7 Heapsort

In this chapter, we introduce another sorting algorithm. Like merge sort, but unlike insertion sort, heapsort's running time is $O(n \lg n)$. Like insertion sort, but unlike merge sort, heapsort sorts in place: only a constant number of array elements are stored outside the input array at any time. Thus, heapsort combines the better attributes of the two sorting algorithms we have already discussed.

Heapsort also introduces another algorithm design technique: the use of a data structure, in this case one we call a "heap," to manage information during the execution of the algorithm. Not only is the heap data structure useful for heapsort, it also makes an efficient priority queue. The heap data structure will reappear in algorithms in later chapters.

We note that the term "heap" was originally coined in the context of heapsort, but it has since come to refer to "garbage-collected storage," such as the programming language Lisp provides. Our heap data structure is *not* garbage-collected storage, and whenever we refer to heaps in this book, we shall mean the structure defined in this chapter.

7.1 Heaps

The ***(binary) heap*** data structure is an array object that can be viewed as a complete binary tree (see Section 5.5.3), as shown in Figure 7.1. Each node of the tree corresponds to an element of the array that stores the value in the node. The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point. An array A that represents a heap is an object with two attributes: $length[A]$, which is the number of elements in the array, and $heap\text{-}size[A]$, the number of elements in the heap stored within array A. That is, although $A[1 \,..\, length[A]]$ may contain valid numbers, no element past $A[heap\text{-}size[A]]$, where $heap\text{-}size[A] \leq length[A]$, is an element of the heap. The root of the tree is $A[1]$, and given the index i of a node, the indices of its parent PARENT(i), left child LEFT(i), and right child RIGHT(i) can be computed simply:

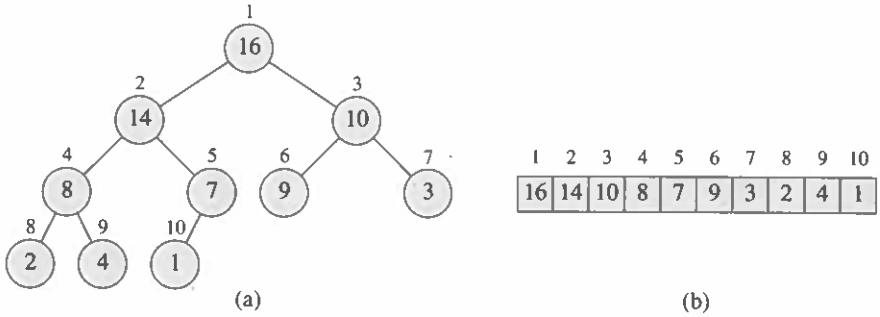

Figure 7.1 A heap viewed as (a) a binary tree and (b) an array. The number within the circle at each node in the tree is the value stored at that node. The number next to a node is the corresponding index in the array.

PARENT(i)
  return $\lfloor i/2 \rfloor$

LEFT(i)
  return $2i$

RIGHT(i)
  return $2i + 1$

On most computers, the LEFT procedure can compute $2i$ in one instruction by simply shifting the binary representation of i left one bit position. Similarly, the RIGHT procedure can quickly compute $2i + 1$ by shifting the binary representation of i left one bit position and shifting in a 1 as the low-order bit. The PARENT procedure can compute $\lfloor i/2 \rfloor$ by shifting i right one bit position. In a good implementation of heapsort, these three procedures are often implemented as "macros" or "in-line" procedures.

Heaps also satisfy the ***heap property***: for every node i other than the root,

$$A[\text{PARENT}(i)] \geq A[i] , \tag{7.1}$$

that is, the value of a node is at most the value of its parent. Thus, the largest element in a heap is stored at the root, and the subtrees rooted at a node contain smaller values than does the node itself.

We define the ***height*** of a node in a tree to be the number of edges on the longest simple downward path from the node to a leaf, and we define the height of the tree to be the height of its root. Since a heap of n elements is based on a complete binary tree, its height is $\Theta(\lg n)$ (see Exercise 7.1-2). We shall see that the basic operations on heaps run in time at most proportional to the height of the tree and thus take $O(\lg n)$ time. The remainder of this chapter presents five basic procedures and

shows how they are used in a sorting algorithm and a priority-queue data structure.

- The HEAPIFY procedure, which runs in $O(\lg n)$ time, is the key to maintaining the heap property (7.1).
- The BUILD-HEAP procedure, which runs in linear time, produces a heap from an unordered input array.
- The HEAPSORT procedure, which runs in $O(n \lg n)$ time, sorts an array in place.
- The EXTRACT-MAX and INSERT procedures, which run in $O(\lg n)$ time, allow the heap data structure to be used as a priority queue.

