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 $A.\mathit{heap\text{-}size}$, 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 $A[1:A.\mathit{heap\text{-}size}]$, where $0 \le A.\mathit{heap\text{-}size} \le n$, are valid elements of the heap. If $A.\mathit{heap\text{-}size} = 0$, then the heap is empty. The root of the tree is $A[1]$, and given the index i of a node,

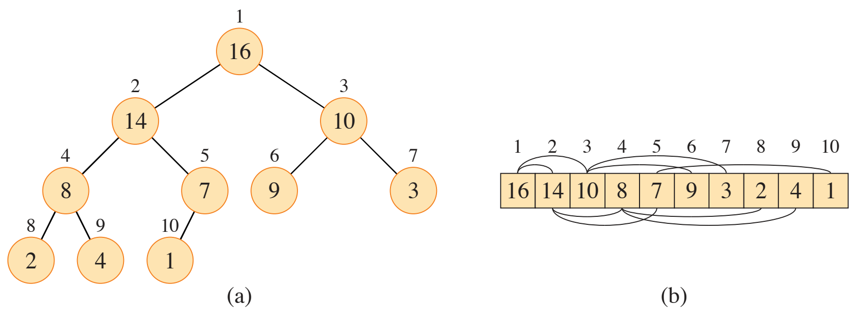

Figure 6.1 A max-heap viewed as **(a)** a binary tree and **(b)** an array. The number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships, 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 $\lfloor i/2 \rfloor$

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,

$$A[\textsc{Parent}(i)] \le A[i] .$$

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 $\lfloor \lg n \rfloor$.