b. Draw a binary tree of height 4 that can be an AVL tree and has the smallest number of nodes among all such trees.

3. Draw diagrams of the single L-rotation and of the double RL-rotation in their general form.

4. For each of the following lists, construct an AVL tree by inserting their elements successively, starting with the empty tree.

a. 1, 2, 3, 4, 5, 6

b. 6, 5, 4, 3, 2, 1

c. 3, 6, 5, 1, 2, 4

5. a. For an AVL tree containing real numbers, design an algorithm for computing the range (i.e., the difference between the largest and smallest numbers in the tree) and determine its worst-case efficiency.

b. True or false: The smallest and the largest keys in an AVL tree can always be found on either the last level or the next-to-last level?

6. Write a program for constructing an AVL tree for a given list of n distinct integers.

7. a. Construct a 2-3 tree for the list C, O, M, P, U, T, I, N, G. Use the alphabetical order of the letters and insert them successively starting with the empty tree.

b. Assuming that the probabilities of searching for each of the keys (i.e., the letters) are the same, find the largest number and the average number of key comparisons for successful searches in this tree.

8. Let T_B and $T_{2\text{-}3}$ be, respectively, a classical binary search tree and a 2-3 tree constructed for the same list of keys inserted in the corresponding trees in the same order. True or false: Searching for the same key in $T_{2\text{-}3}$ always takes fewer or the same number of key comparisons as searching in T_B?

9. For a 2-3 tree containing real numbers, design an algorithm for computing the range (i.e., the difference between the largest and smallest numbers in the tree) and determine its worst-case efficiency.

10. Write a program for constructing a 2-3 tree for a given list of n integers.

6.4 Heaps and Heapsort

The data structure called the "heap" is definitely not a disordered pile of items as the word's definition in a standard dictionary might suggest. Rather, it is a clever, partially ordered data structure that is especially suitable for implementing priority queues. Recall that a ***priority queue*** is a multiset of items with an orderable characteristic called an item's ***priority***, with the following operations:

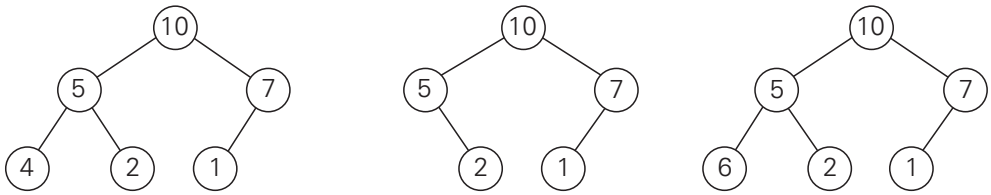

FIGURE 6.9 Illustration of the definition of heap: only the leftmost tree is a heap.

- finding an item with the highest (i.e., largest) priority
- deleting an item with the highest priority
- adding a new item to the multiset

It is primarily an efficient implementation of these operations that makes the heap both interesting and useful. Priority queues arise naturally in such applications as scheduling job executions by computer operating systems and traffic management by communication networks. They also arise in several important algorithms, e.g., Prim's algorithm (Section 9.1), Dijkstra's algorithm (Section 9.3), Huffman encoding (Section 9.4), and branch-and-bound applications (Section 12.2). The heap is also the data structure that serves as a cornerstone of a theoretically important sorting algorithm called heapsort. We discuss this algorithm after we define the heap and investigate its basic properties.

Notion of the Heap

DEFINITION A ***heap*** can be defined as a binary tree with keys assigned to its nodes, one key per node, provided the following two conditions are met:

1. The ***shape property***—the binary tree is ***essentially complete*** (or simply ***complete***), i.e., all its levels are full except possibly the last level, where only some rightmost leaves may be missing.
2. The ***parental dominance*** or ***heap property***—the key in each node is greater than or equal to the keys in its children. (This condition is considered automatically satisfied for all leaves.)[5]

For example, consider the trees of Figure 6.9. The first tree is a heap. The second one is not a heap, because the tree's shape property is violated. And the third one is not a heap, because the parental dominance fails for the node with key 5.

Note that key values in a heap are ordered top down; i.e., a sequence of values on any path from the root to a leaf is decreasing (nonincreasing, if equal keys are allowed). However, there is no left-to-right order in key values; i.e., there is no

5. Some authors require the key at each node to be *less* than or equal to the keys at its children. We call this variation a ***min-heap***.