Exercises

7.1-1
What are the minimum and maximum numbers of elements in a heap of height h?

7.1-2
Show that an n-element heap has height $\lfloor \lg n \rfloor$.

7.1-3
Show that the largest element in a subtree of a heap is at the root of the subtree.

7.1-4
Where in a heap might the smallest element reside?

7.1-5
Is an array that is in reverse sorted order a heap?

7.1-6
Is the sequence $\langle 23, 17, 14, 6, 13, 10, 1, 5, 7, 12 \rangle$ a heap?

7.2 Maintaining the heap property

HEAPIFY is an important subroutine for manipulating heaps. Its inputs are an array A and an index i into the array. When HEAPIFY is called, it is assumed that the binary trees rooted at LEFT(i) and RIGHT(i) are heaps, but that $A[i]$ may be smaller than its children, thus violating the heap property (7.1). The function of HEAPIFY is to let the value at $A[i]$ "float down" in the heap so that the subtree rooted at index i becomes a heap.

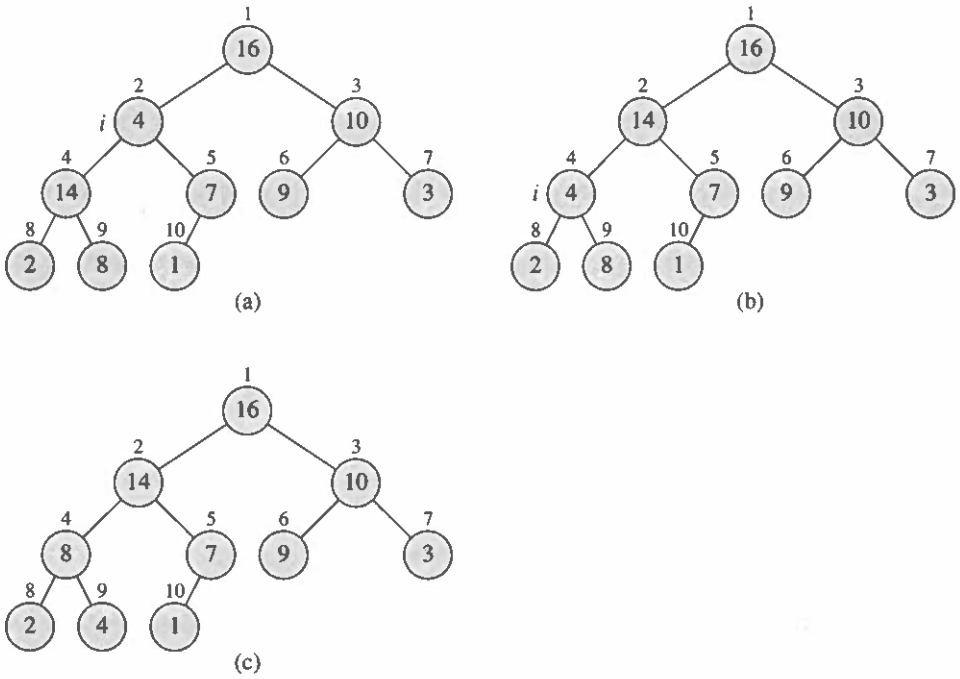

Figure 7.2 The action of HEAPIFY$(A, 2)$, where *heap-size*$[A] = 10$. **(a)** The initial configuration of the heap, with $A[2]$ at node $i = 2$ violating the heap property since it is not larger than both children. The heap property is restored for node 2 in **(b)** by exchanging $A[2]$ with $A[4]$, which destroys the heap property for node 4. The recursive call HEAPIFY$(A, 4)$ now sets $i = 4$. After swapping $A[4]$ with $A[9]$, as shown in **(c)**, node 4 is fixed up, and the recursive call HEAPIFY$(A, 9)$ yields no further change to the data structure.

HEAPIFY(A, i)

```
1  l ← LEFT(i)
2  r ← RIGHT(i)
3  if l ≤ heap-size[A] and A[l] > A[i]
4     then largest ← l
5     else largest ← i
6  if r ≤ heap-size[A] and A[r] > A[largest]
7     then largest ← r
8  if largest ≠ i
9     then exchange A[i] ↔ A[largest]
10          HEAPIFY(A, largest)
```

Figure 7.2 illustrates the action of HEAPIFY. At each step, the largest of the elements $A[i]$, $A[\text{LEFT}(i)]$, and $A[\text{RIGHT}(i)]$ is determined, and its index is stored in *largest*. If $A[i]$ is largest, then the subtree rooted at node i is a heap and the procedure terminates. Otherwise, one of the two children has the largest element, and $A[i]$ is swapped with $A[largest]$, which causes

node i and its children to satisfy the heap property. The node *largest*, however, now has the original value $A[i]$, and thus the subtree rooted at *largest* may violate the heap property. Consequently, HEAPIFY must be called recursively on that subtree.

