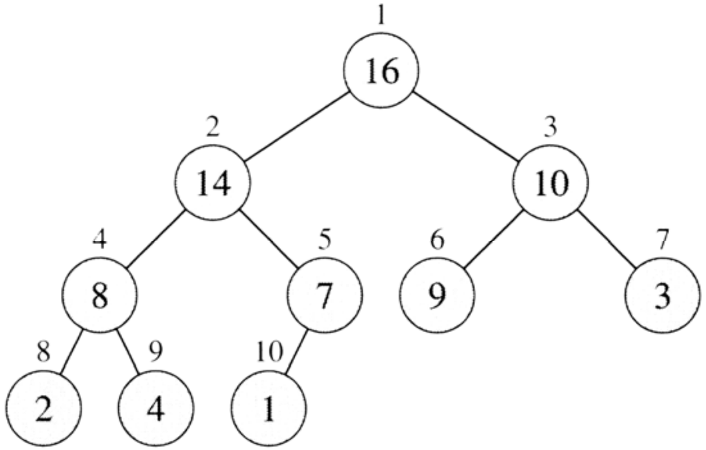
------------------------------ 6 ------------------------------

Heapsort

6.1 ***Heap***: an array that can be viewed as a complete binary tree in which each node has a key not smaller than those of its children (***heap property***).



(a)

1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

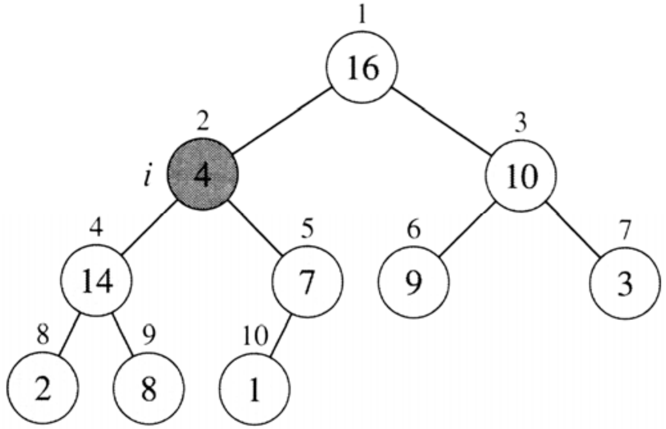
(b)

- *A*[1] is the root
- *Parent*(*i*)=$\lfloor i/2 \rfloor$ (shifting *i* right one bit)
- *Left*(*i*)=2*i* (shifting *i* left one bit)
- *Right*(*i*)=2*i*+1
- A heap of *n* nodes has height Θ(lg *n*)

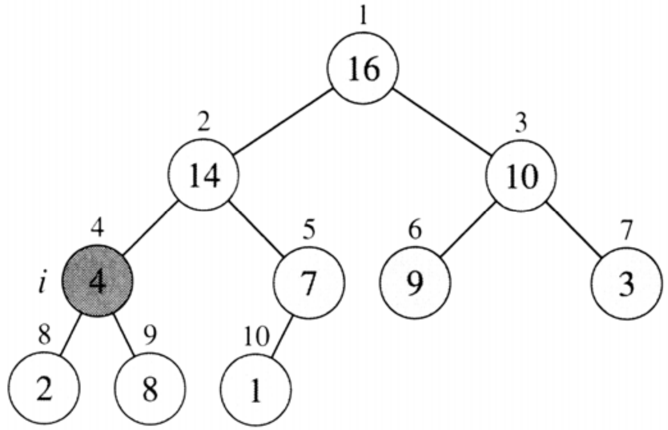
6.2 Maintaining the heap property

***Heapify*(*A*, *i*)**: Binary trees rooted at *Left*(*i*) and *Right*(*i*) are heaps, but *A*[*i*] may be smaller than its children.

