Chapter 15

Priority Queues

Chapter Overview

A *priority queue* is a data structure useful in problems where it is important to be able to rapidly and repeatedly find and remove the largest element from a collection of values. In this chapter we will present two different implementations of priority queues. The first technique uses an abstraction called a *heap*, and is constructed as an *adaptor* built on top another form of container, typically a vector or deque. The heap data structure is then used to demonstrate yet another approach to sorting a collection of values. The second implementation strategy is the *skew heap*. The skew heap is notable in that it does not provide guaranteed performance bounds for any single operation, but it can be shown that if a number of operations are performed over time the average execution time of operations will be small.

As a demonstration one of the more common uses of heap, the chapter concludes with a discussion of discrete event driven simulation. This topic is approached by first developing a general *framework* for simulations, then specializing the framework using *inheritance*.

- The priority queue data abstraction
- Heaps and heap sort
- Skew heaps
- A framework for simulation
- Discrete Event-driven simulation

15.1 The Priority Queue Data Abstraction

An everyday example of a priority queue is the "to do" list of tasks waiting to be performed that most of us maintain to keep ourselves organized. Some jobs, such as "clean desktop", are not imperative and can be postponed arbitrarily. Other tasks, such as "finish report by Monday" or "buy flowers for anniversary", are time crucial and must be addressed more rapidly. Therefore, we sort the tasks waiting to be accomplished in order of their importance (or perhaps based on a combination of their critical importance, their long term benefit, and how much fun they are to do) and choose the most pressing.

For a more computer-related example, an operating system might use a priority queue to maintain a collection of currently active processes, where the value associated with each element in the queue represents the urgency of the task. It may be necessary to respond rapidly to a key pressed at a workstation, for example, before the data is lost when the next key is pressed. Other tasks can be temporarily postponed in order to handle those that are time critical. For this reason a priority queue is used so that the most urgent task can be quickly determined and processed.

Another example might be files waiting to be output on a computer printer. It would be a reasonable policy to print several one page listings before a one hundred-page job, even if the larger task was submitted earlier than the smaller ones. For this reason a queue would maintain files to be printed in order of size, or a combination of size and other factors, and not simply on time of submission.

A simulation, such as the one we will describe in Section 15.4, can use a priority queue of "future events", where each event is marked with a time at which the event is to take place. The element in this collection with the closest following time is the next event that will be simulated. As each event is executed, it may spawn new events to be added to the queue. These are only a few instances of the types of problems for which a priority queue is a useful tool.

In terms of abstract operations, a priority queue is a data structure that takes elements of type `value_type` and implements the following five operations:

void push(value_type)	add a new value to the collection
value_type & top()	return a reference to the largest element in collection
void pop()	delete the largest element from the collection
int size()	return the number of elements in the collection
bool empty()	return true if the collection is empty

Although the definition speaks of removing the largest value, in many problems the value of interest is the smallest item. As we will see in some of the later examples, such uses can be provided by inverting the comparison test between elements.

Note that the name priority *queue* is a misnomer in that the data structure is not a queue, in the sense we used the term in Chapter 10, since it does not return elements in a strict first-in first-out sequence. Nevertheless, the name is now firmly associated with this particular data type.

There are at least three obvious, but inefficient, ways to implement a priority queue. One approach would be to insert new elements at the front of a list, thus requiring only constant time for addition. The generic algorithm max_element() can be used to discover the largest element. The max_element() algorithm traverses the list to find the largest value, requiring $O(n)$ time. Keeping the list ordered by value would make for rapid discovery of the maximum, but necessitate $O(n)$ time for insertion.

Another approach to implementing a priority queue is to take a collection of items and sort them. From the sorted collection we could obtain not only the largest element, but the next largest, and the next, and so on. But as we have seen, sorting is a relatively expensive operation. Furthermore, it is difficult to insert new values into a sorted collection. Since we are interested only in finding the largest element in the collection, we can employ techniques that are much more efficient. In particular, we will develop data structures in which we can find the largest element in a collection of n elements in constant time, and we can find and remove the largest element in time proportional to $\log n$.

There is one more method that is obvious, and not obviously inefficient. In Chapter 12 we examined the STL set data structure. Recall that a new element could be added to a set in $O(\log n)$ steps. If we reverse the comparison operator when we create the set, then the value associated with the iterator aSet.begin() represents the largest value, and can also be used to delete this value from the collection. A careful examination of the iterator procedure will show that this requires no more than $O(\log n)$ steps. Thus, using a set one can perform both insertions and removals in logarithmic time.

The reason for rejecting the set is not asymptotic inefficiency, but practical realistic inefficiency. A set is maintaining more information than we need. We can develop alternative data structures that, while they have no better asymptotic efficiency (they are still $O(\log n)$), generally yield execution times much better than would be possible using the set data type.

We will investigate two data structures that can be used to implement priority queues. The first is provided as a basic data type by the standard library, while the second will be developed independently. The standard data structure is called a *heap*, and maintains the values of the collection in an array. The operations to add or remove an element to or from the heap are relatively efficient, however the heap suffers from the problem common to many array-based algorithms. This is that the array will be expanded as necessary to the maximum size, but will not be made smaller as the heap is reduced. In this fashion the array holding the heap will be the largest size needed to hold the values at any one point in time.

The second data structure, a *skew heap*, avoids the maximum size difficulty by maintaining the heap values in a binary tree. But solving one problem comes only at the cost of introducing another, namely the difficulty of keeping the tree relatively well balanced. Skew heaps are interesting in that the worst case cost for insertions and deletions is a relatively slow $O(n)$, but one can show that this worst case behavior does not occur frequently and cannot be sustained. In particular, the occurrence of a worst case situation must necessarily be followed by several insertions and deletions that are much faster. Thus, amortized over a number of insertions and deletions, the average cost of operation is still relatively good.

Another advantage of the skew heap will be that it provides a fast implementation for the operation of merging two priority-queue heaps to form a new queue.

15.2 Heaps

A *heap* is a binary tree in which every node possesses the property that its value is larger than or equal to the value associated with either child node. This is referred to as the *heap order property*. A simple induction argument establishes that the value associated with each node in a heap must be the largest value held in the subtree rooted at the node. It follows from this property that the largest element in a heap will always be held by the root node. This is unlike a search tree, where the largest element was always held by the rightmost node. Discovering the maximum value in a heap is therefore a trivial operation.

