Chapter 5

Priority Queues

In many applications, we need data structures which support the efficient storage of data items and their retrieval in order of a pre-determined priority. Consider priority-based scheduling, for example. Jobs become available for execution at various times, and as jobs complete, we wish to schedule the available job having highest priority. These priorities may be assigned, for example, according to the relative importance of each job’s being executed in a timely manner. In order to support this form of storage and retrieval, we define a PriorityQueue as a set of items, each having an associated number giving its priority, together with the operations specified in Figure 5.1.

We sometimes wish to have operations MinPriority() and RemoveMin() instead of MaxPriority() and RemoveMax(). The specifications of these operations are the same as those of MaxPriority and RemoveMax, respectively, except that minimum priorities are used instead of maximum priorities. We call the resulting ADT an InvertedPriorityQueue. It is a straightforward matter to convert any implementation of PriorityQueue into an implementation of InvertedPriorityQueue.

In order to facilitate implementations of PriorityQueue, we will use a data structure Keyed for pairing data items with their respective priorities. This structure will consist of two readable representation variables, key and data. We will use a rather general interpretation, namely, that key and data are associated with each other. This generality will allow us to reuse the structure in later chapters with somewhat different contexts. Its structural invariant will simply be true. It will contain a constructor that takes two inputs, x and k, and produces an association with k as the key and x as the data. It will contain no additional operations. 

Strictly speaking, we should use a multiset, because we do not prohibit multiple occurrences of the same item. However, because we ordinarily would not insert multiple occurrences, we will call it a set.
5.1 Sorted Arrays

Our first implementation of PriorityQueue maintains the data in an expandable array, sorted in nondecreasing order of priorities. The representation consists of two variables:

- **elements[0..M − 1]**: an array of Keyed items, each containing a data item with its associated priority as its key, in order of priorities; and
- **size**: an integer giving the number of data items.

Implementation of the REMOVE operation is then trivial — after verifying that size is nonzero, we simply decrement size by 1 and then return elements[size].DATA(). Clearly, this can be done in Θ(1) time. Similarly, the MAX operation can be trivially implemented to run in Θ(1) time.
In order to implement $\text{Put}(x, p)$, we must find the correct place to insert $x$ so that the order of the priorities is maintained. Let us therefore reduce the $\text{Put}$ operation to the problem of finding the correct location to insert a given priority $p$. This location is the index $i$, $0 \leq i \leq \text{size}$, such that

- if $0 \leq j < i$, then $\text{elements}[j].\text{Key}() < p$; and
- if $i \leq j < \text{size}$, then $p \leq \text{elements}[j].\text{Key}()$.

Because the priorities are sorted, $\text{elements}[i].\text{Key}() = p$ iff there is an item in the array whose priority is $p$. Furthermore, if no such item exists, $i$ gives the location at which such an item should be inserted.

We can apply the top-down approach to derive a search technique called \textit{binary search}. Assume we are looking for the insertion point in an array $A[lo..hi-1]$; i.e., the insertion point $i$ will be in the range $lo \leq i \leq hi$. Further assume that $lo < hi$, for otherwise, we must have $lo = i = hi$. Recall that the divide-and-conquer technique, introduced in Section 3.7, reduces large instances to smaller instances that are a fixed fraction of the size of the original instance. In order to apply this technique, we first look at the priority of the middle data item — the item with index $\text{mid} = \lfloor (lo + hi)/2 \rfloor$. If the key of this item is greater than or equal to $p$, then $i$ can be no greater than $\text{mid}$, which in turn is strictly less than $hi$. Otherwise, $i$ must be strictly greater than $\text{mid}$, which in turn is greater than or equal to $lo$. We will therefore have reduced our search to a strictly smaller search containing about half the elements from the original search.

Note that this reduction is actually a transformation — a reduction in which the solution to the smaller problem is exactly the solution to the original problem. Recall from Section 2.4 that a transformation can be implemented as a loop in a fairly straightforward way. Specifically, each iteration of the loop will reduce a large instance to a smaller instance. When the loop terminates, the instance will be the base case, where $lo = hi$.

Prior to the loop, $lo$ and $hi$ must have values 0 and $\text{size}$, respectively. Our invariant will be that $0 \leq lo \leq hi \leq \text{size}$, that items with indices less than $lo$ have a key less than $p$, and that elements with indices greater than or equal to $hi$ have a key greater than or equal to $p$. Thus, the index $i$ to be returned will always be in the range $lo \leq i \leq hi$. When the loop terminates, we will have $lo = hi$; hence, we can return either $lo$ or $hi$. This algorithm is given as the $\text{Find}$ function in Figure 5.2, where a partial implementation of $\text{SortedArrayPriorityQueue}$ is given. The $\text{Expand}$ function copies the contents of its argument into an array of twice the original size, as in
Figure 5.2 SortedArrayPriorityQueue implementation (partial) of the PriorityQueue ADT

Structural Invariant: $0 \leq \text{size} \leq \text{SizeOf}(\text{elements})$, where elements is an array of Keyed items whose keys are numbers in nondecreasing order.

SortedArrayPriorityQueue.Put$(x, p : \text{Number})$

$i \leftarrow \text{Find}(p)$

if $\text{size} = \text{SizeOf}(\text{elements})$

$\text{elements} \leftarrow \text{Expand}(\text{elements})$

for $j \leftarrow \text{size} - 1$ to $i$ by $-1$

$\text{elements}[j + 1] \leftarrow \text{elements}[j]$

$\text{elements}[i] \leftarrow \text{new Keyed}(x, p); \text{size} \leftarrow \text{size} + 1$

— Internal Functions Follow —

Precondition: The structural invariant holds, and $p$ is a Number.

Postcondition: Returns the index $i$, $0 \leq i \leq \text{size}$, such that if $0 \leq j < i$, then $\text{elements}[j].\text{Key}() < p$ and if $i \leq j < \text{size}$, then $p \leq \text{elements}[j].\text{Key}()$.

SortedArrayPriorityQueue.Find$(p)$

$lo \leftarrow 0; hi \leftarrow \text{size}$

// Invariant: $0 \leq lo \leq hi \leq \text{size}$,

// if $0 \leq j < lo$, then $\text{elements}[j].\text{Key}() < p$,

// and if $hi \leq j < \text{size}$, then $\text{elements}[j].\text{Key}() \geq p$.

while $lo < hi$

$mid \leftarrow \lfloor (lo + hi)/2 \rfloor$

if $\text{elements}[mid].\text{Key}() \geq p$

$hi \leftarrow mid$

else

$lo \leftarrow mid + 1$

return $lo$
Section 4.3. The remainder of the implementation and its correctness proof are left as an exercise.

Let us now analyze the running time of \textsc{Find}. Clearly, each iteration of the \textbf{while} loop runs in $\Theta(1)$ time, as does the code outside the loop. We therefore only need to count the number of iterations of the loop.

Let $f(n)$ denote the number of iterations, where $n = hi - lo$ gives the number of elements in the search range. One iteration reduces the number of elements in the range to either $\lfloor n/2 \rfloor$ or $\lceil n/2 \rceil - 1$. The former value occurs whenever the key examined is greater than or equal to $p$. The worst case therefore occurs whenever we are looking for a key smaller than any key in the set. In the worst case, the number of iterations is therefore given by the following recurrence:

$$f(n) = f(\lfloor n/2 \rfloor) + 1$$

for $n > 1$. From Theorem 3.32, $f(n) \in \Theta(\lg n)$. Therefore, \textsc{Find} runs in $\Theta(\lg n)$ time.

Let us now analyze the running time of \textsc{Put}. Let $n$ be the value of \textit{size}. The first statement requires $\Theta(\lg n)$ time, and based on our analysis in Section 4.3, the \textsc{Expand} function should take $O(n)$ time in the worst case. Because we can amortize the time for \textsc{Expand}, let us ignore it for now. Clearly, everything else outside the \textbf{for} loop and a single iteration of the loop run in $\Theta(1)$ time. Furthermore, in the worst case (which occurs when the new key has a value less than all other keys in the set), the loop iterates $n$ times. Thus, the entire algorithm runs in $\Theta(n)$ time in the worst case, regardless of whether we count the time for \textsc{Expand}.

5.2 Heaps

The \texttt{SortedArrayPriorityQueue} has very efficient $\textsc{MaxPriority}$ and $\textsc{RemoveMax}$ operations, but a rather slow $\textsc{Put}$ operation. We could speed up the $\textsc{Put}$ operation considerably by dropping our requirement that the array be sorted. In this case, we could simply add an element at the end of the array, expanding it if necessary. This operation is essentially the same as the \texttt{ExpandableArrayStack.Push} operation, which has an amortized running time in $\Theta(1)$. However, we would no longer be able to take advantage of the ordering of the array in finding the maximum priority. As a result, we would need to search the entire array. The running times for the $\textsc{MaxPriority}$ and $\textsc{RemoveMax}$ operations would therefore be in $\Theta(n)$ time, where $n$ is the number of elements in the priority queue.
In order to facilitate efficient implementations of all three operations, let us try applying the top-down approach to designing an appropriate data structure. Suppose we have a nonempty set of elements. Because we need to be able to find and remove the maximum priority quickly, we should keep track of it. When we remove it, we need to be able to locate the new maximum quickly. We can therefore organize the remaining elements into two (possibly empty) priority queues. (As we will see, using two priority queues for these remaining elements can yield significant performance advantages over a single priority queue.) Assuming for the moment that both of these priority queues are nonempty, the new overall maximum must be the larger of the maximum priorities from each of these priority queues. We can therefore find the new maximum by comparing these two priorities. The cases in which one or both of the two priority queues are empty are likewise straightforward.

We can implement the above idea by arranging the priorities into a heap, as shown in Figure 5.3. This structure will be the basis of all of the remaining PriorityQueue implementations presented in this chapter. In this figure, integer priorities of several data items are shown inside circles, which we will call nodes. The structure is referenced by its root node, containing the priority 89. This value is the maximum of the priorities in the structure. The remaining priorities are accessed via one of two references, one leading to the left, and the other leading to the right. Each of these two groups of priorities forms a priority queue structured in a similar way. Thus, as we follow any path downward in the heap, the values of the priorities are nonincreasing.

A heap is a special case of a more general structure known as a tree. Let $N$ be a finite set of nodes, each containing a data item. We define a rooted tree comprised of $N$ recursively as:

- a special object which we will call the empty tree if $N = \emptyset$; or
- a root node $x \in N$, together with a finite sequence $\langle T_1, \ldots, T_k \rangle$ of children, where
  - each $T_i$ is a rooted tree comprised of some (possibly empty) set $N_i \subseteq N \setminus \{x\}$ (i.e., each element of each $N_i$ is an element of $N$, but not the root node);
  - $\bigcup_{i=1}^k N_i = N \setminus \{x\}$ (i.e., the elements in all of the sets $N_i$ together form the set $N$, without the root node); and
  - for $i \neq j$, $N_i \cap N_j = \emptyset$ (i.e., no two of these sets have any elements in common).
Thus, the structure shown in Figure 5.3 is a rooted tree comprised of 10 nodes. Note that the data items contained in the nodes are not all distinct; however, the nodes themselves are distinct. The root node contains 89. Its first child is a rooted tree comprised of six nodes and having a root node containing 53.

