Chapter 3

Priority Queues and HeapSort

In this chapter, we look at an Abstract Data Type known as a *priority queue*. Like a (normal) queue, a priority queue contains a collection of items that are waiting to be processed. In a queue, items are removed for processing in the same order in which they were added to the queue. In a priority queue, however, each item has a priority, and when it's time to select an item, the item with the highest priority is the one that is chosen. The priority might have nothing to do with the time at which the item was added to the data structure. Note that a priority queue is not, strictly speaking, a queue at all, since a queue is a first-in, first-out data structure and a priority queue is not.

Another way to describe a queue is as an implementation of a "first-come, first served" policy. The items in the queue wait in line in order of arrival. This is the right policy when all the items in the queue are of equal importance. In many situations, however, some items are more important than others, and in that case the more important items should be moved up in the line, ahead of the other items. A priority queue is an implementation of this idea. The priority of an item represents its importance, so items of higher priority should be selected before items of lower priority.

There are many possible applications of priority queues. For example, a priority queue could be used to hold packets of data waiting to be transmitted over a network. Packets that contain important data or data that must arrive in a timely way would be given higher priority and would therefor be transmitted ahead of lower priority data. In an operating system, a priority queue might hold "jobs," that is, programs that are waiting for execution. When processing time becomes available, a job would be removed from the priority queue for processing. Because of the way priority queues work, high priority jobs would be executed before low priority jobs, even if the low priority jobs have been waiting longer. Or a priority queue might contain jobs in a more literal sense—that is, tasks waiting to be assigned to workers as they become available. A computer help desk, for example, might use a priority queue to hold requests for help until tasks become available to process them. (Requests for help from, say, a College President might have a higher priority than a request from a student.)

Let's look at the priority queue as an abstract data type:

Definition 3.1. For any data type, *BaseType*, we define an abstract data type **Priority Queue of BaseType**. A possible value of this type is a collection of items of type *BaseType*, where each

item has an associated number, which is called its *priority*. The operations of the ADT are:

> **insert(x,p)**: Adds an item, x, of type *BaseType* to the priority queue with associated priority p (where p is a number). There is no return value.
>
> **remove()**: Removes from the priority queue an item that has the largest priority among all the items currently in the priority queue. If there are several items with the same maximum priority, it is not specified which one will be removed. It is an error to apply this operation to an empty priority queue.
>
> **makeEmpty()**: Removes all items from the priority queue. No return value.
>
> **isEmpty()**: Returns a value of type *boolean*, which is true if the stack contains no items and is false if there is at least one item on the stack.

There are some obvious implementations of the priority queue ADT. We could simply put the items, along with their priorities, into a list, implemented either as a linked list or as an array. If we keep the items in the list sorted into order of increasing priority, then the next item to be removed would always be at the end of the list and removing it would be a $\Theta(1)$ operation. However, inserting a new item into its correct place in the sorted list would be a $\Theta(n)$ operation, where n is the number of items in the list.

Perhaps we should **not** sort the items. Then a new item could simply be added at the end of the list, again a $\Theta(1)$ operation. However, if the items are not sorted, then finding and removing the item of highest priority takes $\Theta(n)$ time. Perhaps we are just stuck with a $\Theta(n)$ run time for at least one of the priority queue operations, no matter what implementation we use?

As it turns out, this is not the case. There is a very clever data structure called a *heap* that can be used to implement priority queues. (The term "heap" is also used to refer to the section of memory where dynamic variables are allocated, but the heap data structure is completely unrelated to this other use of the term.) In the heap implementation of priority queues, both the insert and remove operations have a worst-case run time that is $\Theta(\log(n))$, which is much better than $\Theta(n)$. Later in the chapter, we'll see that heaps are also used to implement an efficient sorting method known as *HeapSort*. HeapSort has a worst-case running time that is $\Theta(n * \log(n))$, much better than any of the three sorting algorithms that we saw in Chapter 1.