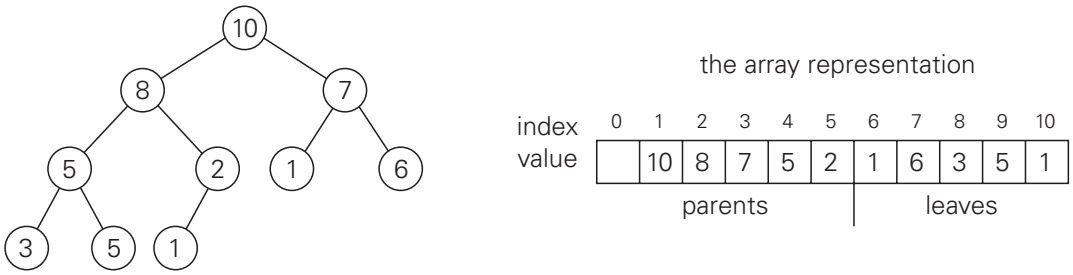

FIGURE 6.10 Heap and its array representation.

relationship among key values for nodes either on the same level of the tree or, more generally, in the left and right subtrees of the same node.

Here is a list of important properties of heaps, which are not difficult to prove (check these properties for the heap of Figure 6.10, as an example).

1. There exists exactly one essentially complete binary tree with n nodes. Its height is equal to $\lfloor \log_2 n \rfloor$.
2. The root of a heap always contains its largest element.
3. A node of a heap considered with all its descendants is also a heap.
4. A heap can be implemented as an array by recording its elements in the top-down, left-to-right fashion. It is convenient to store the heap's elements in positions 1 through n of such an array, leaving $H[0]$ either unused or putting there a sentinel whose value is greater than every element in the heap. In such a representation,
   a. the parental node keys will be in the first $\lfloor n/2 \rfloor$ positions of the array, while the leaf keys will occupy the last $\lceil n/2 \rceil$ positions;
   b. the children of a key in the array's parental position i $(1 \le i \le \lfloor n/2 \rfloor)$ will be in positions $2i$ and $2i + 1$, and, correspondingly, the parent of a key in position i $(2 \le i \le n)$ will be in position $\lfloor i/2 \rfloor$.

Thus, we could also define a heap as an array $H[1..n]$ in which every element in position i in the first half of the array is greater than or equal to the elements in positions $2i$ and $2i + 1$, i.e.,

$$H[i] \ge \max\{H[2i], H[2i + 1]\} \quad \text{for } i = 1, \ldots, \lfloor n/2 \rfloor.$$

(Of course, if $2i + 1 > n$, just $H[i] \ge H[2i]$ needs to be satisfied.) While the ideas behind the majority of algorithms dealing with heaps are easier to understand if we think of heaps as binary trees, their actual implementations are usually much simpler and more efficient with arrays.

How can we construct a heap for a given list of keys? There are two principal alternatives for doing this. The first is the ***bottom-up heap construction*** algorithm illustrated in Figure 6.11. It initializes the essentially complete binary tree with n nodes by placing keys in the order given and then "heapifies" the tree as follows. Starting with the last parental node, the algorithm checks whether the parental

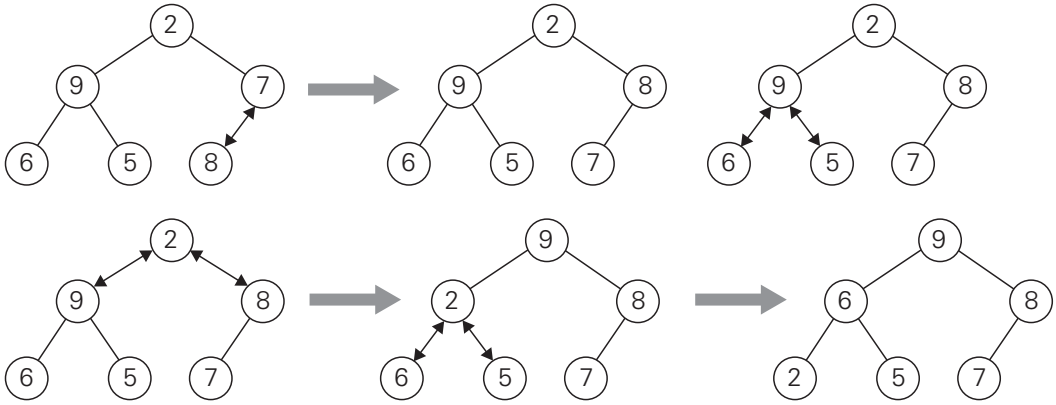

FIGURE 6.11 Bottom-up construction of a heap for the list 2, 9, 7, 6, 5, 8. The double-headed arrows show key comparisons verifying the parental dominance.

