6 Heapsort

This chapter introduces another sorting algorithm: heapsort. 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: using a data structure, in this case one we call a "heap," to manage information. Not only is the heap data structure useful for heapsort, but it also makes an efficient priority queue. The heap data structure will reappear in algorithms in later chapters.

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 languages Java and Python provide. Please don't be confused. The heap data structure is *not* garbage-collected storage. This book is consistent in using the term "heap" to refer to the data structure, not the storage class.

6.1 Heaps

The ***(binary) heap*** data structure is an array object that we can view as a nearly complete binary tree (see Section B.5.3), as shown in Figure 6.1. Each node of the tree corresponds to an element of the array. The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point. An array $A[1:n]$ that represents a heap is an object with an attribute $A.\textit{heap-size}$, which represents how many elements in the heap are stored within array A. That is, although $A[1:n]$ may contain numbers, only the elements in $A[1:A.\textit{heap-size}]$, where $0 \le A.\textit{heap-size} \le n$, are valid elements of the heap. If $A.\textit{heap-size} = 0$, then the heap is empty. The root of the tree is $A[1]$, and given the index i of a node,

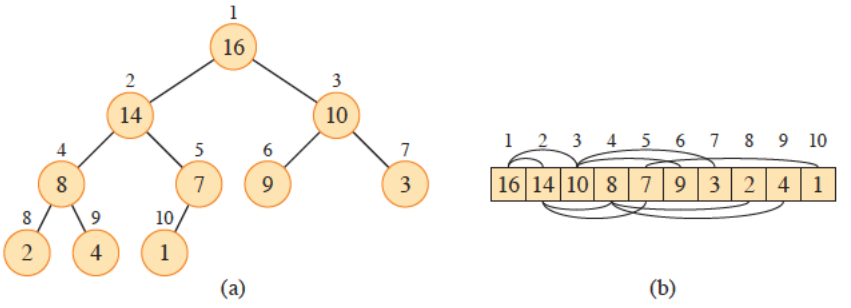

Figure 6.1 A max-heap viewed as (**a**) a binary tree and (**b**) an array. The number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships, with parents always to the left of their children. The tree has height 3, and the node at index 4 (with value 8) has height 1.

there's a simple way to compute the indices of its parent, left child, and right child with the one-line procedures PARENT, LEFT, and RIGHT.

```
PARENT(i)
1  return ⌊i/2⌋

LEFT(i)
1  return 2i

RIGHT(i)
1  return 2i + 1
```

On most computers, the LEFT procedure can compute $2i$ in one instruction by simply shifting the binary representation of i left by one bit position. Similarly, the RIGHT procedure can quickly compute $2i + 1$ by shifting the binary representation of i left by one bit position and then adding 1. The PARENT procedure can compute $\lfloor i/2 \rfloor$ by shifting i right one bit position. Good implementations of heapsort often implement these procedures as macros or inline procedures.

There are two kinds of binary heaps: max-heaps and min-heaps. In both kinds, the values in the nodes satisfy a ***heap property***, the specifics of which depend on the kind of heap. In a ***max-heap***, the ***max-heap property*** is that for every node i other than the root,

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

that is, the value of a node is at most the value of its parent. Thus, the largest element in a max-heap is stored at the root, and the subtree rooted at a node contains values no larger than that contained at the node itself. A ***min-heap*** is organized in the opposite way: the ***min-heap property*** is that for every node i other than the root,

$$A[\text{PARENT}(i)] \leq A[i] .$$

The smallest element in a min-heap is at the root.

The heapsort algorithm uses max-heaps. Min-heaps commonly implement priority queues, which we discuss in Section 6.5. We'll be precise in specifying whether we need a max-heap or a min-heap for any particular application, and when properties apply to either max-heaps or min-heaps, we just use the term "heap."

Viewing a heap as a tree, we define the ***height*** of a node in a heap 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 heap 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 6.1-2). As we'll see, 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 some basic procedures and shows how they are used in a sorting algorithm and a priority-queue data structure.

- The MAX-HEAPIFY procedure, which runs in $O(\lg n)$ time, is the key to maintaining the max-heap property.
- The BUILD-MAX-HEAP procedure, which runs in linear time, produces a max-heap from an unordered input array.
- The HEAPSORT procedure, which runs in $O(n \lg n)$ time, sorts an array in place.
- The procedures MAX-HEAP-INSERT, MAX-HEAP-EXTRACT-MAX, MAX-HEAP-INCREASE-KEY, and MAX-HEAP-MAXIMUM allow the heap data structure to implement a priority queue. They run in $O(\lg n)$ time plus the time for mapping between objects being inserted into the priority queue and indices in the heap.

Exercises

6.1-1

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

6.1-2

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

6.1-3
Show that in any subtree of a max-heap, the root of the subtree contains the largest value occurring anywhere in that subtree.

6.1-4
Where in a max-heap might the smallest element reside, assuming that all elements are distinct?

6.1-5
At which levels in a max-heap might the kth largest element reside, for $2 \le k \le \lfloor n/2 \rfloor$, assuming that all elements are distinct?

6.1-6
Is an array that is in sorted order a min-heap?

6.1-7
Is the array with values $\langle 33, 19, 20, 15, 13, 10, 2, 13, 16, 12 \rangle$ a max-heap?

6.1-8
Show that, with the array representation for storing an n-element heap, the leaves are the nodes indexed by $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$.

6.2 Maintaining the heap property

The procedure MAX-HEAPIFY on the facing page maintains the max-heap property. Its inputs are an array A with the *heap-size* attribute and an index i into the array. When it is called, MAX-HEAPIFY assumes that the binary trees rooted at LEFT(i) and RIGHT(i) are max-heaps, but that $A[i]$ might be smaller than its children, thus violating the max-heap property. MAX-HEAPIFY lets the value at $A[i]$ "float down" in the max-heap so that the subtree rooted at index i obeys the max-heap property.

