Chapter 6 Heapsort

Recall that Insertion-Sort has a worst-case running time of $\theta(n^2)$ while Merge-Sort is better with a worst-case running time of $\theta(n \lg n)$. But Merge-Sort requires extra $\theta(n)$ space for the auxiliary array while Insertion-Sort only needs a constant amount, $\theta(1)$, of extra space. Merge-Sort might run out of memory for large n.

A sorting algorithm sorts ***in place*** if only a constant number of elements in the input array are ever stored outside the array. Our next algorithm is Heapsort; it sorts in place and it has a worst-case running time of $\theta(n \lg n)$ so it conserves both space and time. Heapsort uses heaps and a heap uses a heap data structure.

A **heap data structure** is a complete binary tree. The lowest level of the binary tree is filled from left to right and higher levels are completely filled.

The **height** of the tree is the number of edges along the path from the root to the farthest leaf. What is the height of a tree with n nodes?

n	1	2	3	4	5	6	7
height	0	1	1	2	2	2	2

height = floor(lg n). The height is $\theta(\lg n)$.

A heap data structure of n nodes is stored in an array A[1..n]. The root is stored in A[1], its left child in A[2], and its right child in A[3]. The next level of four nodes is stored from left to right in A[4..7] and then the next level of eight nodes is stored from left to right in A[8..15], etc., with the lowest level stored from left to right at the end of array A.

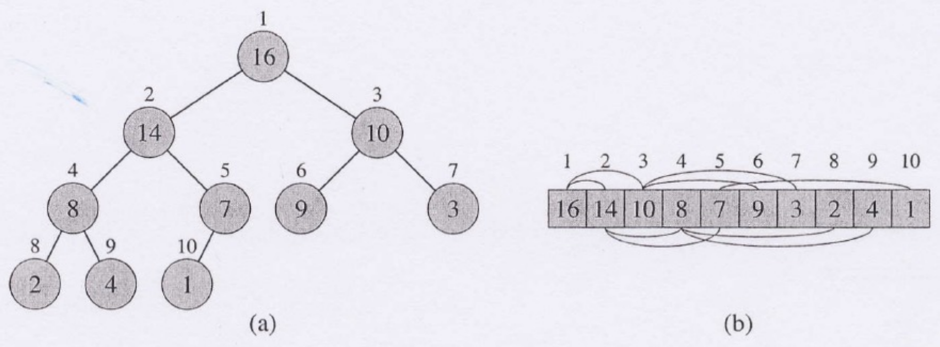

Figure 6.1 A max-heap viewed as **(a)** a binary tree and **(b)** an array. The number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships;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

The storage scheme ensures that for any index i:

- if the node stored in A[i] has a left-child the left-child is stored in A[2i];
- if the node stored in A[i] has a right-child the right-child is stored in A[2i+1]; and
- if the node stored in A[i] has a parent (i.e., if $i > 1$) then the parent is stored in A[floor(i/2)].

Hence, the above properties could be summarized by:

```
PARENT(i)
1       return (floor i/2)

LEFT(i)
1       return 2i

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

The above properties make it easy to move around in the heap structure from parent to child and from child to parent even though the heap structure is really stored in a linear array.

Heap Property

- For max-heaps (largest element at root), ***max-heap property***: for all nodes i, excluding the root, A[Parent(i)] ≥ A[i]
- For min-heaps (smallest element at root), ***min-heap property***: for all nodes i, excluding the root, A[Parent(i)] ≤ A[i]

By induction and transitivity of ≤ , the max-heap property guarantees that the maximum element of a max-heap is at the root. A similar argument for min-heaps holds.

Hence, a heap is a tree in a heap data structure with each node containing a key such that the key in any child node is ≤ the key in its parent node.

MAX-HEAPIFY is a subroutine for manipulating a heap data structures. It is given an array A and an index i into the array. The subtrees rooted at the children of A[i] are heaps but the node A[i] itself may possibly violates the heap property (A[i] might be smaller than A[2i] or A[2i+1]). MAX-HEAPIFY manipulates the tree rooted at A[i] so it becomes a heap.

Here is how it works: It picks the largest child key, A[2i] or A[2i+1], and compares it to the parent key, A[i]. If A[i] ≥ (the largest child key) then MAX-HEAPIFY quits; otherwise it swaps the parent key with the largest child key so the parent is now ≥ its children. The swap may destroy the heap property of the sub-tree rooted at the largest child node so MAX-HEAPIFY calls itself using the largest child node as the new root.

MAX-HEAPFIY is important for manipulating max-heaps. It is used to maintain the max-heap property.

- Before MAX-HEAPIFY, A[i] may be smaller than its children.
- Assume left and right subtrees of i are max-heaps.
- After MAX-HEAPIFY, sub-tree rooted at i is a max-heap.