When a node $c$ is a child of a node $p$, we say that $p$ is the parent of $c$. Also, two children of a given node are called siblings. Thus, in Figure 5.3, the node containing 48 has one node containing 53 as its parent and another as its sibling. We refer to a nonempty tree whose children are all empty as a leaf. Thus, in Figure 5.3, the subtree whose root contains 13 is a leaf.

We define the size of a rooted tree to be the number of nodes which comprise it. Another important measure of a nonempty rooted tree is its height, which we define recursively to be one plus the maximum height of its nonempty children; if it has no nonempty children, we say its height is 0. Thus, the height is the maximum of the distances from the root to the leaves, where we define the distance as the number of steps we must take to get from one node to the other. For example, the tree in Figure 5.3 has size 10 and height 4.

When we draw rooted trees, we usually do not draw empty trees. For
example, in Figure 5.3, the subtree whose root contains 24 has two children, but the first is empty. This practice can lead to ambiguity; for example, it is not clear whether the subtree rooted at 13 contains any children, or if they might all be empty. For this and other reasons, we often consider restricted classes of rooted trees. Here, we wish to define a binary tree as a rooted tree in which each nonempty subtree has exactly two children, either (or both) of which may be empty. In a binary tree, the first child is called the left child, and the other is called the right child. If we then state that the rooted tree in Figure 5.3 is a binary tree, it is clear that the subtree rooted at 13, because it is nonempty, has two empty children.

It is rather difficult to define an ADT for either trees or binary trees in such a way that it can be implemented efficiently. The difficulty is in enforcing as a structural invariant the fact that no two children have nodes in common. In order for an operation to maintain this invariant when adding a new node, it would apparently need to examine the entire structure to see if the new node is already in the tree. As we will see, maintaining this invariant becomes much easier for specific applications of trees. It therefore seems best to think of a rooted tree as a mathematical object, and to mimic its structure in defining a heap implementation of PriorityQueue.

In order to build a heap, we need to be able to implement a single node. For this purpose, we will define a data type BinaryTreeNode. Its representation will contain three variables:

- **root**: the Keyed data item stored in the node;
- **leftChild**: the BinaryTreeNode representing the root of the left child; and
- **rightChild**: the BinaryTreeNode representing the root of the right child.

We will provide read/write access to all three of these variables, and our structural invariant is simply true. The only constructor is shown in Figure 5.4, and no additional operations are included. Clearly, BinaryTreeNode meets its specification (there is very little specified), and each operation and constructor runs in $\Theta(1)$ time.

We can now formally define a heap as a binary tree containing Keyed elements such that if the tree is nonempty, then

- the item stored at the root has the maximum key in the tree; and
- both children are heaps.
Based on the above definition, we can define a representation for PRIORITY-QUEUE using two variables:

- elements: a BINARY-TREE-NODE; and
- size: a natural number.

Our structural invariant will be that elements is a heap whose size is given by size. We interpret the contents of the nodes comprising this heap as the set of items stored in the priority queue, together with their associated priorities.

Implementation of MAX-PRIORITY is now trivial — we just return the key of the root. To implement REMOVE-MAX, we must remove the root (provided the heap is nonempty) and return the data from its contents. When we remove the root, we are left with the two children, which must then be combined into one heap. We therefore will define an internal function MERGE, which takes as input two heaps $h_1$ and $h_2$ with no nodes in common (i.e., the two heaps share no common structure, though they may have keys in common), and returns a single heap containing all of the nodes from $h_1$ and $h_2$. Note that we can also use the MERGE function to implement PUT if we first construct a single-node heap from the element we wish to insert.

Let us consider how to implement MERGE. If either of the two heaps $h_1$ and $h_2$ is nil (i.e., empty), we can simply return the other heap. Otherwise, the root of the result must be the root of either $h_1$ or $h_2$, whichever root contains a KEYED item with larger key (a tie can be broken arbitrarily). Let $L$ denote the heap whose root contains the maximum key, and let $S$ denote the other heap. Then we must form a heap whose root is the root of $L$ and whose two children are heaps containing the nodes in the following three heaps:

- the left child of $L$;
• the right child of $L$; and

• $S$.

We can form these two children by recursively merging two of these three heaps.

A simple implementation, which we call SimpleHeap, is shown in Figure 5.5. Note that we can maintain the structural invariant because we can ensure that the precondition to Merge is always met (the details are left as an exercise). Note also that the above discussion leaves some flexibility in the implementation of Merge. In fact, we will see shortly that this particular implementation performs rather poorly. As a result, we will need to find a better way of choosing the two heaps to merge in the recursive call, and/or a better way to decide which child the resulting heap will be.

Let us now analyze the running time of Merge. Suppose $h_1$ and $h_2$ together have $n$ nodes. Clearly, the running time excluding the recursive call is in $\Theta(1)$. In the recursive call, $L$.RightChild() has at least one fewer node than does $L$; hence the total number of nodes in the two heaps in the recursive call is no more than $n - 1$. The total running time is therefore bounded above by

\[
    f(n) \in f(n - 1) + O(1) \\
    \subseteq O(n)
\]

by Theorem 3.31.

At first it might seem that the bound of $n - 1$ on the number of nodes in the two heaps in the recursive call is overly pessimistic. However, upon close examination of the algorithm, we see that not only does this describe the worst case, it actually describes every case. To see this, notice that nowhere in the algorithm is the left child of a node changed after that node is created. Because each left child is initially empty, no node ever has a nonempty left child. Thus, each heap is single path of nodes going to the right.

The SimpleHeap implementation therefore amounts to a linked list in which the keys are kept in nonincreasing order. The Put operation will therefore require $\Theta(n)$ time in the worst case, which occurs when we add a node whose key is smaller than any in the heap. In the remainder of this chapter, we will examine various ways of taking advantage of the branching potential of a heap in order to improve the performance.
**Figure 5.5** SimpleHeap implementation of PriorityQueue

**Structural Invariant:** elements is a heap whose size is given by size.

SimpleHeap()

\[
elements \leftarrow \text{nil}; \quad \text{size} \leftarrow 0
\]

SimpleHeap.Put(x, p : Number)

\[
h \leftarrow \text{new BinaryTreeNode}(); \quad h.\text{SetRoot(new Keyed}(x, p))
\]

\[
elements \leftarrow \text{Merge}(\text{elements}, h); \quad \text{size} \leftarrow \text{size} + 1
\]

SimpleHeap.MaxPriority()

\[
\text{return elements.Root().Key()}
\]

SimpleHeap.RemoveMax()

\[
x \leftarrow \text{elements.Root().Data}(); \quad \text{size} \leftarrow \text{size} - 1
\]

\[
elements \leftarrow \text{Merge}(\text{elements.LeftChild()}, \text{elements.RightChild}())
\]

\[
\text{return } x
\]

— Internal Functions Follow —

**Precondition:** \(h_1\) and \(h_2\) are (possibly nil) BinaryTreeNodes representing heaps with no nodes in common.

**Postcondition:** Returns a heap containing all of the nodes in \(h_1\) and \(h_2\).

SimpleHeap.Merge(h1, h2)

\[
\text{if } h_1 = \text{nil } \quad \text{return } h_2
\]

\[
\text{else if } h_2 = \text{nil } \quad \text{return } h_1
\]

\[
\text{else } \quad \text{if } h_1.\text{Root().Key()} > h_2.\text{Root().Key()}
\]

\[
L \leftarrow h_1; \quad S \leftarrow h_2
\]

\[
\text{else } \quad \quad L \leftarrow h_2; \quad S \leftarrow h_1
\]

\[
L.\text{SetRightChild(Merge(L.RightChild()), S))}
\]

\[
\text{return } L
\]
5.3 Leftist Heaps

In order to improve the performance of merging two heaps, it would make sense to try to reach one of the base cases as quickly as possible. In SimpleHeap.Merge, the base cases occur when one of the two heaps is empty. In order to simplify the discussion, let us somewhat arbitrarily decide that one of the two heaps to be merged in the recursive call will always be $S$. We therefore need to decide which child of $L$ to merge with $S$. In order to reach a base case as quickly as possible, it would make sense to use the child having an empty subtree nearest to its root.

Let us define, for a given binary tree $T$, the null path length to be the length of the shortest path from the root to an empty subtree. Specifically, if $T$ is empty, then its null path length is 0; otherwise, it is 1 plus the minimum of the null path lengths of its children. Now if, in the recursive call, we were to merge $S$ with the child of $L$ having smaller null path length, then the sum of the null path lengths of the two heaps would always be smaller for the recursive call than for the original call. The running time is therefore proportional to the sum of the null path lengths. This is advantageous due to the following theorem.

**Theorem 5.1** For any binary tree $T$ with $n$ nodes, the null path length of $T$ is at most $\lg(n + 1)$.

The proof of this theorem is typical of many proofs of properties of trees. It proceeds by induction on $n$ using the following general strategy:

- For the base case, prove that the property holds when $n = 0$ — i.e., for an empty tree.
- For the induction step, apply the induction hypothesis to one or more of the children of a nonempty tree.

**Proof of Theorem 5.1:** By induction on $n$.

**Base:** $n = 0$. Then by definition, the null path length is $0 = \lg 1$.

**Induction Hypothesis:** Assume for some $n > 0$ that for $0 \leq i < n$, the null path length of any tree with $i$ nodes is at most $\lg(i + 1)$.

**Induction Step:** Let $T$ be a binary tree with $n$ nodes. Then because the two children together contain $n - 1$ nodes, they cannot both contain more
than \((n - 1)/2\) nodes; hence, one of the two children has no more than \(\lfloor (n - 1)/2 \rfloor\) nodes. By the induction hypothesis, this child has a null path of at most \(\lg(\lfloor (n - 1)/2 \rfloor + 1)\). The null path length of \(T\) is therefore at most
\[
1 + \lg(\lfloor (n - 1)/2 \rfloor + 1) \leq 1 + \lg((n + 1)/2) = \lg(n + 1). 
\]

By the above theorem, if we can always choose the child with smaller null path length for the recursive call, then the merge will operate in \(O(\lg n)\) time, where \(n\) is the number of nodes in the larger of the two heaps. We can develop slightly simpler algorithms if we build our heaps so that the right-hand child always has the smaller null path length, as in Figure 5.6(a). We therefore define a leftist tree to be a binary tree which, if nonempty, has two leftist trees as children, with the right-hand child having a null path length no larger than that of the left-hand child. A leftist heap is then a leftist tree that is also a heap.