The running time of HEAPIFY on a subtree of size n rooted at given node i is the $\Theta(1)$ time to fix up the relationships among the elements $A[i]$, $A[\text{LEFT}(i)]$, and $A[\text{RIGHT}(i)]$, plus the time to run HEAPIFY on a subtree rooted at one of the children of node i. The children's subtrees each have size at most $2n/3$—the worst case occurs when the last row of the tree is exactly half full—and the running time of HEAPIFY can therefore be described by the recurrence

$$T(n) \leq T(2n/3) + \Theta(1) .$$

The solution to this recurrence, by case 2 of the master theorem (Theorem 4.1), is $T(n) = O(\lg n)$. Alternatively, we can characterize the running time of HEAPIFY on a node of height h as $O(h)$.

Exercises

7.2-1
Using Figure 7.2 as a model, illustrate the operation of HEAPIFY$(A, 3)$ on the array $A = \langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0 \rangle$.

7.2-2
What is the effect of calling HEAPIFY(A, i) when the element $A[i]$ is larger than its children?

7.2-3
What is the effect of calling HEAPIFY(A, i) for $i > \textit{heap-size}[A]/2$?

7.2-4
The code for HEAPIFY is quite efficient in terms of constant factors, except possibly for the recursive call in line 10, which might cause some compilers to produce inefficient code. Write an efficient HEAPIFY that uses an iterative control construct (a loop) instead of recursion.

7.2-5
Show that the worst-case running time of HEAPIFY on a heap of size n is $\Omega(\lg n)$. (*Hint:* For a heap with n nodes, give node values that cause HEAPIFY to be called recursively at every node on a path from the root down to a leaf.)

7.3 Building a heap

We can use the procedure HEAPIFY in a bottom-up manner to convert an array $A[1..n]$, where $n = length[A]$, into a heap. Since the elements in the subarray $A[(\lfloor n/2 \rfloor + 1)..n]$ are all leaves of the tree, each is a 1-element heap to begin with. The procedure BUILD-HEAP goes through the remaining nodes of the tree and runs HEAPIFY on each one. The order in which the nodes are processed guarantees that the subtrees rooted at children of a node i are heaps before HEAPIFY is run at that node.

```
BUILD-HEAP(A)
1  heap-size[A] ← length[A]
2  for i ← ⌊length[A]/2⌋ downto 1
3      do HEAPIFY(A, i)
```

Figure 7.3 shows an example of the action of BUILD-HEAP.

We can compute a simple upper bound on the running time of BUILD-HEAP as follows. Each call to HEAPIFY costs $O(\lg n)$ time, and there are $O(n)$ such calls. Thus, the running time is at most $O(n \lg n)$. This upper bound, though correct, is not asymptotically tight.

We can derive a tighter bound by observing that the time for HEAPIFY to run at a node varies with the height of the node in the tree, and the heights of most nodes are small. Our tighter analysis relies on the property that in an n-element heap there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h (see Exercise 7.3-3).

The time required by HEAPIFY when called on a node of height h is $O(h)$, so we can express the total cost of BUILD-HEAP as

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) . \tag{7.2}$$

The last summation can be evaluated by substituting $x = 1/2$ in the formula (3.6), which yields

$$\begin{aligned} \sum_{h=0}^{\infty} \frac{h}{2^h} &= \frac{1/2}{(1-1/2)^2} \\ &= 2 . \end{aligned}$$

Thus, the running time of BUILD-HEAP can be bounded as

$$\begin{aligned} O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) &= O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) \\ &= O(n) . \end{aligned}$$

Hence, we can build a heap from an unordered array in linear time.

