

Solution to Chapter 6

By Y. MENG gmail: y(dot)meng201011

Section 1

exe. 6.1.1 What are the minimum and maximum numbers of elements in a heap of height h ?

◇ A complete binary tree.

$$\text{maximum} = \sum_{i=0}^h 2^i = 2^{h+1} - 1$$

◇ If there is only one node in the lowest level, then all the internal nodes form a complete binary tree with height $(h-1)$.

$$\text{num_internal} = 2^h - 1$$

$$\text{minimum} = \text{num_internal} + 1 = 2^h - 1 + 1 = 2^h$$

exe. 6.1.2 Show that an n -element heap has height $\lfloor \lg n \rfloor$.

Number of nodes of level i in a complete binary tree is $2^i - 1$. Number of nodes from root to level i in a complete binary tree has is $\text{number} = \sum_{k=0}^i 2^k = 2^{i+1} - 1$. Thus, for a complete binary tree with height h , number of nodes is $\text{number} = \sum_{k=0}^h 2^k = 2^{h+1} - 1$. That is, $h = \lg(n+1) - 1$. This is the case that the most nodes a binary tree with height h can have.

For another case that this n -element heap builds an almost-complete binary tree, there is only one leaf at the lowest level. Then number of nodes in this tree excepted the one in the lowest level is $\text{number}' = \sum_{k=0}^{h-1} (2^k) = 2^h - 1$. Then the total number is

$$\text{number} = 2^h - 1 + 1 = 2^h. \text{ That is, } h = \lg(n+1).$$

Thus, $\lg(n) \leq h \leq \lg(n+1) - 1 < \lg(n+1)$. i.e., $h = \lfloor \lg n \rfloor$, since h is an integer.

exe. 6.1.3 Show that in any subtree of a max-heap, the root of the subtree contains the largest value occurring anywhere in that subtree.

This follows the max-heap property.

Subtree rooted by the i th element, according to the max-heap property, both its left-child and right-child are no greater than the element. And since this subtree is also a max-heap, values of all children nodes are smaller than or equal to their parents. And thus, the maximum of this subtree is the value of the i th element.

Following solution is from "Algs. Instructor's Manual".

Assume the claim is FALSE -- i.e., that there is a subtree whose root is not the largest element in the subtree. Then the maximum element is somewhere else in the subtree, possibly even at more than one location. Let m be the index of the maximum (the lowest such index if the maximum appears more than once). Since the maximum is not at the root of the subtree, node m has a parent. And since the parent of a node has a lower index than the node, $A[\text{PARENT}(m)] < A[m]$ (m is the smallest of indices of the maximum). This conflicts to max-heap property that $A[\text{PARENT}(m)] \geq A[m]$. Thus, the assumption is FALSE, which means that the claim is TRUE.

exe. 6.1.4 Where in a max-heap might the smallest element reside, assuming that all elements are distinct?

The smallest element can be any leaf, i.e., it is in subarray $A[\lfloor \frac{n}{2} \rfloor + 1 \dots n]$.

The number of possible elements are $\lfloor \frac{n}{2} \rfloor$ (number of internal nodes $= \sum_{k=0}^{h-1} 2^k = 2^h =$ number of nodes in level h). The running time to find this smallest element is $O(\lg n)$.

exe. 6.1.5 Is an array that is in sorted order a min-heap?

Yes. Sorted array is a min-heap.

◇ The smallest element is $A[1]$, which is also the root of the heap.

◇ For a certain element $A[i]$, elements $A[i * 2]$ and $A[i * 2 + 1]$ are both smaller than or equal to $A[i]$; and $\text{node}(i * 2)$ and $\text{node}(i * 2 + 1)$ are left-child and right-child of $\text{node}(i)$ in the heap, which, according to the min-heap property, should be no greater than $\text{node}(i)$.

Thus, a sorted array is a min-heap.

exe.6.1.6 Is the array with values [23, 17, 14, 6, 13, 10, 1, 5, 7, 12] a max-heap?

No, this is NOT a max-heap.

Children (13 and 10) of the 4th element are greater than the element (6).

exe. 6.1.7 Show that, with the array representation for storing an n-element heap, the leaves are the nodes indexed by $\lfloor \frac{n}{2} \rfloor + 1$, $\lfloor \frac{n}{2} \rfloor + 2$, ..., n .