Figure 6.2 illustrates the action of MAX-HEAPIFY. Each step determines the largest of the elements $A[i]$, $A[\text{LEFT}(i)]$, and $A[\text{RIGHT}(i)]$ and stores the index of the largest element in *largest*. If $A[i]$ is largest, then the subtree rooted at node i is already a max-heap and nothing else needs to be done. Otherwise, one of the two children contains the largest element. Positions i and *largest* swap their contents, which causes node i and its children to satisfy the max-heap property. The node indexed by *largest*, however, just had its value decreased, and thus the subtree rooted at *largest* might violate the max-heap property. Consequently, MAX-HEAPIFY calls itself recursively on that subtree.

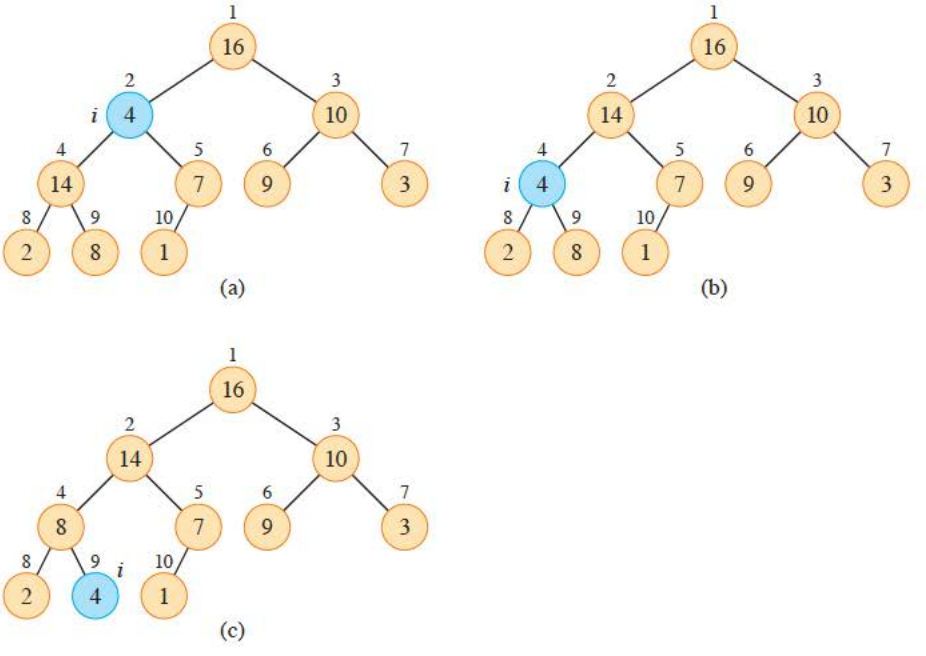

Figure 6.2 The action of MAX-HEAPIFY$(A, 2)$, where $A.heap\text{-}size = 10$. The node that potentially violates the max-heap property is shown in blue. **(a)** The initial configuration, with $A[2]$ at node $i = 2$ violating the max-heap property since it is not larger than both children. The max-heap property is restored for node 2 in **(b)** by exchanging $A[2]$ with $A[4]$, which destroys the max-heap property for node 4. The recursive call MAX-HEAPIFY$(A, 4)$ now has $i = 4$. After $A[4]$ and $A[9]$ are swapped, as shown in **(c)**, node 4 is fixed up, and the recursive call MAX-HEAPIFY$(A, 9)$ yields no further change to the data structure.

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

To analyze MAX-HEAPIFY, let $T(n)$ be the worst-case running time that the procedure takes on a subtree of size at most n. For a tree rooted at a given node i, the running time 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 MAX-HEAPIFY on a subtree rooted at one of the children of node i (assuming that the recursive call occurs). The children's subtrees each have size at most $2n/3$ (see Exercise 6.2-2), and therefore we can describe the running time of MAX-HEAPIFY by the recurrence

$$T(n) \le T(2n/3) + \Theta(1) . \tag{6.1}$$

The solution to this recurrence, by case 2 of the master theorem (Theorem 4.1 on page 102), is $T(n) = O(\lg n)$. Alternatively, we can characterize the running time of MAX-HEAPIFY on a node of height h as $O(h)$.

Exercises

6.2-1
Using Figure 6.2 as a model, illustrate the operation of MAX-HEAPIFY$(A, 3)$ on the array $A = \langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0 \rangle$.

6.2-2
Show that each child of the root of an n-node heap is the root of a subtree containing at most $2n/3$ nodes. What is the smallest constant α such that each subtree has at most αn nodes? How does that affect the recurrence (6.1) and its solution?

6.2-3
Starting with the procedure MAX-HEAPIFY, write pseudocode for the procedure MIN-HEAPIFY(A, i), which performs the corresponding manipulation on a min-heap. How does the running time of MIN-HEAPIFY compare with that of MAX-HEAPIFY?

6.2-4
What is the effect of calling MAX-HEAPIFY(A, i) when the element $A[i]$ is larger than its children?

6.2-5
What is the effect of calling MAX-HEAPIFY(A, i) for $i > A.\textit{heap-size}/2$?

6.2-6
The code for MAX-HEAPIFY is quite efficient in terms of constant factors, except possibly for the recursive call in line 10, for which some compilers might produce inefficient code. Write an efficient MAX-HEAPIFY that uses an iterative control construct (a loop) instead of recursion.

6.2-7
Show that the worst-case running time of MAX-HEAPIFY on a heap of size n is $\Omega(\lg n)$. (*Hint:* For a heap with n nodes, give node values that cause MAX-HEAPIFY to be called recursively at every node on a simple path from the root down to a leaf.)