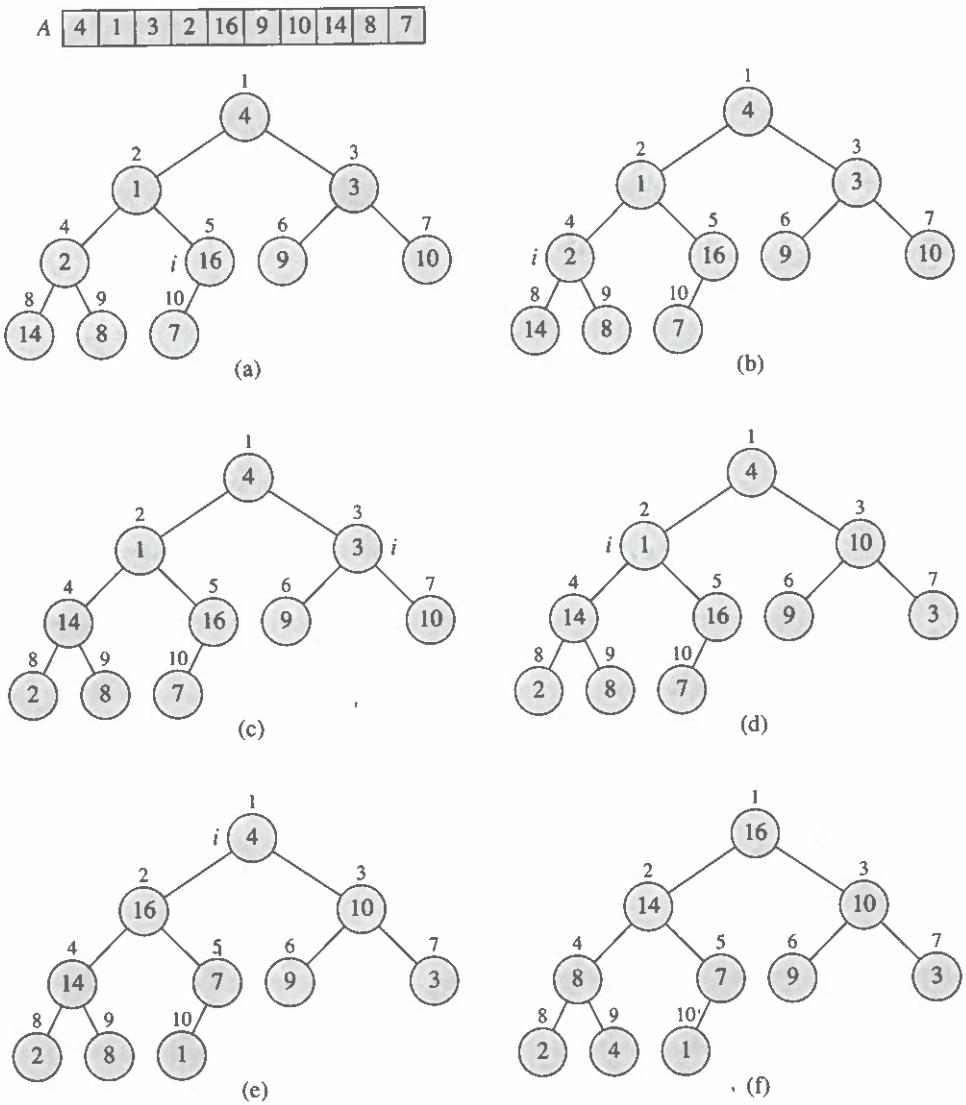

Figure 7.3 The operation of BUILD-HEAP, showing the data structure before the call to HEAPIFY in line 3 of BUILD-HEAP. **(a)** A 10-element input array A and the binary tree it represents. The figure shows that the loop index i points to node 5 before the call HEAPIFY(A, i). **(b)** The data structure that results. The loop index i for the next iteration points to node 4. **(c)–(e)** Subsequent iterations of the **for** loop in BUILD-HEAP. Observe that whenever HEAPIFY is called on a node, the two subtrees of that node are both heaps. **(f)** The heap after BUILD-HEAP finishes.

Exercises

7.3-1
Using Figure 7.3 as a model, illustrate the operation of BUILD-HEAP on the array $A = \langle 5, 3, 17, 10, 84, 19, 6, 22, 9 \rangle$.

7.3-2
Why do we want the loop index i in line 2 of BUILD-HEAP to decrease from $\lfloor length[A]/2 \rfloor$ to 1 rather than increase from 1 to $\lfloor length[A]/2 \rfloor$?

7.3-3
Show that there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h in any n-element heap.

7.4 The heapsort algorithm

The heapsort algorithm starts by using BUILD-HEAP to build a heap on the input array $A[1..n]$, where $n = length[A]$. Since the maximum element of the array is stored at the root $A[1]$, it can be put into its correct final position by exchanging it with $A[n]$. If we now "discard" node n from the heap (by decrementing *heap-size*$[A]$), we observe that $A[1..(n-1)]$ can easily be made into a heap. The children of the root remain heaps, but the new root element may violate the heap property (7.1). All that is needed to restore the heap property, however, is one call to HEAPIFY$(A, 1)$, which leaves a heap in $A[1..(n-1)]$. The heapsort algorithm then repeats this process for the heap of size $n-1$ down to a heap of size 2.

```
HEAPSORT(A)
1  BUILD-HEAP(A)
2  for i ← length[A] downto 2
3      do exchange A[1] ↔ A[i]
4         heap-size[A] ← heap-size[A] − 1
5         HEAPIFY(A, 1)
```

Figure 7.4 shows an example of the operation of heapsort after the heap is initially built. Each heap is shown at the beginning of an iteration of the **for** loop in line 2.