In order to implement a leftist heap, we will use an implementation of a leftist tree. The leftist tree implementation will take care of maintaining the proper shape of the tree. Because we will want to combine leftist trees to form larger leftist trees, we must be able to handle the case in which two given leftist trees have nodes in common. The simplest way to handle this situation is to define the implementation to be an immutable structure. Because no changes can be made to the structure, we can treat all nodes as distinct, even if they are represented by the same storage (in which case they are the roots of identical trees).

In order to facilitate fast computation of null path lengths, we will record the null path length of a leftist tree in one of its representation variables. Thus, when forming a new leftist tree from a root and two existing leftist trees, we can simply compare the null path lengths to decide which tree should be used as the right child. Furthermore, we can compute the null path length of the new leftist tree by adding 1 to the null path length of its right child.

For our representation of \texttt{LeftistTree}, we will therefore use four variables:

- \texttt{root}: a \texttt{Keyed} item;
- \texttt{leftChild}: a \texttt{LeftistTree};
- \texttt{rightChild}: a \texttt{LeftistTree}; and
Figure 5.6 Example of performing a `LeftistHeap.RemoveMax` operation

(a) The original heap
(b) Remove the root (20) and merge the smaller of its children (13) with the right child of the larger of its children (7)

(c) Make 13 the root of the subtree and merge the tree rooted at 7 with the empty right child of 13
(d) Because 13 has a larger null path length than 10, swap them
• **nullPathLength**: a `Nat`.

We will allow read access to all variables. Our structural invariant will be that this structure is a leftist tree such that

- **nullPathLength** gives its null path length; and
- **root = nil** iff **nullPathLength = 0**.

Specifically, we will allow the same node to occur more than once in the structure — each occurrence will be viewed as a copy. Because the structure is immutable, such sharing is safe. The implementation of **LeftistTree** is shown in Figure 5.7. Clearly, each of these constructors runs in $\Theta(1)$ time.

We now represent our **LeftistHeap** implementation of **PriorityQueue** using two variables:

- **elements**: a **LeftistTree**; and
- **size**: a natural number.

Our structural invariant is that **elements** is a leftist heap whose size is given by **size**, and whose nodes are **KEYED** items. We interpret these **KEYED** items as the represented set of elements with their associated priorities. The implementation of **LeftistHeap** is shown in Figure 5.8.

Based on the discussion above, **Merge** runs in $O(\lg n)$ time, where $n$ is the number of nodes in the larger of the two leftist heaps. It follows that **Put** and **RemoveMax** operate in $O(\lg n)$ time, where $n$ is the number of items in the priority queue. Though it requires some work, it can be shown that the lower bound for each of these running times is in $\Omega(\lg n)$.

**Example 5.2** Consider the leftist heap shown in Figure 5.6(a). Suppose we were to perform a **RemoveMax** on this heap. To obtain the resulting heap, we must merge the two children of the root. The larger of the two keys is 15; hence, it becomes the new root. We must then merge its right child with the original right child of 20 (see Figure 5.6(b)). The larger of the two roots is 13, so it becomes the root of this subtree. The subtree rooted at 7 is then merged with the empty right child of 13. Figure 5.6(c) shows the result without considering the null path lengths. We must therefore make sure that in each subtree that we’ve formed, the null path length of the right child is no greater than the null path length of the left child. This is the case for the subtree rooted at 13, but not for the subtree rooted at 15. We therefore must swap the children of 15, yielding the final result shown in Figure 5.6(d).
Figure 5.7 The \texttt{LeftistTree} data structure.

\textbf{Structural Invariant:} This structure forms a leftist tree with \texttt{nullPathLength} giving its null path length, and \texttt{root = nil} iff \texttt{nullPathLength} = 0.

\textbf{Precondition:} true.

\textbf{Postcondition:} Constructs an empty \texttt{LeftistTree}.

\begin{verbatim}
\texttt{LeftistTree()}
    root ← nil; leftChild ← nil; rightChild ← nil; nullPathLength ← 0
\end{verbatim}

\textbf{Precondition:} \texttt{x} is a non-nil \texttt{Keyed} item.

\textbf{Postcondition:} Constructs a \texttt{LeftistTree} containing \texttt{x} at its root and having two empty children.

\begin{verbatim}
\texttt{LeftistTree(x : Keyed)}
    if \texttt{x = nil}
        error
    else
        root ← \texttt{x}; nullPathLength ← 1
        leftChild ← \texttt{new LeftistTree()}
        rightChild ← \texttt{new LeftistTree()}
\end{verbatim}

\textbf{Precondition:} \texttt{x}, \texttt{t} \texttt{1}, and \texttt{t} \texttt{2} are non-nil, \texttt{x} is \texttt{Keyed}, and \texttt{t} \texttt{1} and \texttt{t} \texttt{2} are \texttt{LeftistTrees}.

\textbf{Postcondition:} Constructs a \texttt{LeftistTree} containing \texttt{x} at its root and having children \texttt{t} \texttt{1} and \texttt{t} \texttt{2}.

\begin{verbatim}
\texttt{LeftistTree(x : Keyed, t} \texttt{1 : LeftistTree, t} \texttt{2 : LeftistTree)}
    if \texttt{x = nil or t} \texttt{1 = nil or t} \texttt{2 = nil}
        error
    else if \texttt{t} \texttt{1.nullPathLength \geq t} \texttt{2.nullPathLength}
        leftChild ← \texttt{t} \texttt{1}; rightChild ← \texttt{t} \texttt{2}
    else
        leftChild ← \texttt{t} \texttt{2}; rightChild ← \texttt{t} \texttt{1}
        root ← \texttt{x}; nullPathLength ← 1 + rightChild.nullPathLength
\end{verbatim}
**Figure 5.8** LeftistHeap implementation of PriorityQueue

**Structural Invariant:** elements is a leftist heap whose size is given by size and whose nodes are Keyed items.

```
LEFTISTHEAP()
    elements ← new LEFTISTTREE(); size ← 0

LEFTISTHEAP.PUT(x, p : NUMBER)
    elements ← Merge(elements, new LEFTISTTREE(new Keyed(x, p)))
    size ← size + 1

LEFTISTHEAP.MAXPRIORITY()
    return elements.Root().Key()

LEFTISTHEAP.REMOVEMAX()
    x ← elements.Root().Data()
    elements ← Merge(elements.LeftChild(), elements.RightChild())
    size ← size − 1
    return x

— Internal Functions Follow —

**Precondition:** h₁ and h₂ are LEFTISTTREES storing heaps.

**Postcondition:** Returns a LEFTISTTREE containing the elements of h₁ and h₂ in a heap.