Recall from Chapter 13 that a *complete binary tree* is a binary tree that is entirely filled (in which every node has two children), with the exception of the bottom level, which is filled from left to right. Figure 15.1 shows a complete binary tree that is also a heap. The key insight behind the heap data structure is the observation, which we noted in Chapter 13, that because a complete binary tree is so regular, it can be represented efficiently in an array. The root of the tree will be maintained in position 0 of the array. The two children of node n will be held in positions $2n + 1$ and $2n + 2$. The array corresponding to the tree in Figure 15.1 is the following:

0	1	2	3	4	5	6	7	8	9	10
16	14	9	10	12	7	8	5	2	11	3

Note that a vector that is sorted largest to smallest is also a heap, but the reverse is not true. That is, a vector can maintain a heap and still not be ordered.

Given the index to a node, discovering either the parent or the children of that node is a simple matter of division or multiplication. No explicit pointers need be maintained in the array structure, and traversals of the heap can be performed very efficiently. Notice that the property of completeness is important to ensure there are no "gaps" in the array representation.

The standard library class `priority_queue` constructs a heap on top of a randomly accessible data structure, usually either a `vector` or a `deque`. Like the `stack` and `queue` data types (Chapter 10), the underlying container is provided as a template argument. An optional second template argument (not shown) represents the function to be used in comparing two elements.[1] We will see an example that uses this argument in Section 15.4. To declare a priority queue, the user must state both the priority queue type and the type of the underlying container, as in the following example:

[1]See Section A.3 in Appendix A.

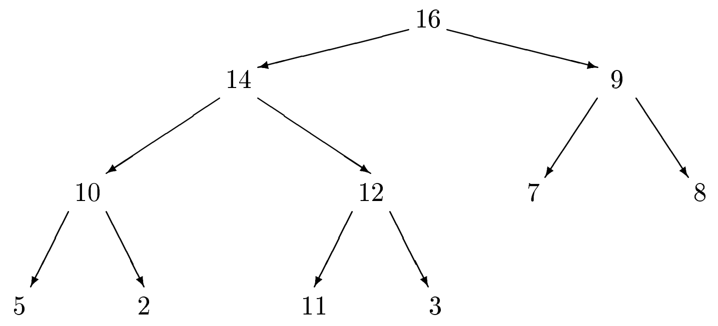

Figure 15.1: A complete binary tree in heap form

```
    // create a priority queue of integers
priority_queue < vector<int> > aQueue;
```

Figure 15.2 gives the class declaration and member functions for this data type. Notice that operations are either defined by the underlying structure, or by one of the generic heap functions we will shortly define.

The operations of insertion (performed by push(newElement)) and deletion (performed by pop()) are more complex than the others, and involve calling an auxiliary function. We will deal with insertion first. When a new element in added to the priority queue, it is obvious that *some* value must be moved to the end of the array, to maintain the complete binary tree property. However, it is likely that the new element cannot be placed there without violating the heap order property. This violation will occur if the new element is larger than the parent element for the location. A solution is to place the new element in the last location, then move it into place by repeatedly exchanging the node with its parent node until the heap order property is restored (that is, until the new node either rises to the top, or until we find a parent node that is larger than the new node). Since the new element rises until it finds a correct location, this process is sometimes known as a *percolate up*.

The insertion method for percolating a value into place is shown in Figure 15.3. We have added the invariants required to prove the correctness of the procedure (see sidebar). Since the `while` loop moves up one level of the tree each cycle, it is obvious it can iterate no more than $\log n$ times. The running time of the insertion procedure is $O(\log n)$. (Note that before calling `push_heap`, the new element is pushed on to the end of the container. For the vector data structure, this operation could, in the worst case, require $O(n)$ steps, if a new buffer is allocated and elements are copied. This could potentially increase the execution time of the entire operation.)

```
//
//    class priority_queue
//        a priority queue managed as a vector heap
//

template <class Container> class priority_queue {
public:
    typedef Container::value_type value_type;

    // priority queue protocol
    bool            empty    ()    { return c.empty(); }
    int             size     ()    { return c.size(); }
    value_type &    top      ()    { return c.front(); }
    void            push     (value_type & newElement)
                    { c.push_back(newElement);
                      push_heap(c.begin(), c.end()); }
    void            pop      ()
                    { pop_heap (c.begin(), c.end());
                      c.pop_back(); }
protected:
    Container        c;    // container of values
};
```

Figure 15.2: Declaration for the class `priority_queue`

Heaps and Heaps

The term *heap* is used for two very different concepts in computer science. The heap *data structure* is an abstract data type used to implement priority queues, as well as being used in a sorting algorithm we will discuss in a later chapter. The terms *heap*, *heap allocation*, and so on, are also frequently used to describe memory that is allocated and released directly by the user, using the `new` and `delete` operators. You should not confuse the two uses of the same term.

```
template <class Iterator>
void push_heap(Iterator start, Iterator stop)
    // initial condition:
    // iterator range describes a heap, except that
    // final element may be out of place
{
        // position is index of out of place element
        // parent is index of parent node
    unsigned int position = (stop - start) - 1;
    unsigned int parent = (position - 1) / 2;

    // now percolate up
    while (position > 0 && start[position] < start[parent]) {
        // inv: tree rooted at position is a heap
        swap (start[position], start[parent]);
        // inv: tree rooted at parent is a heap
        position = parent;
        parent = (position - 1) / 2;
        }
    // inv: entire structure is a heap
}
```

Figure 15.3: Method for insertion into a `heap`

Proof of Correctness: push_heap

The proof of correctness for the algorithm named `push_heap` begins with the assumption that the collection represented by the iterator range represents a valid heap, with the possible exception of the very last element, which may be out of order. The sole task of the algorithm is to move this one element into place.

Throughout the algorithm, the variable `position` will maintain the index position of this value, while the variable `parent` will maintain the index position of the parent node.

