Chapter 5 Priority Queues and Heaps

Wall Street during the Bankers Panic of 1907. From the New York Public Library's Digital Gallery, in the Irma and Paul Milstein Division of United States History, Local History and Genealogy. 1907. Public domain image.

Contents

Instead of having people perform stock transactions by yelling at each other on a trading floor, the NASDAQ uses a massive network of computers to process the buying and selling of stocks of listed companies. At the heart of this network are highly reliable systems known as ***matching engines***, which match the trades of buyers and sellers. A simplification of how such a system works is in terms of a ***continuous limit order book***, where buyers post bids to buy a number of shares in a given stock at or below a specified price and sellers post offers to sell a number of shares of a given stock at or above a specified price. For example, suppose a company, Example.com, has a continuous limit order book that has the following bids and offers currently open:

STOCK: EXAMPLE.COM

Buy Orders			Sell Orders		
Shares	**Price**	**Time**	**Shares**	**Price**	**Time**
1000	4.05	20 s	500	4.06	13 s
100	4.05	6 s	2000	4.07	46 s
2100	4.03	20 s	400	4.07	22 s
1000	4.02	3 s	3000	4.10	54 s
2500	4.01	81 s	500	4.12	2 s
			3000	4.20	58 s
			800	4.25	33 s
			100	4.50	92 s

Buy and sell orders are organized according to a ***price-time priority***, meaning that price has highest priority and if there are multiple orders for the same price, then ones that have been in the order book the longest have higher priority. When a new order is entered into the order book, the matching engine determines if a trade can be immediately executed and if so, then it performs the appropriate matches according to price-time priority. For example, if a new buy order for Example.com came in for 1000 shares at a price of at most 4.10, then it would be matched with the sell order for 500 shares at 4.06 and 500 shares of the sell order for 2000 shares at 4.07. This transaction would reduce the latter sell order to 1500 shares at 4.07. Similarly, if a new sell order came in for 100 shares at 4.00, it would be matched with 100 shares of the buy order for 1000 shares at 4.05. Note in this case that the new sell order would not be matched with the buy order for 100 shares at 4.05, because the one for 1000 shares is older.

Such systems amount to two instances of the data structure we discuss in this chapter—the ***priority queue***—one for buy orders and one for sell orders. This data structure performs element removals based on priorities assigned to elements when they are inserted. Dealing with such priorities efficiently requires more effort than is needed for simple stacks and queues, however, and we give an efficient priority queue implementation in this chapter known as the ***heap***. In addition, we show how priority queues can be used for sorting purposes, giving rise to sorting algorithms known as ***selection-sort***, ***insertion-sort***, and ***heap-sort***.

5.1 Priority Queues

Applications, such as in a matching engine for performing stock trades, often require ranking objects according to parameters or properties, called "keys," that are assigned for each object in a collection. Formally, we define a ***key*** to be an object that is assigned to an element as a specific attribute for that element, which can be used to identify, rank, or weight that element. Note that the key is assigned to an element, typically by a user or application.

Total Orders

The concept of a key as an arbitrary object type is therefore quite general. But, in order to deal consistently with such a general definition for keys and still be able to discuss when one key has priority over another, we need a way of robustly defining a rule for comparing keys. That is, a priority queue needs a comparison rule that will never contradict itself. In order for a comparison rule, denoted by $\leq$, to be robust in this way, it must define a ***total order*** relation, which is to say that the comparison rule is defined for every pair of keys and it must satisfy the following properties:

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

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

Methods for Priority Queues

A ***priority queue*** is a container of elements, each having an associated key that is provided at the time the element is inserted. The name "priority queue" is motivated from the fact that keys determine the "priority" used to pick elements to be removed. The two fundamental methods of a priority queue P are as follows:

insert(k, e):	Insert an element e with key k into P.
removeMin():	Return and remove from P an element with the smallest key, that is, an element whose key is less than or equal to that of every other element in P.

5.2 PQ-Sort, Selection-Sort, and Insertion-Sort

In this section, we discuss how to use a priority queue to sort a set of elements.

PQ-Sort: Using a Priority Queue to Sort

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

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

We give pseudocode for this algorithm in Algorithm 5.1, assuming that C is an array of n elements. The algorithm works correctly for any priority queue P, no matter how P is implemented. However, the running time of the algorithm is determined by the running times of operations insert and removeMin, which do depend on how P is implemented. Indeed, PQ-Sort should be considered more a sorting "scheme" than a sorting "algorithm," because it does not specify how the priority queue P is implemented. The PQ-Sort scheme is the paradigm of several popular sorting algorithms, including selection-sort, insertion-sort, and heap-sort.

```
Algorithm PQ-Sort($C, P$):
    Input: An $n$-element array, $C$, index from 1 to $n$, and a priority queue $P$ that
        compares keys, which are elements of $C$, using a total order relation
    Output: The array $C$ sorted by the total order relation
    for $i \leftarrow 1$ to $n$ do
        $e \leftarrow C[i]$
        $P$.insert$(e, e)$        // the key is the element itself
    for $i \leftarrow 1$ to $n$ do
        $e \leftarrow P$.removeMin()        // remove a smallest element from $P$
        $C[i] \leftarrow e$
```

Algorithm 5.1: Algorithm PQ-Sort. Note that the elements of the input array C serve both as keys and elements of the priority queue P.

5.2.1 Selection-Sort

As our first implementation of a priority queue P, consider storing the elements of P and their keys in an unordered list, S. Let us say that S is a general list implemented with either an array or a doubly linked list (the choice of specific implementation will not affect performance, as we will see). Thus, the elements of S are pairs (k, e), where e is an element of P and k is its key. A simple way of implementing method insert(k, e) of P is to add the new pair object $p = (k, e)$ at the end of the list S. This implementation of method insert takes $O(1)$ time, independent of whether the list is implemented using an array or a linked list (see Section 2.2.2). This choice also means that S will be unsorted, for always inserting items at the end of S does not take into account the ordering of the keys. As a consequence, to perform the operation removeMin on P, we must inspect all the elements of list S to find an element $p = (k, e)$ of S with minimum k. Thus, no matter how the list S is implemented, these search methods on P each take $O(n)$ time, where n is the number of elements in P at the time the method is executed. Moreover, each search method runs in $\Omega(n)$ time even in the best case, since each requires searching the entire list to find a minimum element. That is, each such method runs in $\Theta(n)$ time. Thus, by using an unsorted list to implement a priority queue, we achieve constant-time insertion, but the removeMin operation takes linear time.

Analysis of PQ-Sort with an Unsorted List

If we implement the priority queue P with an unsorted list, as described above, then the first phase of PQ-Sort takes $O(n)$ time, for we can insert each element in constant time. In the second phase, assuming we can compare two keys in constant time, the running time of each removeMin operation is proportional to the number of elements currently in P. Thus, the bottleneck computation in this implementation is the repeated "selection" of the minimum element from an unsorted list in phase 2. For this reason, this sorting algorithm is better known as ***selection-sort***. (See Figure 5.2 for an illustration of this algorithm and Algorithm 5.3 for a pseudocode description.)

The size of P starts at n and incrementally decreases with each removeMin until it becomes 0. Thus, the first removeMin operation takes time $O(n)$, the second one takes time $O(n-1)$, and so on, until the last (nth) operation takes time $O(1)$. Therefore, the total time needed for the second phase is proportional to the sum

$$n + (n-1) + \cdots + 2 + 1 = \sum_{i=1}^{n} i$$

By Theorem 1.13, we have $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$. Thus, the second phase takes time $O(n^2)$, as does the entire selection-sort algorithm.

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

Figure 5.2: An illustration of selection-sort run on list $S = (7,4,8,2,5,3,9)$. This algorithm follows the two-phase PQ-Sort scheme and uses a priority queue P implemented with an unsorted list. In the first phase, we repeatedly remove the first element from S and insert it into P, as the last element of the list implementing P. Note that at the end of the first phase, P is a copy of what was initially S. In the second phase, we repeatedly perform removeMin operations on P, each of which requires that we scan the entire list implementing P, and we add the elements returned at the end of S.

```
Algorithm SelectionSort($A$):
   Input: An array $A$ of $n$ comparable elements, indexed from 1 to $n$
   Output: An ordering of $A$ so that its elements are in nondecreasing order
   for $i \leftarrow 1$ to $n-1$ do
      // Find the index, $s$, of the smallest element in $A[i..n]$.
      $s \leftarrow i$
      for $j \leftarrow i+1$ to $n$ do
         if $A[j] < A[s]$ then
            $s \leftarrow j$
      if $i \neq s$ then
         // Swap $A[i]$ and $A[s]$
         $t \leftarrow A[s]$; $A[s] \leftarrow A[i]$; $A[i] \leftarrow t$
   return $A$
```

Algorithm 5.3: The selection-sort algorithm, described as an ***in-place*** algorithm, where the input list, S, is given as an array, A, and only a constant amount of memory is used in addition to that used by A. In each iteration, $A[1..i-1]$ is the sorted portion and $A[i..n]$ is the unsorted priority queue.

