Building a heap:

- We can use the procedure MAX-HEAPIFY in a bottom-up manner to convert an array A[1 . . n], where n = length[A], into a max-heap.
- The elements in the subarray A [($\left\lfloor \frac{n}{2} \right\rfloor + 1$). . 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   heap-size[A] ← length[A]
2   for i ← ⌊length[A]/2⌋ downto 1
3         do MAX-HEAPIFY(A, i)
```

- 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 $i + 1$, $i + 2$, . . . , n is the root of a max-heap."

➢ **Initialization:** Prior to the first iteration of the loop, $i = \left\lfloor \frac{n}{2} \right\rfloor$.

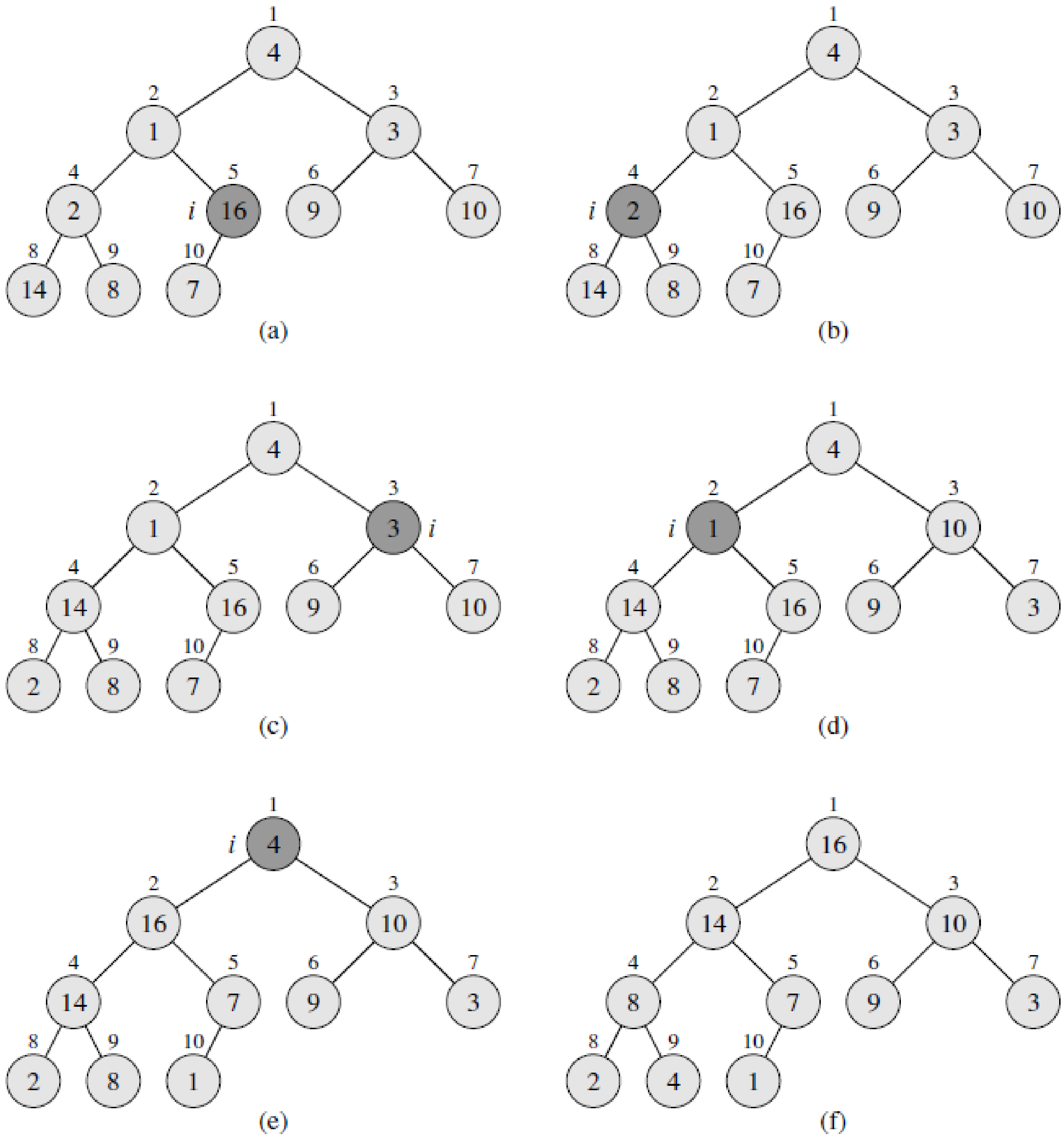
Each node $\left\lfloor \frac{n}{2} \right\rfloor + 1$, $\left\lfloor \frac{n}{2} \right\rfloor + 2$, . . . , n is a leaf and is thus the root of a trivial max-heap.