```
LEFTISTHEAP.MERGE(h₁, h₂)
    if h₁.Root() = nil
        return h₂
    else if h₂.Root() = nil
        return h₁
    else if h₁.Root().Key() > h₂.Root().Key()
        L ← h₁; S ← h₂
    else
        L ← h₂; S ← h₁
    t ← Merge(L.RightChild(), S)
    return new LEFTISTTREE(L.Root(), L.LeftChild(), t)
```
A **Put** operation is performed by creating a single-node heap from the element to be inserted, then merging the two heaps as in the above example. The web site that accompanies this textbook contains a program for viewing and manipulating various kinds of heaps, including leftist heaps and the heaps discussed in the remainder of this chapter. This heap viewer can be useful for generating other examples in order to understand the behavior of heaps.

It turns out that in order to obtain \(O(lg n)\) worst-case performance, it is not always necessary to follow the shortest path to a nonempty subtree. For example, if we maintain a tree such that for each of its \(n\) nodes, the left child has at least as many nodes as the right child, then the distance from the root to the rightmost subtree is still no more than \(lg(n + 1)\). As a result, we can use this strategy for obtaining \(O(lg n)\) worst-case performance for the **PriorityQueue** operations (see Exercise 5.7 for details). However, we really don't gain anything from this strategy, as it is now necessary to maintain the size of each subtree instead of each null path length. In the next two sections, we will see that it is possible to achieve good performance without maintaining any such auxiliary information.

### 5.4 Skew Heaps

In this section, we consider a simple modification to **SimpleHeap** that yields good performance without the need to maintain auxiliary information such as null path lengths. The idea is to avoid the bad performance of **SimpleHeap** by modifying **Merge** to swap the children after the recursive call. We call this modified structure a *skew heap*. The **Merge** function for **SkewHeap** is shown in Figure 5.9; the remainder of the implementation of **SkewHeap** is the same as for **SimpleHeap**.

**Example 5.3** Consider again the heap shown in Figure 5.6(a), and suppose it is a skew heap. Performing a **REMOVE** on this heap proceeds as shown in Figure 5.6 through part (e). At this point, however, for each node at which a recursive **Merge** was performed, the children of this node are swapped. These nodes are 13 and 15. The resulting heap is shown in Figure 5.10.

In order to understand why such a simple modification might be advantageous, observe that in **Merge**, when \(S\) is merged with \(L\).\text{RIGHT}(), we might expect the resulting heap to have a tendency to be larger than \(L\).\text{LEFT}(). As we noted at the end of the previous section, good
Figure 5.9 The SkewHeap.Merge internal function

Precondition: $h_1$ and $h_2$ are (possibly nil) BinaryTreeNodes representing heaps with no nodes in common.

Postcondition: Returns a heap containing all of the nodes in $h_1$ and $h_2$.

SkewHeap.Merge($h_1, h_2$)
  if $h_1 = \text{nil}$
    return $h_2$
  else if $h_2 = \text{nil}$
    return $h_1$
  else
    if $h_1.\text{Root}().\text{Key}() > h_2.\text{Root}().\text{Key}()$
      $L \leftarrow h_1; S \leftarrow h_2$
    else
      $L \leftarrow h_2; S \leftarrow h_1$
      $t \leftarrow \text{merge}(L.\text{RightChild}(), S)$
      $L.\text{SetRightChild}(L.\text{LeftChild}()); L.\text{SetLeftChild}(t)$
  return $L$

worst-case behavior can be obtained by ensuring that the left child of each node has at least as many nodes as the right child. Intuitively, we might be able to approximate this behavior by swapping the children after every recursive call. However, this swapping does not always avoid expensive operations.

Suppose, for example, that we start with an empty skew heap, then insert the sequence of keys $2, 1, 4, 3, \ldots, 2i, 2i - 1, 0$, for some $i \geq 1$. Figure 5.11 shows this sequence of insertions for $i = 3$. Note that each time an even key is inserted, because it is the largest in the heap, it becomes the new root and the original heap becomes its left child. Then when the next key is inserted, because it is smaller than the root, it is merged with the empty right child, then swapped with the other child. Thus, after each odd key is inserted, the heap will contain all the even keys in the rightmost path (i.e., the path beginning at the root and going to the right until it reaches an empty subtree), and for $i \geq 1$, key $2i$ will have key $2i - 1$ as its left child.

Finally, when key 0 is inserted, because it is the smallest key in the heap, it will successively be merged with each right child until it is merged with the empty subtree at the far right. Each of the subtrees on this path to the
right is then swapped with its sibling. Clearly, this last insertion requires \( \Theta(i) \) running time, and \( i \) is proportional to the number of nodes in the heap.

The bad behavior described above results because a long rightmost path is constructed. Note, however, that \( 2^i \) \text{PUT} operations were needed to construct this path. Each of these operations required only \( \Theta(1) \) time. Furthermore, after the \( \Theta(i) \) operation, no long rightmost paths exist from any node in the heap (see Figure 5.11). This suggests that a skew heap might have good amortized running time.

A good measure of the actual cost of the \texttt{SkewHeap} operations is the number of calls to \texttt{Merge}, including recursive calls. In order to derive a bound on the amortized cost, let us try to find a good potential function. Based upon the above discussion, let us say that a node is \textit{good} if its left child has at least as many nodes as its right child; otherwise, it is \textit{bad}. We now make two key observations, whose proofs are left as exercises:

- In any binary tree with \( n \) nodes, the number of good nodes in the rightmost path is no more than \( \log(n + 1) \).

- In the \texttt{Merge} function, if \( L \) is a bad node initially, it will be a good node in the resulting heap.

Due to these observations, we use as our potential function the number of bad nodes in the heap. Because the number of good nodes in each of
the two rightmost paths is logarithmic, the potential function can increase by only a logarithmic amount on any MERGE. Furthermore, because any bad node encountered becomes good, the resulting change in potential will cancel the actual cost associated with this call, leaving only a logarithmic number of calls whose actual costs are not canceled. As a result, we should expect the amortized costs of the SKHEAP operations to be in $O(\lg n)$, where $n$ is the number of elements in the priority queue (the details of the analysis are left as an exercise). Thus, a SKHEAP provides a simple, yet efficient, implementation of PriorityQueue.
**Figure 5.12** The RandomizedHeap.Merge internal function

**Precondition:** \( h_1 \) and \( h_2 \) are (possibly nil) BinaryTreeNode objects representing heaps with no nodes in common.

**Postcondition:** Returns a heap containing all of the nodes in \( h_1 \) and \( h_2 \).

\[
\text{RandomizedHeap.Merge}(h_1, h_2)
\]

\[
\begin{align*}
&\text{if } h_1 = \text{nil} \\
&\quad \text{return } h_2 \\
&\text{else if } h_2 = \text{nil} \\
&\quad \text{return } h_1 \\
&\text{else} \\
&\quad \text{if } h_1.\text{Root().Key()} > h_2.\text{Root().Key()} \\
&\quad \quad L \leftarrow h_1; S \leftarrow h_2 \\
&\quad \text{else} \\
&\quad \quad L \leftarrow h_2; S \leftarrow h_1 \\
&\quad \text{if } \text{FlipCoin()} = \text{heads} \\
&\quad \quad L.\text{SetLeftChild(}\text{Merge}(L.\text{LeftChild()}, S)) \\
&\quad \text{else} \\
&\quad \quad L.\text{SetRightChild(}\text{Merge}(L.\text{RightChild()}, S)) \\
&\text{return } L
\end{align*}
\]

### 5.5 Randomized Heaps

For all of the heap implementations we have seen so far, the merge uses the right child in the recursive call. This choice is not necessary for the correctness of any of the algorithms, but does impact their performance. SimpleHeap.Merge performs badly because all recursive calls use right children, and their results all form right children. Leftist heaps and skew heaps avoid this bad performance by using the results of the recursive calls as left children, at least part of the time. Another approach is to use different children in different calls. Specifically, when we make a recursive call, we can flip a coin to determine which child to use.

The resulting Merge function is shown in Figure 5.12; the remainder of the implementation of RandomizedHeap is identical to the implementations of SimpleHeap and SkewHeap. We assume that the FlipCoin function returns heads or tails randomly with uniform probability. Thus, each call to FlipCoin returns heads with probability 1/2, regardless of the
results of any prior calls. This function can typically be implemented using a built-in random number generator. Most platforms provide a function returning random values uniformly distributed over the range of signed integers on that platform. In a standard signed integer representation, the negative values comprise exactly half the range. The \texttt{FlipCoin} function can therefore generate a random integer and return \texttt{heads} iff that integer is negative.

It usually makes no sense to analyze the worst-case running time for a randomized algorithm, because the running time usually depends on random events. For example, if a given heap consists of a single path with \( n \) nodes, the algorithm could follow exactly that path. However, this could only happen for one particular sequence of \( n \) coin flips. If any of the flips differ from this sequence, the algorithm reaches a base case and terminates at that point. Because the probability of flipping this exact sequence is very small for large \( n \), a worst-case analysis seems inappropriate. Perhaps more to the point, a worst-case analysis would ignore the effect of randomization, and so does not seem appropriate for a randomized algorithm.

Instead, we can analyze the expected running time of a randomized algorithm. The goal of expected-case analysis is to bound the average performance over all possible executions on a worst-case input. For an ordinary deterministic algorithm, there is only one possible execution on any given input, but for randomized algorithms, there can be many possible executions depending on the random choices made.

Expected-case analysis is based on the expected values of random variables over discrete probability spaces. A \textit{discrete probability space} is a countable set of \textit{elementary events}, each having a \textit{probability}. For an elementary event \( e \) in a discrete probability space \( S \), we denote the probability of \( e \) by \( P(e) \). For any discrete probability space \( S \), we require that \( 0 \leq P(e) \leq 1 \) and that

\[
\sum_{e \in S} P(e) = 1.
\]

As a simple example, consider the flipping of a fair coin. The probability space is \{\texttt{heads}, \texttt{tails}\}, and each of these two elementary events has probability 1/2. For a more involved example, let \( T \) be a binary tree, and consider the probability space \( \text{Path}_T \) consisting of paths from the root of \( T \) to empty subtrees. We leave as an exercise to show that if \( T \) has \( n \) nodes, then it has \( n + 1 \) empty subtrees; hence \( \text{Path}_T \) has \( n + 1 \) elements. In order that it be a probability space, we need to assign a probability to each path. The probability of a given path of length \( k \) should be the same as the probability of the sequence of \( k \) coin flips that yields this path in the \texttt{Merge} algorithm;
