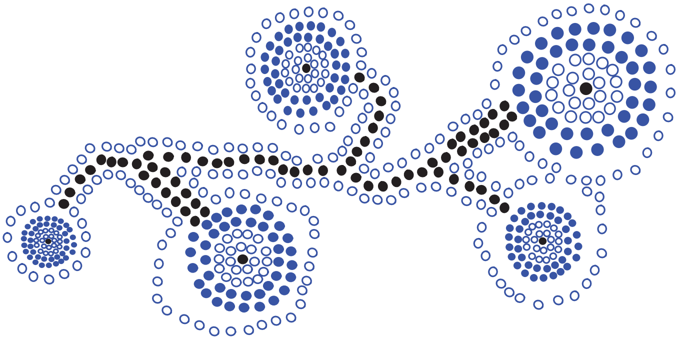
Chapter

8 Heaps and Priority Queues



Contents

8.1 The Priority Queue Abstract Data Type

A ***priority queue*** is an abstract data type for storing a collection of prioritized elements that supports arbitrary element insertion but supports removal of elements in order of priority, that is, the element with first priority can be removed at any time. This ADT is fundamentally different from the position-based data structures such as stacks, queues, deques, lists, and even trees, we discussed in previous chapters. These other data structures store elements at specific positions, which are often positions in a linear arrangement of the elements determined by the insertion and deletion operations performed. The priority queue ADT stores elements according to their priorities, and has no external notion of "position."

8.1.1 Keys, Priorities, and Total Order Relations

Applications commonly require comparing and ranking objects according to parameters or properties, called "keys," that are assigned to each object in a collection. Formally, we define a ***key*** to be an object that is assigned to an element as a specific attribute for that element and that can be used to identify, rank, or weigh that element. Note that the key is assigned to an element, typically by a user or application; hence, a key might represent a property that an element did not originally possess.

The key an application assigns to an element is not necessarily unique, however, and an application may even change an element's key if it needs to. For example, we can compare companies by earnings or by number of employees; hence, either of these parameters can be used as a key for a company, depending on the information we wish to extract. Likewise, we can compare restaurants by a critic's food quality rating or by average entrée price. To achieve the most generality then, we allow a key to be of any type that is appropriate for a particular application.

As in the examples above, the key used for comparisons is often more than a single numerical value, such as price, length, weight, or speed. That is, a key can sometimes be a more complex property that cannot be quantified with a single number. For example, the priority of standby passengers is usually determined by taking into account a host of different factors, including frequent-flyer status, the fare paid, and check-in time. In some applications, the key for an object is data extracted from the object itself (for example, it might be a member variable storing the list price of a book, or the weight of a car). In other applications, the key is not part of the object but is externally generated by the application (for example, the quality rating given to a stock by a financial analyst, or the priority assigned to a standby passenger by a gate agent).

Comparing Keys with Total Orders

A priority queue needs a comparison rule that never contradicts itself. In order for a comparison rule, which we denote by $\leq$, to be robust in this way, it must define a ***total order*** relation, which is to say that the comparison rule is defined for every pair of keys and it must satisfy the following properties:

- **Reflexive property** : $k \leq k$
- **Antisymmetric property**: if $k_1 \leq k_2$ and $k_2 \leq k_1$, then $k_1 = k_2$
- **Transitive property**: if $k_1 \leq k_2$ and $k_2 \leq k_3$, then $k_1 \leq k_3$

Any comparison rule, $\leq$, that satisfies these three properties never leads to a comparison contradiction. In fact, such a rule defines a linear ordering relationship among a set of keys. If a finite collection of keys has a total order defined for it, then the notion of the ***smallest*** key, $k_{\min}$, is well defined as the key, such that $k_{\min} \leq k$, for any other key k in our collection.

A ***priority queue*** is a container of elements, each associated with a key. The name "priority queue" comes from the fact that keys determine the "priority" used to pick elements to be removed. The fundamental functions of a priority queue P are as follows:

insert(e): Insert the element e (with an implicit associated key value) into P.

min(): Return an element of P with the smallest associated key value, that is, an element whose key is less than or equal to that of every other element in P.

removeMin(): Remove from P the element min().

Note that more than one element can have the same key, which is why we were careful to define removeMin to remove not just any minimum element, but the same element returned by min. Some people refer to the removeMin function as extractMin.

There are many applications where operations insert and removeMin play an important role. We consider such an application in the example that follows.

Example 8.1: *Suppose a certain flight is fully booked an hour prior to departure. Because of the possibility of cancellations, the airline maintains a priority queue of standby passengers hoping to get a seat. The priority of each passenger is determined by the fare paid, the frequent-flyer status, and the time when the passenger is inserted into the priority queue. When a passenger requests to fly standby, the associated passenger object is inserted into the priority queue with an* insert *operation. Shortly before the flight departure, if seats become available (for example, due to last-minute cancellations), the airline repeatedly removes a standby passenger with first priority from the priority queue, using a combination of* min *and* removeMin *operations, and lets this person board.*

8.1.2 Comparators

An important issue in the priority queue ADT that we have so far left undefined is how to specify the total order relation for comparing the keys associated with each element. There are a number of ways of doing this, each having its particular advantages and disadvantages.

The most direct solution is to implement a different priority queue based on the element type and the manner of comparing elements. While this approach is arguably simple, it is certainly not very general, since it would require that we make many copies of essentially the same code. Maintaining multiple copies of the nearly equivalent code is messy and error prone.

A better approach would be to design the priority queue as a templated class, where the element type is specified by an abstract template argument, say E. We assume that each concrete class that could serve as an element of our priority queue provides a means for comparing two objects of type E. This could be done in many ways. Perhaps we require that each object of type E provides a function called comp that compares two objects of type E and determines which is larger. Perhaps we require that the programmer defines a function that overloads the C++ comparison operator "<" for two objects of type E. (Recall Section 1.4.2 for a discussion of operator overloading). In C++ jargon this is called a ***function object***.

Let us consider a more concrete example. Suppose that class Point2D defines a two-dimensional point. It has two public member functions, getX and getY, which access its x and y coordinates, respectively. We could define a lexicographical less-than operator as follows. If the x coordinates differ we use their relative values; otherwise, we use the relative values of the y coordinates.

```
bool operator<(const Point2D& p, const Point2D& q) {
  if (p.getX() == q.getX())      return p.getY() < q.getY();
  else                           return p.getX() < q.getX();
}
```

This approach of overloading the relational operators is general enough for many situations, but it relies on the assumption that objects of the same type are always compared in the same way. There are situations, however, where it is desirable to apply different comparisons to objects of the same type. Consider the following examples.

Example 8.2: *There are at least two ways of comparing the* C++ *character strings,* "4" *and* "12"*. In the* **lexicographic ordering***, which is an extension of the alphabetic ordering to character strings, we have* "4" > "12"*. But if we interpret these strings as integers, then* "4" < "12"*.*

Example 8.3: *A geometric algorithm may compare points p and q in two-dimensional space, by their x-coordinate (that is, $p \leq q$ if $p_x \leq q_x$), to sort them from left to right, while another algorithm may compare them by their y-coordinate (that is, $p \leq q$ if $p_y \leq q_y$), to sort them from bottom to top. In principle, there is nothing pertaining to the concept of a point that says whether points should be compared by x- or y-coordinates. Also, many other ways of comparing points can be defined (for example, we can compare the distances of p and q from the origin).*

There are a couple of ways to achieve the goal of independence of element type and comparison method. The most general approach, called the ***composition method***, is based on defining each entry of our priority queue to be a pair (e,k), consisting of an element e and a key k. The element part stores the data, and the key part stores the information that defines the priority ordering. Each key object defines its own comparison function. By changing the key class, we can change the way in which the queue is ordered. This approach is very general, because the key part does not need to depend on the data present in the element part. We study this approach in greater detail in Chapter 9.

The approach that we use is a bit simpler than the composition method. It is based on defining a special object, called a ***comparator***, whose job is to provide a definition of the comparison function between any two elements. This can be done in various ways. In C++, a comparator for element type E can be implemented as a class that defines a single function whose job is to compare two objects of type E. One way to do this is to overload the "()" operator. The resulting function takes two arguments, a and b, and returns a boolean whose value is true if $a < b$. For example, if "isLess" is the name of our comparator object, the comparison function is invoked using the following operation:

isLess(a,b): Return true if $a < b$ and false otherwise.

It might seem at first that defining just a less-than function is rather limited, but note that it is possible to derive all the other relational operators by combining less-than comparisons with other boolean operators. For example, we can test whether a and b are equal with `(!isLess(a,b) && !isLess(b, a))`. (See Exercise R-8.3.)

Defining and Using Comparator Objects

Let us consider a more concrete example of a comparator class. As mentioned in the above example, let us suppose that we have defined a class structure, called Point2D, for storing a two-dimensional point. In Code Fragment 8.1, we present two comparators. The comparator LeftRight implements a left-to-right order by comparing the x-coordinates of the points, and the comparator BottomTop implements a bottom-to-top order by comparing the y-coordinates of the points.

To use these comparators, we would declare two objects, one of each type. Let us call them leftRight and bottomTop. Observe that these objects store no

```
class LeftRight {                              // a left-right comparator
public:
  bool operator()(const Point2D& p, const Point2D& q) const
    { return p.getX() < q.getX(); }
};

class BottomTop {                              // a bottom-top comparator
public:
  bool operator()(const Point2D& p, const Point2D& q) const
    { return p.getY() < q.getY(); }
};
```

Code Fragment 8.1: Two comparator classes for comparing points. The first implements a left-to-right order and the second implements a bottom-to-top order.

data members. They are used solely for the purposes of specifying a particular comparison operator. Given two objects p and q, each of type Point2D, to test whether p is to the left of q, we would invoke leftRight(p,q), and to test whether p is below q, we would invoke bottomTop(p,q). Each invokes the "()" operator for the corresponding class.

Next, let us see how we can use our comparators to implement two different behaviors. Consider the generic function printSmaller shown in Code Fragment 8.2. It prints the smaller of its two arguments. The function definition is templated by the element type E and the comparator type C. The comparator class is assumed to implement a less-than function for two objects of type E. The function is given three arguments, the two elements p and q to be compared and an instance isLess of a comparator for these elements. The function invokes the comparator to determine which element is smaller, and then prints this value.

```
template <typename E, typename C>      // element type and comparator
void printSmaller(const E& p, const E& q, const C& isLess) {
  cout << (isLess(p, q) ? p : q) << endl;  // print the smaller of p and q
}
```

Code Fragment 8.2: A generic function that prints the smaller of two elements, given a comparator for these elements.

Finally, let us see how we can apply our function on two points. The code is shown in Code Fragment 8.3. We declare to points p and q and initialize their coordinates. (We have not presented the class definition for Point2D, but let us assume that the constructor is given the x- and y-coordinates, and we have provided an output operator.) We then declare two comparator objects, one for a left-to-right ordering and the other for a bottom-to-top ordering. Finally, we invoke the function printSmaller on the two points, changing only the comparator objects in each case.

Observe that, depending on which comparator is provided, the call to the func-

```
Point2D p(1.3, 5.7), q(2.5, 0.6);         // two points
LeftRight leftRight;                      // a left-right comparator
BottomTop bottomTop;                      // a bottom-top comparator
printSmaller(p, q, leftRight);            // outputs: (1.3, 5.7)
printSmaller(p, q, bottomTop);            // outputs: (2.5, 0.6)
```

Code Fragment 8.3: The use of two comparators to implement different behaviors from the function printSmaller.

tion isLess in function printSmaller invokes either the "()" operator of class LeftRight or BottomTop. In this way, we obtain the desired result, two different behaviors for the same two-dimensional point class.

Through the use of comparators, a programmer can write a general priority queue implementation that works correctly in a wide variety of contexts. In particular, the priority queues presented in this chapter are generic classes that are templated by two types, the element E and the comparator C.

The comparator approach is a bit less general than the composition method, because the comparator bases its decisions on the contents of the elements themselves. In the composition method, the key may contain information that is not part of the element object. The comparator approach has the advantage of being simpler, since we can insert elements directly into our priority queue without creating element-key pairs. Furthermore, in Exercise R-8.4 we show that there is no real loss of generality in using comparators.

8.1.3 The Priority Queue ADT

Having described the priority queue abstract data type at an intuitive level, we now describe it in more detail. As an ADT, a priority queue P supports the following functions:

size(): Return the number of elements in P.

empty(): Return true if P is empty and false otherwise.

insert(e): Insert a new element e into P.

min(): Return a reference to an element of P with the smallest associated key value (but do not remove it); an error condition occurs if the priority queue is empty.

removeMin(): Remove from P the element referenced by min(); an error condition occurs if the priority queue is empty.

As mentioned above, the primary functions of the priority queue ADT are the insert, min, and removeMin operations. The other functions, size and empty, are generic collection operations. Note that we allow a priority queue to have multiple entries with the same key.