5.2.2 Insertion-Sort

An alternative implementation of a priority queue P also uses a list S, except this time let us store items ordered by key values. Thus, the first element in S is always an element with smallest key in P. Therefore, we can implement the removeMin method of P simply by removing the first element in S. Assuming that S is implemented with a linked list or an array that supports constant-time front-element removal (see Section 2.2.2), finding and removing the minimum in P in this case takes $O(1)$ time. Thus, using a sorted list allows for simple and fast implementations of priority queue access and removal methods.

This benefit comes at a cost, however, for now the insert method of P requires that we scan through the list S to find the appropriate place to insert the new element and key. Thus, implementing the insert method of P now requires $O(n)$ time, where n is the number of elements in P at the time the method is executed. In summary, when using a sorted list to implement a priority queue, insertion runs in linear time whereas finding and removing the minimum can be done in $O(1)$ time.

Analysis of PQ-Sort with a Sorted List

If we implement the priority queue P using a sorted list, as described above, then we improve the running time of the second phase of the PQ-Sort method to $O(n)$, for each operation removeMin on P now takes $O(1)$ time. Unfortunately, the first phase now becomes the bottleneck for the running time. Indeed, in the worst case, the running time of each insert operation is proportional to the number of elements that are currently in the priority queue, which starts out having size zero and increases in size until it has size n. The first insert operation takes time $O(1)$, the second one takes time $O(2)$, and so on, until the last (nth) operation takes time $O(n)$, in the worst case. Thus, if we use a sorted list to implement P, then the first phase becomes the bottleneck phase. This sorting algorithm is therefore better known as ***insertion-sort***, for the bottleneck in this sorting algorithm involves the repeated "insertion" of a new element at the appropriate position in a sorted list. (See Figure 5.4 for an illustration of this algorithm and Algorithm 5.5 for a pseudocode description.)

Analyzing the running time of insertion-sort, we note that the first phase takes $O(\sum_{i=1}^{n} i)$ time in the worst case. Again, by recalling Theorem 1.13, the first phase runs in $O(n^2)$ time, and hence so does the entire algorithm. Therefore, both selection-sort and insertion-sort both have a running time that is $O(n^2)$.

Still, although selection-sort and insertion-sort are similar, they actually have some interesting differences. For instance, note that selection-sort always takes $\Omega(n^2)$ time, for selecting the minimum in each step of the second phase requires scanning the entire priority-queue sequence. The running time of insertion-sort, on the other hand, varies depending on the input sequence. For example, if the input sequence S is in reverse order, then insertion-sort runs in $O(n)$ time.

		list S	***priority queue*** P
Input		$(7,4,8,2,5,3,9)$	$()$
Phase 1	(a)	$(4,8,2,5,3,9)$	(7)
	(b)	$(8,2,5,3,9)$	$(4,7)$
	(c)	$(2,5,3,9)$	$(4,7,8)$
	(d)	$(5,3,9)$	$(2,4,7,8)$
	(e)	$(3,9)$	$(2,4,5,7,8)$
	(f)	(9)	$(2,3,4,5,7,8)$
	(g)	$()$	$(2,3,4,5,7,8,9)$
Phase 2	(a)	(2)	$(3,4,5,7,8,9)$
	(b)	$(2,3)$	$(4,5,7,8,9)$
	$\vdots$	$\vdots$	$\vdots$
	(g)	$(2,3,4,5,7,8,9)$	$()$

Figure 5.4: Schematic visualization of the execution of insertion-sort on list $S = (7,4,8,2,5,3,9)$. This algorithm follows the two-phase PQ-Sort scheme and uses a priority queue P, implemented by means of a sorted list. In the first phase, we repeatedly remove the first element of S and insert it into P, by scanning the list implementing P, until we find the correct position for the element. In the second phase, we repeatedly perform removeMin operations on P, each of which returns the first element of the list implementing P, and we add the element at the end of S.

```
Algorithm InsertionSort($A$):
    Input: An array, $A$, of $n$ comparable elements, indexed from 1 to $n$
    Output: An ordering of $A$ so that its elements are in nondecreasing order.
    for $i \leftarrow 2$ to $n$ do
        $x \leftarrow A[i]$
        // Put $x$ in the right place in $A[1..i]$, moving larger elements up as needed.
        $j \leftarrow i$
        while $j > 1$ and $x < A[j-1]$ do
            $A[j] \leftarrow A[j-1]$        // move $A[j-1]$ up one cell
            $j \leftarrow j-1$
        $A[j] \leftarrow x$
    return $A$
```

Algorithm 5.5: The insertion-sort algorithm, described as an ***in-place*** algorithm, where the input list, S, is given as an array, A, and only a constant amount of memory is used in addition to that used by A. In each iteration, $A[1..i-1]$ is the sorted priority queue and $A[i..n]$ is the unsorted input list.

5.3 Heaps

An implementation of a priority queue that is efficient for both the insert(k, e) and removeMin() operations is to use a ***heap***. This data structure allows us to perform both insertions and removals in logarithmic time. The fundamental way the heap achieves this improvement is to abandon the idea of storing elements and keys in a list and store elements and keys in a binary tree instead.

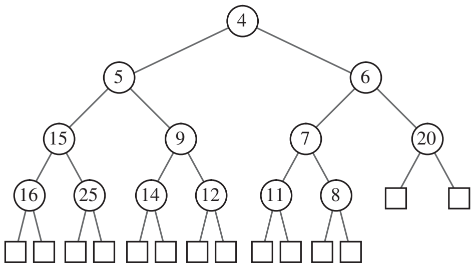

Figure 5.6: Example of a heap storing 13 integer keys. The last node is the one storing key 8, and external nodes are empty.

A heap (see Figure 5.6) is a binary tree T that stores a collection of keys at its internal nodes and that satisfies two additional properties: a relational property defined in terms of the way keys are stored in T and a structural property defined in terms of T itself. Also, in our definition of a heap, we assume the external nodes of T do not store keys or elements and serve only as "place-holders." The relational property for T is the following:

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

As a consequence of this property, the keys encountered on a path from the root to an external node of T are in nondecreasing order. Also, a minimum key is always stored at the root of T. For the sake of efficiency, we want the heap T to have as small a height as possible. We enforce this desire by insisting that the heap T satisfy an additional structural property:

Complete Binary Tree: A binary tree T with height h is ***complete*** if the levels $0, 1, 2, \ldots, h-1$ have the maximum number of nodes possible (that is, level i has 2^i nodes, for $0 \leq i \leq h-1$) and in level $h-1$ all the internal nodes are to the left of the external nodes.

By saying that all the internal nodes on level $h-1$ are "to the left" of the external nodes, we mean that all the internal nodes on this level will be visited before any external nodes on this level in an inorder traversal. (See Figure 5.6.) By insisting that a heap T be complete, we identify another important node in a heap T, other than the root—namely, the ***last node*** of T, which we define to be the right-most, deepest internal node of T. (See Figure 5.6.)

Implementing a Priority Queue with a Heap

Our heap-based priority queue consists of the following (see Figure 5.7):

- ***heap***: A complete binary tree T whose elements are stored at internal nodes and have keys satisfying the heap-order property. For each internal node v of T, we denote the key of the element stored at v as $k(v)$.
- ***last***: A reference to the last node of T.
- ***comp***: A comparison rule that defines the total order relation among the keys. Without loss of generality, we assume that ***comp*** maintains the minimum element at the root. If instead we wish the maximum element to be at the root, then we can redefine our comparison rule accordingly.

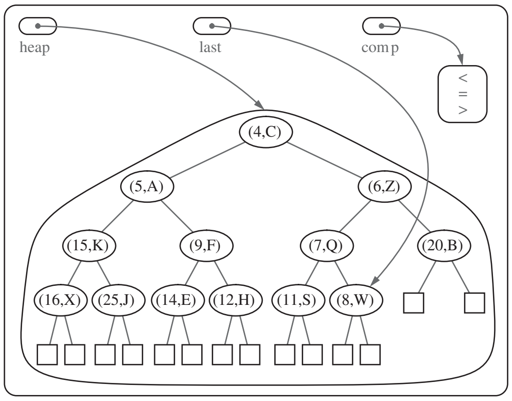

Figure 5.7: A heap-based priority queue storing integer keys and text elements.

The efficiency of this implementation is based on the following fact.

Theorem 5.1: *A heap T storing n keys has height $h = \lceil \log(n+1) \rceil$.*

Proof: Since T is complete, the number of internal nodes of T is at least

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

This lower bound is achieved when there is only one internal node on level $h-1$. Alternately, we observe that the number of internal nodes of T is at most

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

This upper bound is achieved when all the 2^{h-1} nodes on level $h-1$ are internal. Since the number of internal nodes is equal to the number n of keys, $2^{h-1} \le n$ and $n \le 2^h - 1$. Thus, by taking logarithms of both sides of these two inequalities, we see that $h \le \log n + 1$ and $\log(n+1) \le h$, which implies that $h = \lceil \log(n+1) \rceil$. ■

Thus, if we can perform update operations on a heap in time proportional to its height, then those operations will run in logarithmic time.