6.3 Building a heap

The procedure BUILD-MAX-HEAP converts an array $A[1:n]$ into a max-heap by calling MAX-HEAPIFY in a bottom-up manner. Exercise 6.1-8 says that the elements in the subarray $A[\lfloor n/2 \rfloor + 1 : n]$ are all leaves of the tree, and so each is a 1-element heap to begin with. BUILD-MAX-HEAP goes through the remaining nodes of the tree and runs MAX-HEAPIFY on each one. Figure 6.3 shows an example of the action of BUILD-MAX-HEAP.

```
BUILD-MAX-HEAP(A, n)
1  A.heap-size = n
2  for i = ⌊n/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

To show why BUILD-MAX-HEAP works correctly, we use the following loop invariant:

> At the start of each iteration of the **for** loop of lines 2–3, each node $i + 1$, $i + 2, \ldots, n$ is the root of a max-heap.

We need to show that this invariant is true prior to the first loop iteration, that each iteration of the loop maintains the invariant, that the loop terminates, and that the invariant provides a useful property to show correctness when the loop terminates.

Initialization: Prior to the first iteration of the loop, $i = \lfloor n/2 \rfloor$. Each node $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$ is a leaf and is thus the root of a trivial max-heap.

Maintenance: To see that each iteration maintains the loop invariant, observe that the children of node i are numbered higher than i. By the loop invariant, therefore, they are both roots of max-heaps. This is precisely the condition required for the call MAX-HEAPIFY(A, i) to make node i a max-heap root. Moreover, the MAX-HEAPIFY call preserves the property that nodes $i + 1$, $i + 2, \ldots, n$ are all roots of max-heaps. Decrementing i in the **for** loop update reestablishes the loop invariant for the next iteration.

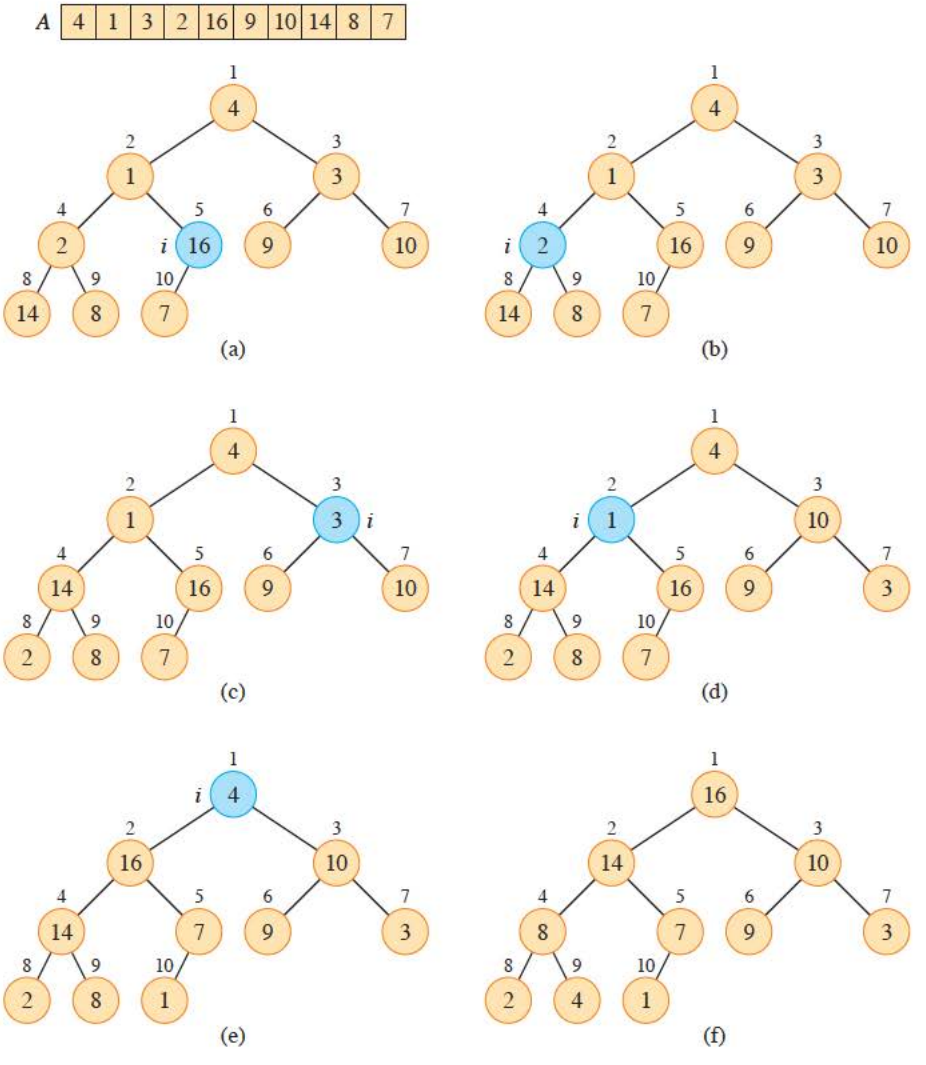

Figure 6.3 The operation of BUILD-MAX-HEAP, showing the data structure before the call to MAX-HEAPIFY in line 3 of BUILD-MAX-HEAP. The node indexed by i in each iteration is shown in blue. **(a)** A 10-element input array A and the binary tree it represents. The loop index i refers to node 5 before the call MAX-HEAPIFY(A, i). **(b)** The data structure that results. The loop index i for the next iteration refers to node 4. **(c)–(e)** Subsequent iterations of the **for** loop in BUILD-MAX-HEAP. Observe that whenever MAX-HEAPIFY is called on a node, the two subtrees of that node are both max-heaps. **(f)** The max-heap after BUILD-MAX-HEAP finishes.

Termination: The loop makes exactly $\lfloor n/2 \rfloor$ iterations, and so it terminates. At termination, $i = 0$. By the loop invariant, each node $1, 2, \ldots, n$ is the root of a max-heap. In particular, node 1 is.

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