```
MAX-HEAPIFY(A, i)
    l = LEFT(i)
    r = RIGHT(i)
    if l ≤ A.heap-size and A[l] > A[i]
        largest = l
    else largest = i
    if r ≤ A.heap-size and A[r] > A[largest]
        largest = r
    if largest ≠ i
        exchange A[i] with A[largest]
        MAX-HEAPIFY(A,largest)
```

At worst MAX-HEAPIFY calls itself h times where h is the height of the tree rooted at i. The worst-case running time is $\theta(\lg n)$ where n is the number of nodes in the tree rooted at i.

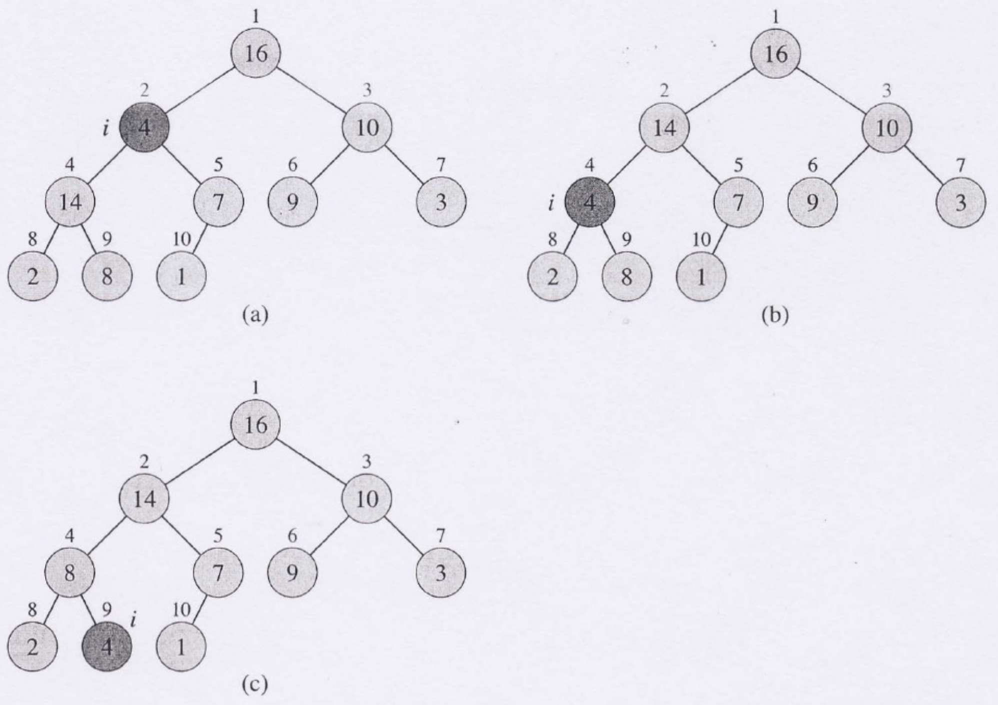

Figure 6.2 The action of MAX-HEAPIFY$(A, 2)$, where $A.heap\text{-}size = 10$. **(a)** The initial configuration, with $A[2]$ at node $i = 2$ violating the max-heap property since it is not larger than both children. The max-heap property is restored for node 2 in **(b)** by exchanging $A[2]$ with $A[4]$, which destroys the max-heap property for node 4. The recursive call MAX-HEAPIFY$(A, 4)$ now has $i = 4$. After swapping $A[4]$ with $A[9]$, 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.

Given an array A[1..n] of n un-ordered keys how can we re-order the keys so that A[1..n] is a heap? Call MAX-HEAPIFY repeatedly to first build a little heaps at the lowest level and then larger and larger heaps at the higher levels until the final call with A[1] as a root builds a heap in the whole array. The order is important; when MAX-HEAPIFY is called with root A[i]; previous calls have already built heaps in the two sub-trees rooted at A[2i] and A[2i+1]. The first call to MAX-HEAPIFY should be at the parent to A[n]; i.e., with the root A[floor(n/2)].