➢ **Maintenance:**

- 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, . . . , 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, . . . , n is the root of a max-heap. In particular, node 1 is.

Eg: A= 4, 1, 3, 2 ,16, 9, 10, 14, 8, 7



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.

Running time of BUILD-MAXHEAP:

(1) Simple Upper Bound:

Each call to MAX-HEAPIFY costs O(lg n) time, and there are O(n) such calls. Thus, the running time is O(n lg n). This upper bound, though correct, is not asymptotically tight.

(2) Tighter Bound Analysis:

- Observe following two things:

  - the time for MAX-HEAPIFY to run at a node varies with the height of the node in the tree.

  - the heights of most nodes are small.

- Further, we know the following two facts:
  - that an n-element heap has height $\lfloor \lg n \rfloor$.
  - There are at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes of any height h

- The height 'h' increases as we move upwards along the tree.
- Let the time required by MAX-HEAPIFY when called on a node of height *h* is *O(h)*, so we can express the total cost of BUILD-MAX-HEAP as follows:

$$\sum_{from\ height\ 0\ to\ \lfloor \lg n \rfloor} (no.\ of\ nodes\ at\ height\ \ h\ ) * (Running\ time\ for\ each\ node)$$

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

$$\Rightarrow \; O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right)$$

$$\Rightarrow \; O(n.2\ )$$

$$\Rightarrow \; O(n\ )$$

Heap sort:

The heapsort algorithm starts by using BUILD-MAX-HEAP to build a max-heap on the input array A[1 . . n], where n = length[A].

Since the maximum element of the array is stored at the root A[1], it can be put into its correct final position by exchanging it with A[n].

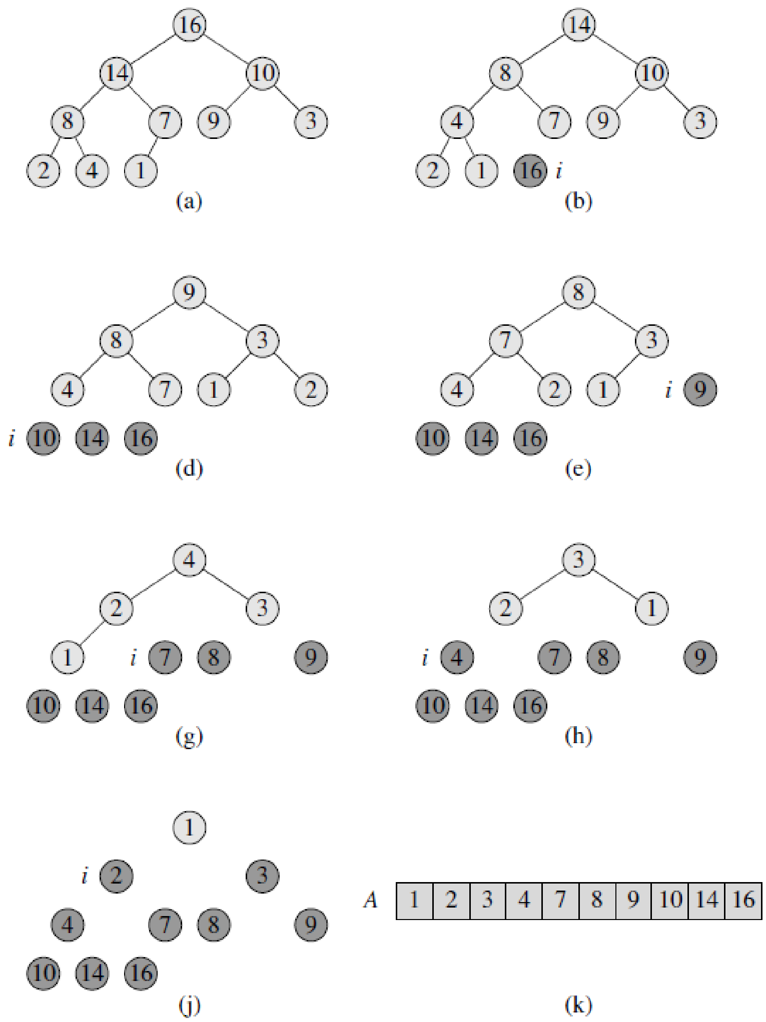
If we now "discard" node n from the heap (by decrementing heap-size[A]), we observe that A[1 . . (n − 1)] can easily be made into a max-heap. The children of the root remain max-heaps, but the new root element may violate the max-heap property. All that is needed to restore the max-heap property, however, is one call to MAX-HEAPIFY(A, 1), which leaves a max-heap in A[1 . . (n − 1)].