The `while` loop at the heart of the algorithm has two test conditions. Both must be true for the loop to execute. Together, the two conditions assert that the position has a parent (that is, the position is not yet the root note), and that the parent value is smaller than the position, in contradiction to the heap order property. The following illustrates this situation.

$$\begin{tikzcd}
& 9 \arrow[dl] \arrow[dr] & & & & \boxed{12} \arrow[dl] \arrow[dr] & \\
5 & & \boxed{12} & => & 5 & & 9
\end{tikzcd}$$

The invariant at the top of the loop body asserts that the subtree rooted at `position` represents a heap. This will trivially be true the first time the while loop is executed, since the subtree represents only a leaf. Since the parent node is already larger than the other child (if the parent has two children), simply swapping the position with the parent is sufficient to locally reestablish the heap order property. (To see why, note that any children of node `position` must have originally been children of node `parent`, and must therefore be smaller than the parent.) Following the swap, the tree rooted at the index value `parent` will therefore be a heap. This value becomes the new position, and we determine the new parent index.

The while loop terminates either when the value percolates all the way to the top of the heap, or when a node is encountered which is larger than the new value. In the first case the final loop invariant is asserting that the entire structure represents a heap. If, on the other hand, the loop terminates because of the second case, we know that the subtree rooted at `parent` has the heap order property. But since this subtree holds the only value that could have been out of order, we therefore can conclude that the entire structure must have the heap order property.

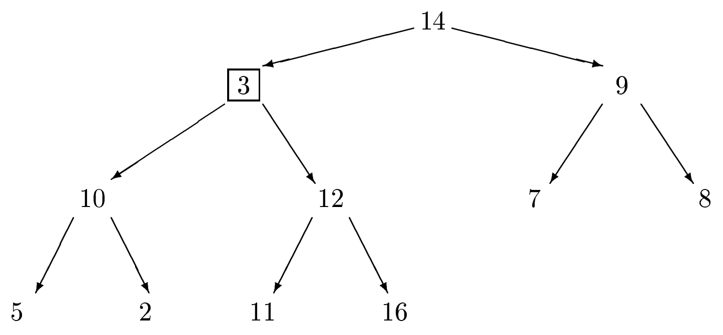

Figure 15.4: A value percolating down into position.

The deletion procedure is handled in a manner similar to insertion. We swap the last position into the first location. Since this may destroy the heap order property, the element must *percolate down* to its proper position by exchanging places with the larger of its children. For reasons that will shortly become clear, we invoke another routine named adjust_heap to do this task.[2] Figure 15.4 shows an intermediate step in this process. The value 3 has been promoted to the root position, where it has subsequently been swapped with the larger of its two children. The value 3 will now be compared to the values 10 and 12, and will be swapped with the larger value. This will continue until either the value is larger than both children, or until we reach the leaf level of the tree. The code to perform deletion is shown in Figure 15.5.

Since at most three comparisons of data values are performed at each level, and the while loop traces out a path from the root of the tree to the leaf, the complexity of the deletion procedure is also $O(\log n)$.

15.2.1 Application – Heap Sort

The heap data structure provides an elegant technique for sorting an array. The basic idea is to first form the array into a heap. To sort the array, the top of the heap (the largest element) is swapped with the last element of the array, and the size of the heap is reduced by one. The effect of the swap, however, may be to destroy the heap property. But this is exactly the same condition we encountered in deleting an element from the heap. And, not surprisingly, we can use the same solution. Heap order is restored by invoking the `adjust_heap` procedure.

[2]The procedure adjust_heap is not defined by the standard library, and is therefore not guaranteed to exist in all implementations.

```
template <class Iterator>
void pop_heap (Iterator start, Iterator stop)
{
    unsigned int lastPosition = (stop - start) - 1;
    // move the largest element into the last location
    swap (start[0], start[lastPosition]);
    // then readjust
    adjust_heap(start, lastPosition, 0);
}

template <class Iterator>
void adjust_heap
    (Iterator start, unsigned heapSize, unsigned position)
    //    initial conditions:
    //    collection represents a heap, except that element
    //    at index value position may be out of order
{
    while (position < heapsize) {
        // To fix, replace position with the larger
        // of the two children
        unsigned int childpos = position * 2 + 1;
        if (childpos < heapsize) {
            if ((childpos + 1 < heapsize) &&
                start[childpos + 1] > start[childpos])
                    childpos++;
            // childpos is larger of two children
            if (start[position] > start[childpos])
                    // structure is now heap
                return;
            else
                swap (start[position], start[childpos]);
            }
        position = childpos;
        }
}
```

Figure 15.5: Method for deletion from a `heap`

Proof of Correctness: adjust_heap

The `adjust_heap` algorithm is in some ways the opposite of the `push_heap` procedure. Here, the assumption is that the structure of the given size beginning at the starting iterator is a heap, except that the value with the index value `position` may be out of order.

To reestablish the heap order property, the two children of node `position` are examined. The value indexed by `childpos` is set to the larger of the two children. If this value is smaller than the value in question, then the heap order property does in fact hold, and therefore the entire structure must be a heap. If, on the other hand, the larger child is also larger than the value in question, then they must be swapped. This is illustrated in the following picture:

$$\begin{tikzcd}
 & \boxed{5} \arrow[dl] \arrow[dr] & & & & 12 \arrow[dl] \arrow[dr] & \\
12 & & 9 & => & \boxed{5} & & 9
\end{tikzcd}$$

Note that since the element was swapped with the larger of the two children, the new root must therefore be not only larger than the element in question, but also larger than the smaller child. Thus we need not consider the subtree rooted at the smaller child. But it is now necessary to continue to examine the subtree rooted at the position into which the value in question was swapped. The larger child position becomes the new value held by the variable `position`, and the while loop continues.

The algorithm terminates either when the value in question finds its location (being larger than both children), or it reaches a point where it has no children.

Slightly more surprising is that we can use the `adjust_heap` procedure to construct an initial heap from an unorganized collection of values held in a vector. To see this, note that a subtree consisting of a leaf node by itself satisfies the heap order property. To build the initial heap, we start with the smallest subtree containing interior nodes, which corresponds to the middle of the data array. Invoking the `adjust_heap` method for this value will ensure the subtree satisfies the heap order property. Walking back towards the root of the tree, we repeatedly invoke the `adjust_heap`, thereby ensuring all subtrees are themselves heaps. When we finally reach the root, the entire tree will have been made into a heap. This algorithm is implemented by the following procedure:

```
template <class Iterator>
```

Proof of Correctness: make_heap

To prove that the `make_heap` algorithm creates a heap it is necessary to understand the task being performed by the loop that is at the heart of the algorithm. At each step of this algorithm, the assumption is that the subtrees representing the children of node `i` are proper heaps, and the task to be performed is to make the subtree rooted at node `i` into a heap.