5.3.1 An Array-Based Structure for Binary Trees

A simple structure for representing a binary tree T, especially when it is complete, is based on a straightforward way of numbering the nodes of T.

Level Numbering the Nodes of a Binary Tree

For every node v of T, let $p(v)$ be the integer defined as follows:

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

The numbering function p is known as a ***level numbering*** of the nodes in a binary tree T, because it numbers the nodes on each level of T in increasing order from left to right, although it may skip some numbers. (See Figure 5.8.)

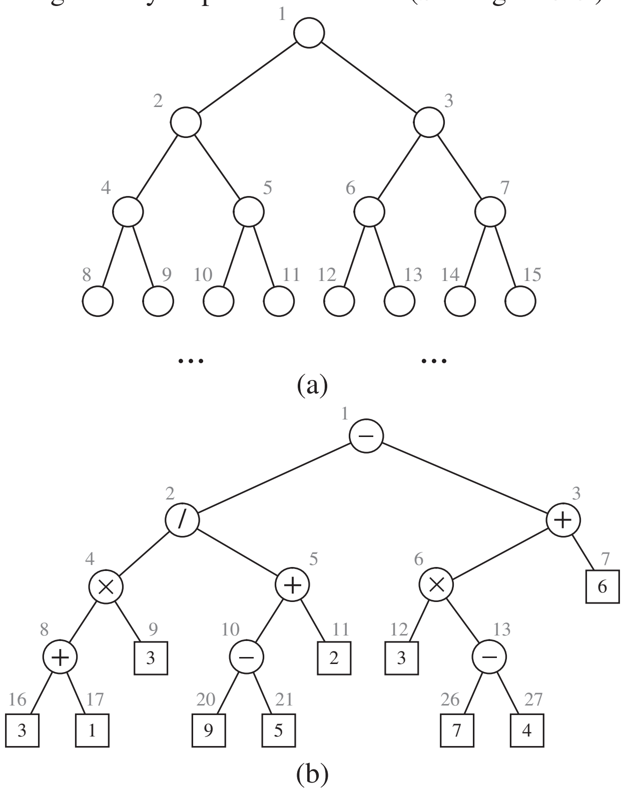

Figure 5.8: Example binary tree level numberings. (a) a complete binary tree; (b) a binary expression tree that is not complete.

Using Level Numbers as Array Indices

The level numbering function p suggests a representation of a binary tree T using an array, S, such that node v of T is associated with the element of S at index $p(v)$. (See Figure 5.9.) Alternatively, we could realize the S using an extendable array. (See Section 1.4.2.)

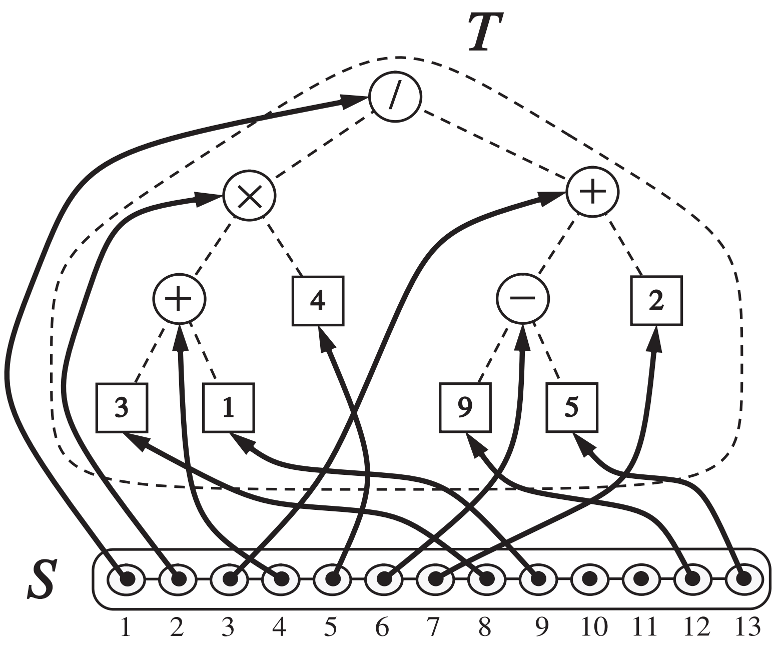

Figure 5.9: Representing a binary tree T using an array S.

Such an implementation is simple and fast, for we can use it to easily perform the methods root, parent, leftChild, rightChild, sibling, isInternal, isExternal, and isRoot by using simple arithmetic operations on the numbers $p(v)$. (See Exercise R-5.3.)

Let n be the number of nodes of T, let p_M be the maximum value of $p(v)$ for T, and note that the array S must have size $N \geq p_M$, since we need to store a value in the array S for each node in T. Also, if T is not complete, then the array S may have empty cells, for non-existing internal nodes of T. In fact, in the worst case, N could be exponential in n, which makes this implementation a poor choice for general binary trees. Nevertheless, if T is a heap, and, hence, T is complete and all its external nodes are empty, then we can save additional space by restricting S to exclude external nodes whose index is past that of the last internal node in the tree. Thus, if T is a heap, then we can set N to be $O(n)$.

An Array-Based Representation of a Heap

When a heap, T, is implemented with an array, as described above, the index of the last node w is always equal to n, and the first empty external node z has index equal to $n + 1$. (See Figure 5.10.) Note that this index for z is valid even for the following cases:

- If the current last node w is the right-most node on its level, then z is the left-most node of the bottommost level (see Figure 5.10b).
- If T has no internal nodes (that is, the priority queue is empty and the last node in T is not defined), then z is the root of T.

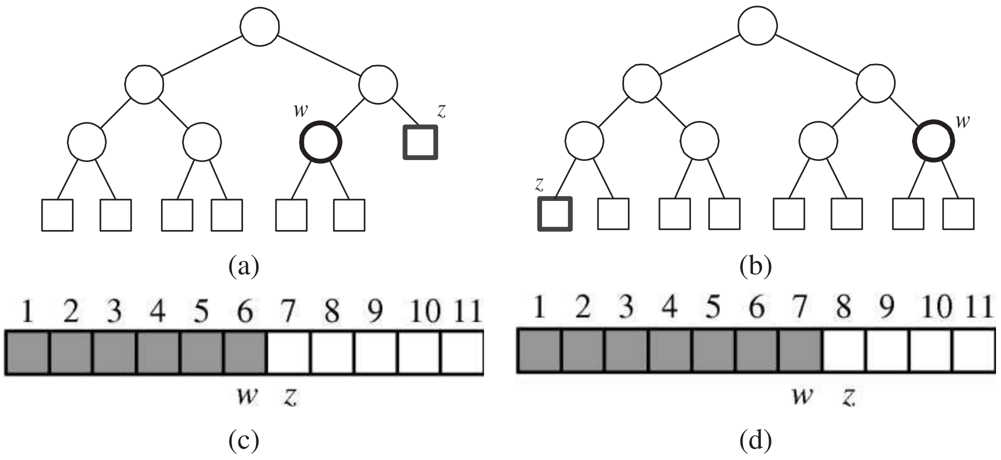

Figure 5.10: Last node w and first external node z in a heap: (a) regular case where z is right of w; (b) case where z is left-most on bottom level. The array representation of (a) is shown in (c); the representation of (b) is shown in (d).

The simplifications that come from representing the heap T with an array aid in our methods for implementing a priority queue. For example, adding a new node in a heap implemented with an array can be done in $O(1)$ time (assuming no array expansion is necessary), for it simply involves assigning an element to a single cell in the array. Likewise, removing a node in a heap implemented with an array just involves clearing out a cell in the array, which also can be done in $O(1)$ time. In addition, finding the element with smallest key can easily be performed in $O(1)$, just by accessing the item stored at the root of the heap (which is at index 1 in the array). Moreover, because T is a complete binary tree, the array associated with heap T in an array-based implementation of a binary tree has $2n+1$ elements, $n+1$ of which are place-holder external nodes by our convention. Indeed, since all the external nodes have indices higher than any internal node, we don't even have to explicitly store all the external nodes. (See Figure 5.10.)

5.3.2 Insertion in a Heap

Let us consider how to perform the insert method for a priority queue implemented with a heap, T. In order to store a new key-element pair (k, e) into T, we need to add a new internal node to T. In order to keep T as a complete binary tree, we must add this new node so that it becomes the new last node of T. That is, we must identify the correct external node z where we can perform an expandExternal(z) operation, which replaces z with an internal node (with empty external-node children), and then insert the new element at z. (See Figure 5.11a–b.) Node z is called the ***insertion position***.

Usually, node z is the external node immediately to the right of the last node w. (See Figure 5.10a.) In any case, by our array implementation of T, the insertion position z is stored at index $n + 1$, where n is the current size of the heap. Thus, we can identify the node z in constant time in the array implementing T. After then performing expandExternal(z), node z becomes the last node, and we store the new key-element pair (k, e) in it, so that $k(z) = k$.