We can derive a tighter asymptotic bound by observing that the time for MAX-HEAPIFY to run at a node varies with the height of the node in the tree, and that the heights of most nodes are small. Our tighter analysis relies on the properties that an n-element heap has height $\lfloor \lg n \rfloor$ (see Exercise 6.1-2) and at most $\lceil n/2^{h+1} \rceil$ nodes of any height h (see Exercise 6.3-4).

The time required by MAX-HEAPIFY when called on a node of height h is $O(h)$. Letting c be the constant implicit in the asymptotic notation, we can express the total cost of BUILD-MAX-HEAP as being bounded from above by $\sum_{h=0}^{\lfloor \lg n \rfloor} \lceil n/2^{h+1} \rceil ch$. As Exercise 6.3-2 shows, we have $\lceil n/2^{h+1} \rceil \geq 1/2$ for $0 \leq h \leq \lfloor \lg n \rfloor$. Since $\lceil x \rceil \leq 2x$ for any $x \geq 1/2$, we have $\lceil n/2^{h+1} \rceil \leq n/2^h$. We thus obtain

$$
\begin{aligned}
&\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil ch \\
&\qquad \leq \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{n}{2^h} ch \\
&\qquad = cn \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \\
&\qquad \leq cn \sum_{h=0}^{\infty} \frac{h}{2^h} \\
&\qquad \leq cn \cdot \frac{1/2}{(1-1/2)^2} \quad \text{(by equation (A.11) on page 1142 with } x = 1/2\text{)} \\
&\qquad = O(n) \,.
\end{aligned}
$$

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

To build a min-heap, use the procedure BUILD-MIN-HEAP, which is the same as BUILD-MAX-HEAP but with the call to MAX-HEAPIFY in line 3 replaced by a call to MIN-HEAPIFY (see Exercise 6.2-3). BUILD-MIN-HEAP produces a min-heap from an unordered linear array in linear time.

Exercises

6.3-1
Using Figure 6.3 as a model, illustrate the operation of BUILD-MAX-HEAP on the array $A = \langle 5, 3, 17, 10, 84, 19, 6, 22, 9 \rangle$.

6.3-2
Show that $\lceil n/2^{h+1} \rceil \geq 1/2$ for $0 \leq h \leq \lfloor \lg n \rfloor$.

6.3-3
Why does the loop index i in line 2 of BUILD-MAX-HEAP decrease from $\lfloor n/2 \rfloor$ to 1 rather than increase from 1 to $\lfloor n/2 \rfloor$?

6.3-4
Show that there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h in any n-element heap.

6.4 The heapsort algorithm

The heapsort algorithm, given by the procedure HEAPSORT, starts by calling the BUILD-MAX-HEAP procedure to build a max-heap on the input array $A[1:n]$. Since the maximum element of the array is stored at the root $A[1]$, HEAPSORT can place it into its correct final position by exchanging it with $A[n]$. If the procedure then discards node n from the heap—and it can do so by simply decrementing $A.\mathit{heap\text{-}size}$—the children of the root remain max-heaps, but the new root element might violate the max-heap property. To restore the max-heap property, the procedure just calls MAX-HEAPIFY$(A, 1)$, which leaves a max-heap in $A[1:n-1]$. The HEAPSORT procedure then repeats this process for the max-heap of size $n-1$ down to a heap of size 2. (See Exercise 6.4-2 for a precise loop invariant.)