thus, if the path corresponds to $k$ flips, its probability should be $2^{-k}$. We leave as an exercise to prove that the sum of these probabilities is 1 for any binary tree.

An important element of expected-case analysis is the notion of a *discrete random variable*, which is a function $f : S \rightarrow \mathbb{R}$, where $S$ is a discrete probability space. In this text, we will restrict our random variables to nonnegative values. For an example of a random variable, let $\text{len}_T(e)$ give the length of a path $e$ in the probability space $\text{Path}_T$ defined above. The *expected value* of a random variable $f$ over a probability space $S$ is defined to be

$$E[f] = \sum_{e \in S} f(e)P(e).$$

Thus, by multiplying the value of the random variable for each elementary event by the probability of that elementary event, we obtain an average value for that variable. Note that it is possible for an expected value to be infinite. If the summation converges, however, it converges to a unique value, because all terms are nonnegative.

**Example 5.4** Let $T$ be a binary tree with $n$ nodes, such that all paths from the root to empty subtrees have the same length. Because the probability of each path is determined solely by its length, all paths must have the same probability. Because there are $n+1$ paths and the sum of their probabilities is 1, each path must have probability $1/(n+1)$. In this case, $E[\text{len}_T]$ is simply the arithmetic mean, or simple average, of all of the lengths:

$$E[\text{len}_T] = \sum_{e \in \text{Path}_T} \text{len}_T(e)P(e)$$

$$= \frac{1}{n+1} \sum_{e \in \text{Path}_T} \text{len}_T(e).$$

Furthermore, because the lengths of all of the paths are the same, $E[\text{len}_T]$ must be this length, which we will denote by $k$.

We have defined the probability of a path of length $k$ to be $2^{-k}$. Furthermore, we have seen that all probabilities are $1/(n+1)$. We therefore have

$$2^{-k} = 1/(n+1).$$

Solving for $k$, we have

$$k = \lg(n + 1).$$

Thus, $E[\text{len}_T] = \lg(n + 1)$.\[More precisely, \text{len}_T(e) is the number of coin flips that are needed to generate e.\]
The discrete random variable \( \text{len}_T \) is always a natural number. When this is the case, its expected value is often easier to analyze. To show why, we first need to define an event, which is any subset of the elementary events in a discrete probability space. The probability of an event \( A \) is the sum of the probabilities of its elementary events; i.e.,

\[
P(A) = \sum_{e \in A} P(e).
\]

Note that because the sum of the probabilities of all elementary events in a discrete probability space is 1, the probability of an event is never more than 1.

The following theorem gives a technique for computing expected values of discrete random variables that range over the natural numbers. It uses predicates like “\( f = i \)” to describe events; e.g., the predicate “\( f = i \)” defines the event in which \( f \) has the value \( i \), and \( P(f = i) \) is the probability of this event.

**Theorem 5.5** Let \( f : S \rightarrow \mathbb{N} \) be a discrete random variable. Then

\[
E[f] = \sum_{i=1}^{\infty} P(f \geq i).
\]

The idea behind the proof is that \( P(f = i) = P(f \geq i) - P(f \geq i + 1) \). The definition of \( E[f] \) then yields

\[
E[f] = \sum_{e \in S} f(e)P(e) = \sum_{i=0}^{\infty} iP(f = i) = \sum_{i=0}^{\infty} i(P(f \geq i) - P(f \geq i + 1)) = \sum_{i=0}^{\infty} (iP(f \geq i) - iP(f \geq i + 1)).
\]

In the above sum, the negative portion \( iP(f \geq i + 1) \) of the \( i \)th term cancels most of the positive portion \((i + 1)P(f \geq i + 1)\) of the \((i + 1)\)st term. The result of this cancellation is the desired sum. However, in order for this reasoning to be valid, it must be the case that the “leftover” term, \(-iP(f \geq i + 1)\), converges to 0 as \( i \) approaches infinity if \( E[f] \) is finite. We leave the details as an exercise.
CHAPTER 5. PRIORITY QUEUES

Example 5.6 Let $T$ be a binary tree in which each of the $n$ nodes has an empty left child; i.e., the nodes form a single path going to the right. Again, the size of $\text{Path}_T$ is $n + 1$, but now the probabilities are not all the same. The length of the path to the rightmost empty subtree is $n$; hence, its probability is $2^{-n}$. For $1 \leq i \leq n$, there is exactly one path that goes right $i - 1$ times and left once. The probabilities for these paths are given by $2^{-i}$. We therefore have

$$E[\text{len}_T] = \sum_{e \in \text{Path}_T} \text{len}_T(e) P(e)$$

$$= n2^{-n} + \sum_{i=1}^{n} i2^{-i}.$$

Because we have no formula to evaluate the above summation, let us instead apply Theorem 5.5. The probability that a given path has length at least $i$, for $1 \leq i \leq n$, is the probability that $i - 1$ coin flips all yield tails. This probability is $2^{1-i}$. The probability that a given path has length at least $i$ for $i > n$ is 0. By Theorem 5.5, we therefore have

$$E[\text{len}_T] = \sum_{i=1}^{\infty} P(\text{len}_T \geq i)$$

$$= \sum_{i=1}^{n} 2^{1-i}$$

$$= \sum_{i=0}^{n-1} (1/2)^i$$

$$= (1/2)^n - 1$$

$$= (1/2)^n - 1$$

(by (2.2))

$$= (1/2) - 1$$

$$= 2 - 2^{1-n}.$$

Thus, $E[\text{len}_T] < 2$.

In order to be able to analyze the expected running time of \texttt{RandomizedHeap.Merge}, we need to know $E[\text{len}_T]$ for a worst-case binary tree $T$ with $n$ nodes. Examples 5.4 and 5.6 give two extreme cases — a completely balanced tree and a completely unbalanced tree. We might guess that the worst case would be one of these extremes. Because $\lg(n + 1) \geq 2 - 2^{1-n}$ for all $n \in \mathbb{N}$, a good guess would be that $\lg(n + 1)$ is an upper bound for the worst case. We can show that this is indeed the case, but we need to use
the following theorem relating the sum of logarithms to the logarithm of a sum.

**Theorem 5.7** If \( x \) and \( y \) are positive real numbers, then

\[
\lg x + \lg y \leq 2 \lg(x + y) - 2.
\]

**Proof:** We first note that \( \lg x + \lg y = \lg xy \). We will therefore show that the right-hand side of the inequality is at least \( \lg xy \). Using the fact that \( \lg 4 = 2 \), we have

\[
2 \lg(x + y) - 2 = \lg((x + y)^2) - \lg 4
\]

\[
= \lg \left( \frac{x^2 + 2xy + y^2}{4} \right).
\]

In order to isolate \( \lg xy \), let us now subtract \( xy \) from the fraction in the above equation. This yields

\[
2 \lg(x + y) - 2 = \lg \left( \frac{x^2 + 2xy + y^2}{4} \right)
\]

\[
= \lg \left( xy + \frac{x^2 - 2xy + y^2}{4} \right)
\]

\[
= \lg \left( xy + \frac{(x - y)^2}{4} \right)
\]

\[
\geq \lg xy,
\]

because \( (x - y)^2/4 \) is always nonnegative and the \( \lg \) function is nondecreasing. \( \square \)

We can now show that \( \lg(n + 1) \) is an upper bound for \( E[\text{len}_T] \) when \( T \) is a binary tree with \( n \) nodes.

**Theorem 5.8** Let \( T \) be any binary tree with size \( n \), where \( n \in \mathbb{N} \). Then \( E[\text{len}_T] \leq \lg(n + 1) \).

**Proof:** By induction on \( n \).

**Base:** \( n = 0 \). Then only one path to an empty tree exists, and its length is 0. Hence, \( E[\text{len}_T] = 0 = \lg 1 \).
Induction Hypothesis: Assume that for some $n > 0$, if $S$ is any binary tree with size $i < n$, then $E[len_S] \leq \lg(i + 1)$.

Induction Step: Suppose $T$ has size $n$. Because $n > 0$, $T$ is nonempty. Let $L$ and $R$ be the left and right children, respectively, of $T$. We then have

$$E[len_T] = \sum_{e \in Path_T} \text{len}_T(e) P(e)$$

$$= \sum_{e \in Path_L} (\text{len}_L(e) + 1) \frac{P(e)}{2} + \sum_{e \in Path_R} (\text{len}_R(e) + 1) \frac{P(e)}{2}, \quad (5.1)$$

because the probability of any path from the root of a child of $T$ to any empty subtree is twice the probability of the path from the root of $T$ to the same empty subtree, and its length is one less.

Because the two sums in (5.1) are similar, we will simplify just the first one. Thus,

$$\sum_{e \in Path_L} (\text{len}_L(e) + 1) \frac{P(e)}{2} = \frac{1}{2} \left( \sum_{e \in Path_L} \text{len}_L(e) P(e) + \sum_{e \in Path_L} P(e) \right)$$

$$= \frac{1}{2} \left( \sum_{e \in Path_L} \text{len}_L(e) P(e) + 1 \right),$$

because in $Path_L$, the sum of the probabilities is 1. We now observe that

$$\sum_{e \in Path_L} \text{len}_L(e) P(e) = E[len_L].$$

Applying a similar simplification to the second sum in 5.1, we have


Suppose $L$ has size $i$. Then $R$ has size $n - i - 1$. Because $0 \leq i < n$, the Induction Hypothesis applies to both $L$ and $R$. Thus,

$$E[len_T] \leq 1 + (\lg(i + 1) + \lg(n - i))/2$$

$$\leq 1 + (2\lg(n + 1) - 2)/2 \quad \text{(by Theorem 5.7)}$$

$$= \lg(n + 1).$$

$\square$
The fact that the expected length of a randomly chosen path in a binary tree of size \( n \) is never more than \( \lg(n + 1) \) gives us reason to believe that the expected running time of `RandomizedHeap.Merge` is in \( O(\lg n) \). However, `Merge` operates on two binary trees. We therefore need a bound on the expected sum of the lengths of two randomly chosen paths, one from each of two binary trees. Hence, we will combine two probability spaces \( \text{Path}_S \) and \( \text{Path}_T \) to form a new discrete probability space \( \text{Paths}_{S,T} \). The elementary events of this space will be pairs consisting of an elementary event from \( \text{Path}_S \) and an elementary event from \( \text{Path}_T \).

We need to assign probabilities to the elementary events in \( \text{Paths}_{S,T} \). In so doing, we need to reflect the fact that the lengths of any two paths from \( S \) and \( T \) are independent of each other; i.e., knowing the length of one path tells us nothing about the length of the other path. Let \( e_1 \) and \( e_2 \) be events over a discrete probability space \( S \). We say that \( e_1 \) and \( e_2 \) are independent if \( P(e_1 \cap e_2) = P(e_1)P(e_2) \).

Suppose we were to define a new discrete probability space \( S_{e_2} \) including only those elementary events in the event \( e_2 \). The sum of the probabilities of these elementary events is \( P(e_2) \). If we were to scale all of these probabilities by dividing by \( P(e_2) \), we would achieve a total probability of 1 while preserving the ratio of any two probabilities. The probability of event \( e_1 \) within \( S_{e_2} \) would be given by

\[
P(e_1 \mid e_2) = \frac{P(e_1 \cap e_2)}{P(e_2)},
\]

where the probabilities on the right-hand side are with respect to \( S \). We call \( P(e_1 \mid e_2) \) the conditional probability of \( e_1 \) given \( e_2 \). Note that if \( P(e_2) \neq 0 \), independence of \( e_1 \) and \( e_2 \) is equivalent to \( P(e_1) = P(e_1 \mid e_2) \). Thus, two events are independent if knowledge of one event does not affect the probability of the other.

The definition of independence tells us how to assign the probabilities in \( \text{Paths}_{S,T} \). Let \( e_1 \) be the event such that the path in \( S \) is \( s \), and let \( e_2 \) be the event such that the path in \( T \) is \( t \). Then \( e_1 \cap e_2 \) is the elementary event consisting of paths \( s \) and \( t \). We need \( P(e_1 \cap e_2) = P(e_1)P(e_2) \) in order to achieve independence. However, \( P(e_1) \) should be the probability of \( s \) in \( \text{Path}_S \), and \( P(e_2) \) should be the probability of \( t \) in \( \text{Path}_T \). Thus the probability of an elementary event in \( \text{Paths}_{S,T} \) must be the product of the probabilities of the constituent elementary events from \( \text{Path}_S \) and \( \text{Path}_T \). It is then not hard to verify that \( P(e_1) \) and \( P(e_2) \) are the probabilities of \( s \) in \( \text{Path}_S \) and of \( t \) in \( \text{Path}_T \), respectively.