Up-Heap Bubbling after an Insertion

After this action, the tree T is complete, but it may violate the heap-order property. Hence, unless node z is the root of T (that is, the priority queue was empty before the insertion), we compare key $k(z)$ with the key $k(u)$ stored at the parent u of z. If $k(u) > k(z)$, then we need to restore the heap-order property, which can be locally achieved by swapping the key-element pairs stored at z and u. (See Figure 5.11c–d.) This swap causes the new key-element pair (k, e) to move up one level. Again, the heap-order property may be violated, and we continue swapping going up in T until no violation of heap-order property occurs. (See Figure 5.11e–h.)

The upward movement by means of swaps is conventionally called ***up-heap bubbling***. A swap either resolves the violation of the heap-order property or propagates it one level up in the heap. In the worst case, up-heap bubbling causes the new key-element pair to move all the way up to the root of heap T. (See Figure 5.11.) Thus, in the worst case, the running time of method insert is proportional to the height of T, that is, it is $O(\log n)$ because T is complete.

If T is implemented with an array, then we can find the new last node z immediately in $O(1)$ time. For example, we could extend an array-based implementation of a binary tree, so as to add a method that returns the node with index $n + 1$, that is, with level number $n + 1$, as defined in Section 5.3.1. Alternately, we could even define an add method, which adds a new element at the first external node z, at rank $n + 1$ in the array. If, on the other hand, the heap T is implemented with a linked structure, then finding the insertion position z is a little more involved. (See Exercise C-5.4.)

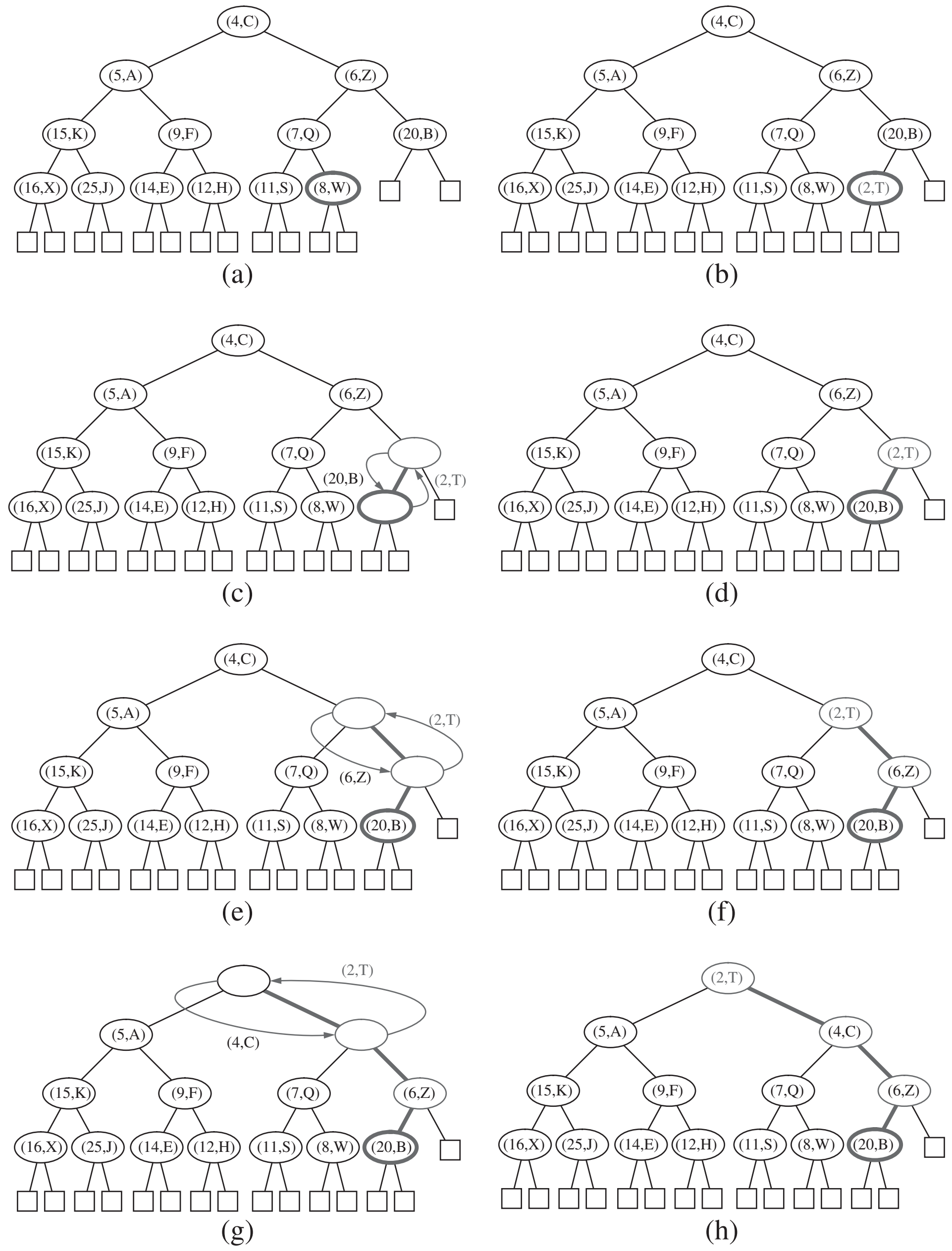

Figure 5.11: Insertion of a new element with key 2 into the heap of Figure 5.7: (a) initial heap; (b) adding a new last node to the right of the old last node; (c)–(d) swap to locally restore the partial order property; (e)–(f) another swap; (g)–(h) final swap.

5.3.3 Removal in a Heap

The algorithm for performing method removeMin using heap T is illustrated in Figure 5.12.

We know that an element with the smallest key is stored at the root r of the heap T (even if there is more than one smallest key). However, unless r is the only internal node of T, we cannot simply delete node r, because this action would disrupt the binary tree structure. Instead, we access the last node w of T, copy its key-element pair to the root r, and then delete the last node by performing the update operation removeAboveExternal(u), where $u = T.\mathsf{rightChild}(w)$. This operation removes the parent, w, of u, together with the node u itself, and replaces w with its left child. (See Figure 5.12a–b.)

After this constant-time action, we need to update our reference to the last node, which can be done simply by referencing the node at rank n (after the removal) in the array implementing the tree T.

Down-Heap Bubbling after a Removal

We are not done, however, for, even though T is now complete, T may now violate the heap-order property. To determine whether we need to restore the heap-order property, we examine the root r of T. If both children of r are external nodes, then the heap-order property is trivially satisfied and we are done. Otherwise, we distinguish two cases:

- If the left child of r is internal and the right child is external, let s be the left child of r.
- Otherwise (both children of r are internal), let s be a child of r with the smallest key.

If the key $k(r)$ stored at r is greater than the key $k(s)$ stored at s, then we need to restore the heap-order property, which can be locally achieved by swapping the key-element pairs stored at r and s. (See Figure 5.12c–d.) Note that we shouldn't swap r with s's sibling. The swap we perform restores the heap-order property for node r and its children, but it may violate this property at s; hence, we may have to continue swapping down T until no violation of the heap-order property occurs. (See Figure 5.12e–h.)

This downward swapping process is called ***down-heap bubbling***. A swap either resolves the violation of the heap-order property or propagates it one level down in the heap. In the worst case, a key-element pair moves all the way down to the level immediately above the bottom level. (See Figure 5.12.) Thus, the running time of method removeMin is, in the worst case, proportional to the height of heap T, that is, it is $O(\log n)$.