dominance holds for the key in this node. If it does not, the algorithm exchanges the node's key K with the larger key of its children and checks whether the parental dominance holds for K in its new position. This process continues until the parental dominance for K is satisfied. (Eventually, it has to because it holds automatically for any key in a leaf.) After completing the "heapification" of the subtree rooted at the current parental node, the algorithm proceeds to do the same for the node's immediate predecessor. The algorithm stops after this is done for the root of the tree.

ALGORITHM *HeapBottomUp*$(H[1..n])$

```
//Constructs a heap from elements of a given array
// by the bottom-up algorithm
//Input: An array H[1..n] of orderable items
//Output: A heap H[1..n]
for i ← ⌊n/2⌋ downto 1 do
    k ← i;   v ← H[k]
    heap ← false
    while not heap and 2 * k ≤ n do
        j ← 2 * k
        if j < n   //there are two children
            if H[j] < H[j + 1]  j ← j + 1
        if v ≥ H[j]
            heap ← true
        else H[k] ← H[j];   k ← j
    H[k] ← v
```

How efficient is this algorithm in the worst case? Assume, for simplicity, that $n = 2^k - 1$ so that a heap's tree is full, i.e., the largest possible number of

nodes occurs on each level. Let h be the height of the tree. According to the first property of heaps in the list at the beginning of the section, $h = \lfloor \log_2 n \rfloor$ or just $\lceil \log_2 (n+1) \rceil - 1 = k - 1$ for the specific values of n we are considering. Each key on level i of the tree will travel to the leaf level h in the worst case of the heap construction algorithm. Since moving to the next level down requires two comparisons—one to find the larger child and the other to determine whether the exchange is required—the total number of key comparisons involving a key on level i will be $2(h - i)$. Therefore, the total number of key comparisons in the worst case will be

$$C_{worst}(n) = \sum_{i=0}^{h-1} \sum_{\text{level } i \text{ keys}} 2(h-i) = \sum_{i=0}^{h-1} 2(h-i)2^i = 2(n - \log_2(n+1)),$$

where the validity of the last equality can be proved either by using the closed-form formula for the sum $\sum_{i=1}^{h} i2^i$ (see Appendix A) or by mathematical induction on h. Thus, with this bottom-up algorithm, a heap of size n can be constructed with fewer than $2n$ comparisons.

The alternative (and less efficient) algorithm constructs a heap by successive insertions of a new key into a previously constructed heap; some people call it the ***top-down heap construction*** algorithm. So how can we insert a new key K into a heap? First, attach a new node with key K in it after the last leaf of the existing heap. Then sift K up to its appropriate place in the new heap as follows. Compare K with its parent's key: if the latter is greater than or equal to K, stop (the structure is a heap); otherwise, swap these two keys and compare K with its new parent. This swapping continues until K is not greater than its last parent or it reaches the root (illustrated in Figure 6.12).

Obviously, this insertion operation cannot require more key comparisons than the heap's height. Since the height of a heap with n nodes is about $\log_2 n$, the time efficiency of insertion is in $O(\log n)$.

How can we delete an item from a heap? We consider here only the most important case of deleting the root's key, leaving the question about deleting an arbitrary key in a heap for the exercises. (Authors of textbooks like to do such things to their readers, do they not?) Deleting the root's key from a heap can be done with the following algorithm, illustrated in Figure 6.13.

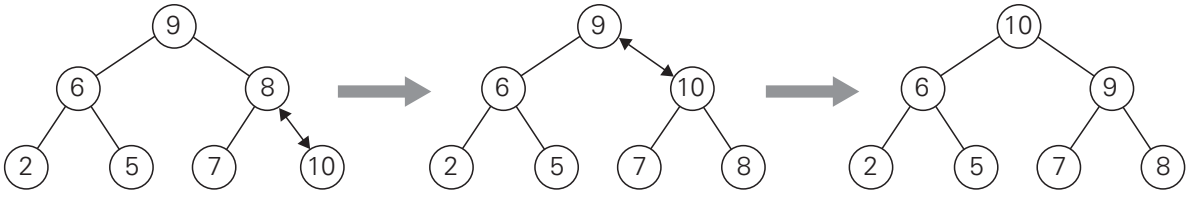

FIGURE 6.12 Inserting a key (10) into the heap constructed in Figure 6.11. The new key is sifted up via a swap with its parent until it is not larger than its parent (or is in the root).