- ◊ An n-element heap has at least (2^{h-1}) leaves and at most (2^h) leaves.
- ◊ An n-element heap has at least (2^h) nodes and at most (2^{h+1}) nodes.
- ◊ Number of internal nodes will be in range $((2^h - 2^{h-1} = 2^{h-1}), (2^{h+1} - 2^h = 2^h))$. That is, $(\lfloor \frac{n}{2} \rfloor, \lfloor \frac{n+1}{2} \rfloor)$

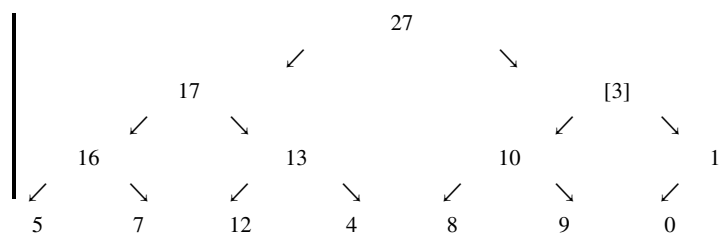
(number of internal nodes = $\sum_{k=0}^{h-1} 2^k = 2^h = \text{number of nodes in level } h$).

Thus, the indices of leaves will be $\lfloor \frac{n}{2} \rfloor + 1$, $\lfloor \frac{n}{2} \rfloor + 2$, ..., n .

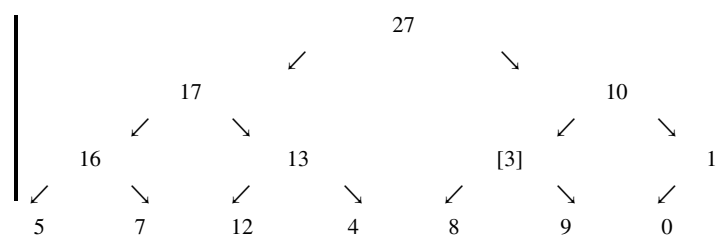
Section 2

exe. 6.2.1 Using Fig.6.2 as a model, illustrate the operation of *MAX-HEAPIFY*(A, 3) on the array A = [27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0].

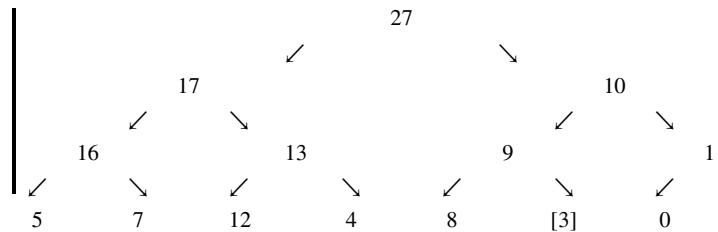
MAX-HEAPIFY(A, 3)



MAX-HEAPIFY(A, 6)



MAX-HEAPIFY(A, 13)



exe.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**?

// Bubbling the *i*th element as high as possible.

MIN-HEAPIFY(A, i)

l = LEFT(*i*)

r = RIGHT(*i*)

if *l* ≤ *A.heap_size* and *A[l*] < *A[i]*

smallest = *l*

else

smallest = *i*

if *r* ≤ *A.heap_size* and *A[r]* < *A[smallest]*

smallest = *r*

if *smallest* ≠ *i*

 SWAP *A[i]* and *A[smallest]*

 MIN-HEAPIFY(*A*, *smallest*)

The running time is same as that of **MAX-HEAPIFY**, i.e., $O(\lg n)$.

exe.6.2.3 What is the effect of calling **MAX-HEAPIFY(A, i)** when the element *A[i]* is larger than its children?

No effect. Since the procedure will terminate directly, no swapping and recursing needed.

exe.6.2.4 What is the effect of calling **MAX-HEAPIFY(A, i)** for $i > \frac{A.heap_size}{2}$?

No effect. In that case, both LEFT(*i*) and RIGHT(*i*) fail the comparison with *A.heap_size* and largest stores *i*, so that, swapping and recursing will not be performed.

exe.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.

MAX-HEAPIFY-ITERATIVE(A, i)

largest = −1

while (*largest* ≠ *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*

 SWAP *A[i]* and *A[largest]*

$i = \text{largest}$
 $\text{largest} = -1$

exe.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.)