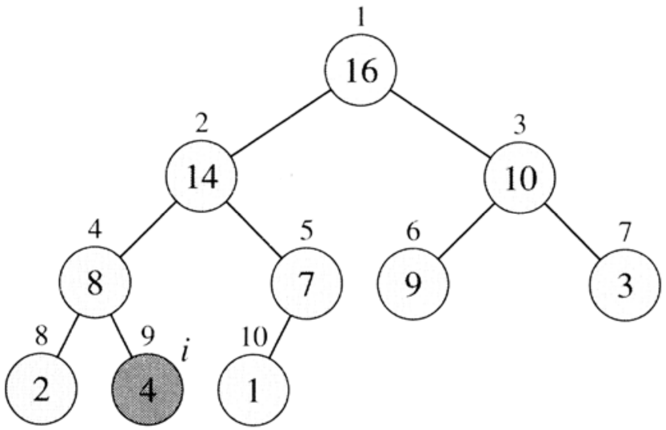
Example:



(a) Heapify(*A*, 2)



(b) Heapify(*A*, 4)



(c) Heapify(A, 9)

- $T(n) \le T(2n/3) + \Theta(1) = O(\lg n) = O(h)$, where n is the number of nodes in the subtree rooted at $A[i]$ and h is the height of $A[i]$.

6.3 Building a heap

Build-Heap(A)

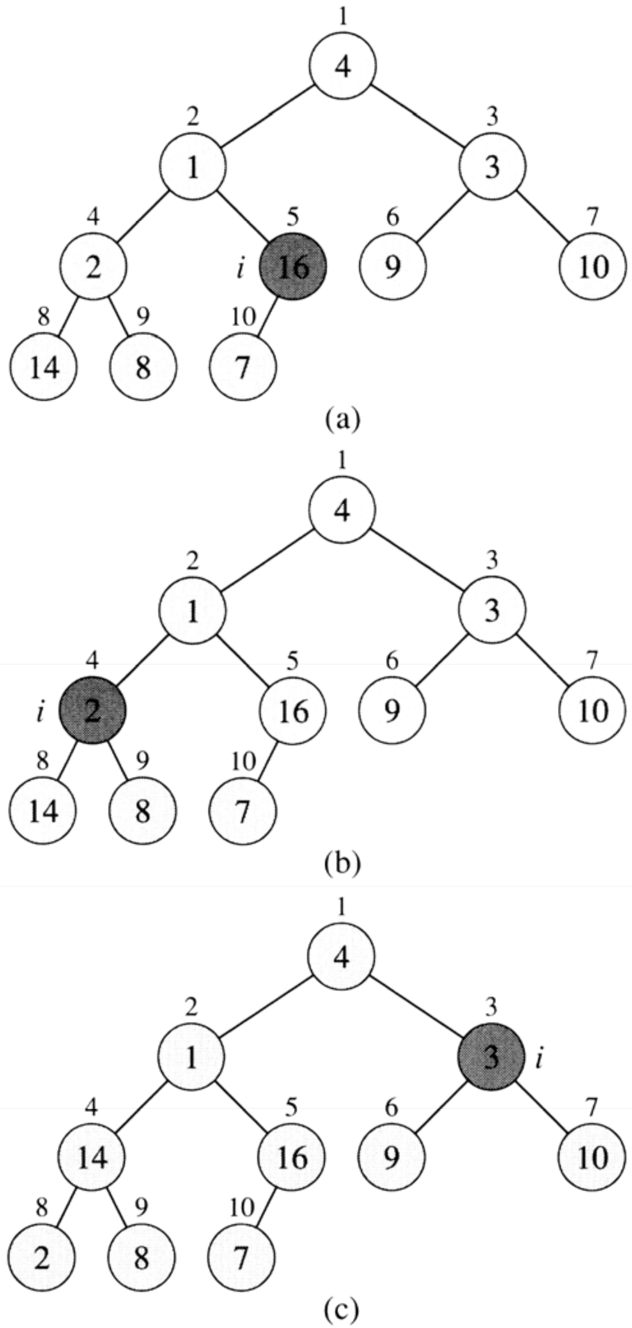
heap-size[A] $\leftarrow$ *length*[A]
for $i \leftarrow \lfloor \text{length}[A]/2 \rfloor$ **downto** 1 **do**
    Heapify(A, i)

