

The solution to this recurrence, by case 2 of the master theorem (Theorem 4.1), is $T(n) = O(\lg n)$. Alternatively, we can characterize the running time of MAX-HEAPIFY on a node of height h as $O(h)$.

Exercises

6.2-1

Using Figure 6.2 as a model, illustrate the operation of MAX-HEAPIFY($A, 3$) on the array $A = \langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0 \rangle$.

6.2-2

Starting with the procedure MAX-HEAPIFY, write pseudocode for the procedure MIN-HEAPIFY(A, i), which performs the corresponding manipulation on a min-heap. How does the running time of MIN-HEAPIFY compare to that of MAX-HEAPIFY?

6.2-3

What is the effect of calling MAX-HEAPIFY(A, i) when the element $A[i]$ is larger than its children?

6.2-4

What is the effect of calling MAX-HEAPIFY(A, i) for $i > A.\text{heap-size}/2$?

6.2-5

The code for MAX-HEAPIFY is quite efficient in terms of constant factors, except possibly for the recursive call in line 10, which might cause some compilers to produce inefficient code. Write an efficient MAX-HEAPIFY that uses an iterative control construct (a loop) instead of recursion.

6.2-6

Show that the worst-case running time of MAX-HEAPIFY on a heap of size n is $\Omega(\lg n)$. (*Hint:* For a heap with n nodes, give node values that cause MAX-HEAPIFY to be called recursively at every node on a simple path from the root down to a leaf.)

6.3 Building a heap

We can use the procedure MAX-HEAPIFY in a bottom-up manner to convert an array $A[1..n]$, where $n = A.\text{length}$, into a max-heap. By Exercise 6.1-7, the elements in the subarray $A[(\lfloor n/2 \rfloor + 1) .. n]$ are all leaves of the tree, and so each is

a 1-element heap to begin with. The procedure BUILD-MAX-HEAP goes through the remaining nodes of the tree and runs MAX-HEAPIFY on each one.

```

BUILD-MAX-HEAP( $A$ )
1   $A.heap\text{-}size = A.length$ 
2  for  $i = \lfloor A.length/2 \rfloor$  downto 1
3      MAX-HEAPIFY( $A, i$ )

```

Figure 6.3 shows an example of the action of BUILD-MAX-HEAP.

To show why BUILD-MAX-HEAP works correctly, we use the following loop invariant:

At the start of each iteration of the **for** loop of lines 2–3, each node $i + 1, i + 2, \dots, n$ is the root of a max-heap.

We need to show that this invariant is true prior to the first loop iteration, that each iteration of the loop maintains the invariant, and that the invariant provides a useful property to show correctness when the loop terminates.

Initialization: Prior to the first iteration of the loop, $i = \lfloor n/2 \rfloor$. Each node $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \dots, n$ is a leaf and is thus the root of a trivial max-heap.

Maintenance: To see that each iteration maintains the loop invariant, observe that the children of node i are numbered higher than i . By the loop invariant, therefore, they are both roots of max-heaps. This is precisely the condition required for the call MAX-HEAPIFY(A, i) to make node i a max-heap root. Moreover, the MAX-HEAPIFY call preserves the property that nodes $i + 1, i + 2, \dots, n$ are all roots of max-heaps. Decrementing i in the **for** loop update reestablishes the loop invariant for the next iteration.

Termination: At termination, $i = 0$. By the loop invariant, each node $1, 2, \dots, n$ is the root of a max-heap. In particular, node 1 is.

We can compute a simple upper bound on the running time of BUILD-MAX-HEAP as follows. Each call to MAX-HEAPIFY costs $O(\lg n)$ time, and BUILD-MAX-HEAP makes $O(n)$ such calls. Thus, the running time is $O(n \lg n)$. This upper bound, though correct, is not asymptotically tight.

We can derive a tighter bound by observing that the time for MAX-HEAPIFY to run at a node varies with the height of the node in the tree, and the heights of most nodes are small. Our tighter analysis relies on the properties that an n -element heap has height $\lceil \lg n \rceil$ (see Exercise 6.1-2) and at most $\lceil n/2^{h+1} \rceil$ nodes of any height h (see Exercise 6.3-3).