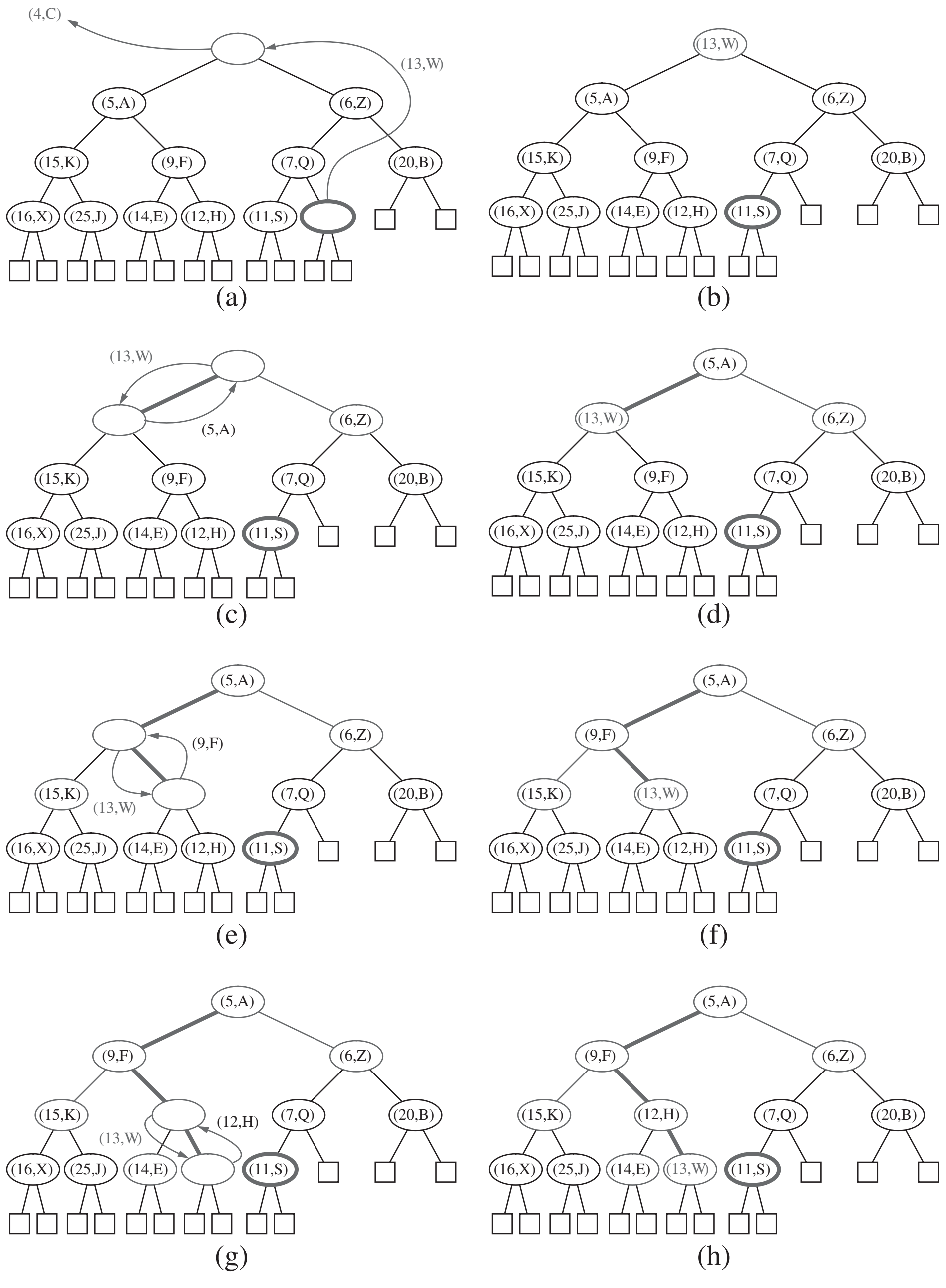

Figure 5.12: Removal of the element with the smallest key from a heap: (a)–(b) deletion of the last node, whose key-element pair gets stored into the root; (c)–(d) swap to locally restore the heap-order property; (e)–(f) another swap; (g)–(h) final swap.

Pseudo-code for the method for performing the insert(k, e) method in a heap storing n items with an extendable array, A, is shown in Algorithm 5.13, and the method for performing the removeMin() method is shown in Algorithm 5.13.

```
Algorithm HeapInsert(k, e):
   Input: A key-element pair
   Output: An update of the array, A, of n elements, for a heap, to add (k, e)
   $n \leftarrow n + 1$
   $A[n] \leftarrow (k, e)$
   $i \leftarrow n$
   while $i > 1$ and $A[\lfloor i/2 \rfloor] > A[i]$ do
      Swap $A[\lfloor i/2 \rfloor]$ and $A[i]$
      $i \leftarrow \lfloor i/2 \rfloor$
```

Algorithm 5.13: Insertion in a heap represented with an array. We identify the parent of i as the cell at index $\lfloor i/2 \rfloor$. The while-loop implements up-heap bubbling.

```
Algorithm HeapRemoveMin():
   Input: None
   Output: An update of the array, A, of n elements, for a heap, to remove and
      return an item with smallest key
   $temp \leftarrow A[1]$
   $A[1] \leftarrow A[n]$
   $n \leftarrow n - 1$
   $i \leftarrow 1$
   while $i < n$ do
      if $2i + 1 \leq n$ then   // this node has two internal children
         if $A[i] \leq A[2i]$ and $A[i] \leq A[2i + 1]$ then
            return $temp$        // we have restored the heap-order property
         else
            Let $j$ be the index of the smaller of $A[2i]$ and $A[2i + 1]$
            Swap $A[i]$ and $A[j]$
            $i \leftarrow j$
      else  // this node has zero or one internal child
         if $2i \leq n$ then   // this node has one internal child (the last node)
            if $A[i] > A[2i]$ then
               Swap $A[i]$ and $A[2i]$
         return $temp$        // we have restored the heap-order property
   return $temp$        // we reached the last node or an external node
```

Algorithm 5.13: Removing the minimum item in a heap represented with an array. We identify the two children of i as the cells at indices $2i$ and $2i+1$. The while-loop implements down-heap bubbling.

Performance

Table 5.14 shows the running time of the priority queue methods for the heap implementation of a priority queue, assuming that the heap T is itself implemented with an array.

Operation	**Time**
insert	$O(\log n)$
removeMin	$O(\log n)$

Table 5.14: Performance of a priority queue that is implemented with a heap, which is in turn implemented with an array-based structure for a binary tree. We denote with n the number of elements in the priority queue at the time a method is executed. The space requirement is $O(N)$, where $N \geq n$ is the size of the array used to implement the heap.

In short, each of the fundamental priority queue methods can be performed in $O(\log n)$ time, where n is the number of elements at the time the method is executed. We can also implement a method for simply returning, but not removing the minimum item, in $O(1)$ time (just by returning the item stored at the root of the heap).

The above analysis of the running time of the methods of a priority queue implemented with a heap is based on the following facts:

- The height of heap T is $O(\log n)$, since T is complete.
- In the worst case, up-heap and down-heap bubbling take time proportional to the height of T.
- Finding the insertion position in the execution of insert and updating the last node position in the execution of removeMin takes constant time.
- The heap T has n internal nodes, each storing a reference to a key and a reference to an element.

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

5.4 Heap-Sort

Let us consider again the PQ-Sort sorting scheme from Section 5.2, which uses a priority queue P to sort a list S. If we implement the priority queue P with a heap, then, during the first phase, each of the n insert operations takes time $O(\log k)$, where k is the number of elements in the heap at the time. Likewise, during the second phase, each of the n removeMin operations also runs in time $O(\log k)$, where k is the number of elements in the heap at the time. Since we always have $k \leq n$, each such operation runs in $O(\log n)$ time in the worst case. Thus, each phase takes $O(n \log n)$ time, so the entire priority-queue sorting algorithm runs in $O(n \log n)$ time when we use a heap to implement the priority queue. This sorting algorithm is better known as ***heap-sort***, and its performance is summarized in the following theorem.

Theorem 5.2: *The heap-sort algorithm sorts a list S of n comparable elements in $O(n \log n)$ time.*

Recalling Table 1.7, we stress that the $O(n \log n)$ running time of heap-sort is much better than the $O(n^2)$ running time for selection-sort and insertion-sort. In addition, there are several modifications we can make to the heap-sort algorithm to improve its performance in practice.

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

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

The above variation of heap-sort is ***in-place***, since we use only a constant amount of space in addition to the list itself. Instead of transferring elements out of the list and then back in, we simply rearrange them. We illustrate in-place heap-sort in Figure 5.15. In general, we say that a sorting algorithm is in-place if it uses only a constant amount of memory in addition to the memory needed for the objects being sorted themselves. The advantage of an in-place sorting algorithm in practice is that such an algorithm can make the most efficient use of the main memory of the computer it is running on.