We now extend the discrete random variables \( \text{len}_S \) and \( \text{len}_T \) to the space \( \text{Paths}_{S,T} \) so that \( \text{len}_S \) gives the length of the path in \( S \) and \( \text{len}_T \) gives the length of the path in \( T \). Because neither the lengths of the paths nor their probabilities change when we make this extension, it is clear that their expected values do not change either.

The running time of \text{RandomizedHeap.Merge} is clearly proportional to the lengths of the paths followed in the two heaps \( S \) and \( T \). These paths may or may not go all the way to an empty subtree, but if not, we can extend them to obtain elementary events \( s \) and \( t \) in \( \text{Path}_S \) and \( \text{Path}_T \), respectively. The running time is then bounded above by \( c(\text{len}_S(s) + \text{len}_T(t)) \), where \( c \) is some fixed positive constant. The expected running time of \text{MERGE} is therefore bounded above by \( E[c(\text{len}_S + \text{len}_T)] \). In order to bound this expression, we need the following theorem.

**Theorem 5.9 (Linearity of Expectation)** Let \( f \), \( g \), and \( h_i \) be discrete random variables for all \( i \in \mathbb{N} \), and let \( a \in \mathbb{R} \geq 0 \). Then

\[
E[af + g] = aE[f] + E[g];
\]

and

\[
E \left[ \sum_{i=0}^{\infty} h_i \right] = \sum_{i=0}^{\infty} E[h_i].
\]

The proof of this theorem is straightforward and left as an exercise. It is important to realize not only what this theorem says, but also what it doesn’t say. For example, it is not necessarily the case that \( E[fg] = E[f]E[g] \), or that \( E[2f] = 2E[f] \) — see Exercise 5.17 for specific counterexamples. We must therefore be very careful in working with expected values, as they do not always behave as our intuition might suggest.

Applying Theorems 5.9 and 5.7 to our analysis, we now see that

\[
E[c(\text{len}_S + \text{len}_T)] = c(E[\text{len}_S] + E[\text{len}_T])
\]

\[
\leq c(\log(|S| + 1) + \log(|T| + 1))
\]

\[
\leq 2c\log(|S| + |T| + 2),
\]

where \( |S| \) and \( |T| \) denote the sizes of \( S \) and \( T \), respectively. Thus, the expected running time of \text{MERGE} is in \( O(\log n) \), where \( n \) is the total number of nodes in the two heaps. It follows that the expected running times of \text{PUT} and \text{REMOVEMAX} are also in \( O(\log n) \).

A close examination of Example 5.4 reveals that the bound of \( \log(n + 1) \) on \( E[\text{len}_T] \) is reached when \( n + 1 \) is a power of 2. Using the fact that \( \log \) is
smooth, we can then show that the expected running time of \textsc{Merge} is in $\Omega(lg \ n)$; the details are left as an exercise. Thus, the expected running times of \textsc{Put} and \textsc{RemoveMax} are in $\Theta(lg \ n)$.

## 5.6 Sorting and Binary Heaps

In Section 3.6, we saw how to sort an array in $\Theta(n^2)$ time. A priority queue can be used to improve this performance. Using either a \textsc{LeftistHeap} or a \textsc{SkewHeap}, we can insert $n$ elements in $\Theta(n \ lg \ n)$ time, by Theorem 3.28. We can then sort the items in the heap by removing the maximum in $\Theta(lg \ n)$ time and sorting the remainder. It is easily seen that this entire algorithm runs in $\Theta(n \ lg \ n)$ time.

In order to improve further the performance of sorting, we would like to avoid the need to use an auxiliary data structure. Specifically, we would like to keep the data items in a single array, which is partitioned into an unsorted part followed by a sorted part, as illustrated in Figure 5.13 (a). The unsorted part will, in essence, be a representation of a priority queue — we will explain the details of this representation in what follows. This priority queue will contain keys that are no larger than any of the keys in the sorted part. When we remove the maximum element from the priority queue, this frees up an array location, as shown in Figure 5.13 (b). We can put the element that we removed from the priority queue into this location. Because this key is at least as large as any key in the priority queue, but no larger than any key in the sorted part, we can extend the sorted part to include this location (see Figure 5.13 (c)).

We therefore need to be able to represent a heap using an array. One way to accomplish this is to number the nodes left-to-right by levels, as shown in Figure 5.14. The numbers we have assigned to the nodes can be used as array indices. In order to avoid ambiguity, there should be no “missing” nodes; i.e., each level except possibly the last should be completely full, and all of the nodes in the last level should be as far to the left as possible. This scheme for storing a heap is known as a \textit{binary heap}.

Notice that a binary heap is very nearly balanced. We saw in Example 5.4 that in a completely balanced binary tree with $n$ nodes, the length of any path to an empty subtree is $lg(n+1)$. This result holds only for tree sizes that can be completely balanced. However, it is not hard to show that for any $n$, if a binary tree with $n$ nodes is balanced as nearly as possible, the length of the longest path to an empty subtree is $\lceil lg(n + 1) \rceil$ (or equivalently, the height is $\lceil lg(n + 1) \rceil - 1$). We will show that this fact allows us to implement
Figure 5.13 Illustration of sorting using a priority queue represented in an array

(a) \[
\begin{array}{cccccccc}
55 & 48 & 52 & 37 & 41 & 50 & 70 & 75 & 85 & 89 & 94 \\
\text{unsorted} & & & & & & & & & & \\
\text{(priority queue)} & & & & & & & & & & \\
\text{sorted} & & & & & & & & & & \\
\end{array}
\]

(b) \[
\begin{array}{cccccccc}
52 & 48 & 50 & 37 & 41 & 70 & 75 & 85 & 89 & 94 \\
\text{unsorted} & & & & & & & & & & \\
\text{(priority queue)} & & & & & & & & & & \\
\text{sorted} & & & & & & & & & & \\
55 & & & & & & & & & & \\
\end{array}
\]

(c) \[
\begin{array}{cccccccc}
52 & 48 & 50 & 37 & 41 & 55 & 70 & 75 & 85 & 89 & 94 \\
\text{unsorted} & & & & & & & & & & \\
\text{(priority queue)} & & & & & & & & & & \\
\text{sorted} & & & & & & & & & & \\
\end{array}
\]

Figure 5.14 A binary heap

\[
\begin{array}{cccccccc}
1 & 2 & 3 \\
\text{89} & \text{53} & \text{32} \\
\end{array}
\]

\[
\begin{array}{cccccccc}
4 & 5 & 6 & 7 \\
\text{48} & \text{53} & \text{17} & \text{27} \\
\end{array}
\]

\[
\begin{array}{cccccccc}
8 & 9 & 10 \\
\text{24} & \text{13} & \text{32} \\
\end{array}
\]
both \textsc{put} and \textsc{removeMax} for a binary heap in $\Theta(\log n)$ time.

Note that each level of a binary heap, except the first and possibly the last, contains exactly twice as many nodes as the level above it. Thus, if we were to number the levels starting with 0 for the top level, then each level $i$ (except possibly the last) contains exactly $2^i$ nodes. It follows from (2.2) that levels 0 through $i - 1$, where $i$ is strictly less than the total number of levels, have a total of $2^i - 1$ nodes. Let $x$ be the $j$th node on level $i$. $x$ would then have index $2^i - 1 + j$. Suppose $x$ has a left child, $y$. In order to compute its index, we observe that level $i$ has $j - 1$ nodes to the left of $x$. Each of these nodes has two children on level $i + 1$ to the left of node $y$. Therefore, the index of $y$ is

$$2^{i+1} - 1 + 2(j - 1) + 1 = 2^{i+1} + 2j - 2,$$

or exactly twice the index of its parent. Likewise, if $x$ has a right child, its index is 1 greater than that of $y$.

As a result of these relationships, we can use simple calculations to find either child or the parent of a node at a given location. Specifically, the left and right children of the element at location $i$ are the elements at locations $2i$ and $2i + 1$, respectively, provided they exist. Furthermore, the parent of the element at location $i > 1$ is at location $\lceil i/2 \rceil$.

Let us consider how we can implement a binary heap as a data structure. We will use two representation variables:

- $\text{elements}[0..m]$: an array of \texttt{Keyed} items; and
- $\text{size}$: a \texttt{Nat}.

We allow read access to $\text{size}$. For reasons that will become clear shortly, $\text{elements}[0]$ will act as a sentinel element, and will have as its key the maximum allowable value. For convenience, we will use a constant \texttt{sentinel} to represent such a data item. Note because $\lfloor 1/2 \rfloor = 0$, we can treat $\text{elements}[0]$ as if it were the parent of $\text{elements}[1]$.

The structural invariant will be:

- $\text{size} \leq \text{SIZEOf} (\text{elements});$
- $\text{elements}[0] = \texttt{sentinel};$ and
- for $1 \leq i \leq \text{size}, \text{elements}[i].\text{Key}() \leq \text{elements}[\lfloor i/2 \rfloor].\text{Key}().$

We interpret $\text{elements}[1..\text{size}]$ as the elements of the set being represented, together with their associated priorities.
Unfortunately, the algorithms for merging heaps don’t work for binary heaps because they don’t maintain the balanced shape. Therefore, let us consider how to insert an element $x$ into a binary heap. If $\text{size}$ is $0$, then we can simply make $x$ the root. Otherwise, we need to compare $x.\text{KEY}()$ with the key of the root. The larger of the two will be the new root, and we can then insert the other into one of the children. We select which child based on where we need the new leaf.

In this insertion algorithm, unless the tree is empty, there will always be a recursive call. This recursive call will always be on the child in the path that leads to the location at which we want to add the new node. Note that the keys along this path from the root to the leaf are in nonincreasing order. As long as the key to be inserted is smaller than the key to which it is compared, it will be the inserted element in the recursive call. When it is compared with a smaller key, that smaller key is used in the recursive call. When this happens, the key passed to the recursive call will always be at least as large as the root of the subtree in which it is being inserted; thus, it will become the new root, and the old root will be used in the recursive call. Thus, the entire process results in inserting the new key at the proper point in the path from the root to the desired insertion location.

For example, suppose we wish to insert the priority $35$ into the binary heap shown in Figure 5.15(a). We first find the path to the next insertion point. This path is $\langle 89, 32, 17 \rangle$. The proper position of $35$ in this path is between $89$ and $32$. We insert $35$ at this point, pushing the following priorities downward. The result is shown in Figure 5.15(b).

Because we can easily find the parent of a node in a $\text{BinaryHeap}$, we can implement this algorithm bottom-up by starting at the location of the new leaf and shifting elements downward one level until we reach a location where the new element will fit. This is where having a sentinel element is convenient — we know we will eventually find some element whose key is at least as large as that of $x$. The resulting algorithm is shown in Figure 5.16. We assume that $\text{Expand}(A)$ returns an array of twice the size of $A$, with the elements of $A$ copied to the first half of the returned array.

The $\text{RemoveMax}$ operation is a bit more difficult. We need to remove the root because it contains the element with maximum priority, but in order to preserve the proper shape of the heap, we need to remove a specific leaf. We therefore first save the value of the root, then remove the proper leaf. We need to form a new heap by replacing the root with the removed leaf. In order to accomplish this, we use the $\text{MakeHeap}$ algorithm shown in Figure 5.17. For ease of presentation, we assume $t$ is formed with $\text{BinaryTreeNode}$s, rather than with an array. If the key of $x$ is at least as
Figure 5.15 Example of inserting 35 into a binary heap

(a) The original heap

(b) 35 is inserted into the path (89, 32, 17)

Figure 5.16 The `BinaryHeap.Put` operation