Example 8.4: *The following table shows a series of operations and their effects on an initially empty priority queue P. Each element consists of an integer, which we assume to be sorted according to the natural ordering of the integers. Note that each call to* min *returns a reference to an entry in the queue, not the actual value. Although the "Priority Queue" column shows the items in sorted order, the priority queue need not store elements in this order.*

Operation	*Output*	*Priority Queue*
insert(5)	–	{5}
insert(9)	–	{5,9}
insert(2)	–	{2,5,9}
insert(7)	–	{2,5,7,9}
min()	[2]	{2,5,7,9}
removeMin()	–	{5,7,9}
size()	3	{5,7,9}
min()	[5]	{5,7,9}
removeMin()	–	{7,9}
removeMin()	–	{9}
removeMin()	–	{}
empty()	*true*	{}
removeMin()	*"error"*	{}

8.1.4 A C++ Priority Queue Interface

Before discussing specific implementations of the priority queue, we first define an informal C++ interface for a priority queue in Code Fragment 8.4. It is not a complete C++ class, just a declaration of the public functions.

```
template <typename E, typename C>                  // element and comparator
class PriorityQueue {                              // priority-queue interface
public:
  int size() const;                                // number of elements
  bool isEmpty() const;                            // is the queue empty?
  void insert(const E& e);                         // insert element
  const E& min() const throw(QueueEmpty);          // minimum element
  void removeMin() throw(QueueEmpty);              // remove minimum
};
```

Code Fragment 8.4: An informal PriorityQueue interface (not a complete class).

Although the comparator type C is included as a template argument, it does not appear in the public interface. Of course, its value is relevant to any concrete implementation. Observe that the function min returns a constant reference to the element in the queue, which means that its value may be read and copied but not modified. This is important because otherwise a user of the class might inadvertently modify the element's associated key value, and this could corrupt the integrity of the data structure. The member functions size, empty, and min are all declared to be **const**, which informs the compiler that they do not alter the contents of the queue.

An error condition occurs if either of the functions min or removeMin is called on an empty priority queue. This is signaled by throwing an exception of type QueueEmpty. Its definition is similar to others we have seen. (See Code Fragment 5.2.)

8.1.5 Sorting with a Priority Queue

Another important application of a priority queue is sorting, where we are given a collection L of n elements that can be compared according to a total order relation, and we want to rearrange them in increasing order (or at least in nondecreasing order if there are ties). The algorithm for sorting L with a priority queue Q, called PriorityQueueSort, is quite simple and consists of the following two phases:

1. In the first phase, we put the elements of L into an initially empty priority queue P through a series of n insert operations, one for each element.
2. In the second phase, we extract the elements from P in nondecreasing order by means of a series of n combinations of min and removeMin operations, putting them back into L in order.

Pseudo-code for this algorithm is given in Code Fragment 8.5. It assumes that L is given as an STL list, but the code can be adapted to other containers.

```
Algorithm PriorityQueueSort($L, P$):
  Input: An STL list $L$ of $n$ elements and a priority queue, $P$, that compares
     elements using a total order relation
  Output: The sorted list $L$
   while !$L$.empty() do
     $e \leftarrow L$.front
     $L$.pop_front()          {remove an element $e$ from the list}
     $P$.insert($e$)          {... and it to the priority queue}
   while !$P$.empty() do
     $e \leftarrow P$.min()
     $P$.removeMin()          {remove the smallest element $e$ from the queue}
     $L$.push_back($e$)       {... and append it to the back of $L$}
```

Code Fragment 8.5: Algorithm PriorityQueueSort, which sorts an STL list L with the aid of a priority queue P.

The algorithm works correctly for any priority queue *P*, no matter how *P* is implemented. However, the running time of the algorithm is determined by the running times of operations insert, min, and removeMin, which do depend on how *P* is implemented. Indeed, PriorityQueueSort should be considered more a sorting "scheme" than a sorting "algorithm," because it does not specify how the priority queue *P* is implemented. The PriorityQueueSort scheme is the paradigm of several popular sorting algorithms, including selection-sort, insertion-sort, and heap-sort, which we discuss in this chapter.

8.1.6 The STL priority_queue Class

The C++ Standard Template Library (STL) provides an implementation of a priority queue, called priority_queue. As with the other STL classes we have seen, such as stacks and queues, the STL priority queue is an example of a container. In order to declare an object of type priority_queue, it is necessary to first include the definition file, which is called "queue." As with other STL objects, the priority queue is part of the std namespace, and hence it is necessary either to use "std::priority_queue" or to provide an appropriate "**using**" statement.

The priority_queue class is templated with three parameters: the base type of the elements, the underlying STL container in which the priority queue is stored, and the comparator object. Only the first template argument is required. The second parameter (the underlying container) defaults to the STL vector. The third parameter (the comparator) defaults to using the standard C++ less-than operator ("<"). The STL priority queue uses comparators in the same manner as we defined in Section 8.1.2. In particular, a comparator is a class that overrides the "()" operator in order to define a boolean function that implements the less-than operator.

The code fragment below defines two STL priority queues. The first stores integers. The second stores two-dimensional points under the left-to-right ordering (recall Section 8.1.2).

```
#include <queue>
using namespace std;                    // make std accessible
priority_queue<int> p1;                 // a priority queue of integers
                              // a priority queue of points with left-to-right order
priority_queue<Point2D, vector<Point2D>, LeftRight> p2;
```

The principal member functions of the STL priority queue are given below. Let *p* be declared to be an STL priority_queue, and let *e* denote a single object whose type is the same as the base type of the priority queue. (For example, *p* is a priority queue of integers, and *e* is an integer.)

size(): Return the number of elements in the priority queue.

empty(): Return true if the priority queue is empty and false otherwise.

push(e): Insert e in the priority queue.

top(): Return a constant reference to the largest element of the priority queue.

pop(): Remove the element at the top of the priority queue.

Other than the differences in function names, the most significant difference between our interface and the STL priority queue is that the functions top and pop access the ***largest*** item in the queue according to priority order, rather than the smallest. An example of the usage of the STL priority queue is shown in Code Fragment 8.6.

```
priority_queue<Point2D, vector<Point2D>, LeftRight> p2;
p2.push( Point2D(8.5, 4.6) );                // add three points to p2
p2.push( Point2D(1.3, 5.7) );
p2.push( Point2D(2.5, 0.6) );
cout << p2.top() << endl; p2.pop();          // output: (8.5, 4.6)
cout << p2.top() << endl; p2.pop();          // output: (2.5, 0.6)
cout << p2.top() << endl; p2.pop();          // output: (1.3, 5.7)
```

Code Fragment 8.6: An example of the use of the STL priority queue.

Of course, it is possible to simulate the same behavior as our priority queue by defining the comparator object so that it implements the greater-than relation rather than the less-than relation. This effectively reverses all order relations, and thus the top function would instead return the smallest element, just as function min does in our interface. Note that the STL priority queue does not perform any error checking.

8.2 Implementing a Priority Queue with a List

In this section, we show how to implement a priority queue by storing its elements in an STL list. (Recall this data structure from Section 6.2.4.) We consider two realizations, depending on whether we sort the elements of the list.

Implementation with an Unsorted List

Let us first consider the implementation of a priority queue P by an unsorted doubly linked list L. A simple way to perform the operation insert(e) on P is by adding each new element at the end of L by executing the function L.push_back(e). This implementation of insert takes $O(1)$ time.

Since the insertion does not consider key values, the resulting list L is unsorted. As a consequence, in order to perform either of the operations min or removeMin on P, we must inspect all the entries of the list to find one with the minimum key value. Thus, functions min and removeMin take $O(n)$ time each, where n is the number of elements in P at the time the function is executed. Moreover, each of these functions runs in time proportional to n even in the best case, since they each require searching the entire list to find the smallest element. Using the notation of Section 4.2.3, we can say that these functions run in $\Theta(n)$ time. We implement functions size and empty by simply returning the output of the corresponding functions executed on list L. Thus, by using an unsorted list to implement a priority queue, we achieve constant-time insertion, but linear-time search and removal.

Implementation with a Sorted List

An alternative implementation of a priority queue P also uses a list L, except that this time let us store the elements sorted by their key values. Specifically, we represent the priority queue P by using a list L of elements sorted by nondecreasing key values, which means that the first element of L has the smallest key.

We can implement function min in this case by accessing the element associated with the first element of the list with the begin function of L. Likewise, we can implement the removeMin function of P as L.pop_front(). Assuming that L is implemented as a doubly linked list, operations min and removeMin in P take $O(1)$ time, so are quite efficient.

This benefit comes at a cost, however, for now function insert of P requires that we scan through the list L to find the appropriate position in which to insert the new entry. Thus, implementing the insert function of P now takes $O(n)$ time, where n is the number of entries in P at the time the function is executed. In summary, when using a sorted list to implement a priority queue, insertion runs in linear time whereas finding and removing the minimum can be done in constant time.

Table 8.1 compares the running times of the functions of a priority queue realized by means of an unsorted and sorted list, respectively. There is an interesting contrast between the two functions. An unsorted list allows for fast insertions but slow queries and deletions, while a sorted list allows for fast queries and deletions, but slow insertions.

Operation	*Unsorted List*	*Sorted List*
size, empty	$O(1)$	$O(1)$
insert	$O(1)$	$O(n)$
min, removeMin	$O(n)$	$O(1)$

Table 8.1: Worst-case running times of the functions of a priority queue of size n, realized by means of an unsorted or sorted list, respectively. We assume that the list is implemented by a doubly linked list. The space requirement is $O(n)$.

8.2.1 A C++ Priority Queue Implementation using a List

In Code Fragments 8.7 through 8.10, we present a priority queue implementation that stores the elements in a sorted list. The list is implemented using an STL list object (see Section 6.3.2), but any implementation of the list ADT would suffice.

In Code Fragment 8.7, we present the class definition for our priority queue. The public part of the class is essentially the same as the interface that was presented earlier in Code Fragment 8.4. In order to keep the code as simple as possible, we have omitted error checking. The class's data members consists of a list, which holds the priority queue's contents, and an instance of the comparator object, which we call isLess.

```
template <typename E, typename C>
class ListPriorityQueue {
public:
  int size() const;                          // number of elements
  bool empty() const;                        // is the queue empty?
  void insert(const E& e);                   // insert element
  const E& min() const;                      // minimum element
  void removeMin();                          // remove minimum
private:
  std::list<E> L;                            // priority queue contents
  C isLess;                                  // less-than comparator
};
```

Code Fragment 8.7: The class definition for a priority queue based on an STL list.

We have not bothered to give an explicit constructor for our class, relying instead on the default constructor. The default constructor for the STL list produces an empty list, which is exactly what we want.

Next, in Code Fragment 8.8, we present the implementations of the simple member functions size and empty. Recall that, when dealing with templated classes, it is necessary to repeat the full template specifications when defining member functions outside the class. Each of these functions simply invokes the corresponding function for the STL list.

```
template <typename E, typename C>           // number of elements
int ListPriorityQueue<E,C>::size() const
  { return L.size(); }

template <typename E, typename C>           // is the queue empty?
bool ListPriorityQueue<E,C>::empty() const
  { return L.empty(); }
```

Code Fragment 8.8: Implementations of the functions size and empty.

Let us now consider how to insert an element e into our priority queue. We define p to be an iterator for the list. Our approach is to walk through the list until we first find an element whose key value is larger than e's, and then we insert e just prior to p. Recall that $*p$ accesses the element referenced by p, and $++p$ advances p to the next element of the list. We stop the search either when we reach the end of the list or when we first encounter a larger element, that is, one satisfying isLess$(e, *p)$. On reaching such an entry, we insert e just prior to it, by invoking the STL list function insert. The code is shown in Code Fragment 8.9.

```
template <typename E, typename C>                    // insert element
void ListPriorityQueue<E,C>::insert(const E& e) {
  typename std::list<E>::iterator p;
  p = L.begin();
  while (p != L.end() && !isLess(e, *p)) ++p;        // find larger element
  L.insert(p, e);                                    // insert e before p
}
```

Code Fragment 8.9: Implementation of the priority queue function insert.

Consider how the above function behaves when e has a key value larger than any in the queue. In such a case, the while loop exits under the condition that p is equal to L.end(). Recall that L.end() refers to an imaginary element that lies just beyond the end of the list. Thus, by inserting before this element, we effectively append e to the back of the list, as desired.

Caution

You might notice the use of the keyword "**typename**" in the declaration of the iterator p. This is due to a subtle issue in C++ involving ***dependent names***, which arises when processing name bindings within templated objects in C++. We do not delve into the intricacies of this issue. For now, it suffices to remember to simply include the keyword **typename** when using a template parameter (in this case E) to define another type.