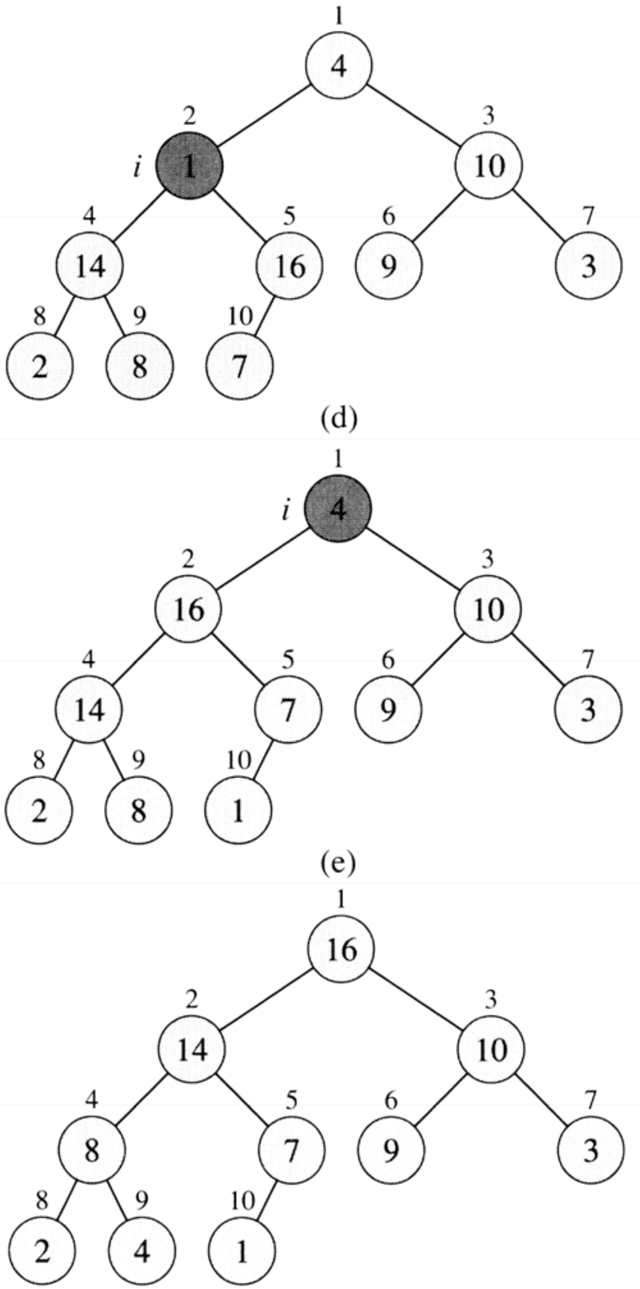
$$T(n) \le \sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \times \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right) = O(n \times 2)$$
$$= O(n).$$

OR: $T(n) = 2T(n/2) + \lg n$ (Assume $n = 2^{h+1} - 1$.)

Example:

A	4	1	3	2	16	9	10	14	8	7





(d) (e) (f)

6.4 Heapsort

Stage 1: Build a heap

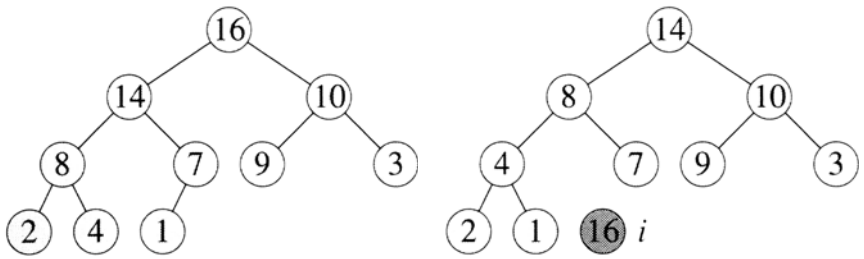
Stage 2: Repeatedly delete the root of the heap.

