 6.24) is from [10]. A solution to Exercise 6.36 can be found in [8].

Min-max heaps (Exercise 6.18) were originally described in [1]. A more efficient implementation of the operations is given in [18] and [25]. Alternative representations for double-ended priority queues are the *deap* and *diamond deque*. Details can be found in [5], [7], and [9]. Solutions to 6.18(e) are given in [12] and [20].

A theoretically interesting priority queue representation is the *Fibonacci heap* [16], which we will describe in Chapter 11. The Fibonacci heap allows all operations to be performed in $O(1)$ amortized time, except for deletions, which are $O(\log N)$. *Relaxed heaps* [13] achieve identical bounds in the worst case (with the exception of `merge`). The procedure of [3] achieves optimal worst-case bounds for all operations. Another interesting implementation is the *pairing heap* [15], which is described in Chapter 12. Finally, priority queues that work when the data consist of small integers are described in [2] and [26].

1. M. D. Atkinson, J. R. Sack, N. Santoro, and T. Strothotte, "Min-Max Heaps and Generalized Priority Queues," *Communications of the ACM,* 29 (1986), 996–1000.
2. J. D. Bright, "Range Restricted Mergeable Priority Queues," *Information Processing Letters,* 47 (1993), 159–164.
3. G. S. Brodal, "Worst-Case Efficient Priority Queues," *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms* (1996), 52–58.
4. M. R. Brown, "Implementation and Analysis of Binomial Queue Algorithms," *SIAM Journal on Computing,* 7 (1978), 298–319.
5. S. Carlsson, "The Deap—A Double-Ended Heap to Implement Double-Ended Priority Queues," *Information Processing Letters,* 26 (1987), 33–36.
6. S. Carlsson and J. Chen, "The Complexity of Heaps," *Proceedings of the Third Symposium on Discrete Algorithms* (1992), 393–402.
7. S. Carlsson, J. Chen, and T. Strothotte, "A Note on the Construction of the Data Structure 'Deap'," *Information Processing Letters,* 31 (1989), 315–317.
8. S. Carlsson, J. I. Munro, and P. V. Poblete, "An Implicit Binomial Queue with Constant Insertion Time," *Proceedings of First Scandinavian Workshop on Algorithm Theory* (1988), 1–13.
9. S. C. Chang and M. W. Due, "Diamond Deque: A Simple Data Structure for Priority Deques," *Information Processing Letters,* 46 (1993), 231–237.
10. D. Cheriton and R. E. Tarjan, "Finding Minimum Spanning Trees," *SIAM Journal on Computing,* 5 (1976), 724–742.
11. C. A. Crane, "Linear Lists and Priority Queues as Balanced Binary Trees," *Technical Report STAN-CS-72-259,* Computer Science Department, Stanford University, Stanford, Calif., 1972.
12. Y. Ding and M. A. Weiss, "The Relaxed Min-Max Heap: A Mergeable Double-Ended Priority Queue," *Acta Informatica,* 30 (1993), 215–231.
13. J. R. Driscoll, H. N. Gabow, R. Shrairman, and R. E. Tarjan, "Relaxed Heaps: An Alternative to Fibonacci Heaps with Applications to Parallel Computation," *Communications of the ACM,* 31 (1988), 1343–1354.
14. R. W. Floyd, "Algorithm 245: Treesort 3," *Communications of the ACM,* 7 (1964), 701.
15. M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan, "The Pairing Heap: A New Form of Self-adjusting Heap," *Algorithmica,* 1 (1986), 111–129.
16. M. L. Fredman and R. E. Tarjan, "Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms," *Journal of the ACM,* 34 (1987), 596–615.
17. G. H. Gonnet and J. I. Munro, "Heaps on Heaps," *SIAM Journal on Computing,* 15 (1986), 964–971.
18. A. Hasham and J. R. Sack, "Bounds for Min-max Heaps," *BIT,* 27 (1987), 315–323.
19. D. B. Johnson, "Priority Queues with Update and Finding Minimum Spanning Trees," *Information Processing Letters,* 4 (1975), 53–57.
20. C. M. Khoong and H. W. Leong, "Double-Ended Binomial Queues," *Proceedings of the Fourth Annual International Symposium on Algorithms and Computation* (1993), 128–137.
21. D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching,* 2d ed., Addison-Wesley, Reading, Mass., 1998.
22. A. LaMarca and R. E. Ladner, "The Influence of Caches on the Performance of Sorting," *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms* (1997), 370–379.

23. C. J. H. McDiarmid and B. A. Reed, "Building Heaps Fast," *Journal of Algorithms,* 10 (1989), 352–365.
24. D. D. Sleator and R. E. Tarjan, "Self-adjusting Heaps," *SIAM Journal on Computing,* 15 (1986), 52–69.
25. T. Strothotte, P. Eriksson, and S. Vallner, "A Note on Constructing Min-max Heaps," *BIT,* 29 (1989), 251–256.
26. P. van Emde Boas, R. Kaas, and E. Zijlstra, "Design and Implementation of an Efficient Priority Queue," *Mathematical Systems Theory,* 10 (1977), 99–127.
27. J. Vuillemin, "A Data Structure for Manipulating Priority Queues," *Communications of the ACM,* 21 (1978), 309–314.
28. J. W. J. Williams, "Algorithm 232: Heapsort," *Communications of the ACM,* 7 (1964), 347–348.