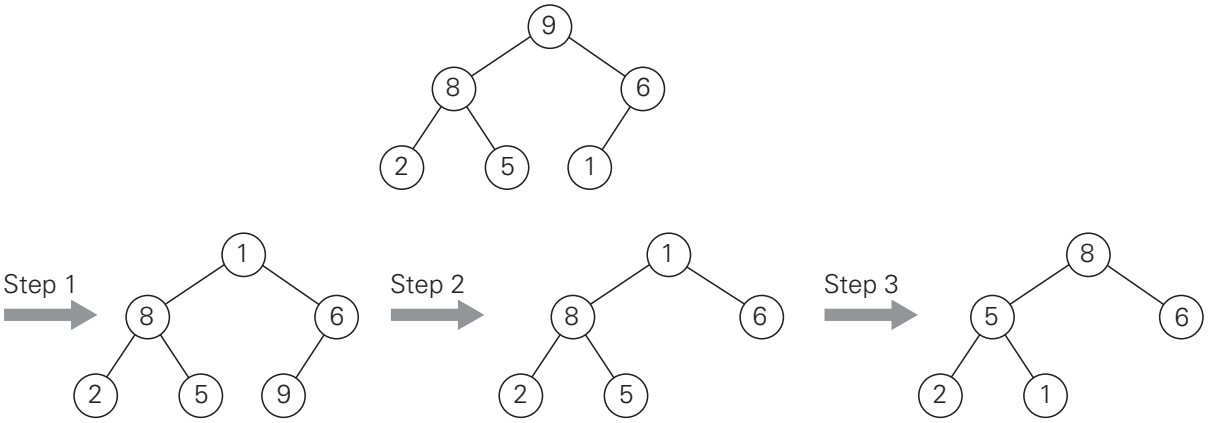

FIGURE 6.13 Deleting the root's key from a heap. The key to be deleted is swapped with the last key after which the smaller tree is "heapified" by exchanging the new key in its root with the larger key in its children until the parental dominance requirement is satisfied.

Maximum Key Deletion from a heap

Step 1 Exchange the root's key with the last key K of the heap.

Step 2 Decrease the heap's size by 1.

Step 3 "Heapify" the smaller tree by sifting K down the tree exactly in the same way we did it in the bottom-up heap construction algorithm. That is, verify the parental dominance for K: if it holds, we are done; if not, swap K with the larger of its children and repeat this operation until the parental dominance condition holds for K in its new position.

The efficiency of deletion is determined by the number of key comparisons needed to "heapify" the tree after the swap has been made and the size of the tree is decreased by 1. Since this cannot require more key comparisons than twice the heap's height, the time efficiency of deletion is in $O(\log n)$ as well.

Heapsort

Now we can describe ***heapsort***—an interesting sorting algorithm discovered by J. W. J. Williams [Wil64]. This is a two-stage algorithm that works as follows.

Stage 1 (heap construction): Construct a heap for a given array.

Stage 2 (maximum deletions): Apply the root-deletion operation $n - 1$ times to the remaining heap.

As a result, the array elements are eliminated in decreasing order. But since under the array implementation of heaps an element being deleted is placed last, the resulting array will be exactly the original array sorted in increasing order. Heapsort is traced on a specific input in Figure 6.14. (The same input as the one