Note that the value `i` is initialized to the value `heapSize / 2`, and moves downwards. Observe that all subtrees with index values larger than `heapSize / 2` represent leaf nodes, and that leaf nodes possess the heap property (trivially, since they have no children). Thus, the first time execution moves from the start of the procedure to the assertion in the body of the loop, the assertions must be true. We have already proven the `adjust_heap` procedure, and therefore the invariant following the procedure call must be true.

Now consider the case where we encounter the assertion at the beginning of the loop body, after having executed some number of previous iterations of the loop. In this case, the children of node `i` must either be leaves, or they must have been previously processed. In either case, the subtrees rooted at the child nodes must be heaps, and therefore following execution of the body of the loop, the subtree rooted at node `i` must be a heap.

The establish the final condition, we simply note that either the loop was executed, in which case the final condition matches one of the loop invariants we previously established, or the loop was never executed, which can only happen if the heap contains only a single leaf. In the latter case, we have already noted that a leaf node is a heap.

```
void make_heap (Iterator start, Iterator stop)
{
    unsigned int heapSize = stop - start;
    for (int i = heapSize / 2; i >= 0; i--)
            // assume children of node i are heaps
        adjust_heap(start, heapSize, i);
            // inv: tree rooted at node i is a heap
    // assert: structure is now a heap
}
```

To convert a heap into a sorted collection, we simply repeatedly swap the first and last positions, then readjust the heap property, reducing the heap size by one element. This is performed by the following procedure:

```
template <class Iterator>
void sort_heap (Iterator start, Iterator stop)
{
    unsigned int lastPosition = stop - start - 1;
    while (lastPosition > 0) {
        swap(start[0], start[lastPosition]);
        adjust_heap(start, lastPosition, 0);
        lastPosition--;
        }
}
```

Combining these two, the heap sort algorithm can be written as follows:[3]

```
template <class Iterator>
void heap_sort(Iterator start, Iterator stop)
{    // sort the vector argument using a heap algorithm

    // first build the initial heap
    make_heap (start, stop);

    // then convert heap into sorted collection
    sort_heap (start, stop);
}
```

To derive the asymptotic running time for this algorithm, recall we noted that the `adjust_heap` procedure requires $O(\log n)$ steps. There are n executions of `adjust_heap` to generate the initial heap, and n further executions to reheap values during the sorting operation. Combining these tells us that the total running time is $O(n \log n)$. This matches that of the merge sort algorithm (Section 8.3.3) and the quick sort algorithm (Section 14.5.1), and is better than the $O(n^2)$ bubble and insertion sort algorithms (Section 5.1.4).

Of a more practical benefit, note that the heap sort algorithm does not require any additional space, since it constructs the heap directly in the vector input value. This was not true of some of the previous sorting algorithms we have seen. Those algorithms must pay the cost not only of the sorting itself, but of the allocation and deallocation of the data structures formed during the process of ordering the elements.

An empirical analysis of the running time of the heap sort algorithm illustrates that for almost all vector sizes heapsort is comparable in speed to quick sort. An advantage of heap sort over quick sort is that the heap sort algorithm is less influenced by the initial

[3]Note that `make_heap` and `sort_heap` are generic algorithms in the standard library, but `heap_sort` is not.

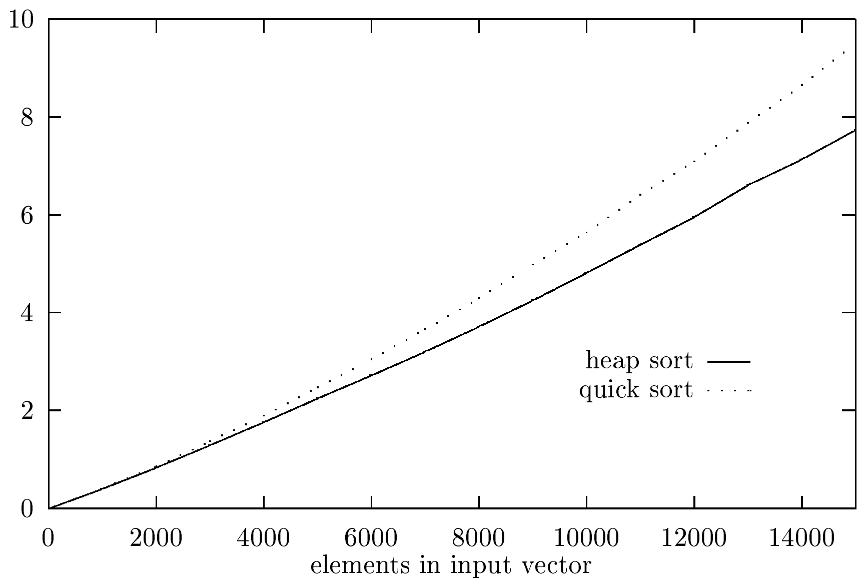

Figure 15.6: Empirical Timing of heap sort

distribution of the input values. You will recall that a poor distribution of values can make quick sort exhibit $O(n^2)$ behavior, while heap sort is $O(n \log n)$ in all circumstances.

15.3 Skew Heaps *

The obvious method to avoid the bounded-size problem of heaps is to use a tree representation. This is not, however, quite as simple as it might seem. The key to obtaining logarithmic performance in the heap data structure is the fact that at each step we were able to guarantee the tree was completely balanced. Finding the next location to be filled in an array representation of a completely balanced binary tree is trivial; it is simply the next location following the current top of the array. In a tree form this is not quite as easy. Consider the tree shown in Figure 15.1 (page 399). Knowing the location of the last element (the value 3) is of no help in discovering where the next element should be inserted in order to maintain the balanced binary tree property. In fact, the next element is part of an entirely different subtree than that containing the current last element.