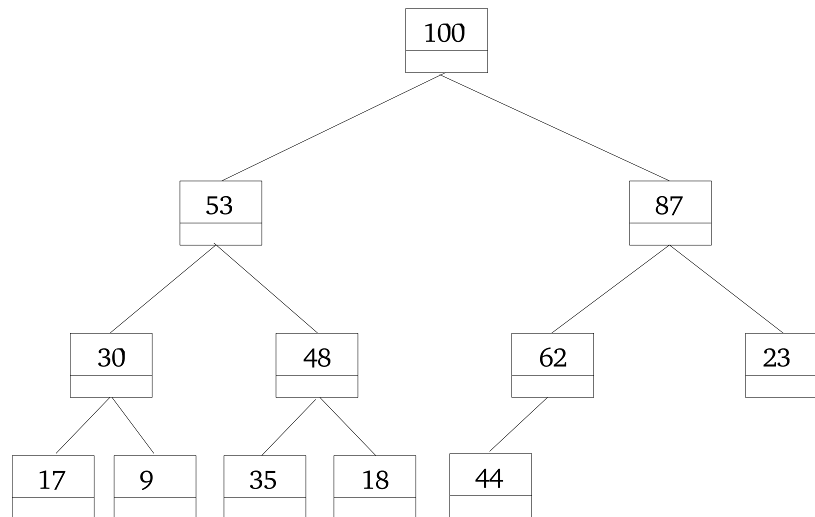
3.1 Heaps

Conceptually, a heap is a kind of binary tree. That is, it is a collection of nodes with a "root node" and in which each node can potentially have a "left child node" and a "right child node." A node is said to be the "parent node" of its child node. Every node in the binary tree, except for the root node, has exactly one parent node.

When a heap is used to implement a priority queue, each node contains one of the items in the queue along with the number that specifies the priority of that item. As far as the structure of the heap goes, it is only the priority, not the item itself, that is important. A heap must satisfy the **heap property**, which says that the priority of every node is greater than or equal to the priority of any child nodes of that node. Another way of saying this is that the priority of each non-root node must be less than or equal to the priority of its parent node. Note that the heap

property implies that the priority of any node is greater than or equal to the priorities of all its descendant nodes, not just the nodes immediately below it. In particular, the priority of the root node is greater than or equal to the priority of every other node in the heap. In terms of priority queues, the root node contains the item that is at the head of the queue.

A heap has one more important property: It is a **full** binary tree. In a full binary tree, there are no missing nodes in the interior of the tree. If you think of the tree being built up by adding nodes one at a time, then the nodes are added level-by-level from top to bottom, and within a level they are added from left to right. Here, for example, is a heap that contains twelve items. Only the priorities of the items are shown in the nodes:



Now, although a heap is conceptually a binary tree, the fact that it is a *full* binary tree makes it possible to physically represent the heap as a simpler data structure. In fact, a heap (or any full binary tree) can be represented as an array. All you have to do is line up the nodes in the array, starting with the root node, then the children of the root, then the grandchildren, and so on. If we do this with above heap, we get the following array:

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
100	53	87	30	48	62	23	17	9	35	18	44

The important thing about this array is that the structure of the binary tree can be described completely in terms of the array: The children of the node at index k in the array are located in the array at index $2 * k + 1$ and at index $2 * k + 2$, provided these numbers are within the range of indices of the array.[1] For example, the root node is at index 0, and its children are at indices $2 * 0 + 1$ and $2 * 0 + 2$, that is, at indices 1 and 2. The children of node 3 are node 7 $(2 * 3 + 1)$ and node 8 $(2 * 3 + 2)$. Node 5 has one child, at position 11 $(2 * 5 + 1)$; its other potential child, at position 12, lies outside the range of indices in the array. Node 7 has no children because both

[1] These formulas can be easily proved using induction.

$2*7+1$ and $2*7+2$ are outside the range of indices. Furthermore, if k is the index of a non-root node in the array, then the parent of that node is at index $(k-1)/2$ (where integer division is used, ignoring any remainder). Using these formulas for the positions of parent and child nodes in the array, we can represent a heap as an array but still think of the operations that we perform on the heap in terms of a binary tree.

In order to use a heap to implement a priority queue, we need to implement the heap's *insert* and *remove* operations in terms of operations on the heap. Remember that after we insert or remove an item, the resulting data structure must still be a full binary tree, and it must still have the heap property.

Assuming that *BaseType* is the type of items stored in the priority queue, we can implement the heap as an array of objects, where each object contains an item of type *BaseType* along with its associated priority. For generality, we should make it a dynamic array, but to keep things simple, let's make it a static array with a maximum capacity given by a constant `MAX_SIZE`. We also need an integer variable to keep track of the actual number of items currently in the queue; In C++, we can use the following data structure:

```
struct HeapItem {
   double priority;
   BaseType item;
};

HeapItem  heap[MAX_SIZE];  // The heap.
int itemCount;             // Number of items on the heap.
```