The heapsort algorithm then repeats this process for the maxheap of size n − 1 down to a heap of size 2.

```
HEAPSORT(A)
BUILD-MAX-HEAP(A)
for i ← length[A] downto 2
    do exchange A[1] ↔ A[i]
       heap-size[A] ← heap-size[A] − 1
       MAX-HEAPIFY(A, 1)
```

Running time of HEAPSORT:

Eg:

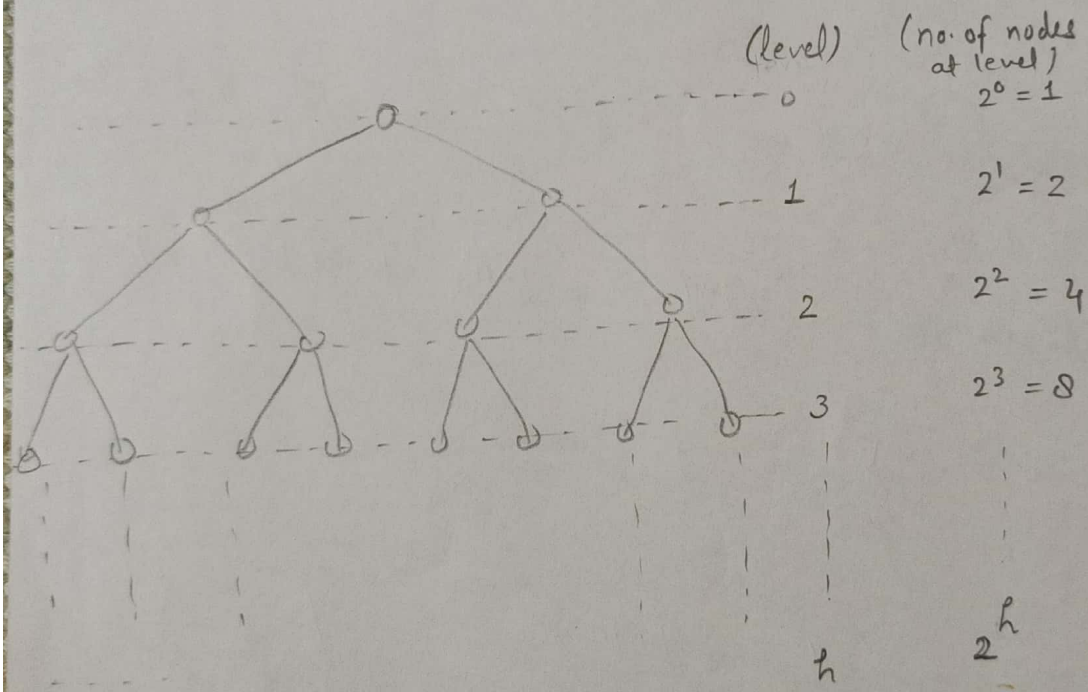


(a) The max-heap data structure just after it has been built by BUILD-MAX-HEAP.