Take the leftmost path in given heap, let the smallest element be the root and left-child is larger than right-child for every element, then **MAX-HEAPIFY** will be called for h many times (h is the height), since it is called at each level in order to sink the smallest element to the leftmost leaf. Since $h = \lfloor \lg n \rfloor$, the worst-case running time of the procedure is $\Omega(\lg n)$.

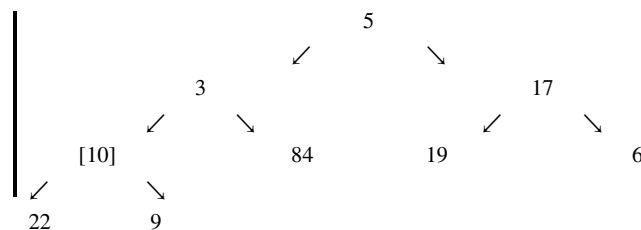
Following solution is from "Algs. Instructor's Manual".

If you put a value at the root that is less than every value in the left and right subtrees, then **MAX-HEAPIFY** will be called recursively until a leaf is reached. To make the recursive calls traverse the longest path to a leaf, choose values that make **MAX-HEAPIFY** always recurse on the left child. It follows the left branch when the left-child \geq right-child, so putting 0 at the root and 1 at all the other nodes, for example, will accomplish that. With such values, **MAX-HEAPIFY** will be called h times (where h is the height of heap, which is the number of edges in the longest path from the root to a leaf), so its running time will be $\Theta(h)$ (since each call does $\Theta(1)$ work), which is $\Theta(\lg n)$. Since we have a case in which **MAX-HEAPIFY**'s running time is $\Theta(\lg n)$, its worst-case running time is $\Omega(\lg n)$.

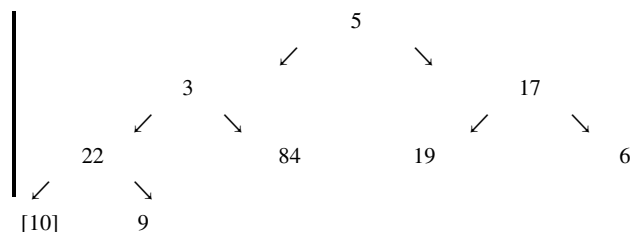
Section 3

exe. 6.3.1 Using Fig. 6.3 as a model, illustrate the operation of **BUILD-MAX-HEAP** on the array $A = [5, 3, 17, 10, 84, 19, 6, 22, 9]$.

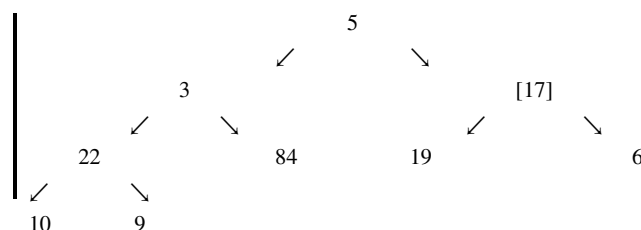
BUILD-MAX-HEAP(A, 4), MAX-HEAPIFY(A, 4)



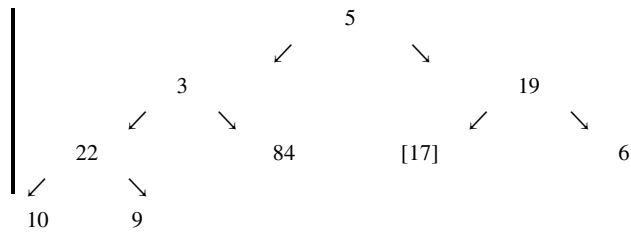
MAX-HEAPIFY(A, 8)



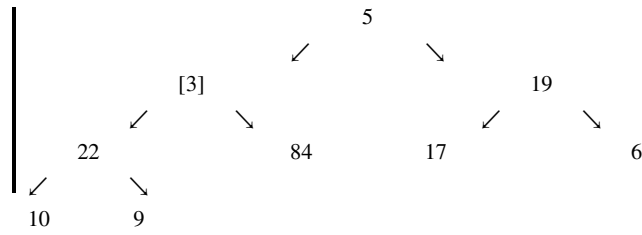
BUILD-MAX-HEAP(A, 3), MAX-HEAPIFY(A, 3)