The time required by MAX-HEAPIFY when called on a node of height h is $O(h)$, and so we can express the total cost of BUILD-MAX-HEAP as being bounded from above by

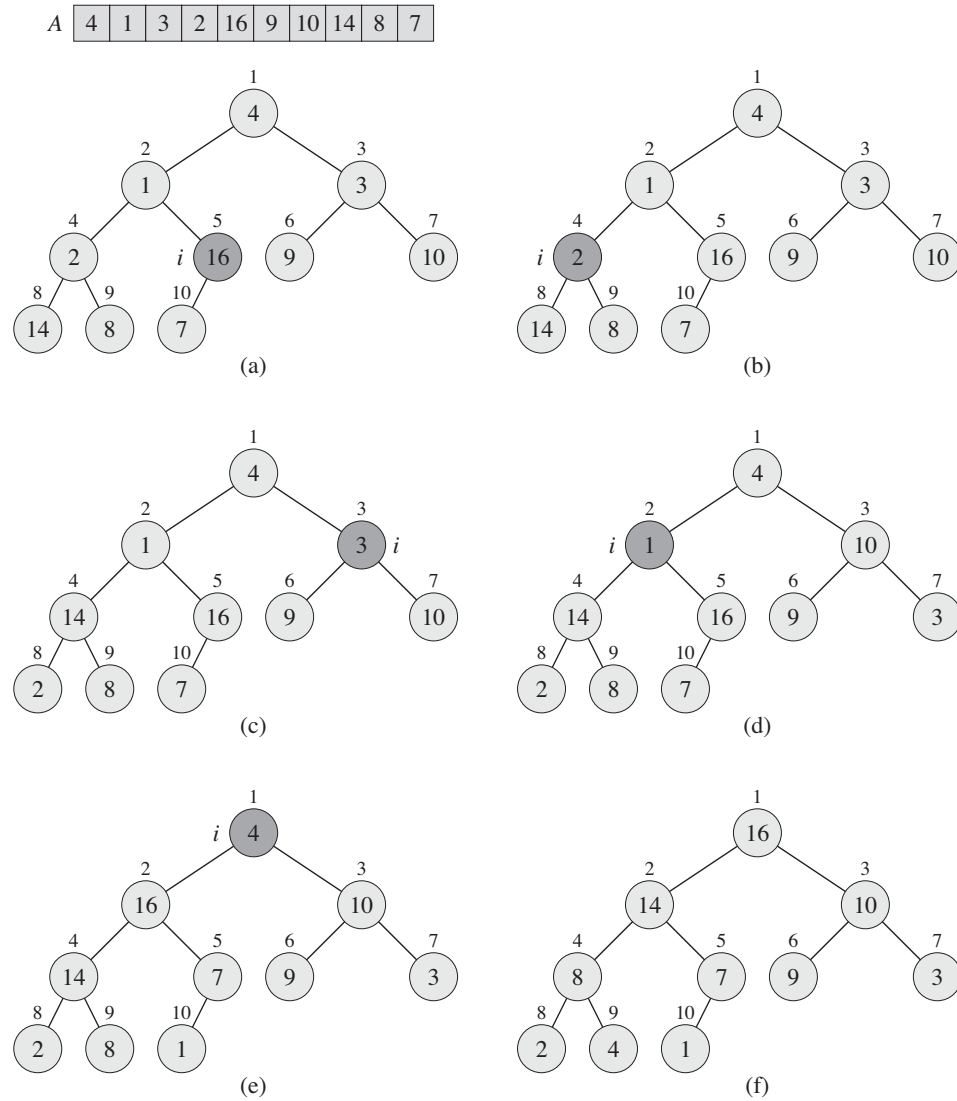


Figure 6.3 The operation of BUILD-MAX-HEAP, showing the data structure before the call to MAX-HEAPIFY in line 3 of BUILD-MAX-HEAP. (a) A 10-element input array A and the binary tree it represents. The figure shows that the loop index i refers to node 5 before the call MAX-HEAPIFY(A, i). (b) The data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

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right).$$

We evaluate the last summation by substituting $x = 1/2$ in the formula (A.8), yielding

$$\begin{aligned} \sum_{h=0}^{\infty} \frac{h}{2^h} &= \frac{1/2}{(1 - 1/2)^2} \\ &= 2. \end{aligned}$$

Thus, we can bound the running time of BUILD-MAX-HEAP as

$$\begin{aligned} O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right) &= O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) \\ &= O(n). \end{aligned}$$

Hence, we can build a max-heap from an unordered array in linear time.

We can build a min-heap by the procedure BUILD-MIN-HEAP, which is the same as BUILD-MAX-HEAP but with the call to MAX-HEAPIFY in line 3 replaced by a call to MIN-HEAPIFY (see Exercise 6.2-2). BUILD-MIN-HEAP produces a min-heap from an unordered linear array in linear time.

Exercises

6.3-1

Using Figure 6.3 as a model, illustrate the operation of BUILD-MAX-HEAP on the array $A = \langle 5, 3, 17, 10, 84, 19, 6, 22, 9 \rangle$.

6.3-2

Why do we want the loop index i in line 2 of BUILD-MAX-HEAP to decrease from $\lfloor A.length/2 \rfloor$ to 1 rather than increase from 1 to $\lfloor A.length/2 \rfloor$?

6.3-3

Show that there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h in any n -element heap.

6.4 The heapsort algorithm

The heapsort algorithm starts by using BUILD-MAX-HEAP to build a max-heap on the input array $A[1..n]$, where $n = A.length$. Since the maximum element of the array is stored at the root $A[1]$, we can put it into its correct final position