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

The running time of BUILD-MAX-HEAP is $\theta(n)$ which is a linear time.

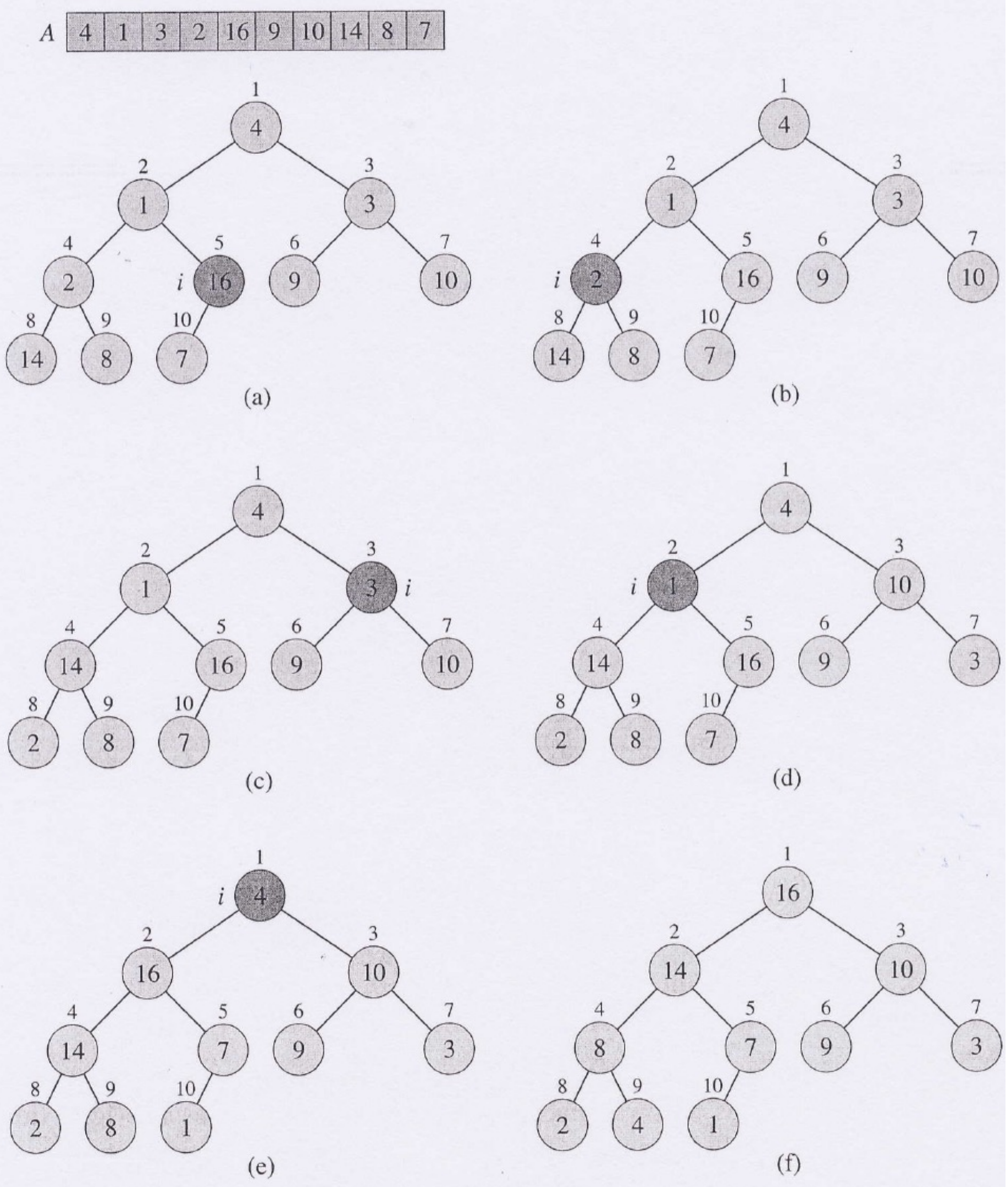

Figure 6.3 The operation of BUILD-MAX-HEAP, showing the data structure before the call t MAX-HEAPIFY in line 3 of BUILD-MAX-HEAP. **(a)** A 10-element input array A and the binary tree it represents. The figure shows that the loop index i refers to node 5 before the call MAX-HEAPIFY(A, i). **(b)** The data structure that results. The loop index i for the next iteratio refers to node 4. **(c)–(e)** Subsequent iterations of the **for** loop in BUILD-MAX-HEAP. Observe tha 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.

The Heapsort Algorithm: Heapsort has two parts: First it uses BUILD-MAX-HEAP to change the un-ordered array A[1..n] into a heap and then it changes the heap into a sorted array.

When the Max-heap is built the largest key will be in its root, A[1]. The largest key should be in A[n] of the sorted array so Heapsort swaps A[1] with A[n] and rearranges the n-1 keys in A[1..n-1] into a heap. The largest of the n-1 keys will now be in A[1] so Heapsort swaps A[1] with A[n-1] and rearranges the n-2 keys in A[1..n-2] into a heap, and so on until the array is sorted.
How does Heapsort rearrange the remaining keys into a heap after each swap? Only the key in A[1] can violate the heap property so it calls MAX-HEAPIFY. Heapsort must change heap-size[A] each time it calls MAX-HEAPIFY so MAX-HEAPIFY doesn't affect the sorted part of the array.