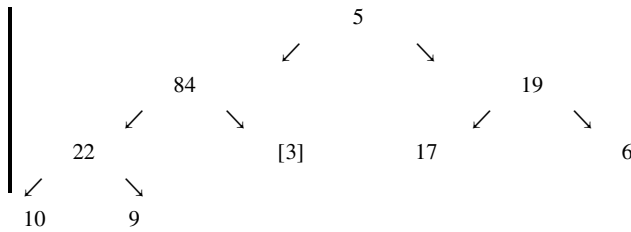
MAX-HEAPIFY(A, 6)



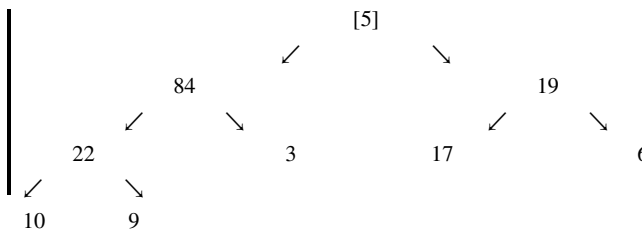
BUILD-MAX-HEAP(A, 2), MAX-HEAPIFY(A, 2)



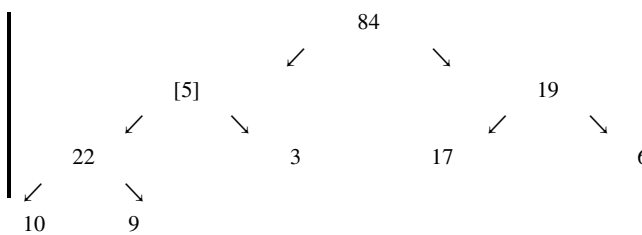
MAX-HEAPIFY(A, 5)



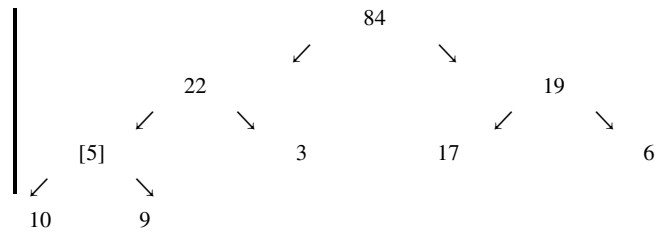
BUILD-MAX-HEAP(A, 1), MAX-HEAPIFY(A, 1)



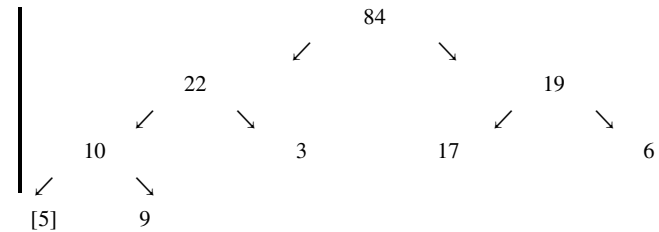
MAX-HEAPIFY(A, 2)



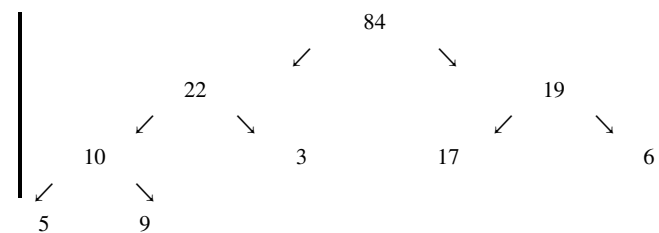
MAX-HEAPIFY(A, 4)



MAX-HEAPIFY(A, 8)



Terminate (i = largest = 1)



exe. 6.3.2 Why do we want the loop index i in line 2 of *BUILD-MAX-HEAP* to decrease from $\lfloor \frac{A.length}{2} \rfloor$ to 1 rather than increase from 1 to $\lfloor \frac{A.length}{2} \rfloor$?