Finally, let us consider the operations min and removeMin. Since the list is sorted in ascending order by key values, in order to implement min, we simply return a reference to the front of the list. To implement removeMin, we remove the front element of the list. The implementations are given in Code Fragment 8.10.

```
template <typename E, typename C>                    // minimum element
const E& ListPriorityQueue<E,C>::min() const
  { return L.front(); }                              // minimum is at the front

template <typename E, typename C>                    // remove minimum
void ListPriorityQueue<E,C>::removeMin()
  { L.pop_front(); }
```

Code Fragment 8.10: Implementations of the priority queue functions min and removeMin.

8.2.2 Selection-Sort and Insertion-Sort

Recall the PriorityQueueSort scheme introduced in Section 8.1.5. We are given an unsorted list L containing n elements, which we sort using a priority queue P in two phases. In the first phase, we insert all the elements, and in the second phase, we repeatedly remove elements using the min and removeMin operations.

Selection-Sort

If we implement the priority queue P with an unsorted list, then the first phase of PriorityQueueSort takes $O(n)$ time, since we can insert each element in constant time. In the second phase, the running time of each min and removeMin operation is proportional to the number of elements currently in P. Thus, the bottleneck computation in this implementation is the repeated "selection" of the minimum element from an unsorted list in the second phase. For this reason, this algorithm is better known as ***selection-sort***. (See Figure 8.1.)

		List L	***Priority Queue P***
Input		$(7,4,8,2,5,3,9)$	$()$
Phase 1	(a)	$(4,8,2,5,3,9)$	(7)
	(b)	$(8,2,5,3,9)$	$(7,4)$
	⋮	⋮	⋮
	(g)	$()$	$(7,4,8,2,5,3,9)$
Phase 2	(a)	(2)	$(7,4,8,5,3,9)$
	(b)	$(2,3)$	$(7,4,8,5,9)$
	(c)	$(2,3,4)$	$(7,8,5,9)$
	(d)	$(2,3,4,5)$	$(7,8,9)$
	(e)	$(2,3,4,5,7)$	$(8,9)$
	(f)	$(2,3,4,5,7,8)$	(9)
	(g)	$(2,3,4,5,7,8,9)$	$()$

Figure 8.1: Execution of selection-sort on list $L = (7,4,8,2,5,3,9)$.

As noted above, the bottleneck is the second phase, where we repeatedly remove an element with smallest key from the priority queue P. The size of P starts at n and decreases to 0 with each removeMin. Thus, the first removeMin operation takes time $O(n)$, the second one takes time $O(n-1)$, and so on. Therefore, the total time needed for the second phase is

$$O\left(n+(n-1)+\cdots+2+1\right) = O\left(\textstyle\sum_{i=1}^{n} i\right).$$

By Proposition 4.3, we have $\sum_{i=1}^{n} i = n(n+1)/2$. Thus, phase two takes $O(n^2)$ time, as does the entire selection-sort algorithm.

Insertion-Sort

If we implement the priority queue P using a sorted list, then we improve the running time of the second phase to $O(n)$, because each operation min and removeMin on P now takes $O(1)$ time. Unfortunately, the first phase now becomes the bottleneck for the running time, since, in the worst case, each insert operation takes time proportional to the size of P. This sorting algorithm is therefore better known as ***insertion-sort*** (see Figure 8.2), for the bottleneck in this sorting algorithm involves the repeated "insertion" of a new element at the appropriate position in a sorted list.

		List L	***Priority Queue*** P
Input		$(7,4,8,2,5,3,9)$	$()$
Phase 1	(a)	$(4,8,2,5,3,9)$	(7)
	(b)	$(8,2,5,3,9)$	$(4,7)$
	(c)	$(2,5,3,9)$	$(4,7,8)$
	(d)	$(5,3,9)$	$(2,4,7,8)$
	(e)	$(3,9)$	$(2,4,5,7,8)$
	(f)	(9)	$(2,3,4,5,7,8)$
	(g)	$()$	$(2,3,4,5,7,8,9)$
Phase 2	(a)	(2)	$(3,4,5,7,8,9)$
	(b)	$(2,3)$	$(4,5,7,8,9)$
	⋮	⋮	⋮
	(g)	$(2,3,4,5,7,8,9)$	$()$

Figure 8.2: Execution of insertion-sort on list $L = (7,4,8,2,5,3,9)$. In Phase 1, we repeatedly remove the first element of L and insert it into P, by scanning the list implementing P until we find the correct place for this element. In Phase 2, we repeatedly perform removeMin operations on P, each of which returns the first element of the list implementing P, and we add the element at the end of L.

Analyzing the running time of Phase 1 of insertion-sort, we note that

$$O(1+2+\ldots+(n-1)+n) = O\left(\textstyle\sum_{i=1}^{n} i\right).$$

Again, by recalling Proposition 4.3, the first phase runs in $O(n^2)$ time; hence, so does the entire algorithm.

Alternately, we could change our definition of insertion-sort so that we insert elements starting from the end of the priority-queue sequence in the first phase, in which case performing insertion-sort on a list that is already sorted would run in $O(n)$ time. Indeed, the running time of insertion-sort is $O(n+I)$ in this case, where I is the number of ***inversions*** in the input list, that is, the number of pairs of elements that start out in the input list in the wrong relative order.

8.3 Heaps

The two implementations of the PriorityQueueSort scheme presented in the previous section suggest a possible way of improving the running time for priority-queue sorting. One algorithm (selection-sort) achieves a fast running time for the first phase, but has a slow second phase, whereas the other algorithm (insertion-sort) has a slow first phase, but achieves a fast running time for the second phase. If we could somehow balance the running times of the two phases, we might be able to significantly speed up the overall running time for sorting. This approach is, in fact, exactly what we can achieve using the priority-queue implementation discussed in this section.

An efficient realization of a priority queue uses a data structure called a ***heap***. This data structure allows us to perform both insertions and removals in logarithmic time, which is a significant improvement over the list-based implementations discussed in Section 8.2. The fundamental way the heap achieves this improvement is to abandon the idea of storing elements and keys in a list and take the approach of storing elements and keys in a binary tree instead.

8.3.1 The Heap Data Structure

A heap (see Figure 8.3) is a binary tree T that stores a collection of elements with their associated keys at its nodes and that satisfies two additional properties: a relational property, defined in terms of the way keys are stored in T, and a structural property, defined in terms of the nodes of T itself. We assume that a total order relation on the keys is given, for example, by a comparator.

The relational property of T, defined in terms of the way keys are stored, is the following:

Heap-Order Property: In a heap T, for every node v other than the root, the key associated with v is greater than or equal to the key associated with v's parent.

As a consequence of the heap-order property, the keys encountered on a path from the root to an external node of T are in nondecreasing order. Also, a minimum key is always stored at the root of T. This is the most important key and is informally said to be "at the top of the heap," hence, the name "heap" for the data structure. By the way, the heap data structure defined here has nothing to do with the free-store memory heap (Section 14.1.1) used in the run-time environment supporting programming languages like C++.

You might wonder why heaps are defined with the smallest key at the top, rather than the largest. The distinction is arbitrary. (This is evidenced by the fact that the STL priority queue does exactly the opposite.) Recall that a comparator

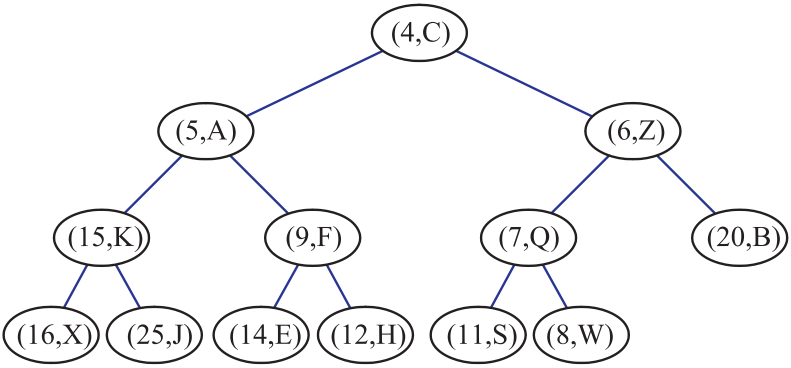


Figure 8.3: Example of a heap storing 13 elements. Each element is a key-value pair of the form (k,v). The heap is ordered based on the key value, k, of each element.

Caution

implements the less-than operator between two keys. Suppose that we had instead defined our comparator to indicate the ***opposite*** of the standard total order relation between keys (so that, for example, isLess(x,y) would return true if x were ***greater than*** y). Then the root of the resulting heap would store the largest key. This versatility comes essentially for free from our use of the comparator pattern. By defining the minimum key in terms of the comparator, the "minimum" key with a "reverse" comparator is in fact the largest. Thus, without loss of generality, we assume that we are always interested in the minimum key, which is always at the root of the heap.

For the sake of efficiency, which becomes clear later, we want the heap T to have as small a height as possible. We enforce this desire by insisting that the heap T satisfy an additional structural property, it must be ***complete***. Before we define this structural property, we need some definitions. We recall from Section 7.3.3 that level i of a binary tree T is the set of nodes of T that have depth i. Given nodes v and w on the same level of T, we say that v is ***to the left of*** w if v is encountered before w in an inorder traversal of T. That is, there is a node u of T such that v is in the left subtree of u and w is in the right subtree of u. For example, in the binary tree of Figure 8.3, the node storing entry $(15,K)$ is to the left of the node storing entry $(7,Q)$. In a standard drawing of a binary tree, the "to the left of" relation is visualized by the relative horizontal placement of the nodes.

Complete Binary Tree Property: A heap T with height h is a ***complete*** binary tree, that is, levels $0,1,2,\ldots,h-1$ of T have the maximum number of nodes possible (namely, level i has 2^i nodes, for $0 \leq i \leq h-1$) and the nodes at level h fill this level from left to right.

The Height of a Heap

Let h denote the height of T. Another way of defining the last node of T is that it is the node on level h such that all the other nodes of level h are to the left of it. Insisting that T be complete also has an important consequence as shown in Proposition 8.5.

Proposition 8.5: *A heap T storing n entries has height*

$$h = \lfloor \log n \rfloor.$$

Justification: From the fact that T is complete, we know that there are 2^i nodes in level, i for $0 \le i \le h-1$, and level h has at least 1 node. Thus, the number of nodes of T is at least

$$\begin{aligned}(1+2+4+\cdots+2^{h-1})+1 &= (2^h-1)+1 \\ &= 2^h.\end{aligned}$$

Level h has at most 2^h nodes, and thus the number of nodes of T is at most

$$(1+2+4+\cdots+2^{h-1})+2^h = 2^{h+1}-1.$$

Since the number of nodes is equal to the number n of entries, we obtain

$$2^h \le n$$

and

$$n \le 2^{h+1}-1.$$

Thus, by taking logarithms of both sides of these two inequalities, we see that

$$h \le \log n$$

and

$$\log(n+1)-1 \le h.$$

Since h is an integer, the two inequalities above imply that

$$h = \lfloor \log n \rfloor.$$

■

Proposition 8.5 has an important consequence. It implies that if we can perform update operations on a heap in time proportional to its height, then those operations will run in logarithmic time. Therefore, let us turn to the problem of how to efficiently perform various priority queue functions using a heap.

8.3.2 Complete Binary Trees and Their Representation

Let us discuss more about complete binary trees and how they are represented.

The Complete Binary Tree ADT

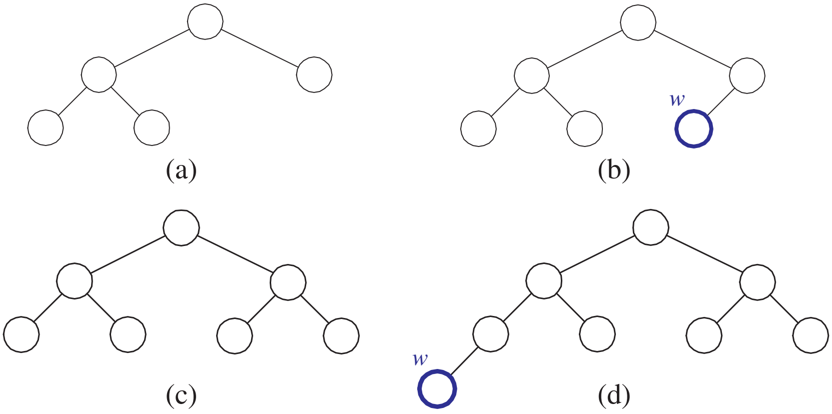
As an abstract data type, a complete binary tree T supports all the functions of the binary tree ADT (Section 7.3.1), plus the following two functions:

add(e): Add to T and return a new external node v storing element e, such that the resulting tree is a complete binary tree with last node v.

remove(): Remove the last node of T and return its element.

By using only these update operations, the resulting tree is guaranteed to be a complete binary. As shown in Figure 8.4, there are essentially two cases for the effect of an add (and remove is similar).

