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:

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

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]

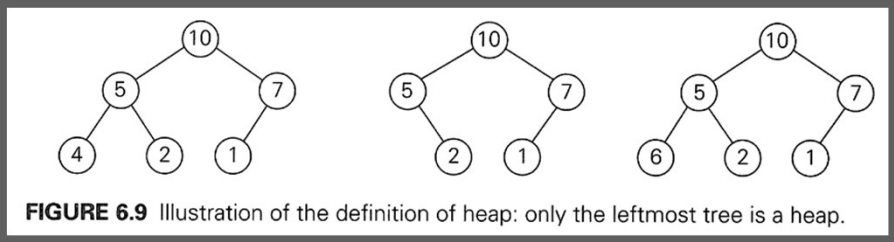

FIGURE 6.9 Illustration of the definition of heap: only the leftmost tree is a heap.

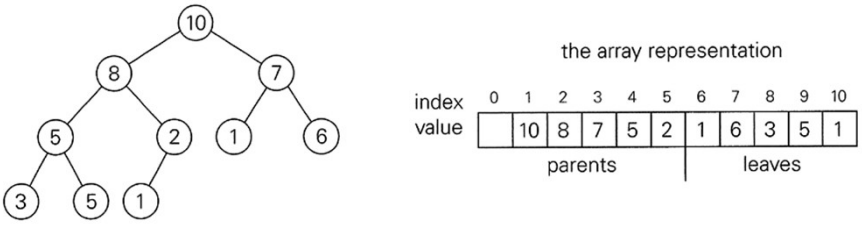

FIGURE 6.10 Heap and its array representation.

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.,

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

(Of course, if $2i + 1 > n$, just $H[i] \geq H[2i]$ 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 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.

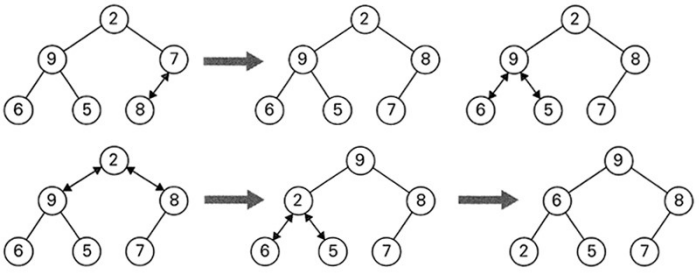

FIGURE 6.11 Bottom-up construction of a heap for the list 2, 9, 7, 6, 5, 8. The double-headed arrows show key comparisons verifying the parental dominance.

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 $i \leftarrow \lfloor n/2 \rfloor$ downto 1 do
    $k \leftarrow i$; $v \leftarrow H[k]$
    heap←false
    while not *heap* and $2 * k \leq n$ do
        $j \leftarrow 2 * k$
        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 2*n* 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).

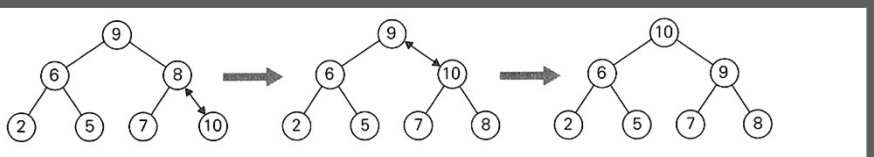

FIGURE 6.12 Inserting a key (10) into the heap constructed in Figure 6.11. The new key is sifted up via a swap with its parent until it is not larger than its parent (or is in the root).

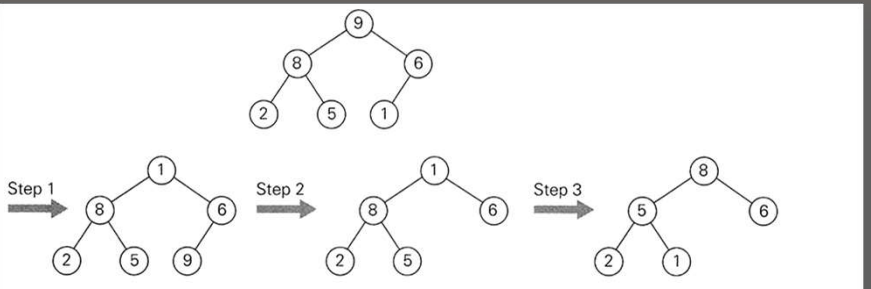

FIGURE 6.13 Deleting the root's key from a heap. The key to be deleted is swapped with the last key after which the smaller tree is "heapified" by exchanging the new key in its root with the larger key in its children until the parental dominance requirement is satisfied.

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.

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.

Stage 1 (heap construction)	Stage 2 (maximum deletions)
2 9 **7** 6 5 8	**9** 6 8 2 5 7
2 **9** 8 6 5 7	7 6 8 2 5 | **9**
2 9 8 6 5 7	**8** 6 7 2 5
9 **2** 8 6 5 7	5 6 7 2 | **8**
9 6 8 2 5 7	**7** 6 5 2
	2 6 5 | **7**
	6 2 5
	5 2 | **6**
	5 2
	2 | **5**
	2

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

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:

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

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