It cannot guarantee that the maximum is moved to the root when the maximum is not at level 1 at beginning. Loop index i from 1 to $\lfloor \frac{A.length}{2} \rfloor$, it can only move the current largest element up to at most level $\lceil \lg i \rceil$.

exe. 6.3.3 Show that there are at most $\lceil \frac{n}{2^{h+1}} \rceil$ nodes of height h in any n -element heap.

My solution (more mathematical)

Base Case:

A max-heap can have at most $\frac{n+1}{2}$ nodes at the lowest level (when the heap is a complete binary tree), and

$$x = \frac{n+1}{2} \leq \lceil \frac{n}{2} \rceil = \lceil \frac{n}{2^1} \rceil = \lceil \frac{n}{2^0+1} \rceil = \lceil \frac{n}{2^{h+1}} \rceil.$$

For non-complete binary trees, number of nodes is less than $\lceil \frac{n}{2} \rceil$. Thus, the base case holds.

Induction:

Assume that it holds for nodes of height $(h - 1)$. Then take a tree, remove all the nodes from the lowest level and get a new tree (which is a complete binary tree) T' with height $h' = h - 1$ and length $n' = n - \lceil \frac{n}{2} \rceil = \lfloor \frac{n}{2} \rfloor$, and the number of nodes at the lowest level of T' is

$$\frac{n'+1}{2} \leq \lceil \frac{n'}{2^{h'+1}} \rceil = \lceil \frac{\lfloor n/2 \rfloor}{2^{h-1+1}} \rceil \leq \lceil \frac{(n/2)}{2^h} \rceil = \lceil \frac{n}{2^{h+1}} \rceil$$

Repeat above process through all level of the tree, and keep in mind that height decrease each round.

KEY OF SOLUTION: height of node is the distance of the node to the lowest leaf!

Following solution is from "Algs. Instructor's Manual". (more logical)

Proof by induction on h .

Basis:

Show that it's TRUE for $h = 0$ (i.e., that #ofleaves $\leq \lceil \frac{n}{2^{h+1}} \rceil = \lceil \frac{n}{2} \rceil$).

The tree leaves (nodes at height 0) are at depths H and $(H - 1)$. They consist of

- ◇ all nodes at depth H , and
- ◇ the nodes at depth $(H + 1)$ that are not parents of depth- H nodes.

Let x be the number of nodes at depth H -- that is, the number of nodes in the bottom (possibly incomplete) level.

Note that $(n - x)$ is odd, because the $(n - x)$ nodes above the bottom level form a complete binary tree, and a complete binary tree has an odd number of nodes (1 less than a power of 2). Thus if n is odd, x is even, otherwise, x is odd.

Now, proof the base case separately when n is even and odd.

n is ODD

◇ If n is odd, then x is even, so all nodes have siblings -- i.e., all internal nodes have 2 children. Thus

$\text{num_internal_nodes} = \text{num_of_leaves} - 1$. So, $n = \text{num_of_leaves} + \text{num_of_internal_nodes} = 2 * \text{num_of_leaves} - 1$. Thus,

$\text{num_of_leaves} = \frac{n+1}{2} = \lceil \frac{n}{2} \rceil$ (The latter equality holds because n is odd).

◇ If n is even, then x is odd, and one leaf does not have sibling. If we gave it a sibling, then we would have $(n + 1)$ nodes, where $x' = (n + 1)$ is odd, so this case has been proved. Observe that we would also increase the number of leaves by 1, since we added a node to a parent that already had a child. By the odd-node case, $\text{num_of_leaves} + 1 = \lceil \frac{n+1}{2} \rceil = \lceil \frac{n}{2} \rceil + 1$. (The latter equality holds since n is even.)

Therefore, in either case, $\text{num_of_leaves} = \lceil \frac{n}{2} \rceil$.

Inductive step:

Show that if it's TRUE for height $(h - 1)$, it's TRUE for h .

Let n_h be the number of nodes at height h in the n -node tree T .

Consider the tree T' formed by removing the leaves of T . It has $n' = n - n_0$ nodes. We know from the base case that $n_0 = \lceil \frac{n}{2} \rceil$, so

$n' = n - \lceil \frac{n}{2} \rceil = \lfloor \frac{n}{2} \rfloor$. Note that the nodes at height h in T would be at height $(h - 1)$ if the leaves of the tree were removed -- that is, they are at height $(h - 1)$ in T' . Letting n'_{h-1} denote the number of nodes at height $(h - 1)$ in T' , we have $n_h = n'_{h-1}$.