- If the bottom level of T is not full, then add inserts a new node on the bottom level of T, immediately after the rightmost node of this level (that is, the last node); hence, T's height remains the same.
- If the bottom level is full, then add inserts a new node as the left child of the leftmost node of the bottom level of T; hence, T's height increases by one.



(a) (b)

(c) (d)

Figure 8.4: Examples of operations add and remove on a complete binary tree, where w denotes the node inserted by add or deleted by remove. The trees shown in (b) and (d) are the results of performing add operations on the trees in (a) and (c), respectively. Likewise, the trees shown in (a) and (c) are the results of performing remove operations on the trees in (b) and (d), respectively.

A Vector Representation of a Complete Binary Tree

The vector-based binary tree representation (recall Section 7.3.5) is especially suitable for a complete binary tree T. We recall that in this implementation, the nodes of T are stored in a vector A such that node v in T is the element of A with index equal to the level number $f(v)$ defined as follows:

- If v is the root of T, then $f(v) = 1$
- If v is the left child of node u, then $f(v) = 2f(u)$
- If v is the right child of node u, then $f(v) = 2f(u) + 1$

much better

With this implementation, the nodes of T have contiguous indices in the range $[1,n]$ and the last node of T is always at index n, where n is the number of nodes of T. Figure 8.5 shows two examples illustrating this property of the last node.

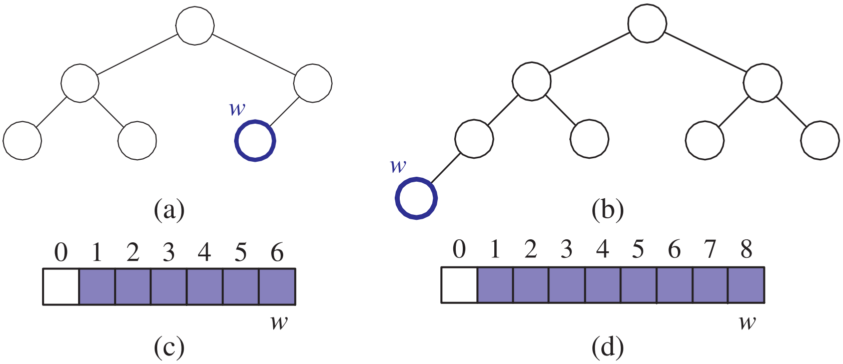


Figure 8.5: Two examples showing that the last node w of a heap with n nodes has level number n: (a) heap T_1 with more than one node on the bottom level; (b) heap T_2 with one node on the bottom level; (c) vector-based representation of T_1; (d) vector-based representation of T_2.

The simplifications that come from representing a complete binary tree T with a vector aid in the implementation of functions add and remove. Assuming that no array expansion is necessary, functions add and remove can be performed in $O(1)$ time because they simply involve adding or removing the last element of the vector. Moreover, the vector associated with T has $n+1$ elements (the element at index 0 is a placeholder). If we use an extendable array that grows and shrinks for the implementation of the vector (for example, the STL vector class), the space used by the vector-based representation of a complete binary tree with n nodes is $O(n)$ and operations add and remove take $O(1)$ amortized time.

A C++ Implementation of a Complete Binary Tree

We present the complete binary tree ADT as an informal interface, called CompleteTree, in Code Fragment 8.11. As with our other informal interfaces, this is not a complete C++ class. It just gives the public portion of the class.

The interface defines a nested class, called Position, which represents a node of the tree. We provide the necessary functions to access the root and last positions and to navigate through the tree. The modifier functions add and remove are provided, along with a function swap, which swaps the contents of two given nodes.

```
template <typename E>
class CompleteTree {                                   // left-complete tree interface
public:                                                // publicly accessible types
  class Position;                                      // node position type
  int size() const;                                    // number of elements
  Position left(const Position& p);                    // get left child
  Position right(const Position& p);                   // get right child
  Position parent(const Position& p);                  // get parent
  bool hasLeft(const Position& p) const;               // does node have left child?
  bool hasRight(const Position& p) const;              // does node have right child?
  bool isRoot(const Position& p) const;                // is this the root?
  Position root();                                     // get root position
  Position last();                                     // get last node
  void addLast(const E& e);                            // add a new last node
  void removeLast();                                   // remove the last node
  void swap(const Position& p, const Position& q);     // swap node contents
};
```

Code Fragment 8.11: Interface CompleteBinaryTree for a complete binary tree.

In order to implement this interface, we store the elements in an STL vector, called V. We implement a tree position as an iterator to this vector. To convert from the index representation of a node to this positional representation, we provide a function pos. The reverse conversion is provided by function idx. This portion of the class definition is given in Code Fragment 8.12.

```
private:                                               // member data
  std::vector<E> V;                                    // tree contents
public:                                                // publicly accessible types
  typedef typename std::vector<E>::iterator Position;  // a position in the tree
protected:                                             // protected utility functions
  Position pos(int i)                                  // map an index to a position
    { return V.begin() + i; }
  int idx(const Position& p) const                     // map a position to an index
    { return p − V.begin(); }
```

Code Fragment 8.12: Member data and private utilities for a complete tree class.

Given the index of a node i, the function pos maps it to a position by adding i to V.begin(). Here we are exploiting the fact that the STL vector supports a ***random-access iterator*** (recall Section 6.2.5). In particular, given an integer i, the expression V.begin() $+ i$ yields the position of the ith element of the vector, and, given a position p, the expression $p - V$.begin() yields the index of position p.

We present a full implementation of a vector-based complete tree ADT in Code Fragment 8.13. Because the class consists of a large number of small one-line functions, we have chosen to violate our normal coding conventions by placing all the function definitions inside the class definition.

```
template <typename E>
class VectorCompleteTree {
  //... insert private member data and protected utilities here
public:
  VectorCompleteTree() : V(1) {}                 // constructor
  int size() const                               { return V.size() − 1; }
  Position left(const Position& p)               { return pos(2*idx(p)); }
  Position right(const Position& p)              { return pos(2*idx(p) + 1); }
  Position parent(const Position& p)             { return pos(idx(p)/2); }
  bool hasLeft(const Position& p) const          { return 2*idx(p) <= size(); }
  bool hasRight(const Position& p) const         { return 2*idx(p) + 1 <= size(); }
  bool isRoot(const Position& p) const           { return idx(p) == 1; }
  Position root()                                { return pos(1); }
  Position last()                                { return pos(size()); }
  void addLast(const E& e)                       { V.push_back(e); }
  void removeLast()                              { V.pop_back(); }
  void swap(const Position& p, const Position& q)
                                                 { E e = *q; *q = *p; *p = e; }
};
```

Code Fragment 8.13: A vector-based implementation of the complete tree ADT.

Recall from Section 7.3.5 that the root node is at index 1 of the vector. Since STL vectors are indexed starting at 0, our constructor creates the initial vector with one element. This element at index 0 is never used. As a consequence, the size of the priority queue is one less than the size of the vector.

Recall from Section 7.3.5 that, given a node at index i, its left and right children are located at indices $2i$ and $2i+1$, respectively. Its parent is located at index $\lfloor i/2 \rfloor$. Given a position p, the functions left, right, and parent first convert p to an index using the utility idx, which is followed by the appropriate arithmetic operation on this index, and finally they convert the index back to a position using the utility pos.

We determine whether a node has a child by evaluating the index of this child and testing whether the node at that index exists in the vector. Operations add and remove are implemented by adding or removing the last entry of the vector, respectively.

8.3.3 Implementing a Priority Queue with a Heap

We now discuss how to implement a priority queue using a heap. Our heap-based representation for a priority queue P consists of the following (see Figure 8.6):

- ***heap***: A complete binary tree T whose nodes store the elements of the queue and whose keys satisfy the heap-order property. We assume the binary tree T is implemented using a vector, as described in Section 8.3.2. For each node v of T, we denote the associated key by $k(v)$.
- ***comp***: A comparator that defines the total order relation among the keys.

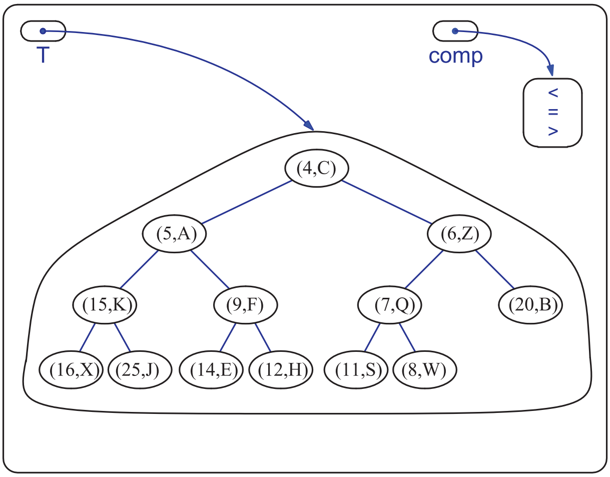


Figure 8.6: Illustration of the heap-based implementation of a priority queue.

With this data structure, functions size and empty take $O(1)$ time, as usual. In addition, function min can also be easily performed in $O(1)$ time by accessing the entry stored at the root of the heap (which is at index 1 in the vector).

Insertion

Let us consider how to perform insert on a priority queue implemented with a heap T. To store a new element e in T, we add a new node z to T with operation add, so that this new node becomes the last node of T, and then store e in this node.

After this action, the tree T is complete, but it may violate the heap-order property. Hence, unless node z is the root of T (that is, the priority queue was empty before the insertion), we compare key $k(z)$ with the key $k(u)$ stored at the parent u of z. If $k(z) \geq k(u)$, the heap-order property is satisfied and the algorithm terminates. If instead $k(z) < k(u)$, then we need to restore the heap-order property, which can be locally achieved by swapping the entries stored at z and u. (See Figures 8.7(c) and (d).) This swap causes the new entry (k,e) to move up one level. Again, the heap-order property may be violated, and we continue swapping, going

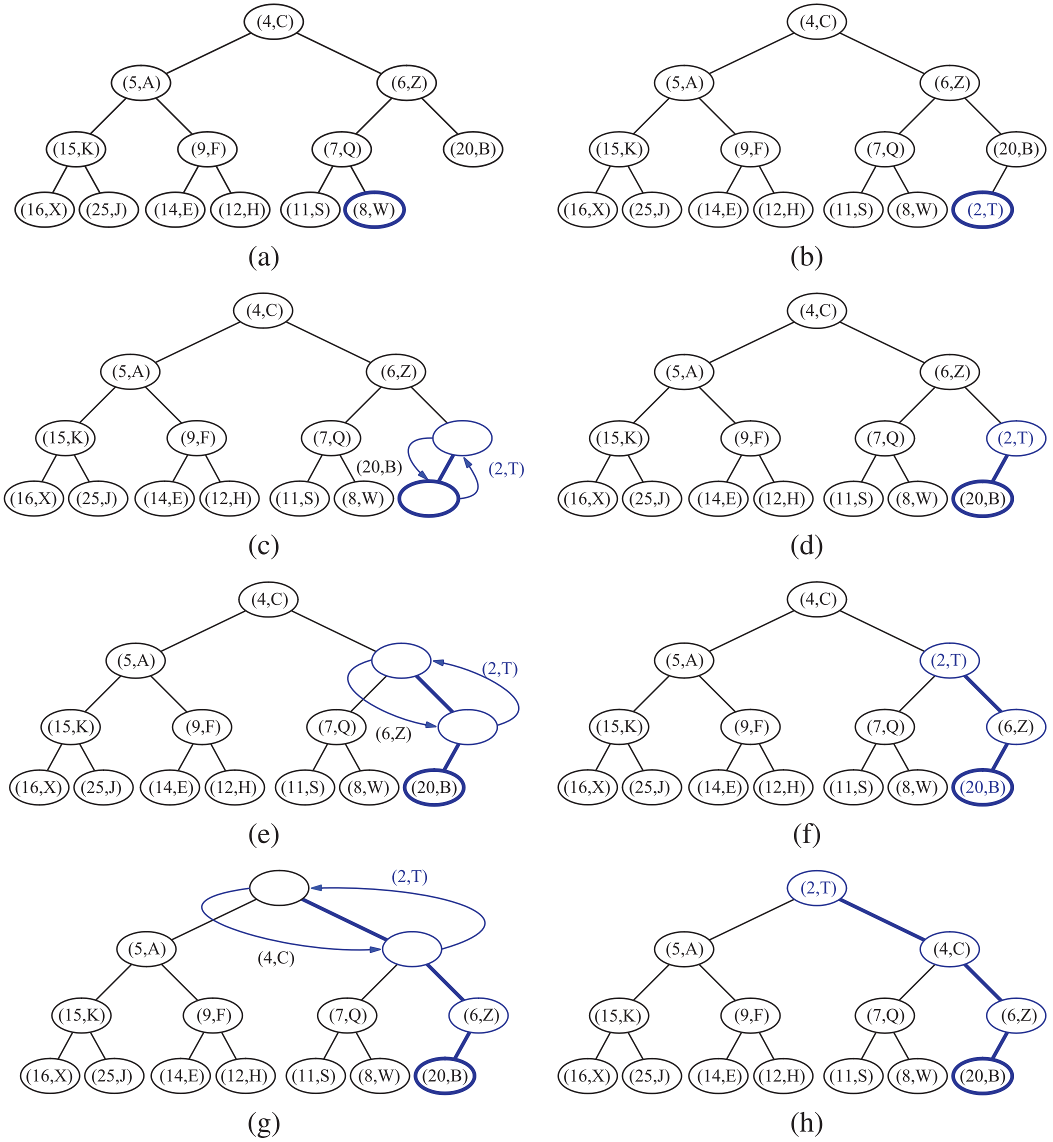


Figure 8.7: Insertion of a new entry with key 2 into the heap of Figure8.6: (a) initial heap; (b) after performing operation add; (c) and (d) swap to locally restore the partial order property; (e) and (f) another swap; (g) and (h)final swap.

up in T until no violation of the heap-order property occurs. (See Figures8.7(e) and (h).)