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

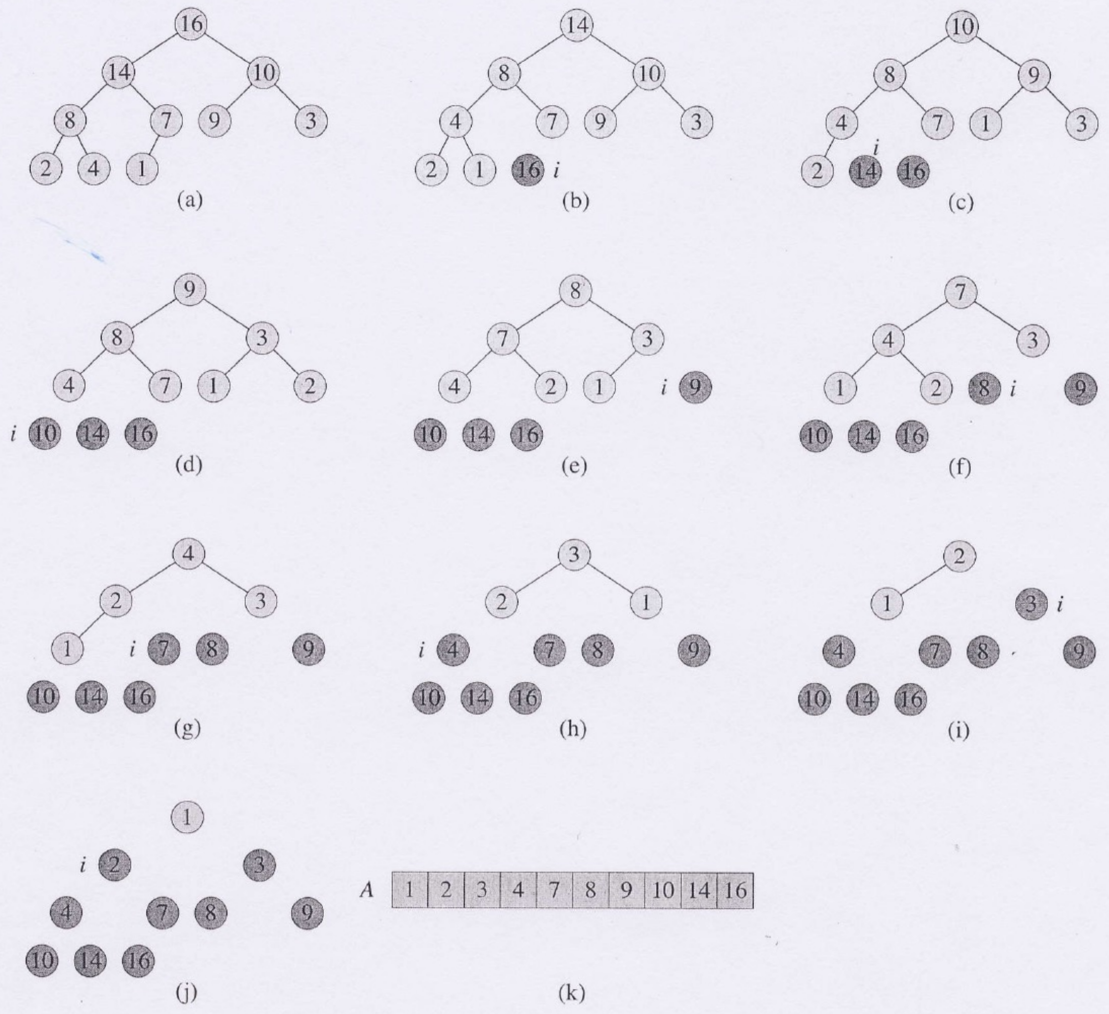

Figure 6.4 The operation of HEAPSORT. **(a)** The max-heap data structure just after BUILD-MAX-HEAP has built it in line 1. **(b)–(j)** The max-heap just after each call of MAX-HEAPIFY in line 5, showing the value of i at that time. Only lightly shaded nodes remain in the heap. **(k)** The resulting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

Analysis

- BUILD-MAX-HEAP: $\theta(n)$
- **for** loop: n-1 times
- exchange elements: $\theta(1)$
- MAX-HEAPIFY: $\theta(\lg n)$

Hence, total time is $\theta(n \lg n)$

Priority Queues: Heaps are also useful for priority queues such as a prioritized job queue in a shared computer. When a job with a high priority enters the queue it should be placed before all jobs with lower priorities. Whenever the computer is free it should pick the highest priority from the queue.

A priority queue of n jobs is stored in an array A[1..n] as a heap.

The job with the highest priority is in A[1] and can be pulled out of the queue in O(lg n) time by moving A[n] to A[1] and calling MAX-HEAPIFY to fix up the heap of the remaining n-1 jobs (just like the body of the for-loop in HEAPSORT).

How can we enter a new job into the priority queue?
HEAP-INSERT does that in O(lg n) time where n is the number of jobs in the queue. A new leaf is added to the heap and HEAP-INSERT runs up through the ancestors of the new leaf moving jobs down until it finds the correct place to put the new job.