(b)–(j) The max-heap just after each call of MAX-HEAPIFY in line 5. The value of i at that time is shown. Only lightly shaded nodes remain in the heap.
(k) The resulting sorted array A.

Build heap Analysis

For simplicity, let us assume $n = 2^{h+1} - 1$, where

h = height of tree [Here, bottommost level is full, and this assumption will save us from worrying about floor & ceilings)



Remember that when heapify is called, the running time depends on how far an element might shift down before process terminates. In worst case, the element might shift down all the way to the leaf-level.

Now, we will calculate cost level by level:

At bottommost level there are 2^h nodes, but we don't call heapify on any of these nodes so total cost $= 0$

At the next to bottommost level, no. of nodes $= 2^{h-1}$, each might be shifted down 1 level

At the 2^{nd} level, from the bottom, no of nodes $= 2^{h-2}$, each might be shifted down 2 levels

!

In general, j^{th} level from the bottom $= 2^{h-j}$, each node might be shifted down j levels

Therefore, total cost can be counted from bottom to top, as follows:

$$T(n) = \sum_{j=0}^{h} 2^{h-j} \cdot j = 2^h \sum_{j=0}^{h} \frac{j}{2^j}$$

Here we have bounded sum, but the infinite series is bounded so we can use it for easy approximation.

$$\left\{ \begin{array}{l} \sum_{k=0}^{\infty} \frac{k}{2^k} = 2 \\ \text{as} \\ \sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2} \\ \text{put } x = 1/2 \end{array} \right.$$

$$\begin{aligned} T(n) &= 2^h \sum_{j=0}^{h} \frac{j}{2^j} \\ &\leq 2^h \cdot 2 \\ &\leq 2^{h+1} \\ &\leq n+1 \\ &\in O(n) \end{aligned}$$

$\boxed{\text{This is the worst case analysis, but it should be noted that algorithm takes at least } \Omega(n) \text{ time since it must access every element of array at least once.}}$

*

Here, It should be noted that a relatively complex structured algorithm, with doubly nested loops, has running time $\Theta(n)$.

**

It is important to observe an important fact about binary tree:

" The vast majority of nodes are at lowest level of the tree "

cf In a complete binary tree of height h

total no of nodes $= 2^{h+1} - 1$ (maximum possible)

$\leq 2^{h+1} - 1$

$n \simeq 2^{h+1}$

so no. of nodes at bottom level $= 2^h$

—— " —— next to bottom " $= 2^{h-1}$

—— " —— 2nd from bottom " $= 2^{h-2}$

$\Rightarrow 2^h + 2^{h-1} + 2^{h-2}$

$\simeq \frac{n}{2} + \frac{n}{4} + \frac{n}{8}$

$\Rightarrow \frac{7n}{8} = 0.875n$

That is, almost 90% of nodes of a complete binary tree reside in the 3 lowest levels.

Thus, we can learn one lesson we can learn from this analysis :-

" when designing algorithms that operate on trees, it is important to be most efficient on the bottommost levels of tree; since that is where most of weight of the tree resides.

Few Results on Heap

1) Max. & Min number of elements in a heap of height 'h'

$$2^h \le n \le (2^{h+1} - 1)$$

2) An n-element heap has height $\lfloor \lg n \rfloor$

3) With the array representation for storing an n-element heap, the leaves are indexed by

$$\lfloor \tfrac{n}{2} \rfloor + 1 , \lfloor \tfrac{n}{2} \rfloor + 2 , \ldots n$$

* Number of leaves in any heap of size n is $\lceil \frac{n}{2} \rceil$

⇒ if n is even ⟶ (2^{nd} half of the heap array)

if n is odd ⟶ (2^{nd} half of the heap array plus the middle element)

** The non-leaves nodes are indexed by

$$1, 2, \ldots \lfloor \tfrac{n}{2} \rfloor$$

(4) There are at most $\lceil n/2^{h+1} \rceil$ nodes of height h in any n-element heap.