A *skew heap* avoids this problem by making no attempt to maintain the heap as a completely balanced binary tree. As we saw when we examined search trees, this means that a tree can potentially become almost linear, and we can place no guarantee on logarithmic

*Section headings followed by an asterisk indicate optional material.

performance for any individual operation. But there is another critical observation we can make concerning heaps, which is that the order of the left and right children for any node is essentially arbitrary. We can exchange the left and right children of any node in a heap without destroying the heap order property. We can make use of this observation by systematically swapping the left and right children of a node as we perform insertions and deletions. A badly unbalanced tree can affect the performance of one operation, but it can be shown that subsequent insertions and deletions must as a consequence be very rapid. In fact, if m insertions or deletions are performed, it can be shown (although the details are beyond the discussion here) that the total time to perform all m operations is bounded by $O(m \log n)$. Thus, *amortized* over time, each operation is no worse than $O(\log n)$.

The second observation critical to the implementation of skew heaps is that both insertions and deletions can be considered as special cases of merging two trees into a single heap. This is obvious in the case of the deletion process. Removing the root of the tree results in two subtrees. The new heap can be constructed by simply merging these two child trees.

```
template <class value_type> void  skewHeap<value_type>::pop ()
    // remove the minimum element from a skew heap
{
    assert (! empty());
    node<value_type> * top = root;
    root = merge(root->right(), root->left());
    delete top;
}
```

Similarly, insertion can be considered a merge of the existing heap and a new heap containing a single element.

```
template <class value_type>
void skewHeap<value_type>::push (value_type & val)
    // to add a new value, simply merge with
    // a tree containing one node
{
    root = merge(root, new node<value_type>(val));
}
```

The `skewHeap` data structure, shown in Figure 15.7, implements both insertions and deletions using an internal method `merge`. The recursive `merge` operation is shown below. If either argument is empty, then the result of a merge is simply the other tree. Otherwise we will assume the largest value is the root of the first tree, by returning the merge of the arguments reversed if this is not the case. To perform the merge, we move the current left child of the left argument to the right child of the result, and recursively merge the right argument with the old right child.

```
//
//     class skewHeap
//         heap priority queue implemented using skew heap merge
//         operations
//

template <class value_type> class skewHeap {
public:
    // constructors
    skewHeap        () : root(0) { }
    ~skewHeap        ();

        // priority queue protocol
    bool            empty    () { return root == 0; }
    int             size     () { return root->size(); }
    value_type &    top      () { return root->value; }
    void            pop      ();
    void            push     (value_type & value);

        // additional method: splice two heaps together
    void            splice        (skewHeap & secondHeap);

protected:
        // root of heap
    node<value_type> *       root;

        // internal method to merge two heaps
    node<value_type> * merge (node<value_type> *, node<value_type> *);
};
```

Figure 15.7: The `skewHeap` class declaration

```
template <class value_type>
node<value_type> * skewHeap<value_type>::merge
    (node<value_type> * h1, node<value_type> * h2)
    // merge two skew heaps to form a new heap
{
        // if either tree is empty, return the other
    if (! h1) return h2;
    if (! h2) return h1;

        // assume largest is root of h1
    if (h2->value > h1->value)
        return merge(h2, h1);

        // reverse children and recurse
    node<value_type> * lchild = h1->left();
    if (lchild) {
        h1->left(merge(h1->right(), h2));
        h1->right(lchild);
        }
    else    // no left child
        h1->left(h2);
    return h1;
}
```

For example, suppose we are merging a heap containing the elements 2, 5 and 7 with a heap containing the two elements 4 and 6. Since the element at the top of the left heap, 7, is larger it becomes the new root. At the same time the old left child of the root becomes the new right child. To form the new right child we recursively merge the old right child and the original right argument.

$$\begin{tikzcd}
{} & 7 \arrow[dl] \arrow[dr] & \leftarrow \mathit{merge} \rightarrow & {} & 6 \arrow[dl] & \Rightarrow & {} & {} & 7 \arrow[dr] & {} \\
5 & {} & 2 & 4 & {} & {} & 2 & \leftarrow \mathit{merge} \rightarrow & 6 \arrow[dl] & 5 \\
{} & {} & {} & {} & {} & {} & {} & 4 & {} & {}
\end{tikzcd}$$

The first step in the recursive call is to flip the arguments, so that the largest element is held in the first argument. The top element of this heap then becomes the new root. As before, the old left child of this value becomes the new right child. A recursive call is made

Merging Heaps

The fact that skew heaps basically operate by merging two heaps to form a new heap means that it is relatively easy to combine together two instances of the data structure. We have taken advantage of this by providing a `splice` method that takes another instance of skew heap as argument.

```
template <class value_type> void skewHeap<value_type>::splice
        (skewHeap<value_type> & secondHeap)
{    // merge elements from a second heap into current heap
    root = merge(root, secondHeap.root);
    // empty values from second heap
    secondHeap.root = 0;
}
```

The merge procedure used is the same as the merge used in implementing the addition and removal methods, and can thus be expected to run very rapidly, in time proportional to the longest path in the largest heap, not the number of elements in the argument heap, as would be the case of the values were simply added one by one.

An important feature to note, however, is that this operation effectively empties the argument heap, by setting its root value to zero. The reason why this is necessary has to do with the way in which our data structures are performing memory management. In our scheme, each node in a tree must be "owned" by one and only one data structure. This data structure is responsible for performing a deletion to free up the memory used by the node when it is no longer being used as part of the structure. If a single node were to be used in two different structures it is possible, indeed inevitable, that it would be deleted at two different times by two different structures.

to insert the right argument, 7, into the now empty former right child of the node 4. This results in the node 7 being returned, and the final result produced.

$$\begin{tikzcd}
 & & & 7 \arrow[dr] & & \Rightarrow & & & 7 \arrow[dl] \arrow[dr] & \\
 & 6 \arrow[dl] & \leftarrow merge \rightarrow & 2 & 5 & & & 6 \arrow[dl] \arrow[dr] & & 5 \\
4 & & & & & & 2 & & 4 &
\end{tikzcd}$$

To illustrate why the amortized analysis of skew heaps can be so good, note that the worst case situation occurs when the left subtree contains a long path along the right child links. For example consider the merging the singleton 2 into such a tree.