When we add a new item to the heap, a new node must be added in the next available position, at the end of the array. However, simply placing the new item in that node is likely to violate the heap property. That is, the new item might have a priority that is greater than the priority of its parent. The solution is simple: Place the new item in the next available spot, but then have a "contest" between the new item and its parent in the tree. If the new item has higher priority, swap it with its parent. Now, the new item might still be out of place in its new position, so have another contest between it and its new parent. Repeat this process until the new item either loses a contest (and therefore is in correct position in the heap) or reaches the root node. Using these ideas we can code the operation *insert(x,p)* as follows:

```
// Add the new item in the next available position in the array:

heap[itemCount].item = x;
heap[itemCount].priority = p;
itemCount++;

// The new item fights its way up the heap to its proper position:

int position = itemCount - 1;  // Current position of item.
```

```
while (true) {
   if (position == 0) {
         // Item is at the root of the tree; no place else to go.
      break;
   }
   int parent = (position - 1) / 2;  // Index in array of parent.
   if ( heap[position].priority <= heap[parent].priority ) {
         // The item loses the contest with its parent,
         // so it's in its correct position.
      break;
   }
   else {
         // The item has higher priority than its parent,
         // so swap the item with its parent and continue.
      HeapItem temp = heap[position];
      heap[position] = heap[parent];
      heap[parent] = temp;
      position = parent;  // Item is now in its parent's position.
   }
}
```

Suppose that the new item wins a contest against its parent. This means that its priority is greater than its parent's priority. After the swap, the previous parent becomes the child, and the new item has priority greater than the priority of that child. But what about the other child (if any) of the new item? To satisfy the heap property, it must have priority greater than or equal to the priorities of *both* its children. Can we be sure that this is the case? In fact, it follows from the fact that the original array had the heap property. The child in question must have a priority that is less than or equal to the priority of its original parent, which in turn is less than the priority of the new item. So, the new item has priority that is less than both the priority of the parent and the priority of the other child of that parent, and the new item belongs above *both* of these items in the heap. After the swap, that is just what we have, so the heap property is satisfied.

We can improve the code for the *insert* operation somewhat by noting that it's not really necessary to put the new item in the array and then repeatedly swap it with its parent. We can hold the contests without putting the new item in the array, moving the losers down out of the way as we go, and then place the new item in the last space that is vacated. The code for *insert(x,p)* then becomes:

```
int position = itemCount;
itemCount++;
while (position > 0 && p > data[(position-1)/2].priority) {
   data[position] = data[(position-1)/2];
   position = (position-1)/2;
```

```
}
data[position].item = x;
data[position].priority = p;
```

The number of times that the while loop can be executed in this code is no more than the number of levels in the binary tree. Since the number of items on each level of the tree is double the number on the previous level, the number of levels is approximately $\log_2(n)$, where n is the number of items in the tree. Another way to see this is to note that the initial value of *position* is in fact n, and that each execution of the while loop divides *position* by 2. We see that for the heap implementation of priority queues, the worst-case run time for the *insert* operation is $\Theta(\log(n))$, where n is the number of items in the queue.

Turing to the implementation of the *remove* operation on a heap, we note that the next item to be removed from the priority queue is always the one in the root node of the binary tree. Equivalently, the item to be removed is at index 0 in the array. However, we can't simply remove this item. We must make sure that after the *remove* operation is performed, the resulting data structure is still a heap. That is, it must be a full binary tree that satisfies the heap property. So, after removing the root item, we must restructure the heap.

We can proceed as follows: After removing the root item, move the item at the *bottom* of the heap (the end of the array) into the vacated root position. The size of the heap decreases by one. This preserves the full-binary-tree property, but it probably violates the heap property. We must rearrange the items in the heap to restore this property. Note that although the root item might be out of place, the rest of the tree—all the items below the root—still satisfy the heap property. To fix the problem, we will apply an operation that I call **heapify** to the root of the tree. *Heapify* assumes that all the descendants of a node satisfy the heap property, but the node itself might be out of place. To apply *heapify* to a node, compare the priority of that node to the priorities of its child nodes. If it has no child nodes, or if its priority is greater than or equal to the priorities of its children, there is nothing to do. Otherwise, swap the node with the child node that has the greater priority. Then apply the same operation to the new position of the node. *Heapify* could be written as a recursive function, but it is just as easy to write it as a while loop. We can now write the code to implement the *remove* operation:

```
// Get the item that is being removed from the root of the tree.

BaseType next = heap[0].item;

// Move the last item from the end of the array into the
// root position, and decrease the size of the heap by one.

heap[0] = heap[itemCount-1];
itemCount--;

// Perform the "heapify" operation on the root node.
```

```
int position = 0;  // Position of possibly out-of-place item.
while (true) {
    int child1 = 2*position + 1;
    int child2 = 2*position + 2;

    if (child1 >= itemCount) {
          // Both child positions are beyond the bounds of the
          // heap.  There is nothing to do.
       break;
    }

    int biggestChild; // Index of child with largest priority.
    if (child2 >= itemCount) {
          // There is no second child.
       biggestChild = child1;
    }
    else if {heap[child1].priority >= heap[child2].priority) {
          // child1 is the child with largest priority.
       biggestChild = child1;
    }
    else {
          // child2 is the child with largest priority.
       biggestChild = child2;
    }

    if (heap[biggestChild].priority <= heap[position].priority) {
          // The child nodes have smaller priority than the node.
       break;
    }

    ItemRecord temp = heap[position]; // Swap node with bigger child
    heap[position] = heap[biggestChild];
    heap[biggestChild] = temp;

    position = biggestChild;   // New position of the node.
}

return next;  // The return value of the remove operation.
```

As with *insert*, the number of times that the while loop in this code will execute is at most the number of levels in the binary tree. So with the heap implementation for priority queues, the worst-case run time for the *remove* operation is also $\Theta(\log(n))$, where n is the number of items on

the heap.