```
BinaryHeap.Put(x, p : Number)
size ← size + 1
if size > SizeOf(elements)
    elements ← Expand(elements)
i ← size; elements[i] ← elements[⌊i/2⌋]
    // Invariant: 1 ≤ i ≤ size, elements[1..size] forms a heap,
    // elements[0..size] contains the elements originally in
    // elements[0..size − 1], with elements[i] and elements[⌊i/2⌋]
    // being duplicates, and p > elements[j].Key() for
    // 2i ≤ j ≤ max(2i + 1, size).
while p > elements[i].Key()
    i ← ⌊i/2⌋; elements[i] ← elements[⌊i/2⌋]
    elements[i] ← new Keyed(x, p)
```
Figure 5.17 The MakeHeap algorithm

**Precondition:** $x$ is a Keyed element and $t$ is a BinaryTreeNode whose children are both heaps.

**Postcondition:** Forms a heap from $t$ containing $x$ and the elements of $t$’s children without changing the shape of $t$.

```plaintext
MakeHeap($x$, $t$)
    $L \leftarrow t$.LeftChild(); $R \leftarrow t$.RightChild()
    if $L = nil$ and $R = nil$
        $t$.SetRoot($x$)
    else
        if $L \neq nil$
            if $R \neq nil$ and $L$.Root().Key() < $R$.Root().Key()
                largerChild $\leftarrow R$
            else
                largerChild $\leftarrow L$
        else
            largerChild $\leftarrow R$
        if $x$.Key() $\geq$ largerChild.Root().Key()
            $t$.SetRoot($x$)
        else
            $t$.SetRoot(largerChild.Root())
            MakeHeap($x$, largerChild)
```

large as the keys of the roots of all children of $t$, we can simply replace the root of $t$ with $x$, and we are finished. Otherwise, we need to move the root of the child with larger key to the root of $t$ and make a heap from this child and $x$. This is just a smaller instance of the original problem.

We can simplify MakeHeap somewhat when we use it with a binary heap. First, we observe that once we have determined that at least one child is nonempty, we can conclude that the left child must be nonempty. We also observe that the reduction is a transformation to a smaller instance; i.e., MakeHeap is tail recursive. We can therefore implement it using a loop. In order to simplify the statement of the loop invariant, we make use of the fact that the entire tree is initially a heap, so that the precondition of MakeHeap could be strengthened to specify that $t$ is a heap. (Later we will use
CHAPTER 5. PRIORITY QUEUES

Figure 5.18 The BinaryHeap.RemoveMax operation.

BinaryHeap.RemoveMax()
    if size = 0
        error
    else
        m ← elements[1].Data(); size ← size − 1; i ← 1
        // Invariant: elements[1..size] forms a heap; 1 ≤ i ≤ size + 1;
        // elements[1..i − 1], elements[i + 1..size + 1], and m are
        // the elements in the original set;
        // elements[size + 1].Key() ≤ elements[(i/2)].Key();
        // and m has maximum key.
        while elements[i] ≠ elements[size + 1]
            j ← 2i
            if j > size
                elements[i] ← elements[size + 1]
            else
                if j < size and elements[j].Key() < elements[j + 1].Key()
                    j ← j + 1
                if elements[j].Key() ≤ elements[size + 1].Key()
                    elements[i] ← elements[size + 1]
                else
                    elements[i] ← elements[j]; i ← j
        return m

MakeHeap in a context in which we need the weaker precondition.) Figure
5.18 gives the entire REMOVEMAX operation without a separate MAKEHEAP
function. Note that elements[size + 1] in Figure 5.18 corresponds to x in
Figure 5.17, elements[i] corresponds to t, and j corresponds to largerChild.

Notice that in REMOVEMAX, i is initialized to 1, the root of the heap,
and on each iteration that does not cause the while condition to be false,
i is set to j, the index of its larger child. Furthermore, on each iteration,
elements[i] is set to either elements[size + 1] or elements[j]. In the latter
case, the larger child of elements[i] is copied to elements[i], and in the former
case, the removed leaf is placed in its proper location. Thus, as in the PUT
operation, an element is inserted into a path in the heap; however, in this
case, the path follows the larger of the children of a node, and the elements
Figure 5.19 Example of the RemoveMax operation for a binary heap

(a) The original heap

(b) 89 is removed, and 41 is inserted into the path ⟨65, 48, 33⟩

preceding the insertion location are moved upward.

For example, suppose we were to perform a RemoveMax on the binary heap shown in Figure 5.19(a). We would remove 89 and find the path that follows the larger child of each node. This path is ⟨65, 48, 33⟩. We would then insert 41, the last leaf, into this path between 48 and 33, moving the preceding priorities upward. The result is shown in Figure 5.19(b).

It is easily seen that both Put and RemoveMax operate in $\Theta(\lg n)$ time, excluding any time needed to expand the array. Furthermore, as we saw in Section 4.3, we can amortize the cost of array expansion to constant time per insertion. The amortized running time for Put is therefore in $\Theta(\lg n)$, and the worst-case time for RemoveMax is in $\Theta(\lg n)$.

We now return to the sorting problem. In order to sort an array $A$, we first need to arrange it into a binary heap. One approach is first to make $A[1..n-1]$ into a heap, then to insert $A[n]$. We can easily implement this bottom-up. The resulting algorithm does $n - 1$ insertions into heaps of sizes ranging from 1 to $n - 1$. The total running time is therefore in

$$
\sum_{i=1}^{n-1} \Theta(\lg i) \subseteq \Theta((n - 1) \lg(n - 1)) \quad \text{(from Theorem 3.28)}
$$

$$
= \Theta(n \lg n).
$$

We can do better, however, by viewing the array $A[1..n]$ as a binary tree
in which the parent of $A[i]$ is $A[i/2]$ for $i > 1$. With this view in mind, the natural approach seems to be to make the children into heaps first, then use MAKEHEAP to make the entire tree into a heap. The resulting algorithm is easiest to analyze when the tree is completely balanced — i.e., when $n + 1$ is a power of 2. Let $N = n + 1$, and let $f(N)$ give the worst-case running time for this algorithm. When $N$ is a power of 2, we have

$$f(N) \in 2f(N/2) + \Theta(lg N).$$

From Theorem 3.32, $f(N) \in \Theta(N) = \Theta(n)$.

This implementation of MAKEHEAP must be more general than the implementation used for BINARYHEAP. Specifically, we must be able to apply MAKEHEAP to arbitrary subtrees in order to be able to use it to form the heap initially. In order to allow us to express the specification of Figure 5.17 in terms of a binary heap, we introduce the notation $\text{Tree}(A, k, n)$ to denote the binary tree formed by using $A[k]$ as the root, $\text{Tree}(A, 2k, n)$ as the left child, and $\text{Tree}(A, 2k + 1, n)$ as the right child, provided $k \leq n$. If $k > n$, $\text{Tree}(A, k, n)$ denotes an empty tree. Thus, $\text{Tree}(A, 1, n)$ denotes the binary tree $T$ implied by the array $A[1..n]$, and for $k \leq n$, $\text{Tree}(A, k, n)$ denotes the subtree of $T$ rooted at $A[k]$. The full implementation of HEAPSORT is shown in Figure 5.20.

It is not hard to show that MAKEHEAP operates in $\Theta(lg(n/k))$ time in the worst case. It is easily seen that the first for loop in HEAPSORT operates in $O(n \lg n)$ time, though in fact a careful analysis shows that it runs in $\Theta(n)$ time, as suggested by the above discussion. It is not hard to show, using Theorem 3.28, that the second for loop operates in $\Theta(n \lg n)$ time in the worst case. Therefore, HEAPSORT runs in $\Theta(n \lg n)$ time in the worst case.

5.7 Summary

A heap provides a clean framework for implementing a priority queue. Although LEFTISTHEAPS yield $\Theta(lg n)$ worst-case performance for the operations PUT and REMOVEMAX, the simpler SKEWHEAPS and RANDOMIZEDHEAPS yield $O(lg n)$ amortized and $\Theta(lg n)$ expected costs, respectively, for these operations. BINARYHEAPS, while providing no asymptotic improvements over LEFTISTHEAPS, nevertheless tend to be more efficient in practice because they require less dynamic memory allocation. They also provide the basis for HEAPSORT, a $\Theta(n \lg n)$ in-place sorting algorithm. A summary of the running times of the PRIORITYQUEUE operations for the various implementations is shown in Figure 5.21.
**Figure 5.20** HeapSort implementation of Sort, specified in Figure 1.1

HeapSort\(A[1..n])

\[\text{// Invariant: } A[1..n] \text{ is a permutation of its original elements such }
\text{// that for } 2(i + 1) \leq j \leq n, A[(j/2)] \geq A[j].\]

\[\text{for } i \leftarrow \lceil n/2 \rceil \text{ to } 1 \text{ by } -1\]
\[\text{MakeHeap}(A[1..n], i, A[i])\]

\[\text{// Invariant: } A[1..n] \text{ is a permutation of its original elements such }
\text{// that for } 2 \leq j \leq i, A[(j/2)] \geq A[j], \text{ and }
\text{// } A[1] \leq A[i + 1] \leq A[i + 2] \leq \cdots \leq A[n].\]

\[\text{for } i \leftarrow n \text{ to } 2 \text{ by } -1\]
\[t \leftarrow A[i]; A[i] \leftarrow A[1]; \text{MakeHeap}(A[1..i - 1], 1, t)\]

**Precondition:** \(A[1..n]\) is an array of Numbers such that Tree\((A, 2k, n)\) and Tree\((A, 2k + 1, n)\) form heaps, \(1 \leq k \leq n\), and \(x\) is a Number.

**Postcondition:** Tree\((A, k, n)\) is a heap containing a permutation of the original values of Tree\((A, 2k, n)\), Tree\((A, 2k + 1, n)\), and \(x\), and no other elements of \(A\) have changed.

MakeHeap\((A[1..n], k, x)\)

\[A[k] \leftarrow \text{sentinel}; i \leftarrow k\]

\[\text{// Invariant: } \text{Tree}(A, k, n) \text{ forms a heap; } k \leq i \leq n;\]

\[\text{// } A[i] \text{ belongs to } \text{Tree}(A, k, n);\]

\[\text{// the elements of } \text{Tree}(A, k, n), \text{ excluding } A[i], \text{ are the elements initially}\]

\[\text{// in } \text{Tree}(A, 2k, n) \text{ and } \text{Tree}(A, 2k + 1, n);\]

\[\text{// all other elements of } A \text{ have their initial values;}\]

\[\text{// and if } i \geq 2k, \text{ then } x < A[(i/2)].\]

\[\text{while } A[i] \neq x\]
\[j \leftarrow 2i\]
\[\text{if } j > n\]
\[A[i] \leftarrow x\]
\[\text{else}\]
\[\text{if } j < n \text{ and } A[j] < A[j + 1]\]
\[j \leftarrow j + 1\]
\[\text{if } A[j] \leq x\]
\[A[i] \leftarrow x\]
\[\text{else}\]
\[A[i] \leftarrow A[j]; i \leftarrow j\]
**Figure 5.21** Running times for the `PriorityQueue` operations for various implementations.

<table>
<thead>
<tr>
<th>PriorityQueue Type</th>
<th>Put</th>
<th>RemoveMax</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SortedArrayPriorityQueue</code></td>
<td>$\Theta(n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td><code>SimpleHeap</code></td>
<td>$\Theta(n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td><code>LeftistHeap</code></td>
<td>$\Theta(lg n)$</td>
<td>$\Theta(lg n)$</td>
</tr>
<tr>
<td><code>SkewHeap</code></td>
<td>$O(lg n)$</td>
<td>$O(lg n)$</td>
</tr>
<tr>
<td></td>
<td>amortized</td>
<td>amortized</td>
</tr>
<tr>
<td><code>RandomizedHeap</code></td>
<td>$\Theta(lg n)$</td>
<td>$\Theta(lg n)$</td>
</tr>
<tr>
<td></td>
<td>expected</td>
<td>expected</td>
</tr>
<tr>
<td><code>BinaryHeap</code></td>
<td>$\Theta(lg n)$</td>
<td>$\Theta(lg n)$</td>
</tr>
<tr>
<td></td>
<td>amortized</td>
<td></td>
</tr>
</tbody>
</table>

**Notes:**

- $n$ is the number of elements in the priority queue.
- Unless otherwise noted, all running times are worst-case.
- The constructor and the `MaxPriority` and `Size` operations all run is $\Theta(1)$ worst-case time for all implementations.