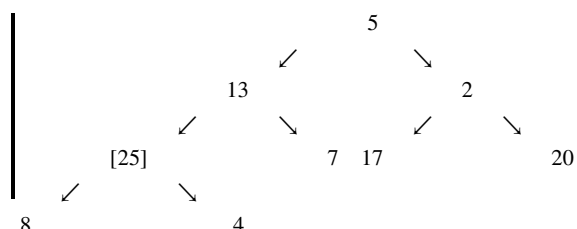
By **inductoin**, we can bound n'_{h-1} :

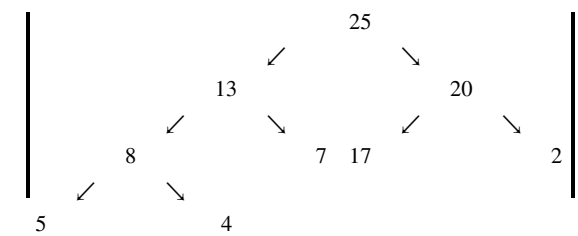
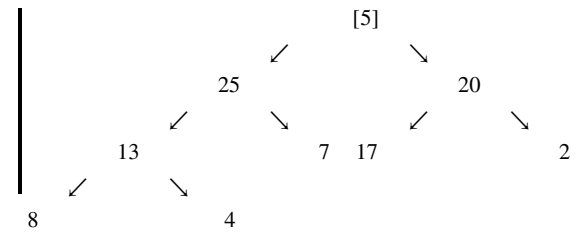
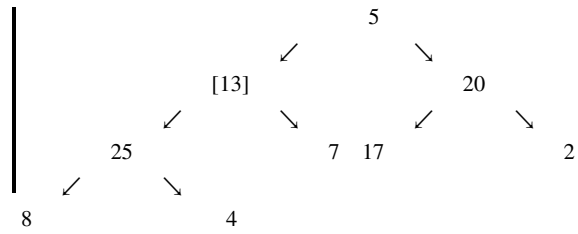
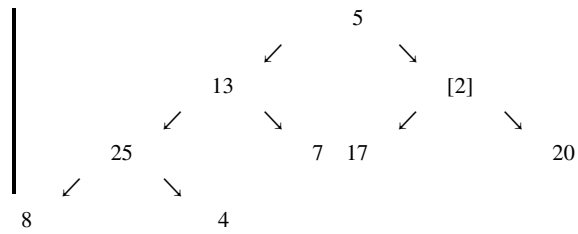
$$n_h = \lceil \frac{n'_{h-1}}{2} \rceil \leq \lceil \frac{\lfloor \frac{n}{2} \rfloor}{2} \rceil \leq \lceil \frac{n/2}{2} \rceil = \lceil \frac{n}{2^{h+1}} \rceil.$$

Section 4

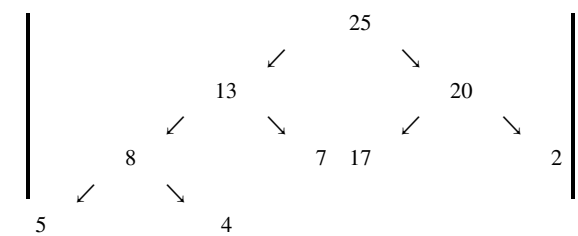
exe. 6.4.1 Using Fig. 6.4 as a model, illustrate the operation of HEAPSORT on the array $A = [5, 13, 2, 25, 7, 17, 20, 8, 4]$.