```
HEAP-MAXIMUM (A)
1       return A[1]
```

```
HEAP-EXTRACT-MAX (A)
1       if A.heap-size < 1
2           error "heap underflow"
3       max = A[1]
4       A[1] = A[A.heap-size]
5       A.heap-size = A.heap-size - 1
6       MAX-HEAPIFY(A, 1)
7       return max
```

```
HEAP-INCREASE-KEY (A, i, key)
1       if key < A[i]
2           error "new key is smaller than current key"
3       A[i] = key
4       while i > 1 and A[PARENT(i)] < A[i]
5           exchange A[i] with A[PARENT(i)]
6           i = PARENT(i)
```

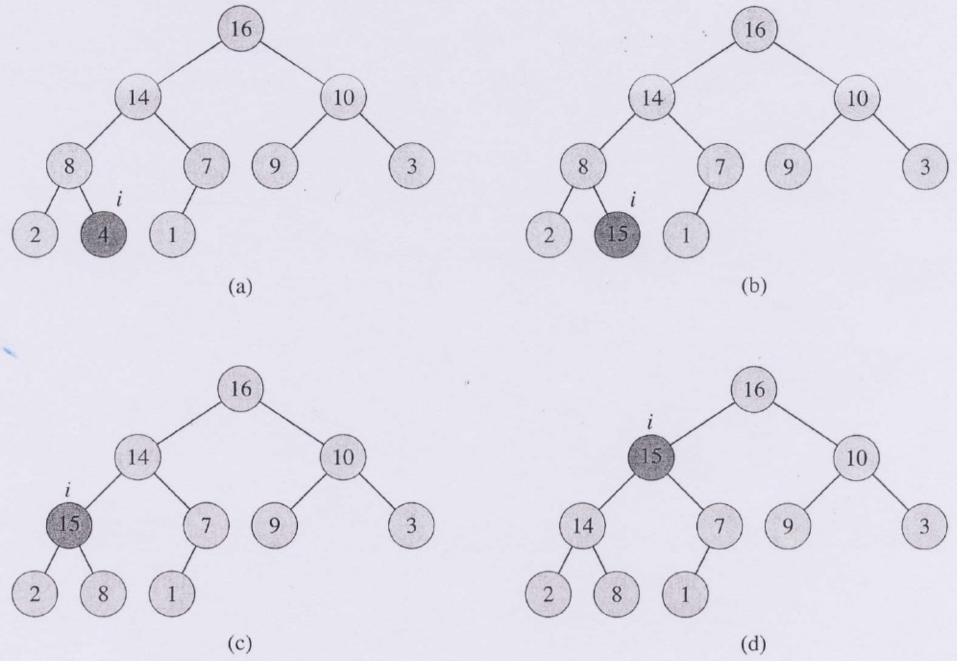

Figure 6.5 The operation of HEAP-INCREASE-KEY. **(a)** The max-heap of Figure 6.4(a) with a node whose index is i heavily shaded. **(b)** This node has its key increased to 15. **(c)** After one iteration of the **while** loop of lines 4–6, the node and its parent have exchanged keys, and the index i 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.

```
MAX-HEAP-INSERT(A, key)
1       A.heap-size = A.heap-size + 1
2       A[A.heap-size] = - ∞
3       HEAP-INCREASE-KEY(A, A.heap-size, key)
```