```
HEAPSORT(A, n)
1  BUILD-MAX-HEAP(A, n)
2  for i = n downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

Figure 6.4 shows an example of the operation of HEAPSORT after line 1 has built the initial max-heap. The figure shows the max-heap before the first iteration of the **for** loop of lines 2–5 and after each iteration.

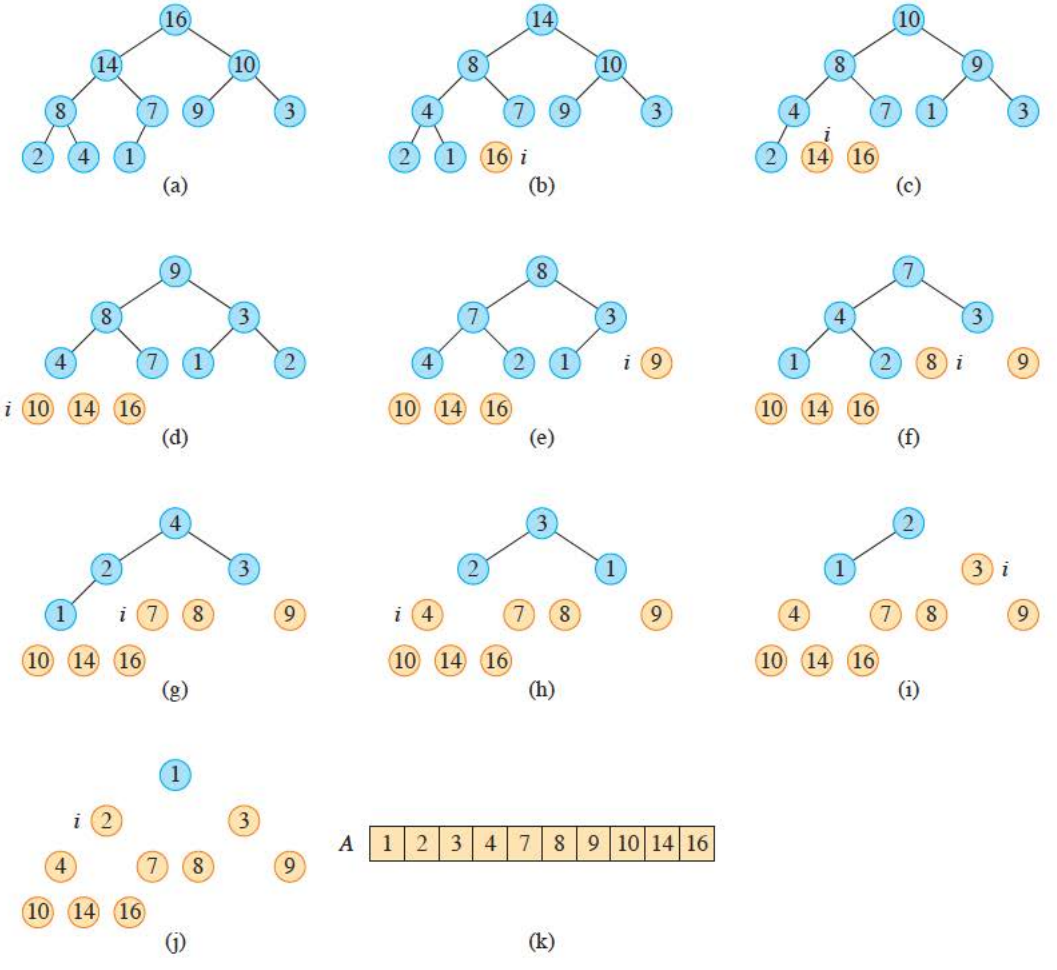

Figure 6.4 The operation of HEAPSORT. **(a)** The max-heap data structure just after BUILD-MAX-HEAP has built it in line 1. **(b)–(j)** The max-heap just after each call of MAX-HEAPIFY in line 5, showing the value of i at that time. Only blue nodes remain in the heap. Tan nodes contain the largest values in the array, in sorted order. **(k)** The resulting sorted array A.

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

Exercises

6.4-1
Using Figure 6.4 as a model, illustrate the operation of HEAPSORT on the array $A = \langle 5, 13, 2, 25, 7, 17, 20, 8, 4 \rangle$.

6.4-2
Argue the correctness of HEAPSORT using the following loop invariant:

> At the start of each iteration of the **for** loop of lines 2–5, the subarray $A[1:i]$ is a max-heap containing the i smallest elements of $A[1:n]$, and the subarray $A[i+1:n]$ contains the $n-i$ largest elements of $A[1:n]$, sorted.

6.4-3
What is the running time of HEAPSORT on an array A of length n that is already sorted in increasing order? How about if the array is already sorted in decreasing order?

6.4-4
Show that the worst-case running time of HEAPSORT is $\Omega(n \lg n)$.

★ ***6.4-5***
Show that when all the elements of A are distinct, the best-case running time of HEAPSORT is $\Omega(n \lg n)$.

6.5 Priority queues

In Chapter 8, we will see that any comparison-based sorting algorithm requires $\Omega(n \lg n)$ comparisons and hence $\Omega(n \lg n)$ time. Therefore, heapsort is asymptotically optimal among comparison-based sorting algorithms. Yet, a good implementation of quicksort, presented in Chapter 7, usually beats it in practice. Nevertheless, the heap data structure itself has many uses. In this section, we present one of the most popular applications of a heap: as an efficient priority queue. As with heaps, priority queues come in two forms: max-priority queues and min-priority queues. We'll focus here on how to implement max-priority queues, which are in turn based on max-heaps. Exercise 6.5-3 asks you to write the procedures for min-priority queues.

A ***priority queue*** is a data structure for maintaining a set S of elements, each with an associated value called a ***key***. A ***max-priority queue*** supports the following operations:

INSERT(S, x, k) inserts the element x with key k into the set S, which is equivalent to the operation $S = S \cup \{x\}$.

MAXIMUM(S) returns the element of S with the largest key.

EXTRACT-MAX(S) removes and returns the element of S with the largest key.

INCREASE-KEY(S, x, k) increases the value of element x's key to the new value k, which is assumed to be at least as large as x's current key value.

Among their other applications, you can use max-priority queues to schedule jobs on a computer shared among multiple users. The max-priority queue keeps track of the jobs to be performed and their relative priorities. When a job is finished or interrupted, the scheduler selects the highest-priority job from among those pending by calling EXTRACT-MAX. The scheduler can add a new job to the queue at any time by calling INSERT.

Alternatively, a ***min-priority queue*** supports the operations INSERT, MINIMUM, EXTRACT-MIN, and DECREASE-KEY. A min-priority queue can be used in an event-driven simulator. The items in the queue are events to be simulated, each with an associated time of occurrence that serves as its key. The events must be simulated in order of their time of occurrence, because the simulation of an event can cause other events to be simulated in the future. The simulation program calls EXTRACT-MIN at each step to choose the next event to simulate. As new events are produced, the simulator inserts them into the min-priority queue by calling INSERT. We'll see other uses for min-priority queues, highlighting the DECREASE-KEY operation, in Chapters 21 and 22.

When you use a heap to implement a priority queue within a given application, elements of the priority queue correspond to objects in the application. Each object contains a key. If the priority queue is implemented by a heap, you need to determine which application object corresponds to a given heap element, and vice versa. Because the heap elements are stored in an array, you need a way to map application objects to and from array indices.