The HEAPSORT procedure takes time $O(n \lg n)$, since the call to BUILD-HEAP takes time $O(n)$ and each of the $n-1$ calls to HEAPIFY takes time $O(\lg n)$.

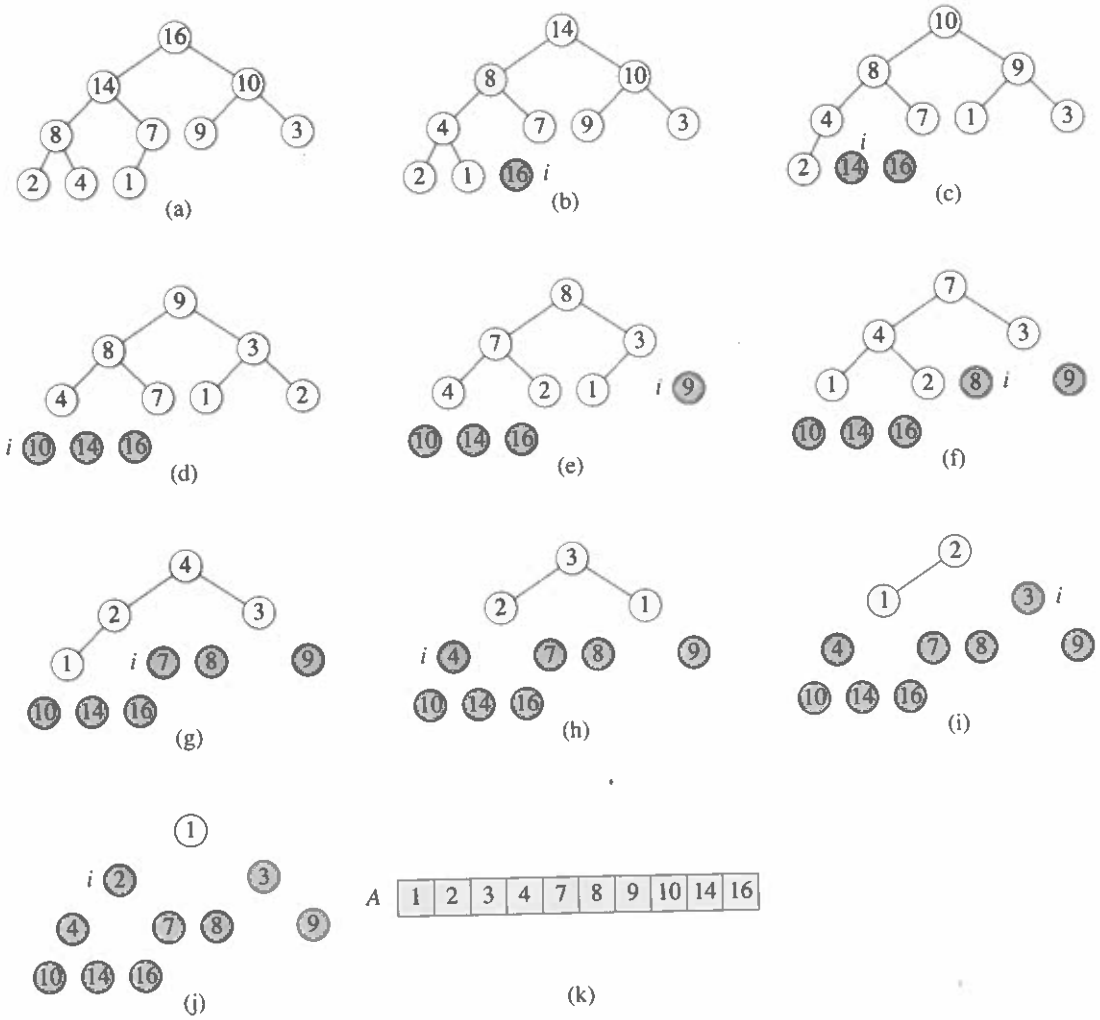

Figure 7.4 The operation of HEAPSORT. **(a)** The heap data structure just after it has been built by BUILD-HEAP. **(b)–(j)** The heap just after each call of HEAPIFY in line 5. The value of i at that time is shown. Only lightly shaded nodes remain in the heap. **(k)** The resulting sorted array A.