The upward movement of the newly inserted entry by means of swaps is conventionally called ***up-heap bubbling***. A swap either resolves the violation of the heap-order property or propagates it one level up in the heap. In the worst case, up-heap bubbling causes the new entry to move all the way up to the root of heap T. (See Figure 8.7.) Thus, in the worst case, the number of swaps performed in the execution of function insert is equal to the height of T, that is, it is $\lfloor \log n \rfloor$ by Proposition 8.5.

Removal

Let us now turn to function removeMin of the priority queue ADT. The algorithm for performing function removeMin using heap T is illustrated in Figure 8.8.

We know that an element with the smallest key is stored at the root r of T (even if there is more than one entry with the smallest key). However, unless r is the only node of T, we cannot simply delete node r, because this action would disrupt the binary tree structure. Instead, we access the last node w of T, copy its entry to the root r, and then delete the last node by performing operation remove of the complete binary tree ADT. (See Figure 8.8(a) and (b).)

Down-Heap Bubbling after a Removal

We are not necessarily done, however, for, even though T is now complete, T may now violate the heap-order property. If T has only one node (the root), then the heap-order property is trivially satisfied and the algorithm terminates. Otherwise, we distinguish two cases, where r denotes the root of T:

- If r has no right child, let s be the left child of r
- Otherwise (r has both children), let s be a child of r with the smaller key

If $k(r) \leq k(s)$, the heap-order property is satisfied and the algorithm terminates. If instead $k(r) > k(s)$, then we need to restore the heap-order property, which can be locally achieved by swapping the entries stored at r and s. (See Figure 8.8(c) and (d).) (Note that we shouldn't swap r with s's sibling.) The swap we perform restores the heap-order property for node r and its children, but it may violate this property at s; hence, we may have to continue swapping down T until no violation of the heap-order property occurs. (See Figure 8.8(e) and (h).)

This downward swapping process is called ***down-heap bubbling***. A swap either resolves the violation of the heap-order property or propagates it one level down in the heap. In the worst case, an entry moves all the way down to the bottom level. (See Figure 8.8.) Thus, the number of swaps performed in the execution of function removeMin is, in the worst case, equal to the height of heap T, that is, it is $\lfloor \log n \rfloor$ by Proposition 8.5

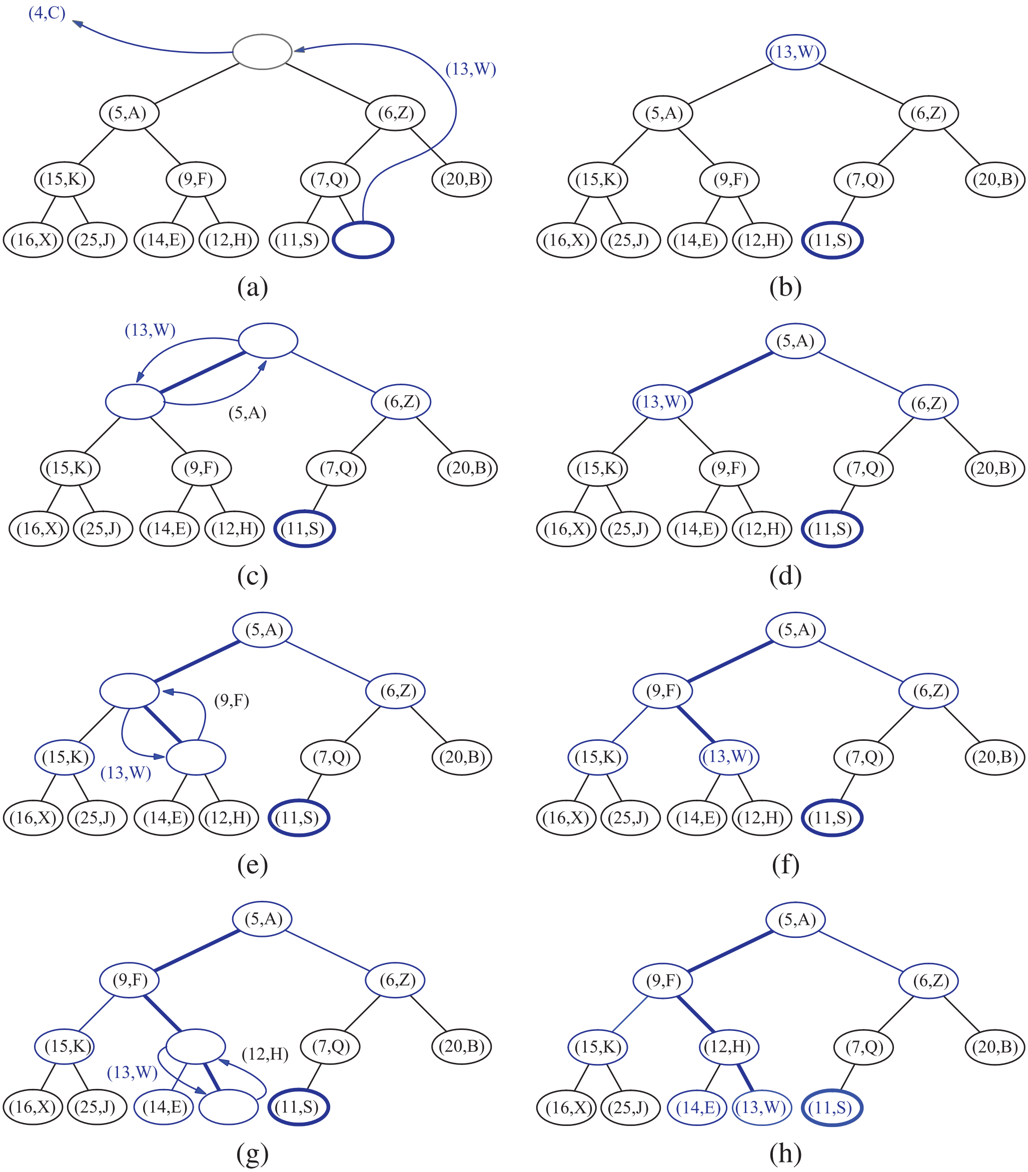


Figure 8.8: Removing the element with the smallest key from a heap: (a) and (b) deletion of the last node, whose element is moved to the root; (c) and (d) swap to locally restore the heap-order property; (e) and (f) another swap; (g) and (h) final swap.

Analysis

Table 8.2 shows the running time of the priority queue ADT functions for the heap implementation of a priority queue, assuming that two keys can be compared in $O(1)$ time and that the heap T is implemented with either a vector or linked structure.

Operation	***Time***
size, empty	$O(1)$
min	$O(1)$
insert	$O(\log n)$
removeMin	$O(\log n)$

Table 8.2: Performance of a priority queue realized by means of a heap, which is in turn implemented with a vector or linked structure. We denote with n the number of entries in the priority queue at the time a method is executed. The space requirement is $O(n)$. The running time of operations insert and removeMin is worst case for the array-list implementation of the heap and amortized for the linked representation.

In short, each of the priority queue ADT functions can be performed in $O(1)$ time or in $O(\log n)$ time, where n is the number of elements at the time the function is executed. This analysis is based on the following:

- The heap T has n nodes, each storing a reference to an entry
- Operations add and remove on T take either $O(1)$ amortized time (vector representation) or $O(\log n)$ worst-case time
- In the worst case, up-heap and down-heap bubbling perform a number of swaps equal to the height of T
- The height of heap T is $O(\log n)$, since T is complete (Proposition 8.5)

Thus, if heap T is implemented with the linked structure for binary trees, the space needed is $O(n)$. If we use a vector-based implementation for T instead, then the space is proportional to the size N of the array used for the vector representing T.

We conclude that the heap data structure is a very efficient realization of the priority queue ADT, independent of whether the heap is implemented with a linked structure or a vector. The heap-based implementation achieves fast running times for both insertion and removal, unlike the list-based priority queue implementations. Indeed, an important consequence of the efficiency of the heap-based implementation is that it can speed up priority-queue sorting to be much faster than the list-based insertion-sort and selection-sort algorithms.

8.3.4 C++ Implementation

In this section, we present a heap-based priority queue implementation. The heap is implemented using the vector-based complete tree implementation, which we presented in Section 8.3.2.

In Code Fragment 8.7, we present the class definition. The public part of the class is essentially the same as the interface, but, in order to keep the code simple, we have ignored error checking. The class's data members consists of the complete tree, named T, and an instance of the comparator object, named *isLess*. We have also provided a type definition for a node position in the tree, called Position.

```
template <typename E, typename C>
class HeapPriorityQueue {
public:
  int size() const;                          // number of elements
  bool empty() const;                        // is the queue empty?
  void insert(const E& e);                   // insert element
  const E& min();                            // minimum element
  void removeMin();                          // remove minimum
private:
  VectorCompleteTree<E> T;                   // priority queue contents
  C isLess;                                  // less-than comparator
                                             // shortcut for tree position
  typedef typename VectorCompleteTree<E>::Position Position;
};
```

Code Fragment 8.14: A heap-based implementation of a priority queue.

In Code Fragment 8.15, we present implementations of the simple member functions size, empty, and min. The function min returns a reference to the root's element through the use of the "*" operator, which is provided by the Position class of VectorCompleteTree.

```
template <typename E, typename C>            // number of elements
int HeapPriorityQueue<E,C>::size() const
  { return T.size(); }

template <typename E, typename C>            // is the queue empty?
bool HeapPriorityQueue<E,C>::empty() const
  { return size() == 0; }

template <typename E, typename C>            // minimum element
const E& HeapPriorityQueue<E,C>::min()
  { return *(T.root()); }                    // return reference to root element
```

Code Fragment 8.15: The member functions size, empty, and min.

Next, in Code Fragment 8.16, we present an implementation of the insert operation. As outlined in the previous section, this works by adding the new element to the last position of the tree and then it performs up-heap bubbling by repeatedly swapping this element with its parent until its parent has a smaller key value.

```
template <typename E, typename C>         // insert element
void HeapPriorityQueue<E,C>::insert(const E& e) {
  T.addLast(e);                           // add e to heap
  Position v = T.last();                  // e's position
  while (!T.isRoot(v)) {                  // up-heap bubbling
    Position u = T.parent(v);
    if (!isLess(*v, *u)) break;           // if v in order, we're done
    T.swap(v, u);                         // ...else swap with parent
    v = u;
  }
}
```

Code Fragment 8.16: An implementation of the function insert.

Finally, let us consider the removeMin operation. If the tree has only one node, then we simply remove it. Otherwise, we swap the root's element with the last element of the tree and remove the last element. We then apply down-heap bubbling to the root. Letting *u* denote the current node, this involves determining *u*'s smaller child, which is stored in *v*. If the child's key is smaller than *u*'s, we swap *u*'s contents with this child's. The code is presented in Code Fragment 8.17.

```
template <typename E, typename C>         // remove minimum
void HeapPriorityQueue<E,C>::removeMin() {
  if (size() == 1)                        // only one node?
    T.removeLast();                       // ...remove it
  else {
    Position u = T.root();                // root position
    T.swap(u, T.last());                  // swap last with root
    T.removeLast();                       // ...and remove last
    while (T.hasLeft(u)) {                // down-heap bubbling
      Position v = T.left(u);
      if (T.hasRight(u) && isLess(*(T.right(u)), *v))
        v = T.right(u);                   // v is u's smaller child
      if (isLess(*v, *u)) {               // is u out of order?
        T.swap(u, v);                     // ...then swap
        u = v;
      }
      else break;                         // else we're done
    }
  }
}
```

Code Fragment 8.17: A heap-based implementation of a priority queue.