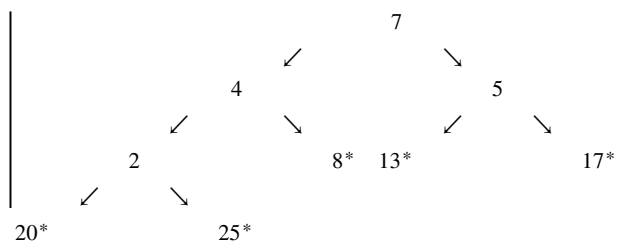
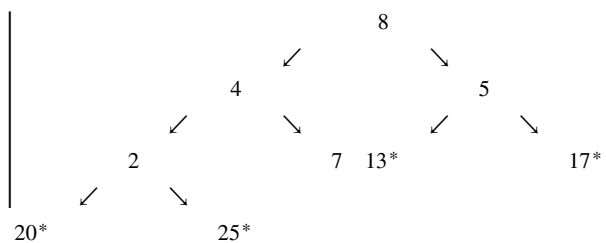
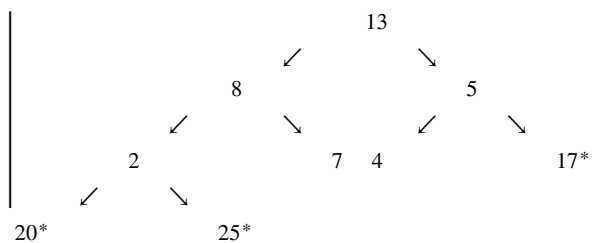
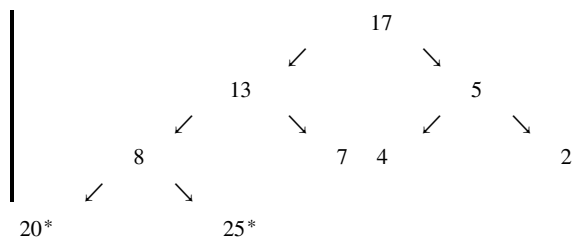
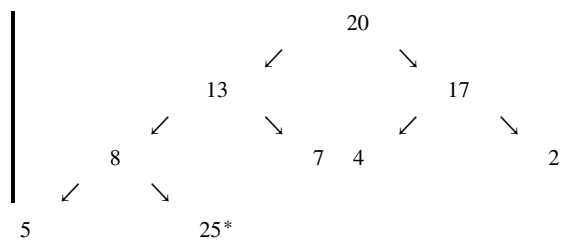
Build the max-heap

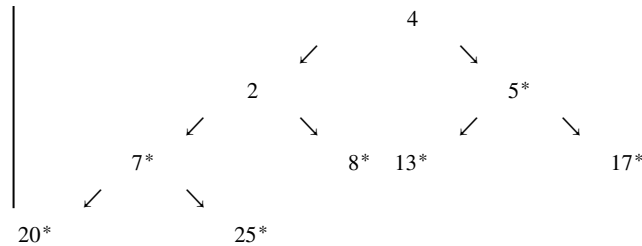
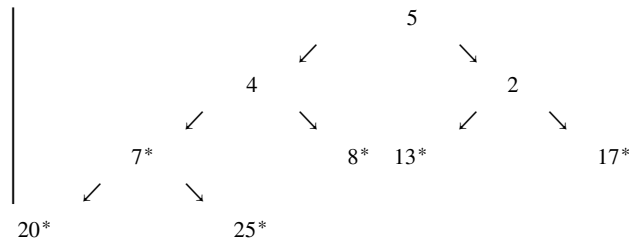




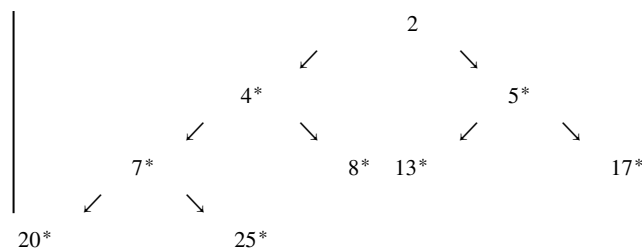
Heapsort







Terminate (only one element in the tree)



exe. 6.4.2 Argue the correctness of HEAPSORT using the following loop invariant:

At the start of each iteration of the **for** loop of lines 2-5, the subarray $A[1..i]$ is a max-heap containing the i smallest elements of $A[1..n]$, and the subarray $A[i+1..n]$ contains the $n-i$ largest elements of $A[1..n]$, sorted.

exe. 6.4.3 What is the running time of HEAPSORT on an array A of length n that is already sorted in increasing order? What about decreasing order?

exe. 6.4.4 Show that the worst-case running time of HEAPSORT is $\Omega(n \lg n)$.

exe. 6.4.5* Show that when all elements are distinct, the best-case running time of HEAPSORT is $\Omega(n \lg n)$.

In []: