3.5 Priority Queues, Binary Heaps, and Heapsort

Priority Queues

The abstract data type known as a **priority queue** allows us to insert an item with a specified priority (given by a number) and to delete an item having the *highest* priority. For example, if we are processing jobs with specified priorities and there is a single processor available, as the jobs arrive we can insert them into a priority queue. When the processor becomes available, we can delete a job from the priority queue—which, by definition, has the highest priority—and process it.

In the discussion that follows, we list only the priority of an item. In practice, a data item has other attributes besides its priority (e.g., an identification number, time started, time stopped).

Example 3.5.1. Suppose that items with the priorities

$$57, 32, 100, 56, 44$$

are inserted into an initially empty priority queue. If we delete an item from the priority queue, 100 will be removed, regardless of the order in which the items were inserted, because the highest priority is 100. The priority queue contains

$$57, 32, 56, 44.$$

If we delete another item from the priority queue, 57 will be removed since it now has the highest priority. The priority queue contains

$$32, 56, 44.$$

If we now insert 37, the priority queue contains

$$32, 56, 44, 37.$$

If we delete another item from the priority queue, 56 will be removed since it now has the highest priority. The priority queue contains

$$32, 44, 37.$$ □

Consider implementing a priority queue using an array. If an item is inserted by putting it at the end of the array, insertion takes time $\Theta(1)$. To delete an item, we must first locate the item having the greatest priority. Since the items are in no particular order, we would have to scan all the items to find one having the largest priority. Scanning an n-element array takes time $\Theta(n)$. After finding an item having the largest priority, we then have to remove it. This entails shifting all items to its right one cell to the left, which takes time $O(n)$. Thus, deletion always takes time $\Theta(n)$.

When using a priority queue, we typically perform many insertions and deletions. Suppose, for example, that we perform n insertions and n deletions in an initially empty priority queue. Since each insertion takes constant

time, the total time for the insertions is $\Theta(n)$. The time for a deletion is $O(n)$ and, since we perform n deletions, the total time for the deletions is $O(n^2)$. The time for deletions dominates. Thus, the total time to perform n insertions and n deletions in an initially empty priority queue is $O(n^2)$. This estimate is sharp. In the worst case, performing n insertions and n deletions takes time $\Theta(n^2)$ (see Exercise 3).

Suppose that we still implement a priority queue using an array, but we maintain nondecreasing order. Now, when we insert an item, we first have to determine where it should be inserted to maintain nondecreasing order. If the item is to be inserted in the cell at index i, for each $j \geq i$ we will have to shift the item in the cell at index j to the cell at index $j + 1$ to make room for the inserted item. In the worst case (when the item to be inserted is smaller than all the others), all items have to move. Moving all items in an n-element array takes time $\Theta(n)$. We can determine where to insert the item by scanning all the items, which takes time $\Theta(n)$. Thus, in the worst case, insertion takes time $\Theta(n)$. [Using binary search (see Section 4.1), we can determine where to insert the item in time $O(\lg n)$. However, in the worst case, moving the items still takes time $\Theta(n)$; so, binary search does not improve the asymptotic worst-case time of the insertion algorithm.] Deleting an item takes time $\Theta(1)$, since the item with largest priority is at the end of the array.

Again, suppose that we perform n insertions and n deletions in an initially empty priority queue. Since each deletion takes constant time, the total time for the deletions is $\Theta(n)$. The time for an insertion is $O(n)$ and, since we perform n insertions, the total time for the insertions is $O(n^2)$. The time for insertions dominates. Thus, the total time to perform n insertions and n deletions in an initially empty priority queue is $O(n^2)$. This estimate is sharp. In the worst case, performing n insertions and n deletions takes time $\Theta(n^2)$ (see Exercise 4).

The two proposed ways to implement a priority queue represent the extremes of data organization and times for inserting and deleting. In the first implementation, the array is not sorted, insertion takes time $\Theta(1)$, and deletion takes time $O(n)$. In the second implementation, the array is sorted, insertion takes time $O(n)$, and deletion takes time $\Theta(1)$. For both implementations, in the worst case performing n insertions and n deletions takes time $\Theta(n^2)$. We can use a **heap** to maintain a "weak order," an organization that is not completely sorted but is not random either. By doing so, we can perform both insertions and deletions in time $O(\lg n)$. Thus, performing n insertions and n deletions takes time $O(n \lg n)$, which, in the worst case, is better than either of the other implementations.

Heaps

We begin by defining a *heap structure.*

Definition 3.5.2. A *heap structure* is a binary tree in which all levels, except possibly the last (bottom) level, have as many nodes as possible. On the last level, all of the nodes are at the left.

Example 3.5.3. Figure 3.5.1 shows a heap structure. Levels 0 (the root), 1, and 2 (the next-to-last level) have as many nodes as possible. On level 3 (the last level), all of the nodes are at the left.

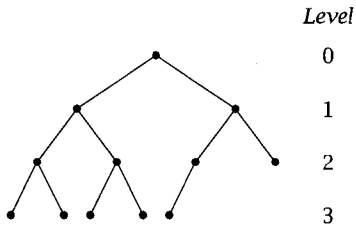

Figure 3.5.1 A heap structure. All levels, except the last level, 3, have as many nodes as possible. On the last level, all of the nodes are at the left. □

Definition 3.5.4. A *binary minheap* is a heap structure in which values are assigned to the nodes so that the value of each node is less than or equal to the values of its children (if any). A *binary maxheap* is a heap structure in which values are assigned to the nodes so that the value of each node is greater than or equal to the values of its children (if any).

Throughout the remainder of this section, we will discuss binary maxheaps and abbreviate "binary maxheap" to "heap." (The algorithms for manipulating minheaps are similar to those for manipulating maxheaps.)

Example 3.5.5. Figure 3.5.2 shows a heap. The value of each node is greater than or equal to the values of its children (if any).

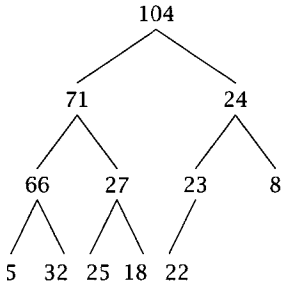

Figure 3.5.2 A heap. The value of each node is greater than or equal to the values of its children (if any). For example, the value 24 is greater than or equal to the values of the children, 23 and 8. □

A heap is "weakly sorted" in the sense that the values along a path from the root to a terminal node are in nonincreasing order. For example, in Figure 3.5.2 the values 104, 71, 66, 5, along the path from the root to the terminal node with value 5, are in nonincreasing order. At the same time,

the values along a *level* are, in general, in no particular order. For example, in Figure 3.5.2 the values 5, 32, 25, 18, 22 on level 3 are in neither nonincreasing nor nondecreasing order. (See also Exercise 21.)

In a heap, the maximum value is at the root. Thus, if we implement a priority queue using a heap, the item with highest priority is at the root. In order to use a heap to implement a priority queue, we must represent the heap. We do so using an array! We number the nodes level by level, left to right, starting at the root. In Figure 3.5.3, we have numbered the nodes in the heap structure of Figure 3.5.1.

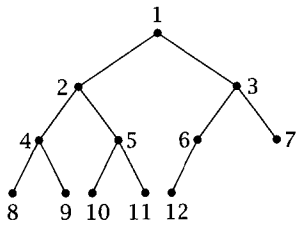

Figure 3.5.3 A heap structure with the nodes numbered level by level, left to right, starting at the root.

To represent a heap, we store the value in node number i in cell i of an array. In Figure 3.5.4, we represent the heap of Figure 3.5.2 as an array.

104	71	24	66	27	23	8	5	32	25	18	22
1	2	3	4	5	6	7	8	9	10	11	12

Figure 3.5.4 The heap of Figure 3.5.2 as an array. The value in node number i is stored in cell i of the array.

It is apparent from Figure 3.5.3 that the parent of node i, assuming that the node is not the root, is $\lfloor i/2 \rfloor$. Also, the left child of node i, assuming that the node has a left child, is $2 * i$, and the right child of node i, assuming that the node has a right child, is $2 * i + 1$. We will use these formulas in our subsequent heap algorithms.

In order to implement a priority queue as a heap, we must write algorithms to return the largest value in the heap, to delete the largest value from a heap, and to insert an arbitrary value into a heap. Returning the largest value in the heap is straightforward since the largest value is at the root.

Algorithm 3.5.6 Largest. This algorithm returns the largest value in a heap. The array v represents the heap.

Input Parameter: v
Output Parameters: None

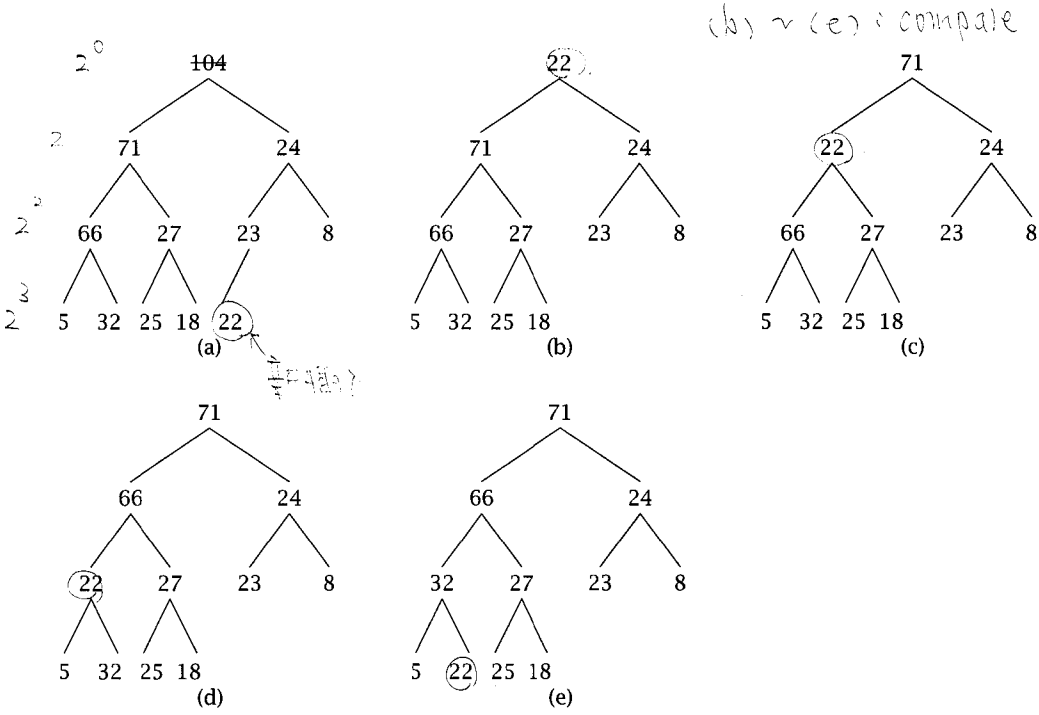

Figure 3.5.5 Deleting from a heap. The root, which contains the largest value, 104, is to be deleted [see (a)]. To maintain a heap structure, we move the value at the bottom level, farthest right, 22, to the root [see (b)]. The root is not greater than or equal to its children; so, we swap the root's value, 22, with the largest child, 71, [see (c)]. The node with value 22 is not greater than or equal to its children; so, we again swap 22 with the largest child, 66, [see (d)]. Again, the node with value 22 is not greater than or equal to its children; so, we again swap 22 with the largest child, 32, [see (e)]. Since 22 now has no children, the algorithm terminates. The structure shown in (e) is a heap.

```
heap_largest(v) {
  return v[1]
}
```

Algorithm 3.5.6 runs in constant time.

We next discuss the delete algorithm, whose implementation is shown in Figure 3.5.5. Before writing the delete algorithm, we write a separate algorithm to repeatedly swap a value with the larger child. We allow the operation to begin at an arbitrary node rather than at the root as in Figure 3.5.5, since we will need this more general version later. Thus, we assume that v is an array representing a heap structure indexed from 1 to n. We further assume that the left subtree of node i is a heap and the right subtree of node i is also a heap [see Figure 3.5.6(a)]. After *siftdown*(v, i, n) is called, the subtree rooted at i is a heap [see Figure 3.5.6(b)]. To make the algorithm more efficient, we do not actually swap values; rather, we copy the value $v[i]$ to a temporary variable and repeatedly move the larger child up, if necessary.

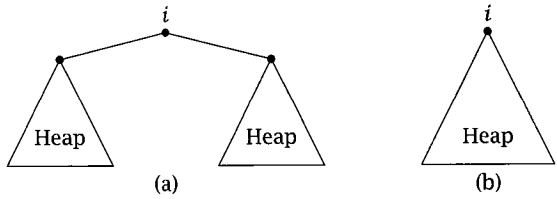

Figure 3.5.6 The siftdown algorithm. Initially, the left and right subtrees of node i are heaps [see (a)]. After *siftdown* is called, the subtree rooted at i is a heap [see (b)].

After locating the cell where $v[i]$ goes, we copy it to that cell.

Algorithm 3.5.7 Siftdown. The array v represents a heap structure indexed from 1 to n. The left and right subtrees of node i are heaps. After

$siftdown(v, i, n)$

is called, the subtree rooted at i is a heap.

Input Parameters: v, i, n
Output Parameter: v

```
siftdown(v, i, n) {
  temp = v[i]
  // 2 * i ≤ n tests for a left child
  while (2 * i ≤ n) {
    child = 2 * i
    // if there is a right child and it is bigger than the left child, move child
    if (child < n && v[child + 1] > v[child])
      child = child + 1
    // move child up?
    if (v[child] > temp)
      v[i] = v[child]
    else
      break // exit while loop
    i = child
  }
  // insert original v[i] in correct spot
  v[i] = temp
}
```

In the worst case, the variable i in Algorithm 3.5.7 travels all the way from the root to the last level; thus, in the worst-case, the while loop in Algorithm 3.5.7 executes $\Theta(h)$ times, where h is the height of the subtree rooted at i. We show that if the subtree rooted at i has m nodes, $h = \lfloor \lg m \rfloor$; thus, the worst-case time of Algorithm 3.5.7 is $\Theta(\lg m)$.

Theorem 3.5.8. *The height of a heap structure containing m nodes is* $\lfloor \lg m \rfloor$.

Proof. Suppose that a heap structure containing m nodes has height $h > 0$. Then, level i, $i < h$, contains the maximum number of nodes 2^i (see Figure 3.5.1). Thus, levels 0 through $h - 1$ contain

$$2^0 + 2^1 + \cdots + 2^{h-1} = 2^h - 1$$

nodes. (We have used the formula for the geometric sum; see Example 2.2.2.) Level h contains 1 to 2^h nodes. Thus, m satisfies

$$(2^h - 1) + 1 \leq m \leq (2^h - 1) + 2^h = 2^{h+1} - 1$$

or

$$2^h \leq m < 2^{h+1}.$$

Notice that the last inequality is also true if $h = 0$. Taking the logarithm to the base 2 of each expression in the last inequality yields

$$h \leq \lg m < h + 1.$$

Therefore,

$$h = \lfloor \lg m \rfloor.$$ ■

We now write the delete algorithm.

Algorithm 3.5.9 Delete. This algorithm deletes the root (the item with largest value) from a heap containing n elements. The array v represents the heap.

Input Parameters: v, n
Output Parameters: v, n

```
heap_delete(v, n) {
  v[1] = v[n]
  n = n - 1
  siftdown(v, 1, n)
}
```

When the input is a heap structure of size n, *siftdown* runs in time $\Theta(\lg n)$ in the worst case; therefore, the worst-case time of *heap_delete* is $\Theta(\lg n)$.

We turn next to the insert algorithm, whose implementation is shown in Figure 3.5.7. As in the delete algorithm, we do not actually swap values in the insert algorithm; rather we repeatedly move the parent down, if necessary. After locating the cell where the added value goes, we copy it to that cell.

Algorithm 3.5.10 Insert. This algorithm inserts the value *val* into a heap containing n elements. The array v represents the heap.

Input Parameters: val, v, n
Output Parameters: v, n

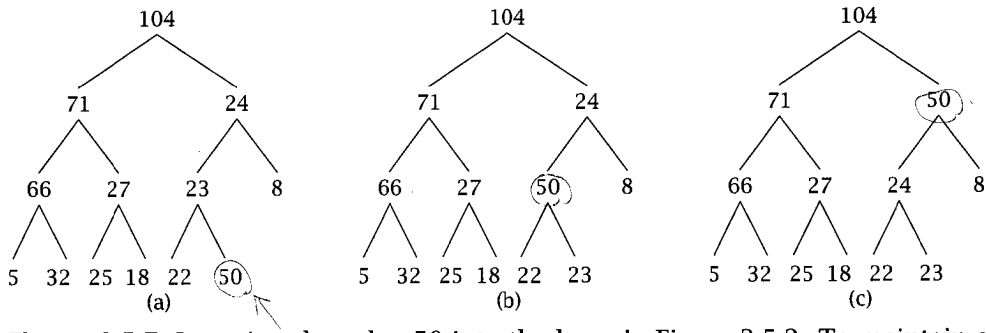

Figure 3.5.7 Inserting the value 50 into the heap in Figure 3.5.2. To maintain a heap structure, the value added, 50, is inserted in the bottom level, farthest right[see (a)].(If the bottom level were full, we would begin another level at the left.) Since 50 is larger than its parent, we swap 50 with its parent [see (b)]. Again 50 is larger than its parent; so, we again swap 50 with its parent [see (c)]. This time 50 is not larger than its parent, so the insert algorithm terminates.

```
heap_insert(val, v, n) {
  i = n = n + 1
  // i is the child and i/2 is the parent.
  // If i > 1, i is not the root.
  while (i > 1 && val > v[i/2]) {
    v[i] = v[i/2]
    i = i/2
  }
  v[i] = val
}
```

Suppose that we use Algorithm 3.5.10 to insert into a heap containing n elements. In the worst case, the value added travels all the way from last level to the root; thus, the worst-case time of Algorithm 3.5.10 is $\Theta(h)$, where h is the height of the tree, which now contains $n + 1$ elements. By Theorem 3.5.8, the height of the tree is $\lfloor \lg(n + 1) \rfloor$. Therefore, the worst-case time of Algorithm 3.5.10 is $\Theta(\lg n)$.

Suppose that we have an array of n elements that we want to organize into a heap. We could use Algorithm 3.5.10 to insert the elements one at a time into an initially empty heap. Since each insertion takes time $O(\lg n)$, the total time is $O(n \lg n)$. This estimate is sharp; that is, the worst-case time to construct a heap in this way is $\Theta(n \lg n)$ (see Exercise 25). We can do better by repeatedly using *siftdown* (Algorithm 3.5.7).

Example 3.5.11. Consider the problem of organizing the data shown in Figure 3.5.8(a) into a heap. Notice that the left and right subtrees of the parent having the largest index 4 are trivially heaps. Thus we may call *siftdown* on node 4. The result is shown in Figure 3.5.8(b).

The left and right subtrees of the node having the next largest index, 3, are heaps. Thus we may call *siftdown* on node 3. The result is shown in Figure 3.5.8(c).

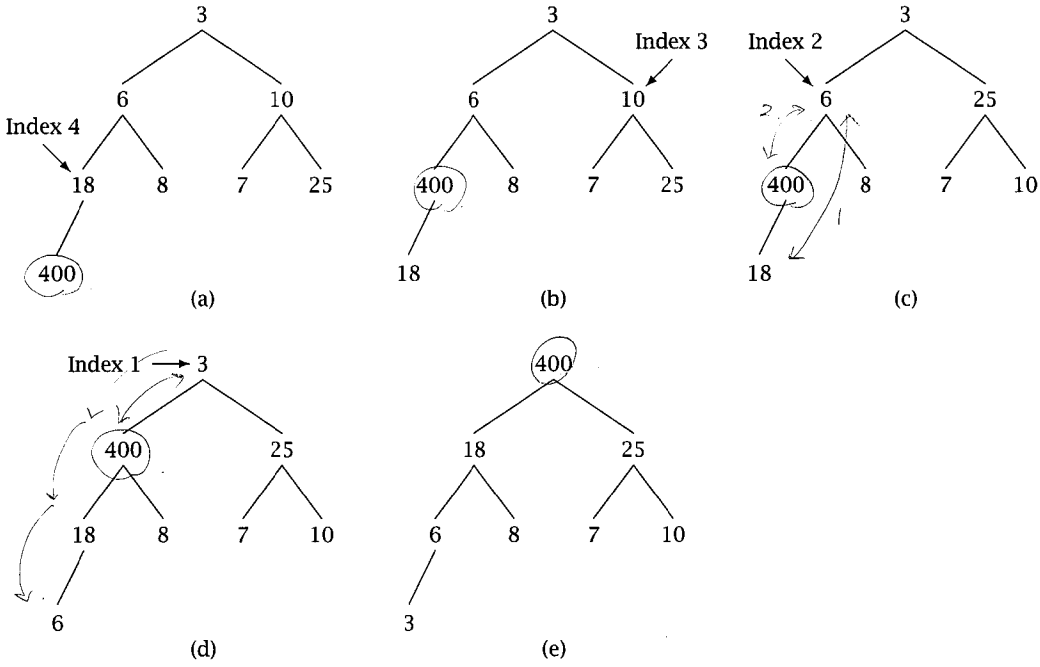

Figure 3.5.8 Making a heap. The input is shown in (a). *siftdown* is first called on the parent having the largest index, 4. The result is shown in (b). *siftdown* is next called on the node having the next largest index, 3. The result is shown in (c). *siftdown* is next called on the node having the next largest index, 2. The result is shown in (d). Finally, *siftdown* is called on the root. The result, shown in (e), is a heap.

Now the left and right subtrees of the node having the next largest index, 2, are heaps. Thus we may call *siftdown* on node 2. The result is shown in Figure 3.5.8(d).

Finally, the left and right subtrees of the root are heaps. Thus we may call *siftdown* on node 1. The result, shown in Figure 3.5.8(e), is a heap. □

We state the algorithm to make a heap as Algorithm 3.5.12.

Algorithm 3.5.12 Heapify. This algorithm rearranges the data in the array v, indexed from 1 to n, so that it represents a heap.

Input Parameters: v, n
Output Parameters: v

```
heapify(v, n) {
  // n/2 is the index of the parent of the last node
  for i = n/2 downto 1
    siftdown(v, i, n)
}
```

We show that Algorithm 3.5.12 runs in linear time.

Theorem 3.5.13. *The time for Algorithm 3.5.12 is* $\Theta(n)$.

Proof. The time for Algorithm 3.5.12 is bounded by the worst-case time of all of the calls to *siftdown*.

There is one node on level 0, namely the root. The worst-case time for this call to *siftdown* (counting iterations of *siftdown*'s while loop) is h, where $h = \lfloor \lg n \rfloor$ is the height of the tree.

There are two nodes on level 1; so, the worst-case time for these two calls is $2(h-1)$.

In general, on level i, $i < h$, there are 2^i nodes; so, the worst-case time for these 2^i calls is bounded by $2^i(h-i)$. [*siftdown* may not be called on all nodes on level $h-1$ (see Figure 3.5.8); however, its worst-case time is bounded by $2^{h-1} \cdot 1$.] It follows that the time of Algorithm 3.5.12 is bounded by

$$\sum_{i=0}^{h-1} 2^i(h-i).$$

Now

$$\sum_{i=0}^{h-1} 2^i(h-i) = \sum_{i=1}^{h} 2^{h-i}i = 2^h \sum_{i=1}^{h} \frac{i}{2^i}.$$

Taking $r = 1/2$ in Theorem 2.2.3, we obtain

$$\sum_{i=1}^{h} \frac{i}{2^i} < 2.$$

Therefore,

$$2^h \sum_{i=1}^{h} \frac{i}{2^i} < 2^h \cdot 2 = 2^{\lfloor \lg n \rfloor} \cdot 2 \leq 2^{\lg n} \cdot 2 = 2n,$$

and the time for Algorithm 3.5.12 is $O(n)$.

On the other hand, the for loop in Algorithm 3.5.12 executes $\Theta(n)$ times; so, the time for Algorithm 3.5.12 is $\Omega(n)$. Therefore, the time for Algorithm 3.5.12 is $\Theta(n)$. ■

Indirect Heaps

In some applications (see, e.g., Sections 7.3 and 7.4), we need to modify a value in a heap. To specify which element to modify, we provide its index. The problem is that in the current implementation, the indexes of the various values change. In order to modify a value in a heap efficiently, we maintain an array of values, *key*, which does not change unless a *value* (as opposed to the *index* of a value) in the heap is modified. For example, the siftdown algorithm would not modify *key*. We then maintain two additional arrays: *into*$[i]$, whose value is the index in the heap structure where *key*$[i]$ is found; and *outof*$[j]$, whose value is the index in *key* where the value of node j in

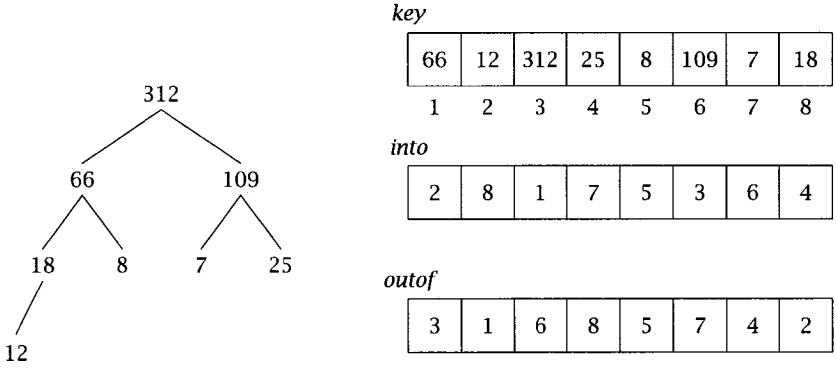

Figure 3.5.9 An indirect heap. The heap structure is shown at the left. The *key* array stores the values contained in the heap structure. The value *key*[*i*] is stored at index *into*[*i*] in the heap structure. For example, the value 25 = *key*[4] is stored at index *into*[4] = 7 in the heap structure. The value at index *j* in the heap structure is at index *outof*[*j*] in the *key* array. For example, the value, 18, at index 4 in the heap structure is at index *outof*[4] = 8 in the *key* array. Notice that *into* and *outof* are inverses of each other in the sense that *into*[*outof*[*j*]] = *j* for all *j*, and *outof*[*into*[*i*]] = *i* for all *i*.

the heap structure is found (see Figure 3.5.9). Such a structure is sometimes called an **indirect heap.**

Example 3.5.14. As an example of manipulating an indirect heap, we show how to swap the value 66 at index 2 and the value 18 at index 4 in the indirect heap in Figure 3.5.9.

In Figure 3.5.9, *outof*[2] = 1 gives the index of 66 in the *key* array, and *outof*[4] = 8 gives the index of 18 in the *key* array. These values in *outof* must be swapped:

temp = *outof*[2]
outof[2] = *outof*[4]
outof[4] = *temp*

The changes are shown in Figure 3.5.10 (next page).

Similarly, the *into* values must also be swapped:

temp = *into*[*outof*[2]]
into[*outof*[2]] = *into*[*outof*[4]]
into[*outof*[4]] = *temp*

(see Figure 3.5.10). □

Consider writing an *increase* algorithm. The input is an index into the *key* array, which specifies the value to be increased, and the replacement value. The algorithm is illustrated in Figure 3.5.11.

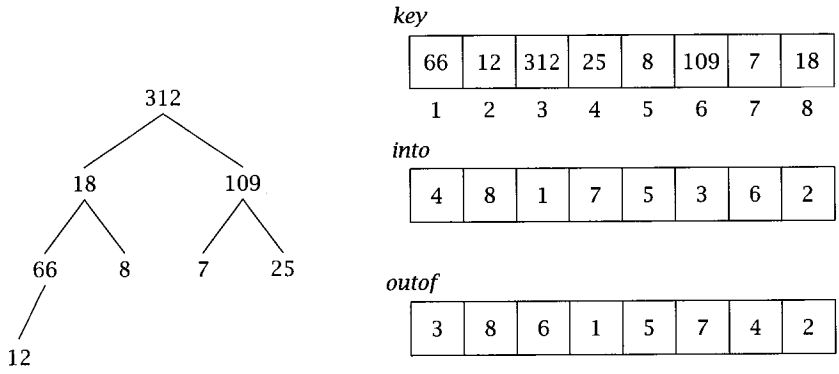

Figure 3.5.10 The heap structure of Figure 3.5.9 after swapping the value 66 at index 2 and the value 18 at index 4. The values *outof*[2] and *outof*[4] are swapped as are the corresponding *into* values, *into*[*outof*[2]] and *into*[*outof*[4]].

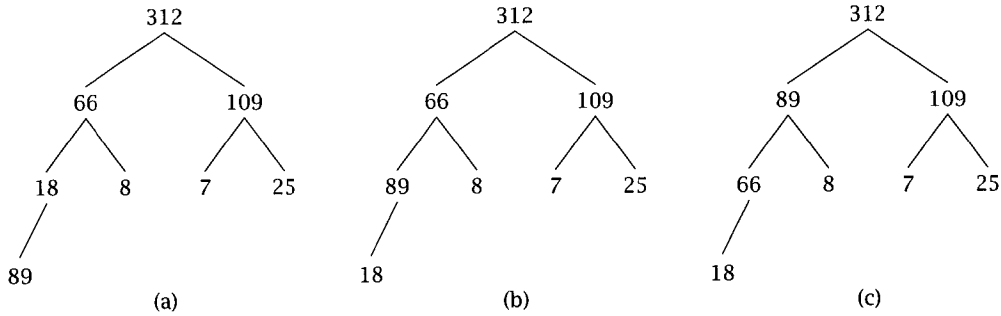

Figure 3.5.11 The increase algorithm. The value 12 in the heap in Figure 3.5.9 is to be increased to 89 [see (a)]. Since 89 is larger than its parent (18), 18 must move down [see (b)]. Again, 89 is larger than its parent (66), so 66 must move down [see (c)]. This time 89 is not larger than its parent (312); thus, the heap is restored and the increase algorithm terminates.

Algorithm 3.5.15 Increase. This algorithm increases a value in an indirect heap and then restores the heap. The input is an index i into the *key* array, which specifies the value to be increased, and the replacement value *newval*.

Input Parameters: i, *newval*
Output Parameters: None

```
increase(i, newval) {
  key[i] = newval
  // p is the parent index in the heap structure
  // c is the child index in the heap structure
  c = into[i]
```

```
  p = c/2
  while (p ≥ 1) {
    if (key[outof[p]] ≥ newval)
      break // exit while loop
    // move value at p down to c
    outof[c] = outof[p]
    into[outof[c]] = c
    // move p and c up
    c = p
    p = c/2
  }
  // "put" newval in heap structure at index c
  outof[c] = i
  into[i] = c
}
```

The worst-case time of Algorithm 3.5.15 occurs when the value to be increased is at the bottom of the heap and migrates to the root. In this case, the while loop iterates h times, where h is the height of the heap. By Theorem 3.5.8, $h = \lfloor \lg n \rfloor$; thus, the worst-case time of Algorithm 3.5.15 is $\Theta(\lg n)$, where n is the number of items in the heap.

We leave the details of implementing other heap algorithms, when the heap is implemented as an indirect heap, to the exercises (see Exercises 27–32).

Heapsort

Figure 3.5.12 shows how a heap can be used to sort an array. The algorithm that results is called **heapsort**. We write it as Algorithm 3.5.16.

Algorithm 3.5.16 Heapsort. This algorithm sorts the array $v[1], \ldots, v[n]$ in nondecreasing order. It uses the siftdown and heapify algorithms (Algorithms 3.5.7 and 3.5.12).

Input Parameters: v, n
Output Parameter: v

```
heapsort(v, n) {
  // make v into a heap
  heapify(v, n)
  for i = n downto 2 {
    // v[1] is the largest among v[1], ..., v[i].
    // Put it in the correct cell.
    swap(v[1], v[i])
    // Heap is now at indexes 1 through i - 1.
    // Restore heap.
    siftdown(v, 1, i - 1)
  }
}
```

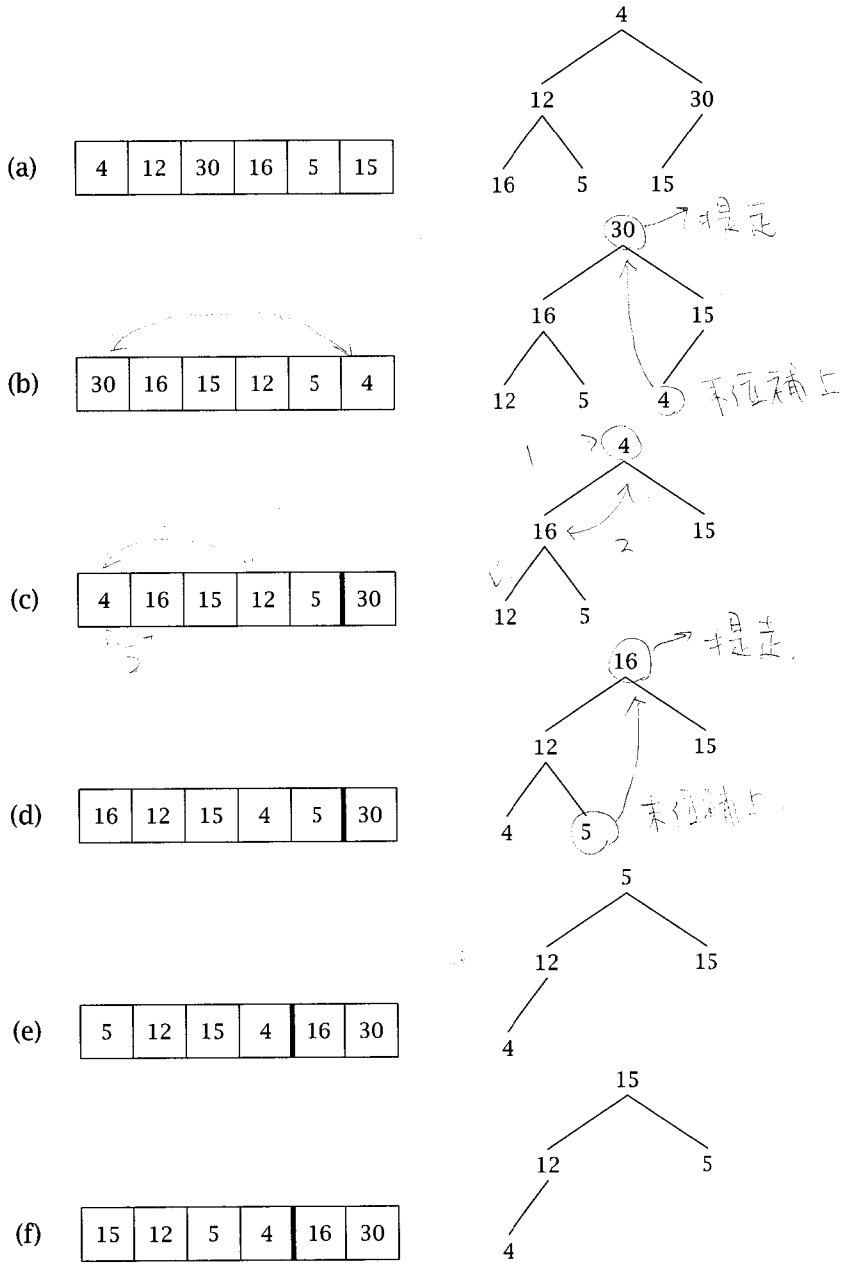

Figure 3.5.12 The first steps in the heapsort algorithm. The input array is considered a heap structure [see (a)]. First, *heapify* is called [see (b)]. The largest element, at index 1, is swapped with the last element; and the elements, except the last, are considered a heap structure [see (c)]. Next, *siftdown* is called [see (d)]. The second-largest element, at index 1, is swapped with the next-to-last element; and the elements, except the last two, are considered a heap structure [see (e)]. Again, *siftdown* is called [see (f)]. The process is repeated until the array is sorted.

In the heapsort algorithm, the call to *heapify* takes time $\Theta(n)$. Each call to *siftdown* takes time $O(\lg n)$. Since the calls to *siftdown* are in a for loop that runs in time $\Theta(n)$, the total time for the calls to *siftdown* is $O(n \lg n)$. Therefore, heapsort runs in time $O(n \lg n)$. Corollary 6.3.3 shows that any comparison-based sorting algorithm, of which heapsort is an example, has worst-case time $\Omega(n \lg n)$. Therefore, the worst-case time of heapsort is $\Theta(n \lg n)$. It is also possible to show directly that the worst-case time of heapsort is $\Theta(n \lg n)$ by constructing input that requires time $\Theta(n \lg n)$ (see Exercise 3.12).

Notice that except for the input array, heapsort uses a constant amount of storage. In practice, heapsort is the fastest general sorting algorithm that, except for the input array, uses a constant amount of storage and guarantees worst-case time $\Theta(n \lg n)$. [In practice, quicksort (see Section 6.2) is faster on average than heapsort. However, in addition to the input array, quicksort, if properly implemented, uses $O(\lg n)$ storage and has worst-case time $\Theta(n^2)$.]

Exercises

1S. A priority queue is implemented using an array. An item is inserted by putting it at the end of the array. Write algorithms to initialize a priority queue to empty, to delete an item with the highest priority, to return an item with the highest priority, and to insert an item with a specified priority.

2. A priority queue is implemented using an array. An item is inserted by putting it in a cell that maintains nondecreasing order. Write algorithms to initialize a priority queue to empty, to delete an item with the highest priority, to return an item with the highest priority, and to insert an item with a specified priority.

3. A priority queue is implemented using an array. An item is inserted by putting it at the end of the array. Show that, in the worst case, performing n insertions and n deletions in an initially empty priority queue takes time $\Theta(n^2)$.

4S. A priority queue is implemented using an array. An item is inserted by putting it in a cell that maintains nondecreasing order. Show that, in the worst case, performing n insertions and n deletions in an initially empty priority queue takes time $\Theta(n^2)$.

5. Show how to implement a stack using a priority queue.

6. Show how to implement a queue using a priority queue.

Unless specified otherwise, all heaps are maxheaps.