One way to map between application objects and heap elements uses ***handles***, which are additional information stored in the objects and heap elements that give enough information to perform the mapping. Handles are often implemented to be opaque to the surrounding code, thereby maintaining an abstraction barrier between the application and the priority queue. For example, the handle within an application object might contain the corresponding index into the heap array. But since only the code for the priority queue accesses this index, the index is entirely hidden from the application code. Because heap elements change locations within

the array during heap operations, an actual implementation of the priority queue, upon relocating a heap element, must also update the array indices in the corresponding handles. Conversely, each element in the heap might contain a pointer to the corresponding application object, but the heap element knows this pointer as only an opaque handle and the application maps this handle to an application object. Typically, the worst-case overhead for maintaining handles is $O(1)$ per access.

As an alternative to incorporating handles in application objects, you can store within the priority queue a mapping from application objects to array indices in the heap. The advantage of doing so is that the mapping is contained entirely within the priority queue, so that the application objects need no further embellishment. The disadvantage lies in the additional cost of establishing and maintaining the mapping. One option for the mapping is a hash table (see Chapter 11).[1] The added expected time for a hash table to map an object to an array index is just $O(1)$, though the worst-case time can be as bad as $\Theta(n)$.

Let's see how to implement the operations of a max-priority queue using a max-heap. In the previous sections, we treated the array elements as the keys to be sorted, implicitly assuming that any satellite data moved with the corresponding keys. When a heap implements a priority queue, we instead treat each array element as a pointer to an object in the priority queue, so that the object is analogous to the satellite data when sorting. We further assume that each such object has an attribute *key*, which determines where in the heap the object belongs. For a heap implemented by an array A, we refer to $A[i].key$.

The procedure MAX-HEAP-MAXIMUM on the facing page implements the MAXIMUM operation in $\Theta(1)$ time, and MAX-HEAP-EXTRACT-MAX implements the operation EXTRACT-MAX. MAX-HEAP-EXTRACT-MAX is similar to the **for** loop body (lines 3–5) of the HEAPSORT procedure. We implicitly assume that MAX-HEAPIFY compares priority-queue objects based on their *key* attributes. We also assume that when MAX-HEAPIFY exchanges elements in the array, it is exchanging pointers and also that it updates the mapping between objects and array indices. The running time of MAX-HEAP-EXTRACT-MAX is $O(\lg n)$, since it performs only a constant amount of work on top of the $O(\lg n)$ time for MAX-HEAPIFY, plus whatever overhead is incurred within MAX-HEAPIFY for mapping priority-queue objects to array indices.

The procedure MAX-HEAP-INCREASE-KEY on page 176 implements the INCREASE-KEY operation. It first verifies that the new key k will not cause the key in the object x to decrease, and if there is no problem, it gives x the new key value. The procedure then finds the index i in the array corresponding to object x,

[1] In Python, dictionaries are implemented with hash tables.

```
MAX-HEAP-MAXIMUM(A)
1  if A.heap-size < 1
2       error "heap underflow"
3  return A[1]

MAX-HEAP-EXTRACT-MAX(A)
1  max = MAX-HEAP-MAXIMUM(A)
2  A[1] = A[A.heap-size]
3  A.heap-size = A.heap-size − 1
4  MAX-HEAPIFY(A, 1)
5  return max
```

so that $A[i]$ is x. Because increasing the key of $A[i]$ might violate the max-heap property, the procedure then, in a manner reminiscent of the insertion loop (lines 5–7) of INSERTION-SORT on page 19, traverses a simple path from this node toward the root to find a proper place for the newly increased key. As MAX-HEAP-INCREASE-KEY traverses this path, it repeatedly compares an element's key to that of its parent, exchanging pointers and continuing if the element's key is larger, and terminating if the element's key is smaller, since the max-heap property now holds. (See Exercise 6.5-7 for a precise loop invariant.) Like MAX-HEAPIFY when used in a priority queue, MAX-HEAP-INCREASE-KEY updates the information that maps objects to array indices when array elements are exchanged. Figure 6.5 shows an example of a MAX-HEAP-INCREASE-KEY operation. In addition to the overhead for mapping priority queue objects to array indices, the running time of MAX-HEAP-INCREASE-KEY on an n-element heap is $O(\lg n)$, since the path traced from the node updated in line 3 to the root has length $O(\lg n)$.

The procedure MAX-HEAP-INSERT on the next page implements the INSERT operation. It takes as inputs the array A implementing the max-heap, the new object x to be inserted into the max-heap, and the size n of array A. The procedure first verifies that the array has room for the new element. It then expands the max-heap by adding to the tree a new leaf whose key is $-\infty$. Then it calls MAX-HEAP-INCREASE-KEY to set the key of this new element to its correct value and maintain the max-heap property. The running time of MAX-HEAP-INSERT on an n-element heap is $O(\lg n)$ plus the overhead for mapping priority queue objects to indices.

In summary, a heap can support any priority-queue operation on a set of size n in $O(\lg n)$ time, plus the overhead for mapping priority queue objects to array indices.

```
MAX-HEAP-INCREASE-KEY(A, x, k)
1  if k < x.key
2       error "new key is smaller than current key"
3  x.key = k
4  find the index i in array A where object x occurs
5  while i > 1 and A[PARENT(i)].key < A[i].key
6       exchange A[i] with A[PARENT(i)], updating the information that maps
            priority queue objects to array indices
7       i = PARENT(i)

MAX-HEAP-INSERT(A, x, n)
1  if A.heap-size == n
2       error "heap overflow"
3  A.heap-size = A.heap-size + 1
4  k = x.key
5  x.key = −∞
6  A[A.heap-size] = x
7  map x to index heap-size in the array
8  MAX-HEAP-INCREASE-KEY(A, x, k)
```

Exercises

6.5-1

Suppose that the objects in a max-priority queue are just keys. Illustrate the operation of MAX-HEAP-EXTRACT-MAX on the heap $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$.