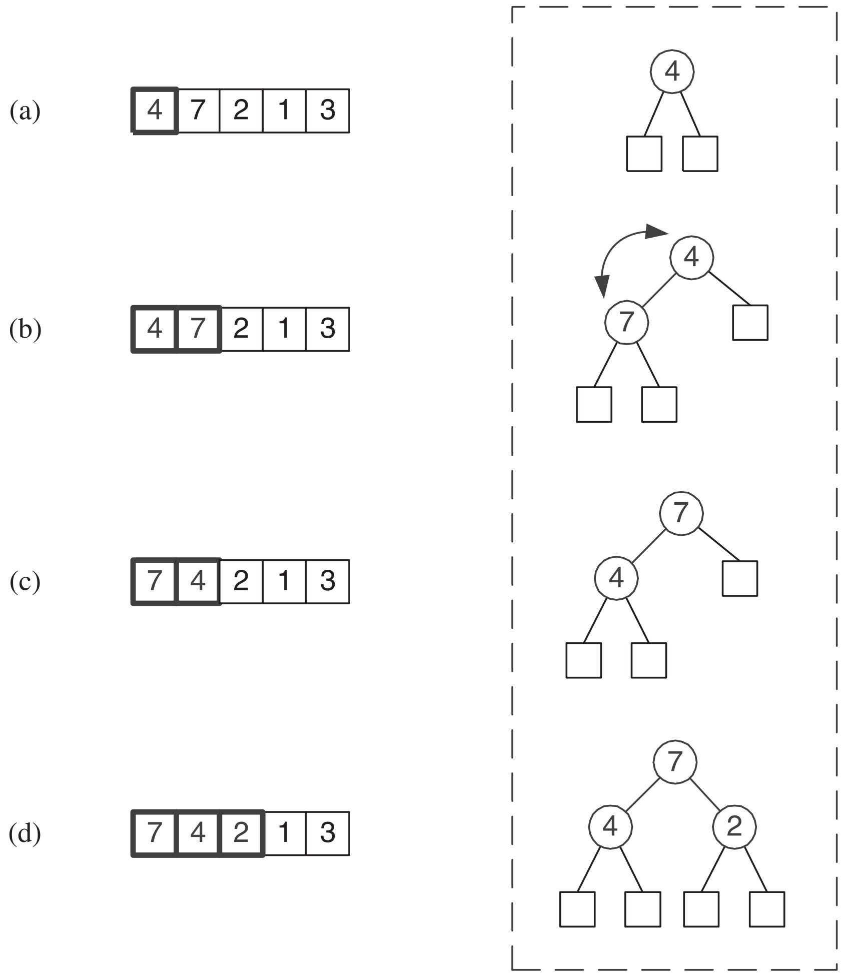

Figure 5.15: First three steps of Phase 1 of in-place heap-sort. The heap portion of the array is highlighted with thick lines. Next to the array, we draw a binary tree view of the heap, even though this tree is not actually constructed by the in-place algorithm.

Bottom-Up Heap Construction

The analysis of the heap-sort algorithm shows that we can construct a heap storing n key-element pairs in $O(n \log n)$ time, by means of n successive insert operations, and then use that heap to extract the elements in order. However, if all the keys to be stored in the heap are given in advance, there is an alternative ***bottom-up*** construction method that runs in $O(n)$ time.

We describe this method here, observing that it could be used in the heap-sort algorithm instead of filling up the heap using a series of n insert operations. For simplicity of exposition, we describe this bottom-up heap construction assuming the number n of keys is an integer of the type

$$n = 2^h - 1.$$

That is, the heap is a complete binary tree with every level being full, so the heap has height

$$h = \log(n + 1).$$

We describe bottom-up heap construction as a recursive algorithm, as shown in Algorithm 5.16, which we call by passing a list storing the keys for which we wish to build a heap. We describe the construction algorithm as acting on keys, with the understanding that their elements accompany them. That is, the items stored in the tree T are key-element pairs.

```
Algorithm BottomUpHeap($S$):
  Input: A list $S$ storing $n = 2^h - 1$ keys
  Output: A heap $T$ storing the keys in $S$.
  if $S$ is empty then
      return an empty heap (consisting of a single external node).
  Remove the first key, $k$, from $S$.
  Split $S$ into two lists, $S_1$ and $S_2$, each of size $(n-1)/2$.
  $T_1 \leftarrow$ BottomUpHeap($S_1$)
  $T_2 \leftarrow$ BottomUpHeap($S_2$)
  Create binary tree $T$ with root $r$ storing $k$, left subtree $T_1$, and right subtree $T_2$.
  Perform a down-heap bubbling from the root $r$ of $T$, if necessary.
  return $T$
```

Algorithm 5.16: Recursive bottom-up heap construction.

This construction algorithm is called “bottom-up” heap construction because of the way each recursive call returns a subtree that is a heap for the elements it stores. That is, the “heapification” of T begins at its external nodes and proceeds up the tree as each recursive call returns. For this reason, some authors refer to the bottom-up heap construction as the “heapify” operation.

We illustrate bottom-up heap construction in Figure 5.17 for $h = 4$.

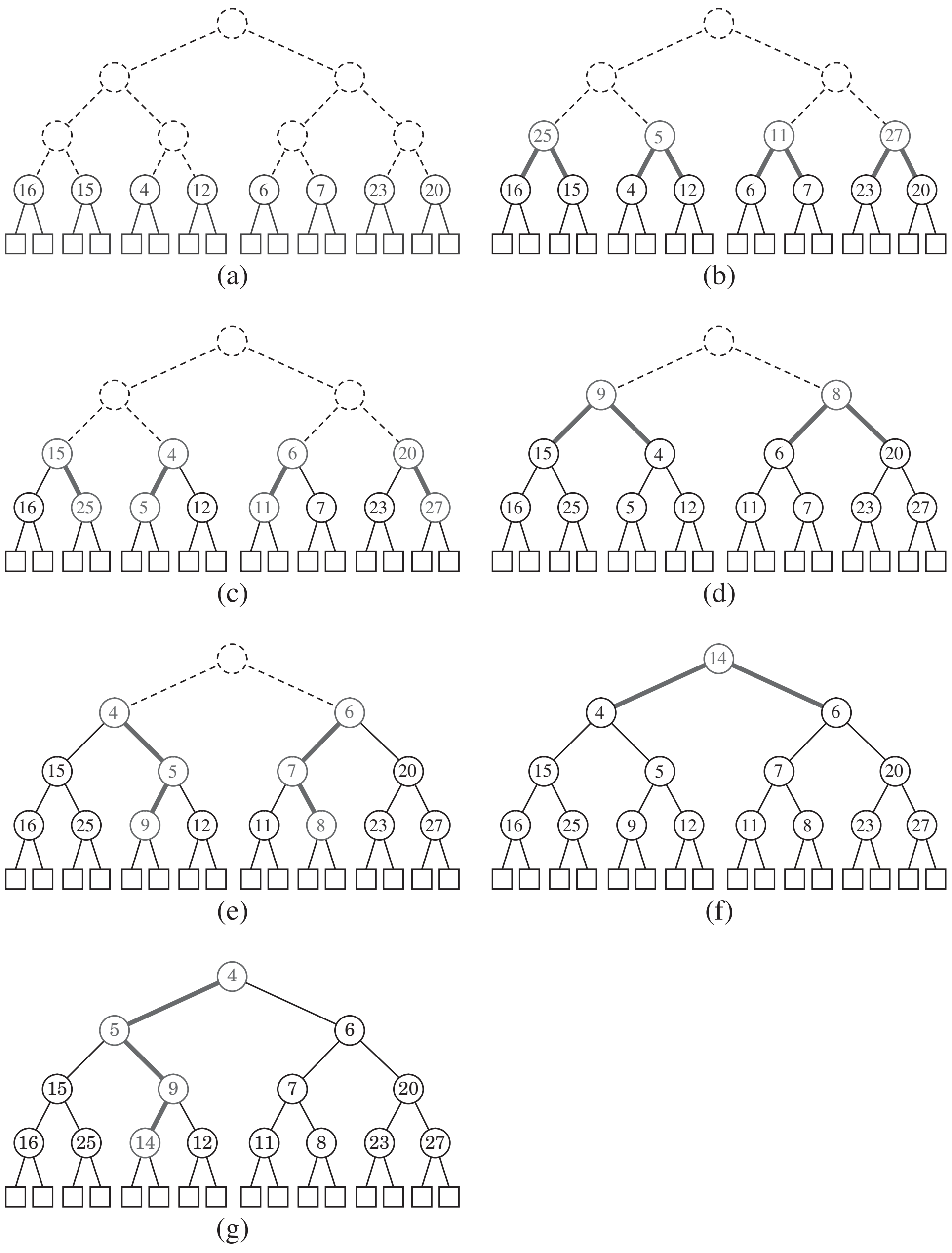

Figure 5.17: Bottom-up construction of a heap with 15 keys: (a) we begin by constructing 1-key heaps on the bottom level; (b)–(c) we combine these heaps into 3-key heaps and then (d)–(e) 7-key heaps, until (f)–(g) we create the final heap. The paths of the down-heap bubblings are highlighted with thick lines.

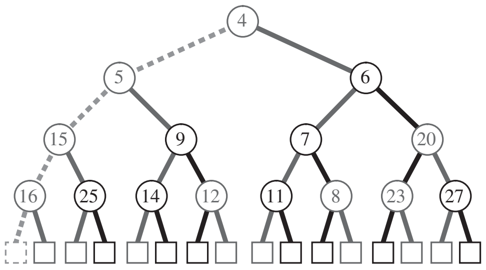

Figure 5.18: Visual justification of the linear running time of bottom-up heap construction, where the paths associated with the internal nodes have been highlighted alternating gray and black. For example, the path associated with the root consists of the internal nodes storing keys 4, 6, 7, and 11, plus an external node.

Bottom-up heap construction is asymptotically faster than incrementally inserting n keys into an initially empty heap, as the following theorem shows.

Theorem 5.3: *The bottom-up construction of a heap with n items takes $O(n)$ time.*

Proof: We analyze bottom-up heap construction using a "visual" approach, which is illustrated in Figure 5.18.