3.2 HeapSort

There is an obvious way to do sorting with priority queues: Take the items that you want to sort, and insert them into the priority queue (using the item itself as its own priority). Then remove items from the priority queue until it is empty. The items will come off the queue in order from largest to smallest. However, rather than work with priority queues, we can use heaps directly to implement the efficient sorting method known as *HeapSort*. For this application, a heap is simply an array of items, rather than an array of items with priorities. The heap property applies to the items themselves; that is, each item must be greater than or equal to its child items. Given this property, the root will contain the largest item in the heap.

To apply HeapSort to an array, we first rearrange the array into a heap. We can do this by applying *heapify* to each item the array, starting from the end of the array and working backwards. Note that when we get to a position k in the array, all the descendents of k have already been heapified, so the preconditions for applying *heapify* to position k have been met. In fact, we don't really have to start at the end of the array—we can start at the first array position that has children in the array. We start with the parent of the last element of the array. If there are n items in the array, this means we can start with position $(n-1)/2$.

Once the entire array has been heapified, we can remove items one-by-one from the heap, obtaining them in reverse order. We could simply save the items in a second array as we remove them from the first, but in fact the second array is not necessary! When we remove an item from the heap, the size of the part of the array occupied by the heap shrinks by one item. We just put the newly removed item in the space that has been vacated by the shrinking heap. The first item removed goes in the last array position, the second item removed goes in the next-to-last array position, and so on. The items are removed from the heap in descending order, and they are replaced in the array from the end of the array to the beginning. When the process is finished, all the items are arranged in *ascending* order, from the beginning of the array to the end.

Remember that removing an item from the heap means moving the last item into position 0 and then applying the *heapify* operation to position 0. In *HeapSort*, the newly removed item goes into the vacated space at the end of the heap. So, the operation becomes: Swap the last item in the heap with the item in position 0, decrease the heap size by one, and apply *heapify* to position 0. This operation is repeated until the heap is empty (though in practice, we can stop when the heap size is one, since the final item is already in its correct location at that time). The *HeapSort* algorithm for sorting an array of integers can be expressed in code as follows:

```
void heapify( int heap[], int size, int top ) {
      // Apply the heapify operation to position "top"
      // in a heap that contains "size" items.
   while (true) {
       int child1 = 2*top+1;
       int child2 = 2*top+2;
```

```
            if (child1 >= size)
               break;
            int biggestChild;
            if (child2 >= size || heap[child1] >= heap[child2])
               biggestChild = child1;
            else
               biggestChild = child2;
            if (heap[biggestChild] <= heap[top])
               break;
            int temp = heap[top];
            heap[top] = heap[biggestChild];
            heap[biggestChild] = temp;
            top = biggestChild;
      }
   }

   void heapsort( int list[], int n ) {
         // Apply heapsort to an array of n integers.
      for (int top = n/2; top >= 0; top--) { // Make array into a heap.
         heapify(list,n,top)
      }
      for (int top = n-1; top > 0; top--) {
            // Swap item 0 with item top, then apply heapify
            // to position 0.  Note that the value of top is
            // also the size of the heap when heapify is called.
         int temp = list[top];
         list[top] = list[0];
         list[0] = temp;
         heapify(list,top,0);
      }
   }
```

In this algorithm, *heapify* is called about $1.5 * n$ times. Each time it is called, the while loop in the *heapify* function is executed $\log(n)$ times or less. So, the worst-case total running time of all the calls to *heapify* is bounded by a constant times $n * \log(n)$. The total running time of the other lines in *heapsort* is bounded by a constant times n, which has a lower growth rate than $n * log(n)$. We see that the worst-case run time of *HeapSort* is $\Theta(n * \log(n))$.

Exercises

1. What is the best-case run time of *HeapSort*, and for which array does it occur?

2. Suppose that the following priorities are inserted into an initially empty heap, in the order given. Draw the heap as a binary tree as it appears after each insertion:

```
18  26  52  30  97  43  7  22
```

Now show what happens to the binary tree when the *remove* operation is applied.

3. Write a class to implement the abstract data type *priority queue of string*, and write a main program to test your class. (A better approach would be to write a template class to represent a priority queue of any base type.)