$$\begin{tikzcd}
 & 9 \arrow[dl] \arrow[dr] & & \leftarrow merge \rightarrow & 2 & & & & & 9 \arrow[dr] & \\
8 & & 7 \arrow[dl] \arrow[dr] & & & \Rightarrow & & 7 \arrow[dl] \arrow[dr] & \leftarrow merge \rightarrow & 2 & 8 \\
 & 6 & & 5 \arrow[dl] \arrow[dr] & & & 6 & & 5 \arrow[dl] \arrow[dr] & & \\
 & & 4 & & 3 & & & 4 & & 3 &
\end{tikzcd}$$

$$\begin{tikzcd}
 & & & & & & & & & & & & 9 \arrow[dl] \arrow[dr] & \\
 & & & & & & 9 \arrow[dll] \arrow[dr] & & \Rightarrow & & & 7 \arrow[dl] \arrow[dr] & & 8 \\
 & & & & 7 \arrow[dr] & & & 8 & & & 5 \arrow[dl] \arrow[dr] & & 6 & \\
 & 5 \arrow[dl] \arrow[dr] & & \leftarrow merge \rightarrow & & 2 & 6 & & & 3 \arrow[dl] & & 4 & & \\
4 & & 3 & & & & & & 2 & & & & &
\end{tikzcd}$$

The merge requires 4 steps. However, note that now the long right path has been converted into a long left path. Thus, the next insertion will be relatively quick. Assume, for example, we now insert the value 1.

$$\begin{tikzcd}
 & & 9 \arrow[dl] \arrow[dr] & & \\
 & 8 \arrow[dl] & & 7 \arrow[dl] \arrow[dr] & \\
1 & & 5 \arrow[dl] \arrow[dr] & & 6 \\
 & 3 \arrow[dl] & & 4 & \\
2 & & & &
\end{tikzcd}$$

It is tempting to conjecture that after two insertions we would be back to the original poor configuration. But note that this has not occurred. The longest path is still a left path, although it is now on the right side. It will be quite a few steps before the situation can arise where a long right path can slow insertions.

15.4 Application – Discrete Event-Driven Simulation

Imagine you are thinking about opening an ice cream store on a popular beach location. You need to decide how large the store should be; how many seats you should have and so on. If you plan too small, customers will be turned away when there is insufficient space and you will lose profits. On the other hand if you plan too large, most of the seats will be unused and you will be paying useless rent on the space, and hence losing profits. So you need to choose approximately the right number – but how do you decide?

One approach would be to perform a simulation. You first examine similar operations in comparable locations, and form a model which includes, among other factors, an estimation of the number of customers you can expect to arrive in any period of time, the length of time they will take to decide upon an order, and the length of time they will stay after having been served. Based on this you can design a simulation.

A *discrete event-driven simulation* is a popular simulation technique. Objects in the simulation model objects in the real world, and are programmed to react as much as possible as the real objects would react. A priority queue is used to store a representation of "events" that are waiting to happen. This queue is stored in order based on the time the event should occur, so the smallest element will always be the next event to be modeled. As an event occurs, it can spawn other events. These subsequent events are placed into the queue as well. Execution continues until all events have occurred, or until a preset time for the simulation is exceeded.

To see how we might design a simulation of our ice cream store, consider a typical scenario. A group of customers arrive at the ice cream store. From our measurements of similar stores we derive a probability that indicates how frequently this occurs. For example suppose we assume that groups will consist of from 1 to 5 people, selected uniformly over that range. (In actual simulations the distribution would seldom be uniform. For example groups of size 2 and 3 might predominate, with groups of size 1 and groups larger than 3 being relatively less frequent. The mathematics involved in forming non-uniform distributions is subtle, and not particularly relevant to our discussion. We will therefore use uniform distributions throughout.) These groups will arrive at times spaced from 1 to 10 minutes apart, again selected uniformly. Once they arrive, a group will either be seated, or see that there are no seats and leave. If seated they will take from 2 to 10 minutes to order, and once they order they will remain from 15 to 35 minutes in the store. We know that every customer will order from 1 to 3 scoops of ice cream, and that the store makes a profit of $0.35 on each scoop.

To create a random integer between two values we can write a simple function that uses the `randomInteger` class we introduced in Chapter 2. This new function takes the integer endpoints, and returns a new value from a uniform distribution between the two points:

```
integer randBetween (integer low, integer high)
    // return random integer between low and high
{
    randomInteger randomizer;
    return low + randomizer(high - low);
}
```

The primary object in the simulation is the store itself. It might seem odd to provide "behavior" for an inanimate object such as a store, however we can think of the store as a useful abstraction for the servers and managers who work in the store. The store manages two data items; the number of available seats and the amount of profit generated. The behavior of the store can be described by the following list:

- When a customer group arrives, the size of the group is compared to the number of seats. If insufficient seats are available the group leaves. Otherwise the group is seated and the number of seats decreased.
- When a customer orders and is served the amount of profit is computed.
- When a customer group leaves the seats are released for another customer group.

A class description for `IceCreamStore` is shown in Figure 15.8. The implementation of the methods are shown in Figure 15.9.

A Framework for Simulations

Rather than simply code a simulation of this one problem, we will generalize the problem and first produce a generic *framework* for simulations. This is similar to the framework for backtracking problems we presented in Chapter 10.

```
class IceCreamStore {
public:
    IceCreamStore()
        : freeChairs(35), profit(0.0) { }

    bool canSeat (unsigned int numberOfPeople);
    void order(unsigned int numberOfScoops);
    void leave(unsigned int numberOfPeople);

    unsigned int freeChairs;
    double profit;
};
```

Figure 15.8: The class `IceCreamStore`.