Let T be the final heap, let v be an internal node of T, and let $T(v)$ denote the subtree of T rooted at v. In the worst case, the time for forming $T(v)$ from the two recursively formed subtrees rooted at v's children is proportional to the height of $T(v)$. The worst case occurs when down-heap bubbling from v traverses a path from v all the way to a bottommost external node of $T(v)$. Consider now the path $p(v)$ of T from node v to its inorder successor external node, that is, the path that starts at v, goes to the right child of v, and then goes down leftward until it reaches an external node. We say that path $p(v)$ is ***associated with*** node v. Note that $p(v)$ is not necessarily the path followed by down-heap bubbling when forming $T(v)$. Clearly, the length (number of edges) of $p(v)$ is equal to the height of $T(v)$. Hence, forming $T(v)$ takes time proportional to the length of $p(v)$, in the worst case. Thus, the total running time of bottom-up heap construction is proportional to the sum of the lengths of the paths associated with the internal nodes of T.

Note that for any two internal nodes u and v of T, paths $p(u)$ and $p(v)$ do not share edges, although they may share nodes. (See Figure 5.18.) Therefore, the sum of the lengths of the paths associated with the internal nodes of T is no more than the number of edges of heap T, that is, no more than $2n$. We conclude that the bottom-up construction of heap T takes $O(n)$ time. ■

To summarize, Theorem 5.3 says that the first phase of heap-sort can be implemented to run in $O(n)$ time. Unfortunately, the running time of the second phase of heap-sort is $\Omega(n \log n)$ in the worst case. We will not justify this lower bound until Chapter 8, however.

5.5 Extending Priority Queues

We can additionally augment the two fundamental methods for a priority queue with supporting methods, such as size(), isEmpty(), and the following:

minElement():	Return (but do not remove) an element of P with the smallest key.
minKey():	Return (but do not remove) the smallest key in P.

Both of these methods return error conditions if the priority queue is empty. If we implement a priority queue using an ordered list or a heap, then we can perform these operations in $O(1)$ time each.

Comparators

In addition, we can augment a priority queue to support parameterizing the data structure in the way it does comparisons, by using a concept known a ***comparator***. This pattern specifies the way in which we compare keys, and is designed to support the most general and reusable form of a priority queue. For such a design, we should not rely on the keys to provide their comparison rules, for such rules might not be what a user desires (particularly for multidimensional data). Instead, we use special ***comparator*** objects that are external to the keys to supply the comparison rules. A comparator is an object that compares two keys. We assume that a priority queue P is given a comparator when P is constructed, and we might also imagine the ability of a priority queue to be given a new comparator if its old one ever becomes "out of date." When P needs to compare two keys, it uses the comparator it was given to perform the comparison. Thus, a programmer can write a general priority queue implementation that can work correctly in a wide variety of contexts. Formally, a comparator object provides the following methods, each of which takes two keys and compares them (or reports an error if the keys are incomparable). The methods of a comparator include the following:

isLess(a, b):	True if and only if a is less than b.
isLessOrEqualTo(a, b):	True if and only if a is less than or equal to b.
isEqualTo(a, b):	True if and only if a and b are equal.
isGreater(a, b):	True if and only if a is greater than b.
isGreaterOrEqualTo(a, b):	True if and only if a is greater than or equal to b.
isComparable(a):	True if and only if a can be compared.

Locators

We conclude this section by discussing a concept that allows us to extend a priority queue to have additional functionality, which will be useful, for example, in some of the graph algorithms discussed later in this book. As we saw with lists and binary trees, abstracting positional information in a container is a very powerful tool.

There are applications where we need to keep track of elements as they are being moved around inside a container, however. A concept that fulfills this need is the ***locator***. A locator is a mechanism for maintaining the association between an element and its current position in a container. A locator "sticks" with a specific element, even if the element changes its position in the container.

A locator is like a coat check; we can give our coat to a coat-room attendant, and we receive back a coat check, which is a "locator" for our coat. The position of our coat relative to the other coats can change, as other coats are added and removed, but our coat check can always be used to retrieve our coat. The important thing to remember about a locator is that it follows its item, even if it changes position.

Like a coat check, we can now imagine getting something back when we insert an element in a container—we can get back a locator for that element. This locator in turn can be used later to refer to the element within the container, for example, to specify that this element should be removed from the container. Viewed abstractly, a locator ℓ supports the following methods:

element():	Return the element of the item associated with ℓ.
key():	Return the key of the item associated with ℓ.

For the sake of concreteness, we next discuss how we can use locators to extend the repertoire of operations of a priority queue to include methods that return locators and take locators as arguments.

Using Locators with a Priority Queue

We can use locators in a natural way in the context of a priority queue. A locator in such a scenario stays attached to an item inserted in the priority queue, and allows us to access the item in a generic manner, independent of the specific implementation of the priority queue. This ability is important for a priority queue implementation, for there are no positions, *per se*, in a priority queue, since we do not refer to items by any notions of "index" or "node," and instead are based on the use of keys.

Locator-Based Priority Queue Methods

By using locators, we can extend a priority queue with the following methods that access and modify a priority queue P:

$\mathsf{min}()$:	Return the locator to an item of P with smallest key.
$\mathsf{insert}(k, e)$:	Insert a new item with element e and key k into P and return a locator to the item.
$\mathsf{remove}(\ell)$:	Remove from P the item with locator ℓ.
$\mathsf{replaceElement}(\ell, e)$:	Replace with e and return the element of the item of P with locator ℓ.
$\mathsf{replaceKey}(\ell, k)$:	Replace with k and return the key of the item of P with locator ℓ.

Locator-based access runs in $O(1)$ time, while a key-based access, which must look for the element via a search in an entire list or heap, runs in $O(n)$ time in the worst case. In addition, some applications call for us to restrict the operation replaceKey so that it only increases or decreases the key. This restriction can be done by defining new methods increaseKey or decreaseKey, for example, which would take a locator as an argument, which can be implemented to run in amortized $O(1)$ time using more sophisticated heap structures, such as Fibonacci heaps, which we don't discuss here. Further applications of such priority queue methods are given in Chapter 14.

Comparison of Different Priority Queue Implementations

In Table 5.19, we compare running times of the priority queue methods defined in this section for the unsorted-list, sorted-list, and heap implementations.

Method	**Unsorted List**	**Sorted List**	**Heap**
key, replaceElement	$O(1)$	$O(1)$	$O(1)$
minElement, min, minKey	$O(n)$	$O(1)$	$O(1)$
insert	$O(1)$	$O(n)$	$O(\log n)$
removeMin	$O(n)$	$O(1)$	$O(\log n)$
remove	$O(1)$	$O(1)$	$O(\log n)$
replaceKey	$O(1)$	$O(n)$	$O(\log n)$

Table 5.19: Comparison of the running times of the priority queue methods for the unsorted-list, sorted-list, and heap implementations. We denote with n the number of elements in the priority queue at the time a method is executed.

5.6 Exercises

Reinforcement

R-5.1 Give an in-place pseudocode description of selection-sort assuming that arrays are indexed from 0 to $n-1$, as they are in Java and C/C++, instead of from 1 to n, as given in Algorithm 5.3.

R-5.2 Give an in-place pseudocode description of insertion-sort assuming that arrays are indexed from 0 to $n-1$, as they are in Java and C/C++, instead of from 1 to n, as given in Algorithm 5.5.

R-5.3 Let T be a binary tree with n nodes that is realized with a array, S, and let p be the level numbering of the nodes in T, as given in Section 5.3.1. Give pseudocode descriptions of each of the methods root, parent, leftChild, rightChild, isInternal, isExternal, and isRoot.

R-5.4 Illustrate the performance of the selection-sort algorithm on the following input sequence: $(22, 15, 36, 44, 10, 3, 9, 13, 29, 25)$.

R-5.5 Illustrate the performance of the insertion-sort algorithm on the input sequence of the previous problem.

R-5.6 Give an example of a worst-case list with n elements for insertion-sort, and show that insertion-sort runs in $\Omega(n^2)$ time on such a list.

R-5.7 Describe how to modify the pseudocode of Algorithm 5.13 so that it uses a standard array of size N instead of an extendable array.

R-5.8 Where may an item with largest key be stored in a heap?

R-5.9 Illustrate the performance of the heap-sort algorithm on the following input list: $(2, 5, 16, 4, 10, 23, 39, 18, 26, 15)$.

R-5.10 Suppose a binary tree T is implemented using a array S, as described in Section 5.3.1. If n items are stored in S in sorted order, starting with index 1, is the tree T a heap?

R-5.11 Is there a heap T storing seven distinct elements such that a preorder traversal of T yields the elements of T in sorted order? How about an inorder traversal? How about a postorder traversal?

R-5.12 Show that the sum $\sum_{i=1}^{n} \log i$, which appears in the analysis of heap-sort, is $\Omega(n \log n)$.

R-5.13 Show the steps for removing key 16 from the heap of Figure 5.6.

R-5.14 Show the steps for replacing 5 with 18 in the heap of Figure 5.6.