8.3.5 Heap-Sort

As we have previously observed, realizing a priority queue with a heap has the advantage that all the functions in the priority queue ADT run in logarithmic time or better. Hence, this realization is suitable for applications where fast running times are sought for all the priority queue functions. Therefore, let us again consider the PriorityQueueSort sorting scheme from Section 8.1.5, which uses a priority queue P to sort a list L.

During Phase 1, the i-th insert operation ($1 \leq i \leq n$) takes $O(1+\log i)$ time, since the heap has i entries after the operation is performed. Likewise, during Phase 2, the j-th removeMin operation ($1 \leq j \leq n$) runs in time $O(1+\log(n-j+1)$, since the heap has $n-j+1$ entries at the time the operation is performed. Thus, each phase takes $O(n\log n)$ time, so the entire priority-queue sorting algorithm runs in $O(n\log n)$ time when we use a heap to implement the priority queue. This sorting algorithm is better known as ***heap-sort***, and its performance is summarized in the following proposition.

Proposition 8.6: *The heap-sort algorithm sorts a list L of n elements in $O(n\log n)$ time, assuming two elements of L can be compared in $O(1)$ time.*

Let us stress that the $O(n\log n)$ running time of heap-sort is considerably better than the $O(n^2)$ running time of selection-sort and insertion-sort (Section 8.2.2) and is essentially the best possible for any sorting algorithm.

Implementing Heap-Sort In-Place

If the list L to be sorted is implemented by means of an array, we can speed up heap-sort and reduce its space requirement by a constant factor using a portion of the list L itself to store the heap, thus avoiding the use of an external heap data structure. This performance is accomplished by modifying the algorithm as follows:

1. We use a reverse comparator, which corresponds to a heap where the largest element is at the top. At any time during the execution of the algorithm, we use the left portion of L, up to a certain rank $i-1$, to store the elements in the heap, and the right portion of L, from rank i to $n-1$ to store the elements in the list. Thus, the first i elements of L (at ranks $0,\ldots,i-1$) provide the vector representation of the heap (with modified level numbers starting at 0 instead of 1), that is, the element at rank k is greater than or equal to its "children" at ranks $2k+1$ and $2k+2$.
2. In the first phase of the algorithm, we start with an empty heap and move the boundary between the heap and the list from left to right, one step at a time. In step i ($i = 1,\ldots,n$), we expand the heap by adding the element at rank $i-1$ and perform up-heap bubbling.

3. In the second phase of the algorithm, we start with an empty list and move the boundary between the heap and the list from right to left, one step at a time. At step i ($i = 1,\ldots,n$), we remove a maximum element from the heap and store it at rank $n - i$.

The above variation of heap-sort is said to be ***in-place***, since we use only a constant amount of space in addition to the list itself. Instead of transferring elements out of the list and then back in, we simply rearrange them. We illustrate in-place heap-sort in Figure 8.9. In general, we say that a sorting algorithm is in-place if it uses only a constant amount of memory in addition to the memory needed for the objects being sorted themselves. A sorting algorithm is considered space-efficient if it can be implemented in-place.

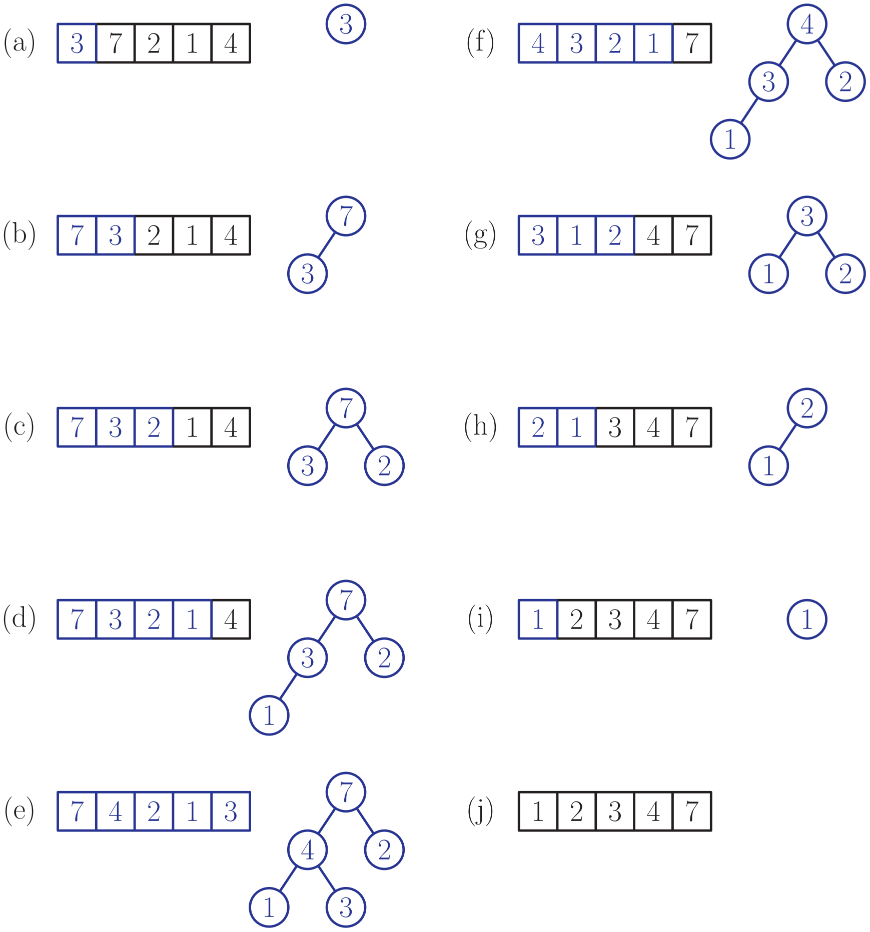


Figure 8.9: In-place heap-sort. Parts (a) through (e) show the addition of elements to the heap; (f) through (j) show the removal of successive elements. The portions of the array that are used for the heap structure are shown in blue.

8.3.6 Bottom-Up Heap Construction ⋆

The analysis of the heap-sort algorithm shows that we can construct a heap storing n elements in $O(n \log n)$ time, by means of n successive insert operations, and then use that heap to extract the elements in order. However, if all the elements to be stored in the heap are given in advance, there is an alternative ***bottom-up*** construction function that runs in $O(n)$ time. We describe this function in this section, observing that it can be included as one of the constructors in a Heap class instead of filling a heap using a series of n insert operations. For simplicity, we describe this bottom-up heap construction assuming the number n of keys is an integer of the type $n = 2^h - 1$. That is, the heap is a complete binary tree with every level being full, so the heap has height $h = \log(n+1)$. Viewed nonrecursively, bottom-up heap construction consists of the following $h = \log(n+1)$ steps:

1. In the first step (see Figure 8.10(a)), we construct $(n+1)/2$ elementary heaps storing one entry each.
2. In the second step (see Figure 8.10(b)–(c)), we form $(n+1)/4$ heaps, each storing three entries, by joining pairs of elementary heaps and adding a new entry. The new entry is placed at the root and may have to be swapped with the entry stored at a child to preserve the heap-order property.
3. In the third step (see Figure 8.10(d)–(e)), we form $(n+1)/8$ heaps, each storing 7 entries, by joining pairs of 3-entry heaps (constructed in the previous step) and adding a new entry. The new entry is placed initially at the root, but may have to move down with a down-heap bubbling to preserve the heap-order property.

$\vdots$

i. In the generic ith step, $2 \leq i \leq h$, we form $(n+1)/2^i$ heaps, each storing $2^i - 1$ entries, by joining pairs of heaps storing $(2^{i-1} - 1)$ entries (constructed in the previous step) and adding a new entry. The new entry is placed initially at the root, but may have to move down with a down-heap bubbling to preserve the heap-order property.

$\vdots$

$h+1$. In the last step (see Figure 8.10(f)–(g)), we form the final heap, storing all the n entries, by joining two heaps storing $(n-1)/2$ entries (constructed in the previous step) and adding a new entry. The new entry is placed initially at the root, but may have to move down with a down-heap bubbling to preserve the heap-order property.

We illustrate bottom-up heap construction in Figure 8.10 for $h = 3$.

$n = 2^h - 1$ $T(h) = 2^h - h - 2$ $a = -1$ $b = -2$

wt $T(h) = T(h-1) + T(h-1) + h$

$T(h) = n$

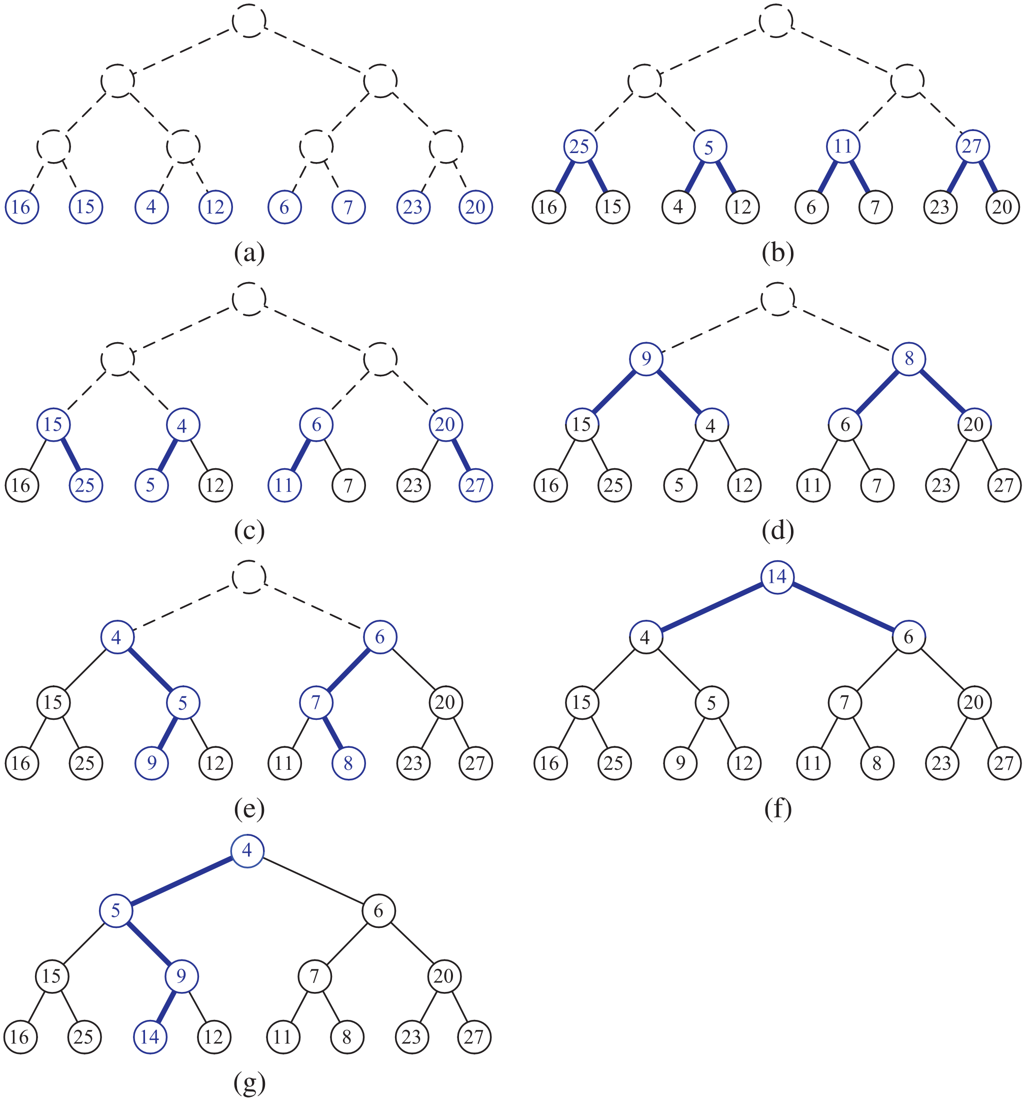


Figure 8.10: Bottom-up construction of a heap with 15 entries: (a) we begin by constructing one-entry heaps on the bottom level; (b) and (c) we combine these heaps into three-entry heaps; (d) and (e) seven-entry heaps; (f) and (g) we create the final heap. The paths of the down-heap bubblings are highlighted in blue. For simplicity, we only show the key within each node instead of the entire entry.

Recursive Bottom-Up Heap Construction

We can also describe bottom-up heap construction as a recursive algorithm, as shown in Code Fragment 8.18, which we call by passing a list storing the keys for which we wish to build a heap.

```
Algorithm BottomUpHeap($L$):
  Input: An STL list $L$ storing $n = 2^{h+1} - 1$ entries
  Output: A heap $T$ storing the entries of $L$.
   if $L$.empty() then
     return an empty heap
   $e \leftarrow L$.front()
   $L$.pop_front()
   Split $L$ into two lists, $L_1$ and $L_2$, each of size $(n-1)/2$
   $T_1 \leftarrow$ BottomUpHeap($L_1$)
   $T_2 \leftarrow$ BottomUpHeap($L_2$)
   Create binary tree $T$ with root $r$ storing $e$, left subtree $T_1$, and right subtree $T_2$
   Perform a down-heap bubbling from the root $r$ of $T$, if necessary
   return $T$
```

Code Fragment 8.18: Recursive bottom-up heap construction.