At the heart of a simulation is the concept of an *event*. An event will be represented by an instance of class `event`. The only value held by the class will be the time the event is to occur. The method `processEvent` will be invoked to "execute" the event when the appropriate time is reached.

```
//
//    class event
//        execution event in a discrete event driven simulation
//

class event {
public:
        // constructor requires time of event
    event (unsigned int t) : time(t) { }

        // time is a public data field
    unsigned int time;

        // execute event by invoking this method
    virtual void processEvent() { }
};
```

The simulation queue will need to maintain a collection of different types of events. Each different form of event will be represented by a different derived classes of class `event`. Not all events will have the same type, although they will all be derived from class `event`.

```
bool IceCreamStore::canSeat (unsigned int numberOfPeople)
    // if sufficient room, then seat customers
{
    cout << "Time: " << time;
    cout << " group of " << numberOfPeople << " customers arrives";
    if (numberOfPeople < freeChairs) {
        cout << " is seated" << endl;
        freeChairs -= numberOfPeople;
        return true;
        }
    else {
        cout << " no room, they leave" << endl;
        return false;
        }
}

void IceCreamStore::order (unsigned int numberOfScoops)
    // serve ice-cream, compute profits
{
    cout << "Time: " << time;
    cout << " serviced order for " << numberOfScoops << endl;
    profit += 0.35 * numberOfScoops;
}

void IceCreamStore::leave (unsigned int numberOfPeople)
    // people leave, free up chairs
{
    cout << "Time: " << time;
    cout << " group of size " << numberOfPeople << " leaves" << endl;
    freeChairs += numberOfPeople;
}
```

Figure 15.9: The methods implementing the class `IceCreamStore`.

Polymorphic Variables

A variable that can hold many different *types* of values is called *polymorphic* (poly = many, morph = form). In object-oriented languages, such as C++, polymorphic variables are linked to the class-derived class hierarchy. A variable declared as a pointer to a parent class, such as the class event, can in fact hold a value that is a derived class type, such as arriveEvent.

In C++ polymorphic variables can only occur through the use of pointers or references. This is due to the way memory is allocated by the C++ system. Note that the storage required by the child class is *larger* than the storage required by the parent class. (The parent class had only one integer data field, while the child class has two). Memory for all elements of a *vector* must be the same–this is necessary for the efficient indexing ability characteristic of vectors. These two requirements conflict with one another. However, the space required to hold a pointer is fixed, regardless of the type of value it points to. It is for this reason that C++ only allows pointer values to be polymorphic.

(This is sometimes called a *heterogeneous* collection, and the value that points to an event is sometimes called a *polymorphic* variable.) For this reason the collection must store *pointers* to events, instead of the events themselves.

Since comparison of pointers cannot be specialized on the basis of the pointer types, we must instead define an explicit comparison function for pointers to events. When using the standard library this is accomplished by defining a new structure, the sole purpose of which is to define the function invocation operator (the () operator) in the appropriate fashion. Since in this particular example we wish to use the priority queue to return the *smallest* element each time, rather than the largest, the order of the comparison is reversed, as follows:

```
class eventComparison {
public:
    bool operator () (event * left, event * right)
        { return left->time > right->time; }
};
```

We are now ready to define the class simulation, which provides the basic structure for the simulation activities. The class simulation provides two basic functions. The first is used to insert a new event into the queue, while the second runs the simulation. A data field is also provided to hold the current simulation “time”.

```
class simulation {
public:
    simulation () : eventQueue(), currentTime(0) { }

    void scheduleEvent (event * newEvent)
        { eventQueue.push (newEvent); }

    void run();

    unsigned int currentTime;

protected:
    priority_queue<vector<event *>, eventComparison> eventQueue;
};
```

Notice the declaration of the priority queue used to hold the pending events. In this case we are using a vector as the underlying container. We could just as easily have used a deque. Note also the way in which the comparison function class is provided as the second template argument.

The heart of the simulation is the member function `run()`, which defines the event loop. This procedure makes use of three of the five priority queue operations, namely `top()`, `pop()`, and `empty()`. It is implemented as follows:

```
void simulation::run()
    // execute events until event queue becomes empty
{
    while (! eventQueue.empty()) {
        event * nextEvent = eventQueue.top();
        eventQueue.pop();
        time = nextEvent->time;
        nextEvent->processEvent();
        delete nextEvent;
        }
}
```

Ice Cream Store Simulation

Having created a framework for simulations in general, we now return to the specific simulation in hand, the ice cream store. An instance of class `simulation` is defined as a global variable, called `theSimulation`. An instance of `iceCreamStore` is accessible via the name `theStore`.

As we noted already, each activity is matched by a derived class of `event`. Each derived class of `event` includes an integer data field, which represents the size of a group of customers. The arrival event occurs when a group enters. When executed, the arrival event creates and installs a new order event:

```
class arriveEvent : public event {
public:
    arriveEvent (unsigned int time, unsigned int gs)
        : event(time), groupSize(gs) { }
    virtual void processEvent ();
protected:
    unsigned int groupSize;
};

void arriveEvent::processEvent()
{
    if (theStore.canSeat(groupSize))
        theSimulation.scheduleEvent
            (new orderEvent(time + randBetween(2,10), groupSize));
}
```

An order event similarly spawns a leave event:

```
class orderEvent : public event {
public:
    orderEvent (unsigned int time, unsigned int gs)
        : event(time), size(gs) { }
    virtual void processEvent ();
protected:
    unsigned int groupSize;
};

void orderEvent::processEvent()
{
        // each person orders some number of scoops
    for (int i = 0; i < groupSize; i++)
        theStore.order(1 + rand(3));
    theSimulation.scheduleEvent
        (new leaveEvent(time + randBetween(15,35), groupSize));
};
```

Finally, leave events free up chairs, but do not spawn any new events:

```
class leaveEvent : public event {
public:
    leaveEvent (unsigned int time, unsigned int gs)
        : event(time), groupSize(gs) { }
    virtual void processEvent ();
protected:
    unsigned int groupSize;
};

void leaveEvent::processEvent ()
{
    theStore.leave(groupSize);
}
```

The main program simply creates a certain number of initial events, then sets the simulation in motion. In our case we will simulate two hours (120 minutes) of operation, with groups arriving with random distribution between 2 and 5 minutes apart.

```
void main() {
    // load queue with some number of initial events
    unsigned int t = 0;
    while (t < 120) {
        t += randBetween(2,5);
        theSimulation.scheduleEvent(new arriveEvent(t, randBetween(1,5)));
        }

    // then run simulation and print profits
    theSimulation.run();
    cout << "Total profits " << theStore.profit << endl;
}
```

An example execution might produce a log such as the following:

```
customer group of size 4 arrives at time 11
customer group of size 4 orders 5 scoops of ice cream at time 13
customer group size 4 leaves at time 15
customer group of size 2 arrives at time 16
customer group of size 1 arrives at time 17
customer group of size 2 orders 2 scoops of ice cream at time 19
customer group of size 1 orders 1 scoops of ice cream at time 19
customer group size 1 leaves at time 22
```

```
...
customer group of size 2 orders 3 scoops of ice cream at time 136
customer group size 2 leaves at time 143
total profits are 26.95
```

15.5 Chapter Summary

A priority queue is not a queue at all, but is a data structure designed to permit rapid access and removal of the largest element in a collection. Priority queues can be structured by building them on top of lists, sets, vectors or deques. The vector or deque version of a priority queue is called a heap. The heap structure forms the basis of a very efficient sorting algorithm.