For the implementations that use a `Merge` function, it is possible to provide `Merge` as an operation. However, this operation is not very appropriate for the `PriorityQueue` ADT because we may need to require the two priority queues to be of the same type. For example, if we added a `Merge` operation to `LeftistHeap`, we would need to require that the parameter is also a `LeftistHeap` — `Merge(PriorityQueue)` would be insufficient. Furthermore, we would need to be concerned with security because the resulting heap would share storage with the original heaps.

Using an immutable structure, as we did for `LeftistHeap`, would take care of the security issue. With such implementations, the `Merge` operation could be done in $\Theta(lg n)$ worst-case time for a `LeftistHeap`, or in $\Theta(lg n)$ expected time for a `RandomizedHeap`, where $n$ is the sum of the sizes of the two priority queues. The amortized time for `SkewHeap.Merge`, however, is not in $O(lg n)$ unless we restrict the sequences of operations so that after two priority queues are merged, the original priority queues are not used in
CHAPTER 5. PRIORITY QUEUES

any subsequent operations; otherwise, we can repeatedly perform the same expensive Merge.

In Section 5.5, we introduced the basics of expected-case analysis for randomized algorithms. Specifically, we showed how discrete random variables can be defined and manipulated in order to analyze expected running time. In Section 6.4, we will develop this theory more fully.

5.8 Exercises

Exercise 5.1 Complete the implementation of SortedArrayPriorityQueue shown in Figure 5.2 by adding a constructor and implementations of the MaxPriority and RemoveMax operations. Prove that your implementation meets its specification.

Exercise 5.2 Prove that SimpleHeap, shown in Figure 5.5, meets its specification.

Exercise 5.3 Show the result of first inserting the sequence of priorities below into a leftist heap, then executing one RemoveMax.

\[34, 12, 72, 15, 37, 49, 17, 55, 45\]

Exercise 5.4 Prove that LeftistTree, shown in Figure 5.7, meets its specification.

Exercise 5.5 Prove that LeftistHeap, shown in Figure 5.8, meets its specification.

* Exercise 5.6 Prove that for any \(n \in \mathbb{N}\), if we insert a sequence of \(n\) strictly decreasing priorities into an initially empty leftist heap, we obtain a leftist heap with null path length \(\lfloor \lg(n + 1) \rfloor\).

Exercise 5.7 Instead of keeping track of the null path lengths of each node, a variation on LeftistTree keeps track of the number of nodes in each subtree, and ensures that the left child has as many nodes as the right child. We call this variation a LeftHeavyTree.

a. Give an implementation of LeftHeavyTree. The structure must be immutable, and each constructor must require only \(\Theta(1)\) time.
Figure 5.22 The HasPriority interface

Precondition: true.
Postcondition: Returns a number representing a priority.

HasPriority.Priority()

b. Prove by induction on the number of nodes $n$ in the tree that in any LeftHeavyTree, the distance from the root to the rightmost empty subtree is no more than $\lg(n + 1)$.

c. Using the result of part b, show that if we use LeftHeavyTrees instead of LeftistTrees in the implementation of LeftistHeap, the running times of the operations are still in $O(\lg n)$, where $n$ is the number of elements in the priority queue.

Exercise 5.8 Repeat Exercise 5.3 using a skew heap instead of a leftist heap.

Exercise 5.9 Prove that SkewHeap, obtained by replacing the Merge function in SimpleHeap (Figure 5.5) with the function shown in Figure 5.9, meets its specification.

* Exercise 5.10 Another way of specifying a priority queue is to define an interface HasPriority, as shown in Figure 5.22. Rather than supplying two arguments to the Put operation, we could instead specify that it takes a single argument of type HasPriority, where the priority of the item is given by its Priority operation. Discuss the potential security problems for this approach. How could these problems be avoided if such a specification were adopted?

Exercise 5.11 The goal of this exercise is to complete the analysis of the amortized running times of the SkewHeap operations.

a. Prove by induction on $n$ that in any binary tree $T$ with $n$ nodes, the number of good nodes on its rightmost path is no more than $\lg(n + 1)$, where the definition of a good node is as in Section 5.4.

b. Prove that in the SkewHeap.Merge operation (shown in Figure 5.9 on page 166) if $L$ is initially a bad node, then it is a good node in the resulting heap.
c. Given two skew heaps to be merged, let us define the potential of each node to be 0 if the node is good, or 1 if the node is bad. Using the results from parts a and b above, prove that the actual cost of the MERGE operation, plus the sum of the potentials of the nodes in the resulting heap, minus the sum of potentials of the nodes in the two original heaps, is in $O(\lg n)$ where $n$ is the number of keys in the two heaps together.

d. Using the result of part c, prove that the amortized running times of the SkewHeap operations are in $O(\lg n)$, where $n$ is the number of nodes in the heap.

Exercise 5.12 Prove that RANDOMIZEDHeap, obtained by replacing the MERGE function in SIMPLEHeap (Figure 5.5) with the function shown in Figure 5.12, meets its specification.

Exercise 5.13 Prove by induction on $n$ that any binary tree with $n$ nodes has exactly $n + 1$ empty subtrees.

Exercise 5.14 Prove by induction on the number of nodes in a binary tree $T$, that the sum of the probabilities in Path$_T$ is 1.

Exercise 5.15 The goal of this exercise is to prove Theorem 5.5. Let $f : S \to N$ be a discrete random variable.

a. Prove by induction on $n$ that
\[ \sum_{i=0}^{n} i P(f = i) = \sum_{i=1}^{n} P(f \geq i) - n P(f \geq n + 1). \]

b. Prove that for every $n \in N$,
\[ \sum_{i=0}^{\infty} i P(f = i) \geq \sum_{i=0}^{n} i P(f = i) + n P(f \geq n + 1). \]

c. Using the fact that if $g(i) \geq 0$ for all $i$, then
\[ \sum_{i=0}^{\infty} g(i) = \lim_{n \to \infty} \sum_{i=0}^{n} g(i), \]
prove that if $E[f]$ is finite, then
\[ \lim_{n \to \infty} n P(f \geq n + 1) = 0. \]
d. Prove Theorem 5.5.

**Exercise 5.16** Prove Theorem 5.9.

**Exercise 5.17** Let $S$ be the set of all sequences of four flips of a fair coin, where each sequence has probability $1/16$. Let $h$ be the discrete random variable giving the number of heads in the sequence.

a. Compute $E[h]$.

b. Compute $E[h^2]$, and show that $E[h^2] ≠ (E[h])^2$.


**Exercise 5.18** Use Example 5.4 to show that the expected running time of `RANDOMIZED_HEAP.MERGE`, shown in Figure 5.12, is in $\Omega(\lg n)$ in the worst case, where $n$ is the number of elements in the two heaps combined.

**Exercise 5.19** Complete the implementation of `BINARY_HEAP` by adding a constructor and a `MAX_PRIORITY` operation to the operations shown in Figures 5.16 and 5.18. Prove that the resulting implementation meets its specification.

**Exercise 5.20** Repeat Exercise 5.3 using a binary heap instead of a leftist heap. Show the result as both a tree and an array.

**Exercise 5.21** Prove that `HEAPSORT`, shown in Figure 5.20, meets its specification.

**Exercise 5.22** Prove that the first loop in `HEAPSORT` runs in $\Theta(n)$ time in the worst case.

**Exercise 5.23** Prove that `HEAPSORT` runs in $\Theta(n \lg n)$ time in the worst case.

**Exercise 5.24** We can easily modify the `SORT` specification (Figure 1.2 on page 6) so that instead of sorting numbers, we are sorting `KEYED` items in nondecreasing order of their keys. `HEAPSORT` can be trivially modified to meet this specification. Any sorting algorithm meeting this specification is said to be *stable* if the resulting sorted array always has elements with equal keys in the same order as they were initially. Show that `HEAPSORT`, when modified to sort `KEYED` items, is not stable.
Exercise 5.25 Consider the following scheduling problem. We have a collection of jobs, each having a natural number ready time $r_i$, a positive integer execution time $e_i$, and a positive integer deadline $d_i$, such that $d_i \geq r_i + e_i$. At each natural number time instant $t$, we wish to schedule the job with minimum deadline satisfying the following conditions:

- $t \geq r_i$ (i.e., the job is ready); and
- if the job has already been executed for $a < e_i$ time units, then $t + e_i - a \leq d_i$ (i.e., the job can meet its deadline).

Note that this scheduling strategy may preempt jobs, and that it will discard jobs that have been delayed so long that they can no longer meet their deadlines. Give an algorithm to produce such a schedule, when given a sequence of jobs ordered by ready time. Your algorithm should store the ready jobs in an InvertedPriorityQueue. (You do not need to give an implementation of InvertedPriorityQueue.) Show that your algorithm operates in $O(k \lg n)$ time, where $k$ is length of the schedule and $n$ is the number of jobs. You may assume that $k \geq n$ and that PUT and REMOVE MIN both operate in $\Theta(\lg n)$ time in the worst case.

Exercise 5.26 The game of craps consists of a sequence of rolls of two six-sided dice with faces numbered 1 through 6. The first roll is known as the come-out roll. If the come-out roll is a 7 or 11 (the sum of the top faces of the two dice), the shooter wins. If the come-out roll is a 2, 3, or 12, the shooter loses. Otherwise, the result is known as the point. The shooter continues to roll until the result is either the point (in which case the shooter wins) or a 7 (in which case the shooter loses).

a. For each of the values 2 through 12, compute the probability that any single roll is that value.

b. A field bet can be made on any roll. For each dollar bet, the payout is determined by the roll as follows:

- 2 or 12: $3$ (i.e., the bettor pays $1$ and receives $3$, netting $2$);
- 3, 4, 9, 10 or 11: $2$;
- 5, 6, 7, or 8: 0.

Calculate the expected payout for a field bet.
c. A pass-line bet is a bet, placed prior to the come-out roll, that the shooter will win. For each dollar bet, the payout for a win is $2, whereas the payout for a loss is 0. Compute the expected payout for a pass-line bet. [Hint: The problem is much easier if you define a finite probability space, ignoring those rolls that don’t affect the outcome. In order to do this you will need to use conditional probabilities (e.g., given that the roll is either a 5 or a 7, the probability that it is a 5).]

Exercise 5.27 Let $S$ be a discrete probability space, and let $f$ be a discrete random variable over $S$. Let $a$ be any positive real number. Prove Markov’s Inequality:

$$P(f \geq a) \leq \frac{E[f]}{a}.$$  \hspace{1cm} (5.3)

5.9 Chapter Notes

Both heaps and heap sort were introduced by Williams [113]. The linear-time construction of a binary heap is due to Floyd [39]. Leftist heaps were introduced by Crane [26]; see also Knuth [80]. Skew heaps were introduced by Sleator and Tarjan [100]. Randomized heaps were introduced by Gambin and Malinowski [46].

Other implementations of priority queues have been defined based on the idea of a heap. For example, binomial queues were introduced by Vuillemin [108]. Lazy binomial queues and Fibonacci heaps, each of which provide $\text{PUT}$ and $\text{REMOVEMAX}$ operations with amortized running times in $O(1)$ and $O(\lg n)$, respectively, were introduced by Fredman and Tarjan [44].

The information on craps in Exercise 5.26 is taken from Silberstang [98].