Although the algorithm has been expressed in terms of an STL list, the construction could have been performed equally well with a vector. In such a case, the splitting of the vector is performed conceptually, by defining two ranges of indices, one representing the front half L_1 and the other representing the back half L_2.

At first glance, it may seem that there is no substantial difference between this algorithm and the incremental heap construction used in the heap-sort algorithm of Section 8.3.5. One works by down-heap bubbling and the other uses up-heap bubbling. It is somewhat surprising, therefore, that the bottom-up heap construction is actually asymptotically faster than incrementally inserting n keys into an initially empty heap. The following proposition shows this.

Proposition 8.7: *Bottom-up construction of a heap with n entries takes $O(n)$ time, assuming two keys can be compared in $O(1)$ time.*

Justification: We analyze bottom-up heap construction using a "visual" approach, which is illustrated in Figure 8.11.

Let T be the final heap, let v be a node of T, and let $T(v)$ denote the subtree of T rooted at v. In the worst case, the time for forming $T(v)$ from the two recursively formed subtrees rooted at v's children is proportional to the height of $T(v)$. The worst case occurs when down-heap bubbling from v traverses a path from v all the way to a bottommost node of $T(v)$.

Now consider the path $p(v)$ of T from node v to its inorder successor external node, that is, the path that starts at v, goes to the right child of v, and then goes down leftward until it reaches an external node. We say that path $p(v)$ is ***associated with*** node v. Note that $p(v)$ is not necessarily the path followed by down-heap bubbling when forming $T(v)$. Clearly, the size (number of nodes) of $p(v)$ is equal to the height of $T(v)$ plus one. Hence, forming $T(v)$ takes time proportional to the size of $p(v)$, in the worst case. Thus, the total running time of bottom-up heap construction is proportional to the sum of the sizes of the paths associated with the nodes of T.

Observe that each node v of T belongs to at most two such paths: the path $p(v)$ associated with v itself and possibly also the path $p(u)$ associated with the closest ancestor u of v preceding v in an inorder traversal. (See Figure 8.11.) In particular, the root r of T and the nodes on the leftmost root-to-leaf path each belong only to one path, the one associated with the node itself. Therefore, the sum of the sizes of the paths associated with the internal nodes of T is at most $2n-1$. We conclude that the bottom-up construction of heap T takes $O(n)$ time. ■

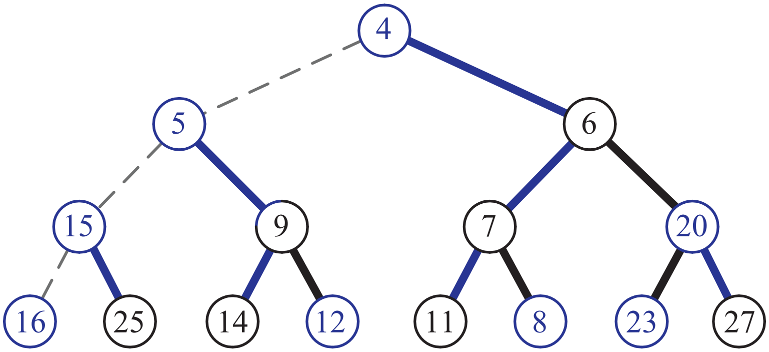


Figure 8.11: Visual justification of the linear running time of bottom-up heap construction, where the paths associated with the internal nodes have been highlighted with alternating colors. For example, the path associated with the root consists of the nodes storing keys 4, 6, 7, and 11. Also, the path associated with the right child of the root consists of the internal nodes storing keys 6, 20, and 23.

To summarize, Proposition 8.7 states that the running time for the first phase of heap-sort can be reduced to be $O(n)$. Unfortunately, the running time of the second phase of heap-sort cannot be made asymptotically better than $O(n\log n)$ (that is, it will always be $\Omega(n\log n)$ in the worst case). We do not justify this lower bound until Chapter 11, however. Instead, we conclude this chapter by discussing a design pattern that allows us to extend the priority queue ADT to have additional functionality.

8.4 Adaptable Priority Queues

The functions of the priority queue ADT given in Section 8.1.3 are sufficient for most basic applications of priority queues such as sorting. However, there are situations where additional functions would be useful as shown in the scenarios below that refer to the standby airline passenger application.

- A standby passenger with a pessimistic attitude may become tired of waiting and decide to leave ahead of the boarding time, requesting to be removed from the waiting list. Thus, we would like to remove the entry associated with this passenger from the priority queue. Operation removeMin is not suitable for this purpose, since it only removes the entry with the lowest priority. Instead, we want a new operation that removes an arbitrary entry.
- Another standby passenger finds her gold frequent-flyer card and shows it to the agent. Thus, her priority has to be modified accordingly. To achieve this change of priority, we would like to have a new operation that changes the information associated with a given entry. This might affect the entry's key value (such as frequent-flyer status) or not (such as correcting a misspelled name).

Functions of the Adaptable Priority Queue ADT

The above scenarios motivate the definition of a new ADT for priority queues, which includes functions for modifying or removing specified entries. In order to do this, we need some way of indicating which entry of the queue is to be affected by the operation. Note that we cannot use the entry's key value, because keys are not distinct. Instead, we assume that the priority queue operation insert(e) is augmented so that, after inserting the element e, it returns a reference to the newly created entry, called a ***position*** (recall Section 6.2.1). This position is permanently attached to the entry, so that, even if the location of the entry changes within the priority queue's internal data structure (as is done when performing bubbling operations in a heap), the position remains fixed to this entry. Thus, positions provide us with a means to uniquely specify the entry to which each operation is applied.

We formally define an ***adaptable priority queue*** P to be a priority queue that, in addition to the standard priority queue operations, supports the following enhancements.

insert(e): Insert the element e into P and return a position referring to its entry.

remove(p): Remove the entry referenced by p from P.

replace(p, e): Replace with e the element associated with the entry referenced by p and return the position of the altered entry.

8.4.1 A List-Based Implementation

In this section, we present a simple implementation of an adaptable priority queue, called AdaptPriorityQueue. Our implementation is a generalization of the sorted-list priority queue implementation given in Section 8.2.

In Code Fragment 8.7, we present the class definition, with the exception of the class Position, which is presented later. The public part of the class is essentially the same as the standard priority queue interface, which was presented in Code Fragment 8.4, together with the new functions remove and replace. Note that the function insert now returns a position.

```
template <typename E, typename C>
class AdaptPriorityQueue {                        // adaptable priority queue
protected:
  typedef std::list<E> ElementList;               // list of elements
public:
  // ...insert Position class definition here
public:
  int size() const;                               // number of elements
  bool empty() const;                             // is the queue empty?
  const E& min() const;                           // minimum element
  Position insert(const E& e);                    // insert element
  void removeMin();                               // remove minimum
  void remove(const Position& p);                 // remove at position p
  Position replace(const Position& p, const E& e); // replace at position p
private:
  ElementList L;                                  // priority queue contents
  C isLess;                                       // less-than comparator
};
```

Code Fragment 8.19: The class definition for an adaptable priority queue.

We next define the class Position, which is nested within the public part of class AdaptPriorityQueue. Its data member is an iterator to the STL list. This list contains the contents of the priority queue. The main public member is a function that returns a "const" reference the underlying element, which is implemented by overloading the "*" operator. This is presented in Code Fragment 8.20.

```
  class Position {                                // a position in the queue
  private:
    typename ElementList::iterator q;             // a position in the list
  public:
    const E& operator*() { return *q; }           // the element at this position
    friend class AdaptPriorityQueue;              // grant access
  };
```

Code Fragment 8.20: The class representing a position in AdaptPriorityQueue.

The operation insert is presented in Code Fragment 8.21. It is essentially the same as presented in the standard list priority queue (see Code Fragment 8.9). Since it is declared outside the class, we need to provide the complete template specifications for the function. We search for the first entry *p* whose key value exceeds ours, and insert *e* just prior to this entry. We then create a position that refers to the entry just prior to *p* and return it.

```
template <typename E, typename C>                    // insert element
typename AdaptPriorityQueue<E,C>::Position
AdaptPriorityQueue<E,C>::insert(const E& e) {
  typename ElementList::iterator p = L.begin();
  while (p != L.end() && !isLess(e, *p)) ++p;        // find larger element
  L.insert(p, e);                                    // insert before p
  Position pos;  pos.q = −−p;
  return pos;                                        // inserted position
}
```

Code Fragment 8.21: The function insert for class AdaptPriorityQueue.

We omit the definitions of the member functions size, empty, min, and removeMin, since they are the same as in the standard list-based priority queue implementation (see Code Fragments 8.8 and 8.10). Next, in Code Fragment 8.22, we present the implementations of the functions remove and replace. The function remove invokes the erase function of the STL list to remove the entry referred to by the given position.

```
template <typename E, typename C>                    // remove at position p
void AdaptPriorityQueue<E,C>::remove(const Position& p)
  { L.erase(p.q); }

template <typename E, typename C>                    // replace at position p
typename AdaptPriorityQueue<E,C>::Position
AdaptPriorityQueue<E,C>::replace(const Position& p, const E& e) {
  L.erase(p.q);                                      // remove the old entry
  return insert(e);                                  // insert replacement
}
```

Code Fragment 8.22: The functions remove and replace for AdaptPriorityQueue.

We have chosen perhaps the simplest way to implement the function replace. We remove the entry to be modified and simply insert the new element *e* into the priority queue. In general, the key information may have changed, and therefore it may need to be moved to a new location in the sorted list. Under the assumption that key changes are rare, a more clever solution would involve searching forwards or backwards to determine the proper position for the modified entry. While it may not be very efficient, our approach has the virtue of simplicity.

8.4.2 Location-Aware Entries

In our implementation of the adaptable priority queue, AdaptPriorityQueue, presented in the previous section, we exploited a nice property of the list-based priority queue implementation. In particular, once a new entry is added to the sorted list, the element associated with this entry never changes. This means that the positions returned by the insert and replace functions always refer to the same element.

Note, however, that this same approach would fail if we tried to apply it to the heap-based priority queue of Section 8.3.3. The reason is that the heap-based implementation moves the entries around the heap (for example, through up-heap bubbling and down-heap bubbling). When an element e is inserted, we return a reference to the entry p containing e. But if e were to be moved as a result of subsequent operations applied to the priority queue, p does not change. As a result, p might be pointing to a different element of the priority queue. An attempt to apply remove(p) or replace(p,e'), would not be applied to e but instead to some other element.

The solution to this problem involves decoupling positions and entries. In our implementation of AdaptPriorityQueue, each position p is essentially a pointer to a node of the underlying data structure (for this is how an STL iterator is implemented). If we move an entry, we need to also change the associated pointer. In order to deal with moving entries, each time we insert a new element e in the priority queue, in addition to creating a new entry in the data structure, we also allocate memory for an object, called a ***locator***. The locator's job is to store the current position p of element e in the data structure. Each entry of the priority queue needs to know its associated locator l. Thus, rather than just storing the element itself in the priority queue, we store a pair $(e, \&l)$, consisting of the element e and a pointer to its locator. We call this a ***locator-aware entry***. After inserting a new element in the priority queue, we return the associated locator object, which points to this pair.

How does this solve the decoupling problem? First, observe that whenever the user of the priority queue wants to locate the position p of a previously inserted element, it suffices to access the locator that stores this position. Suppose, however, that the entry moves to a different position p' within the data structure. To handle this, we first access the location-aware entry $(e, \&l)$ to access the locator l. We then modify l so that it refers to the new position p'. The user may find the new position by accessing the locator.

The price we pay for this extra generality is fairly small. For each entry, we need to two store two additional pointers (the locator and the locator's address). Each time we move an object in the data structure, we need to modify a constant number of pointers. Therefore, the running time increases by just a constant factor.

8.5 Exercises

For help with exercises, please visit the web site, www.wiley.com/college/goodrich.

Reinforcement

R-8.1 What are the running times of each of the functions of the (standard) priority queue ADT if we implement it by adapting the STL priority_queue?

R-8.2 How long would it take to remove the $\lceil \log n \rceil$ smallest elements from a heap that contains n entries using the removeMin() operation?

R-8.3 Show that, given only the less-than operator (<) and the boolean operators *and* (&&), *or* (||), and *not* (!), it is possible to implement all of the other comparators: >, <=, >=, ==, !=.

R-8.4 Explain how to implement a priority queue based on the composition method (of storing key-element pairs) by adapting a priority queue based on the comparator approach.

R-8.5 Suppose you label each node v of a binary tree T with a key equal to the preorder rank of v. Under what circumstances is T a heap?