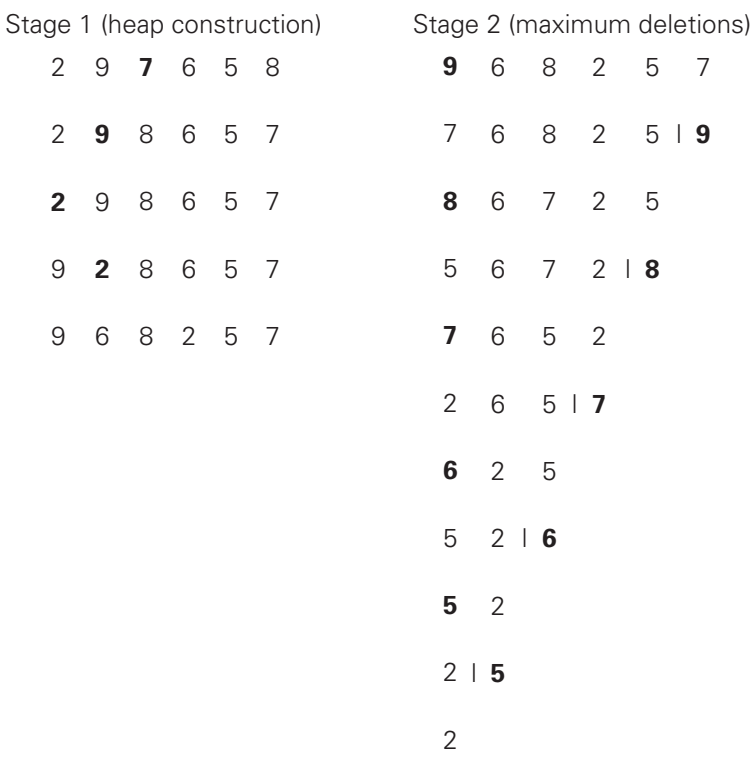

FIGURE 6.14 Sorting the array 2, 9, 7, 6, 5, 8 by heapsort.

in Figure 6.11 is intentionally used so that you can compare the tree and array implementations of the bottom-up heap construction algorithm.)

Since we already know that the heap construction stage of the algorithm is in $O(n)$, we have to investigate just the time efficiency of the second stage. For the number of key comparisons, $C(n)$, needed for eliminating the root keys from the heaps of diminishing sizes from n to 2, we get the following inequality:

$$C(n) \le 2\lfloor \log_2(n-1) \rfloor + 2\lfloor \log_2(n-2) \rfloor + \cdots + 2\lfloor \log_2 1 \rfloor \le 2 \sum_{i=1}^{n-1} \log_2 i$$

$$\le 2 \sum_{i=1}^{n-1} \log_2(n-1) = 2(n-1) \log_2(n-1) \le 2n \log_2 n.$$

This means that $C(n) \in O(n \log n)$ for the second stage of heapsort. For both stages, we get $O(n) + O(n \log n) = O(n \log n)$. A more detailed analysis shows that the time efficiency of heapsort is, in fact, in $\Theta(n \log n)$ in both the worst and average cases. Thus, heapsort's time efficiency falls in the same class as that of mergesort. Unlike the latter, heapsort is in-place, i.e., it does not require any extra storage. Timing experiments on random files show that heapsort runs more slowly than quicksort but can be competitive with mergesort.

Exercises 6.4

1. **a.** Construct a heap for the list 1, 8, 6, 5, 3, 7, 4 by the bottom-up algorithm.
   b. Construct a heap for the list 1, 8, 6, 5, 3, 7, 4 by successive key insertions (top-down algorithm).
   c. Is it always true that the bottom-up and top-down algorithms yield the same heap for the same input?
2. Outline an algorithm for checking whether an array $H[1..n]$ is a heap and determine its time efficiency.
3. **a.** Find the smallest and the largest number of keys that a heap of height h can contain.
   b. Prove that the height of a heap with n nodes is equal to $\lfloor \log_2 n \rfloor$.
4. Prove the following equality used in Section 6.4:
$$\sum_{i=0}^{h-1} 2(h-i)2^i = 2(n - \log_2(n+1)), \quad \text{where } n = 2^{h+1} - 1.$$
5. **a.** Design an efficient algorithm for finding and deleting an element of the smallest value in a heap and determine its time efficiency.
   b. Design an efficient algorithm for finding and deleting an element of a given value v in a heap H and determine its time efficiency.
6. Indicate the time efficiency classes of the three main operations of the priority queue implemented as
   a. an unsorted array.
   b. a sorted array.
   c. a binary search tree.
   d. an AVL tree.
   e. a heap.
7. Sort the following lists by heapsort by using the array representation of heaps.
   a. 1, 2, 3, 4, 5 (in increasing order)
   b. 5, 4, 3, 2, 1 (in increasing order)
   c. S, O, R, T, I, N, G (in alphabetical order)
8. Is heapsort a stable sorting algorithm?
9. What variety of the transform-and-conquer technique does heapsort represent?
10. Which sorting algorithm other than heapsort uses a priority queue?
11. Implement three advanced sorting algorithms—mergesort, quicksort, and heapsort—in the language of your choice and investigate their performance on arrays of sizes $n = 10^3$, 10^4, 10^5, and 10^6. For each of these sizes consider