R-5.15 Draw an example of a heap whose keys are all the odd numbers from 1 to 59 (with no repeats), such that the insertion of an item with key 32 would cause up-heap bubbling to proceed all the way up to a child of the root (replacing that child's key with 32).

Creativity

C-5.1 Consider a modification of the selection-sort algorithm where instead of finding the smallest element in the unsorted part of the array and swapping it to be in its proper place, as given in Algorithm 5.3, we instead perform a sequence of swaps to move the smallest element to its proper location, as shown in Algorithm 5.20. Show that this algorithm, which is known as ***bubble-sort***, runs in $\Theta(n^2)$ time.

```
Algorithm BubbleSort($A$):
  Input: An array $A$ of $n$ comparable elements, indexed from 1 to $n$
  Output: An ordering of $A$ so that its elements are in nondecreasing order.
   for $i \leftarrow 1$ to $n - 1$ do
       // Move the smallest element in $A[i + 1 .. n]$ to $A[i]$ by swaps.
       for $j \leftarrow n$ down to $i + 1$ do
           if $A[j - 1] > A[j]$ then
               Swap $A[j - 1]$ and $A[j]$
   return $A$
```

Algorithm 5.20: The bubble-sort algorithm.

C-5.2 Given an array, A, of elements that come from a total order, an ***inversion*** in A is a pair of elements that are in the wrong order, that is, a pair (i, j) where $i < j$ and $A[i] > A[j]$. Show that the in-place version of insertion-sort, as given in Algorithm 5.5, runs in $O(n + I)$ time, where I is the number of inversions in the array A.

C-5.3 Assuming the input to the sorting problem is given in an array A, give a pseudocode description of in-place heap-sort. That is, describe how to implement the heap-sort algorithm using only the array A and at most six additional (base-type) variables.

C-5.4 Suppose the binary tree T used to implement a heap can be accessed using only the methods of a binary tree. That is, we cannot assume T is implemented as an array. Given a reference to the current last node, v, describe an efficient algorithm for finding the insertion point (that is, the new last node) using just the methods of the binary tree interface. Be sure and handle all possible cases. What is the running time of this method?

C-5.5 Show that, for any n, there is a sequence of insertions in a heap that requires $\Omega(n \log n)$ time to process.

C-5.6 We can represent a path from the root to a node of a binary tree by means of a binary string, where 0 means "go to the left child" and 1 means "go to the right child." Design a logarithmic-time algorithm for finding the last node of a heap holding n elements based on the this representation.

C-5.7 Show that the problem of finding the kth smallest element in a heap takes at least $\Omega(k)$ time in the worst case.

C-5.8 Develop an algorithm that computes the kth smallest element of a set of n distinct integers in $O(n + k \log n)$ time.

C-5.9 Let T be a heap storing n keys. Give an efficient algorithm for reporting all the keys in T that are smaller than or equal to a given query key x (which is not necessarily in T). For example, given the heap of Figure 5.6 and query key $x = 7$, the algorithm should report $4, 5, 6, 7$. Note that the keys do not need to be reported in sorted order. Ideally, your algorithm should run in $O(k)$ time, where k is the number of keys reported.

C-5.10 Define a ***min-max stack*** to be a data structure that supports the stack operations of push() and pop() for objects that come from a total order, as well as operations min() and max(), which return, but do not delete the minimum or maximum element in the min-max stack, respectively. Describe an implementation for a min-max stack that can perform each of these operations in $O(1)$ time.

C-5.11 Define a ***min-max queue*** to be a data structure that supports the queue operations of enqueue() and dequeue() for objects that come from a total order, as well as operations min() and max(), which return, but do not delete the minimum or maximum element in the min-max queue, respectively. Describe an implementation for a min-max queue that can perform each of these operations in amortized $O(1)$ time.

Applications

A-5.1 In data smoothing applications, such as in visualizing trends in stock averages over time, it is useful to keep track of the median of a set, S, of numbers as values are inserted or removed from S. Describe a method for maintaining the median of an initially empty set, S, subject to an operation, insert(x), which inserts the value x, and an operation, median$()$, which returns the median in S. Each of these methods should run in at most $O(\log n)$ time, where n is the number of values in S.

A-5.2 In a ***discrete event simulation***, a physical system, such as a galaxy or solar system, is modeled as it changes over time based on simulated forces. The objects being modeled define events that are scheduled to occur in the future. Each event, e, is associated with a time, t_e, in the future. To move the simulation forward, the event, e, that has smallest time, t_e, in the future needs to be processed. To process such an event, e is removed from the set of pending events, and a set of physical forces are calculated for this event, which can optionally create a constant number of new events for the future, each with its own event time. Describe a way to support event processing in a discrete event simulation in $O(\log n)$ time, where n is the number of events in the system.

A-5.3 Suppose you work for a major airline and are given the job of writing the algorithm for processing upgrades into first class on various flights. Any frequent flyer can request an upgrade for his or her up-coming flight using this online system. Frequent flyers have different priorities, which are determined first by frequent flyer status (which can be, in order, silver, gold, platinum, and super) and then, if there are ties, by length of time in the waiting list. In addition, at any time prior to the flight, a frequent flyer can cancel his or her upgrade request (for instance, if he or she wants to take a different flight), using a confirmation code they got when he or she made his or her upgrade request. When it is time to determine upgrades for a flight that is about to depart, the gate agents inform the system of the number, k, of seats available in first class, and it needs to match those seats with the k highest-priority passengers on the waiting list. Describe a system that can process upgrade requests and cancellations in $O(\log n)$ time and can determine the k highest-priority flyers on the waiting list in $O(k \log n)$ time, where n is the number of frequent flyers on the waiting list.

A-5.4 Suppose you are designing a system for buying and selling stocks using a continuous limit order book strategy, as described in the beginning of this chapter. Describe how to use priority queues to process such buy and sell orders, including those that can be processed immediately and those that have some or all of their shares needing to wait until some future match to be processed. You don't need to list all pending buy and sell orders, but you do need to always display the pending buy order(s) with the highest and lowest prices and the pending order(s) with the highest and lowest prices. You may assume that buy and sell orders are never cancelled and remain in the system until they are matched. Describe a system that can process buy and sell orders in amortized $O(\log n)$ time, where n is the number of orders in the system.

A-5.5 The problem of accurately summing a set S of n floating-point numbers, $S = \{x_1, x_2, \ldots, x_n\}$, on a real-world computer is more challenging than might first appear. For example, using the standard accumulating-sum algorithm to compute the harmonic numbers,

$$H_n = \sum_{i=1}^{n} \frac{1}{n},$$

in floating-point produces a sequence that converges to a constant instead of growing to infinity as a function close to $\ln n$. This phenomenon is due to the fact that floating-point addition can suffer from round-off errors, especially if the two numbers being added differ greatly in magnitude. Fortunately, there are several useful heuristic summation algorithms for dealing with this issue, with one of them being the following. Let x_i and x_j be two numbers in S with smallest magnitudes. Sum x_i and x_j and return the result to S. Repeat this process until only one number remains in S, which is the sum. This algorithm will often produce a more accurate sum than the straightforward summation algorithm, because it tends to minimize the round-off errors of each partial sum. Describe how to implement this floating-point summation algorithm in $O(n \log n)$ time.

A-5.6 One of the oldest applications used on the Internet is FTP, the file transfer protocol. The definition for this protocol traces its roots back to 1971, before the Internet even existed, and its formulation for the Internet was given in 1980. Its primary purpose is for transferring files from one computer to another over the Internet. Suppose you are writing an Internet file transfer application, similar to FTP, and it is your job to write the code that takes a raw sequence of network packets that were sent from one computer and reassembles that sequence to reconstitute the file at the receiving computer. The n packets in the input are numbered by their sequence numbers, which starts at some value, N, and goes incrementally in order to $N + n - 1$. The packets are usually not received in this order, however. Thus, to reconstruct the file at the receiving computer, your method must re-order the packets so that they are sorted by their sequence numbers. In studying example inputs, you notice that the input sequence of packets is usually not that far from being sorted. That is, you notice in the examples you studied that each input packet is ***proximate*** to its output position, which means that if a packet is at position i in the input sequence, then its position in the sorted output sequence is somewhere in the range $[i-50,\ i+50]$. Describe a method for sorting a sequence of n such packets by their sequence numbers in $O(n)$ time, assuming that every input packet is proximate to its output position.

Chapter Notes

Knuth's book on sorting and searching [131] describes the motivation and history for the selection-sort, insertion-sort, and heap-sort algorithms. The heap-sort algorithm is due to Williams [216], and the linear-time heap construction algorithm is due to Floyd [73]. Additional algorithms and analyses for heaps and heap-sort variations can be found in papers by Bentley [29], Carlsson [41], Gonnet and Munro [86], McDiarmid and Reed [150], and Schaffer and Sedgewick [185]. The locator pattern (also described in [89]) appears to be new. Higham [97] studies the numerical stability of several floating-point summation algorithms, including the one discussed in Exercise A-5.5.