R-8.6 Show the output from the following sequence of priority queue ADT operations. The entries are key-element pairs, where sorting is based on the key value: insert$(5,a)$, insert$(4,b)$, insert$(7,i)$, insert$(1,d)$, removeMin(), insert$(3,j)$, insert$(6,c)$, removeMin(), removeMin(), insert$(8,g)$, removeMin(), insert$(2,h)$, removeMin(), removeMin().

R-8.7 An airport is developing a computer simulation of air-traffic control that handles events such as landings and takeoffs. Each event has a *time-stamp* that denotes the time when the event occurs. The simulation program needs to efficiently perform the following two fundamental operations:

- Insert an event with a given time-stamp (that is, add a future event)
- Extract the event with smallest time-stamp (that is, determine the next event to process)

Which data structure should be used for the above operations? Why?

R-8.8 Although it is correct to use a "reverse" comparator with our priority queue ADT so that we retrieve and remove an element with the maximum key each time, it is confusing to have an element with the maximum key returned by a function named "removeMin." Write a short adapter class that can take any priority queue P and an associated comparator C and implement a priority queue that concentrates on the element with the maximum key, using functions with names like removeMax.
(Hint: Define a new comparator C' in terms of C.)

R-8.9 Illustrate the performance of the selection-sort algorithm on the following input sequence: $(22,15,36,44,10,3,9,13,29,25)$.

R-8.10 Illustrate the performance of the insertion-sort algorithm on the input sequence of the previous problem.

R-8.11 Give an example of a worst-case sequence with n elements for insertion-sort, and show that insertion-sort runs in $\Omega(n^2)$ time on such a sequence.

R-8.12 At which nodes of a heap can an entry with the largest key be stored?

R-8.13 In defining the relation "to the left of" for two nodes of a binary tree (Section 8.3.1), can we use a preorder traversal instead of an inorder traversal? How about a postorder traversal?

R-8.14 Illustrate the performance of the heap-sort algorithm on the following input sequence: $(2,5,16,4,10,23,39,18,26,15)$.

R-8.15 Let T be a complete binary tree such that node v stores the key-entry pairs $(f(v),0)$, where $f(v)$ is the level number of v. Is tree T a heap? Why or why not?

R-8.16 Explain why the case where the right child of r is internal and the left child is external was not considered in the description of down-heap bubbling.

R-8.17 Is there a heap T storing seven distinct elements such that a preorder traversal of T yields the elements of T in sorted order? How about an inorder traversal? How about a postorder traversal?

R-8.18 Consider the numbering of the nodes of a binary tree defined in Section 7.3.5, and show that the insertion position in a heap with n keys is the node with number $n+1$.

R-8.19 Let H be a heap storing 15 entries using the vector representation of a complete binary tree. What is the sequence of indices of the vector that are visited in a preorder traversal of H? What about an inorder traversal of H? What about a postorder traversal of H?

R-8.20 Show that the sum $\sum_{i=1}^{n} \log i$, which appears in the analysis of heap-sort, is $\Omega(n \log n)$.

R-8.21 Bill claims that a preorder traversal of a heap will list its keys in nondecreasing order. Draw an example of a heap that proves him wrong.

R-8.22 Hillary claims that a postorder traversal of a heap will list its keys in non-increasing order. Draw an example of a heap that proves her wrong.

R-8.23 Show all the steps of the algorithm for removing key 16 from the heap of Figure 8.3.

R-8.24 Draw an example of a heap whose keys are all the odd numbers from 1 to 59 (with no repeats), such that the insertion of an entry with key 32 would cause up-heap bubbling to proceed all the way up to a child of the root (replacing that child's key with 32).

R-8.25 Give a pseudo-code description of a nonrecursive in-place heap-sort algorithm.

R-8.26 A group of children want to play a game, called ***Unmonopoly***, where in each turn the player with the most money must give half of his/her money to the player with the least amount of money. What data structure(s) should be used to play this game efficiently? Why?

Creativity

C-8.1 An online computer system for trading stock needs to process orders of the form "buy 100 shares at \$$x$ each" or "sell 100 shares at \$$y$ each." A buy order for \$$x$ can only be processed if there is an existing sell order with price \$$y$ such that $y \leq x$. Likewise, a sell order for \$$y$ can only be processed if there is an existing buy order with price \$$x$ such that $x \geq y$. If a buy or sell order is entered but cannot be processed, it must wait for a future order that allows it to be processed. Describe a scheme that allows for buy and sell orders to be entered in $O(\log n)$ time, independent of whether or not they can be immediately processed.

C-8.2 Extend a solution to the previous problem so that users are allowed to update the prices for their buy or sell orders that have yet to be processed.

C-8.3 Write a comparator for integer objects that determines order based on the number of 1s in each number's binary expansion, so that $i < j$ if the number of 1s in the binary representation of i is less than the number of 1s in the binary representation of j.

C-8.4 Show how to implement the stack ADT using only a priority queue and one additional member variable.

C-8.5 Show how to implement the (standard) queue ADT using only a priority queue and one additional member variable.

C-8.6 Describe, in detail, an implementation of a priority queue based on a sorted array. Show that this implementation achieves $O(1)$ time for operations min and removeMin and $O(n)$ time for operation insert.

C-8.7 Describe an in-place version of the selection-sort algorithm that uses only $O(1)$ space for member variables in addition to an input array itself.

C-8.8 Assuming the input to the sorting problem is given in an array A, describe how to implement the insertion-sort algorithm using only the array A and, at most, six additional (base-type) variables.

C-8.9 Assuming the input to the sorting problem is given in an array A, describe how to implement the heap-sort algorithm using only the array A and, at most, six additional (base-type) variables.

C-8.10 Describe a sequence of n insertions to a heap that requires $\Omega(n \log n)$ time to process.

C-8.11 An alternative method for finding the last node during an insertion in a heap T is to store, in the last node and each external node of T, a pointer to the external node immediately to its right (wrapping to the first node in the next lower level for the rightmost external node). Show how to maintain such a pointer in $O(1)$ time per operation of the priority queue ADT, assuming T is implemented as a linked structure.

C-8.12 We can represent a path from the root to a given node of a binary tree by means of a binary string, where 0 means "go to the left child" and 1 means "go to the right child." For example, the path from the root to the node storing 8 in the heap of Figure 8.3 is represented by the binary string 101. Design an $O(\log n)$-time algorithm for finding the last node of a complete binary tree with n nodes based on the above representation. Show how this algorithm can be used in the implementation of a complete binary tree by means of a linked structure that does not keep a reference to the last node.

C-8.13 Suppose the binary tree T used to implement a heap can be accessed using only the functions of the binary tree ADT. That is, we cannot assume T is implemented as a vector. Given a pointer to the current last node, v, describe an efficient algorithm for finding the insertion point (that is, the new last node) using just the functions of the binary tree interface. Be sure and handle all possible cases as illustrated in Figure 8.12. What is the running time of this function?

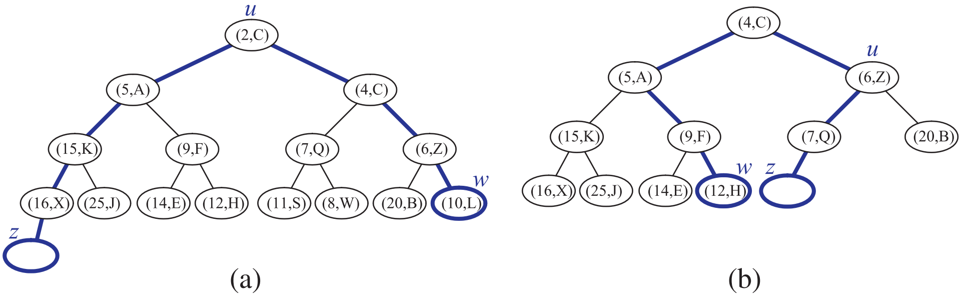


Figure 8.12: Updating the last node in a complete binary tree after operation add or remove. Node w is the last node before operation add or after operation remove. Node z is the last node after operation add or before operation remove.

C-8.14 Given a heap T and a key k, give an algorithm to compute all the entries in T with a key less than or equal to k. For example, given the heap of Figure 8.12(a) and query $k = 7$, the algorithm should report the entries with keys 2, 4, 5, 6, and 7 (but not necessarily in this order). Your algorithm should run in time proportional to the number of entries returned.

C-8.15 Show that, for any n, there is a sequence of insertions in a heap that requires $\Omega(n \log n)$ time to process.

C-8.16 Provide a justification of the time bounds in Table 8.1.

C-8.17 Develop an algorithm that computes the kth smallest element of a set of n distinct integers in $O(n + k \log n)$ time.

C-8.18 Suppose the internal nodes of two binary trees, T_1 and T_2 respectively, hold items that satisfy the heap-order property. Describe a method for combining these two trees into a tree T, whose internal nodes hold the union of the items in T_1 and T_2 and also satisfy the heap-order property. Your algorithms should run in time $O(h_1 + h_2)$ where h_1 and h_2 are the respective heights of T_1 and T_2.

C-8.19 Give an alternative analysis of bottom-up heap construction by showing that, for any positive integer h, $\sum_{i=1}^{h}(i/2^i)$ is $O(1)$.

C-8.20 Let T be a heap storing n keys. Give an efficient algorithm for reporting all the keys in T that are smaller than or equal to a given query key x (which is not necessarily in T). For example, given the heap of Figure 8.3 and query key $x = 7$, the algorithm should report 4, 5, 6, 7. Note that the keys do not need to be reported in sorted order. Ideally, your algorithm should run in $O(k)$ time, where k is the number of keys reported.

C-8.21 Give an alternate description of the in-place heap-sort algorithm that uses a standard comparator instead of a reverse one.

C-8.22 Describe efficient algorithms for performing operations remove(e) on an adaptable priority queue realized by means of an unsorted list with location-aware entries.

C-8.23 Let S be a set of n points in the plane with distinct integer x- and y-coordinates. Let T be a complete binary tree storing the points from S at its external nodes, such that the points are ordered left-to-right by increasing x-coordinates. For each node v in T, let $S(v)$ denote the subset of S consisting of points stored in the subtree rooted at v. For the root r of T, define $top(r)$ to be the point in $S = S(r)$ with maximum y-coordinate. For every other node v, define $top(r)$ to be the point in S with highest y-coordinate in $S(v)$ that is not also the highest y-coordinate in $S(u)$, where u is the parent of v in T (if such a point exists). Such labeling turns T into a ***priority search tree***. Describe a linear-time algorithm for turning T into a priority search tree.

Projects

P-8.1 Generalize the Heap data structure of Section 8.3 from a binary tree to a k-ary tree, for an arbitrary $k \geq 2$. Study the relative efficiencies of the resulting data structure for various values of k, by inserting and removing a large number of randomly generated keys into each data structure.

P-8.2 Give a C++ implementation of a priority queue based on an unsorted list.

P-8.3 Develop a C++ implementation of a priority queue that is based on a heap and supports the locator-based functions.

P-8.4 Implement the in-place heap-sort algorithm. Compare its running time with that of the standard heap-sort that uses an external heap.

P-8.5 Implement a heap-based priority queue that supports the following additional operation in linear time:

replaceComparator(c): Replace the current comparator with c.

After changing the comparator, the heap will need to be restructured. (Hint: Utilize the bottom-up heap construction algorithm.)

P-8.6 Write a program that can process a sequence of stock buy and sell orders as described in Exercise C-8.1.

P-8.7 One of the main applications of priority queues is in operating systems—for ***scheduling jobs*** on a CPU. In this project you are to build a program that schedules simulated CPU jobs. Your program should run in a loop, each iteration of which corresponds to a ***time slice*** for the CPU. Each job is assigned a priority, which is an integer between -20 (highest priority) and 19 (lowest priority), inclusive. From among all jobs waiting to be processed in a time slice, the CPU must work on the job with highest priority. In this simulation, each job will also come with a ***length*** value, which is an integer between 1 and 100, inclusive, indicating the number of time slices that are needed to process this job. For simplicity, you may assume jobs cannot be interrupted—once it is scheduled on the CPU, a job runs for a number of time slices equal to its length. Your simulator must output the name of the job running on the CPU in each time slice and must process a sequence of commands, one per time slice, each of which is of the form “add job *name* with length n and priority p” or “no new job this slice.”

Chapter Notes

Knuth’s book on sorting and searching [57] describes the motivation and history for the selection-sort, insertion-sort, and heap-sort algorithms. The heap-sort algorithm is due to Williams [103], and the linear-time heap construction algorithm is due to Floyd [33]. Additional algorithms and analyses for heaps and heap-sort variations can be found in papers by Bentley [12], Carlsson [20], Gonnet and Munro [38], McDiarmid and Reed [70], and Schaffer and Sedgewick [88]. The design pattern of using location-aware entries (also described in [39]), appears to be new.