A skew heap is a form of heap that does not have the fixed size characteristic of the vector heap. The skew heap data structure is interesting in that it can potentially have a very poor worst case performance. However, it can be shown that the worst case performance cannot be maintained, and following any occurrence the next several operations of insertion or removal must be very rapid. Thus, when measured over several operations the performance of a skew heap is very impressive.

A common problem addressed using priority queues is the idea of discrete event-driven simulation. We have illustrated the use of heaps in an example simulation of an ice-cream store. In Chapter 19 we will once more use a priority queue in the development of an algorithm for computing the shortest path between pairs of points in a graph.

Key Concepts

- Priority Queue
- Heap
- Heap order property
- Heap sort
- Skew heap
- Discrete event driven simulation

References

The binary heap was first proposed by John Williams in conjunction with the heapsort algorithm [Williams 64]. Although heapsort is now considered to be one of the standard classic algorithms, a thorough theoretical analysis of the algorithm has proven to be surprisingly difficult. It was only in 1991 that the best case and average case execution time analysis of heapsort was reported [Schaffer 91].

Skew heaps were first described by Donald Sleator and Robert Tarjan in [Sleator 86]. An explanation of the amortized analysis of skew heaps is presented in [Weiss 92]. The ice cream store simulation is derived from a similar simulation in my earlier book on Smalltalk [Budd 87].

Study Questions

1. What is the primary characterization of a priority queue?
2. Why is a priority queue not a true queue?
3. How could a priority queue be constructed using a list?
4. What is the heap data structure?
5. What are the two most common uses of the term heap in computer science?
6. What is the heap order property?
7. What is a complete binary tree?
8. Give a vector of ten elements ordered largest to smallest, then show the corresponding complete binary tree. Why will the tree always be a heap?
9. What is the difference between the procedures `sort_heap` and `heap_sort`?
10. In the skew heap data structure, what is similar about the `push` and `pop` routines?
11. What is a discrete event-driven simulation?
12. What is a heterogeneous collection?
13. What is a polymorphic variable?

Exercises

1. Complete the design of a priority queue constructed using lists. That is, develop a data structure with the same interface as the `priority_queue` data type, but which uses lists as the underlying container and implements the interface using list operations.
2. While it is possible to implement iterators for the heap data structure, argue why it makes little sense to do so.
3. Give an example of a priority queue that occurs in a non-computer science situation.

4. Explain how the `skewHeap` data structure could be changed so that the smallest item was the value most easily accessible, instead of the largest value.

5. Show what a `heap` data structure looks like subsequent to insertions of each of the following values:

   4 2 5 8 3 6 1 10 14

6. Show what a `skewHeap` data structure looks like subsequent to insertions of each of the following values:

   4 2 5 8 3 6 1 10 14

7. Consider the following alternative implementation of `push_heap`:

```
template <class Iterator>
void push_heap (Iterator start, Iterator stop)
{
    unsigned int heapsize = stop - start;
    unsigned int position = heapsize - 1;

    // now restore the possibly lost heap property
    while (position > 0) {
        // reheap the subtree
        adjust_heap(data, heapsize, position);
        // move up the tree
        position = (position-1)/2;
        }
}
```

   (a) Prove that this algorithm will in fact successfully add a new element to the heap.

   (b) Explain why as a practical matter this algorithm is less desirable than the algorithm presented in the text.

8. Using the techniques described in Problem 4.3, Chapter 4 (page 86), test the hypothesis that heap sort is an $O(n \log n)$ algorithm. Using the coefficient c you compute, estimate for the heap sort algorithm how long it would take to sort a vector of 100,000 elements. Is your value in agreement with the actual time presented is the second table in Appendix B? Why do you think the values computed for small vectors represent a better predictor of performance for heap sort than did the equivalent analysis performed for tree sort?

9. Another heap-based sorting algorithm can be constructed using skew heaps. The idea is to simply copy the values from a vector into a skew heap, then copy the values one-by-one back out of the heap. Write the C++ procedure to do this.

10. Perform empirical timings on the algorithm you wrote for the previous question. Use as input vectors of various sizes containing random numbers. Compare the running time of this algorithm to that of tree sort and heap sort.

11. Design a simulation of an airport. The airport has two runways. Planes arrive from the air and request permission to land, and independently planes on the ground request permission to take off.

12. One alternative to the use of uniform distributions is the idea of a weighted discrete probability. Suppose we observe a real store and note that 65% of the time customers will order one scoop, 25% of the time they will order two scoops, and only 10% of the time will they order three scoops. This is certainly a far different distribution from the uniform distribution we used in the simulation. In order to simulate this behavior, we can add a new method to our class random.

    (a) Add a method named `weightedDiscrete` to the class `random`. This method will take, as an argument, a vector of unsigned integer values. For example, to generate the distribution above the programmer would pass the method a vector of three elements, containing 65, 25 and 10.

    (b) The method first sums the values in the array, resulting in a maximum value. In this case the value would be 100. A random number between 1 and this maximum value is then generated.

    (c) The method then decides in which category the number belongs. This can be discovered by looping through the values. In our example, if the number is less than 65, then the method should return 0 (remember, index values start at 0), if less than or equal to 90, return 1, and otherwise return 2.

13. Modify the ice cream store simulation so it uses the weighted discrete random number generated function implemented in the previous question. Select reasonable numbers for the weights. Compare a run of the resulting program to a run of the program using the uniform distribution.