6.5-2

Suppose that the objects in a max-priority queue are just keys. Illustrate the operation of MAX-HEAP-INSERT$(A, 10)$ on the heap $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$.

6.5-3

Write pseudocode to implement a min-priority queue with a min-heap by writing the procedures MIN-HEAP-MINIMUM, MIN-HEAP-EXTRACT-MIN, MIN-HEAP-DECREASE-KEY, and MIN-HEAP-INSERT.

6.5-4

Write pseudocode for the procedure MAX-HEAP-DECREASE-KEY(A, x, k) in a max-heap. What is the running time of your procedure?

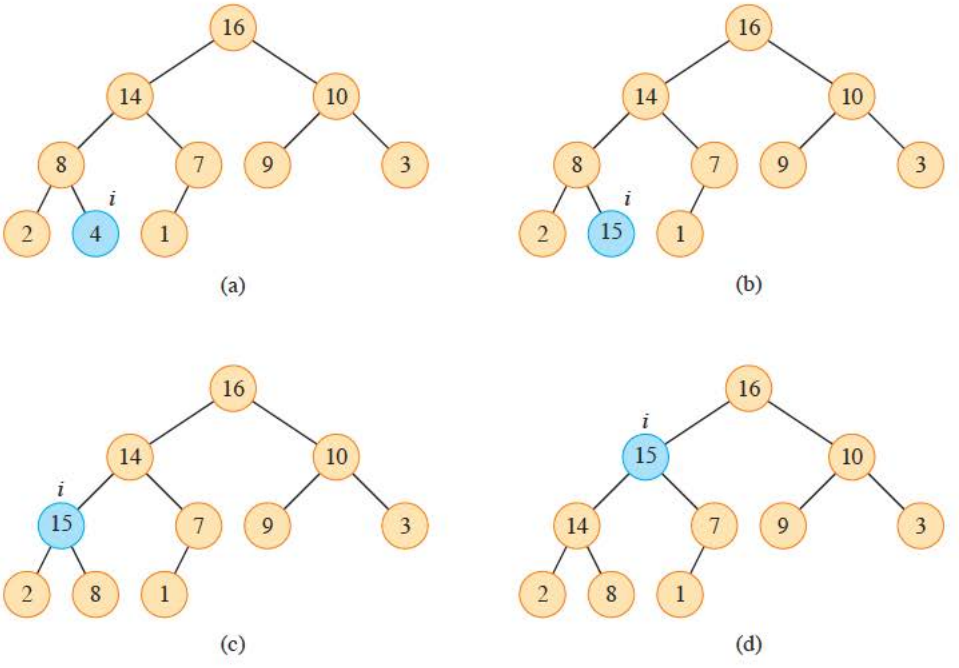

Figure 6.5 The operation of MAX-HEAP-INCREASE-KEY. Only the key of each element in the priority queue is shown. The node indexed by i in each iteration is shown in blue. **(a)** The max-heap of Figure 6.4(a) with i indexing the node whose key is about to be increased. **(b)** This node has its key increased to 15. **(c)** After one iteration of the **while** loop of lines 5–7, the node and its parent have exchanged keys, and the index i moves up to the parent. **(d)** The max-heap after one more iteration of the **while** loop. At this point, $A[\text{PARENT}(i)] \geq A[i]$. The max-heap property now holds and the procedure terminates.

6.5-5

Why does MAX-HEAP-INSERT bother setting the key of the inserted object to $-\infty$ in line 5 given that line 8 will set the object's key to the desired value?

6.5-6

Professor Uriah suggests replacing the **while** loop of lines 5–7 in MAX-HEAP-INCREASE-KEY by a call to MAX-HEAPIFY. Explain the flaw in the professor's idea.

6.5-7

Argue the correctness of MAX-HEAP-INCREASE-KEY using the following loop invariant:

At the start of each iteration of the **while** loop of lines 5–7:

a. If both nodes PARENT(i) and LEFT(i) exist, then $A[\text{PARENT}(i)].key \geq A[\text{LEFT}(i)].key$.
b. If both nodes PARENT(i) and RIGHT(i) exist, then $A[\text{PARENT}(i)].key \geq A[\text{RIGHT}(i)].key$.
c. The subarray $A[1 : A.heap\text{-}size]$ satisfies the max-heap property, except that there may be one violation, which is that $A[i].key$ may be greater than $A[\text{PARENT}(i)].key$.

You may assume that the subarray $A[1 : A.heap\text{-}size]$ satisfies the max-heap property at the time MAX-HEAP-INCREASE-KEY is called.

6.5-8
Each exchange operation on line 6 of MAX-HEAP-INCREASE-KEY typically requires three assignments, not counting the updating of the mapping from objects to array indices. Show how to use the idea of the inner loop of INSERTION-SORT to reduce the three assignments to just one assignment.

6.5-9
Show how to implement a first-in, first-out queue with a priority queue. Show how to implement a stack with a priority queue. (Queues and stacks are defined in Section 10.1.3.)

6.5-10
The operation MAX-HEAP-DELETE(A, x) deletes the object x from max-heap A. Give an implementation of MAX-HEAP-DELETE for an n-element max-heap that runs in $O(\lg n)$ time plus the overhead for mapping priority queue objects to array indices.

6.5-11
Give an $O(n \lg k)$-time algorithm to merge k sorted lists into one sorted list, where n is the total number of elements in all the input lists. (*Hint:* Use a min-heap for k-way merging.)

Problems

6-1 Building a heap using insertion
One way to build a heap is by repeatedly calling MAX-HEAP-INSERT to insert the elements into the heap. Consider the procedure BUILD-MAX-HEAP′ on the facing page. It assumes that the objects being inserted are just the heap elements.

```
BUILD-MAX-HEAP′(A, n)
1  A.heap-size = 1
2  for i = 2 to n
3      MAX-HEAP-INSERT(A, A[i], n)
```

a. Do the procedures BUILD-MAX-HEAP and BUILD-MAX-HEAP′ always create the same heap when run on the same input array? Prove that they do, or provide a counterexample.

b. Show that in the worst case, BUILD-MAX-HEAP′ requires $\Theta(n \lg n)$ time to build an n-element heap.

6-2 Analysis of d-ary heaps

A ***d-ary heap*** is like a binary heap, but (with one possible exception) nonleaf nodes have d children instead of two children. In all parts of this problem, assume that the time to maintain the mapping between objects and heap elements is $O(1)$ per operation.

a. Describe how to represent a d-ary heap in an array.

b. Using Θ-notation, express the height of a d-ary heap of n elements in terms of n and d.

c. Give an efficient implementation of EXTRACT-MAX in a d-ary max-heap. Analyze its running time in terms of d and n.

d. Give an efficient implementation of INCREASE-KEY in a d-ary max-heap. Analyze its running time in terms of d and n.

e. Give an efficient implementation of INSERT in a d-ary max-heap. Analyze its running time in terms of d and n.

6-3 Young tableaus

An $m \times n$ ***Young tableau*** is an $m \times n$ matrix such that the entries of each row are in sorted order from left to right and the entries of each column are in sorted order from top to bottom. Some of the entries of a Young tableau may be ∞, which we treat as nonexistent elements. Thus, a Young tableau can be used to hold $r \leq mn$ finite numbers.

a. Draw a 4×4 Young tableau containing the elements $\{9, 16, 3, 2, 4, 8, 5, 14, 12\}$.

b. Argue that an $m \times n$ Young tableau Y is empty if $Y[1,1] = \infty$. Argue that Y is full (contains mn elements) if $Y[m,n] < \infty$.

c. Give an algorithm to implement EXTRACT-MIN on a nonempty $m \times n$ Young tableau that runs in $O(m+n)$ time. Your algorithm should use a recursive subroutine that solves an $m \times n$ problem by recursively solving either an $(m-1) \times n$ or an $m \times (n-1)$ subproblem. (*Hint:* Think about MAX-HEAPIFY.) Explain why your implementation of EXTRACT-MIN runs in $O(m+n)$ time.

d. Show how to insert a new element into a nonfull $m \times n$ Young tableau in $O(m+n)$ time.

e. Using no other sorting method as a subroutine, show how to use an $n \times n$ Young tableau to sort n^2 numbers in $O(n^3)$ time.

f. Give an $O(m+n)$-time algorithm to determine whether a given number is stored in a given $m \times n$ Young tableau.

Chapter notes

The heapsort algorithm was invented by Williams [456], who also described how to implement a priority queue with a heap. The BUILD-MAX-HEAP procedure was suggested by Floyd [145]. Schaffer and Sedgewick [395] showed that in the best case, the number of times elements move in the heap during heapsort is approximately $(n/2) \lg n$ and that the average number of moves is approximately $n \lg n$.

We use min-heaps to implement min-priority queues in Chapters 15, 21, and 22. Other, more complicated, data structures give better time bounds for certain min-priority queue operations. Fredman and Tarjan [156] developed Fibonacci heaps, which support INSERT and DECREASE-KEY in $O(1)$ amortized time (see Chapter 16). That is, the average worst-case running time for these operations is $O(1)$. Brodal, Lagogiannis, and Tarjan [73] subsequently devised strict Fibonacci heaps, which make these time bounds the actual running times. If the keys are unique and drawn from the set $\{0, 1, \ldots, n-1\}$ of nonnegative integers, van Emde Boas trees [440, 441] support the operations INSERT, DELETE, SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, and SUCCESSOR in $O(\lg \lg n)$ time.

If the data are b-bit integers, and the computer memory consists of addressable b-bit words, Fredman and Willard [157] showed how to implement MINIMUM in $O(1)$ time and INSERT and EXTRACT-MIN in $O(\sqrt{\lg n})$ time. Thorup [436] has

improved the $O(\sqrt{\lg n})$ bound to $O(\lg \lg n)$ time by using randomized hashing, requiring only linear space.

An important special case of priority queues occurs when the sequence of EXTRACT-MIN operations is ***monotone***, that is, the values returned by successive EXTRACT-MIN operations are monotonically increasing over time. This case arises in several important applications, such as Dijkstra's single-source shortest-paths algorithm, which we discuss in Chapter 22, and in discrete-event simulation. For Dijkstra's algorithm it is particularly important that the DECREASE-KEY operation be implemented efficiently. For the monotone case, if the data are integers in the range $1, 2, \ldots, C$, Ahuja, Mehlhorn, Orlin, and Tarjan [8] describe how to implement EXTRACT-MIN and INSERT in $O(\lg C)$ amortized time (Chapter 16 presents amortized analysis) and DECREASE-KEY in $O(1)$ time, using a data structure called a radix heap. The $O(\lg C)$ bound can be improved to $O(\sqrt{\lg C})$ using Fibonacci heaps in conjunction with radix heaps. Cherkassky, Goldberg, and Silverstein [90] further improved the bound to $O(\lg^{1/3+\epsilon} C)$ expected time by combining the multilevel bucketing structure of Denardo and Fox [112] with the heap of Thorup mentioned earlier. Raman [375] further improved these results to obtain a bound of $O\left(\min\left\{\lg^{1/4+\epsilon} C, \lg^{1/3+\epsilon} n\right\}\right)$, for any fixed $\epsilon > 0$.

Many other variants of heaps have been proposed. Brodal [72] surveys some of these developments.