HeapSort(*A*)

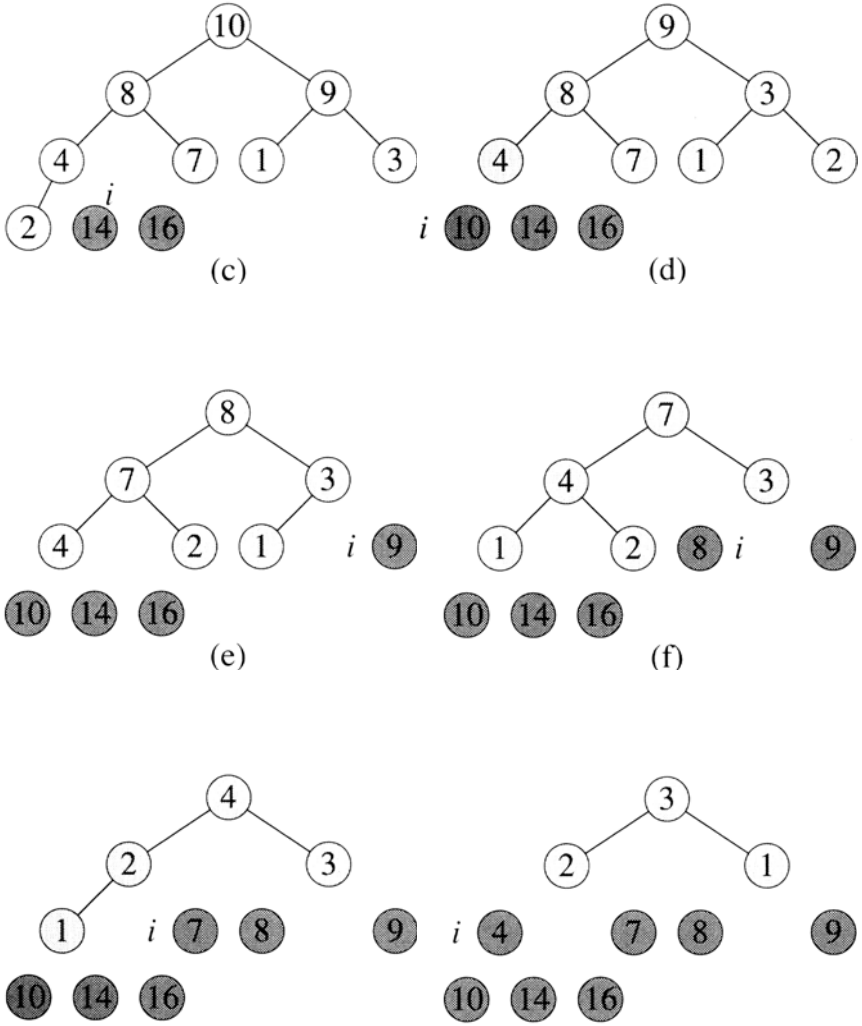
```
Build-Heap(A)
for i ← length[A] downto 2
    do  exchange A[1] ↔ A[i]
        heap-size[A] ← heap-size[A]–1
        Heapify(A, 1)
```

- $T(n) = O(n) + O((n\text{-}1) \times \lg n)$
  $= O(n\lg n)$

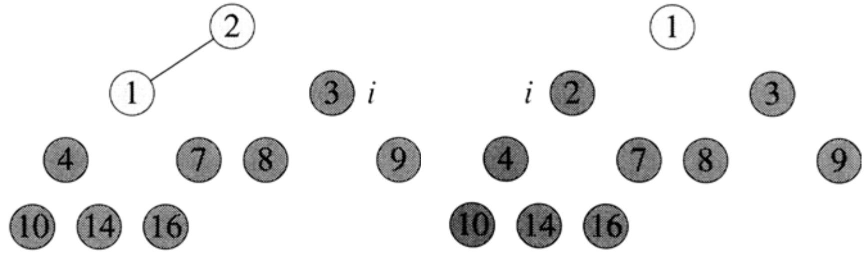
Example: sort *A*={4, 1, 3, 2, 16, 9, 10, 14, 8, 7}.



(a) (b)



(c) (d) (e) (f) (g) (h)



(i) (j)

A	1	2	3	4	7	8	9	10	14	16

(k)

6.5 Priority queues

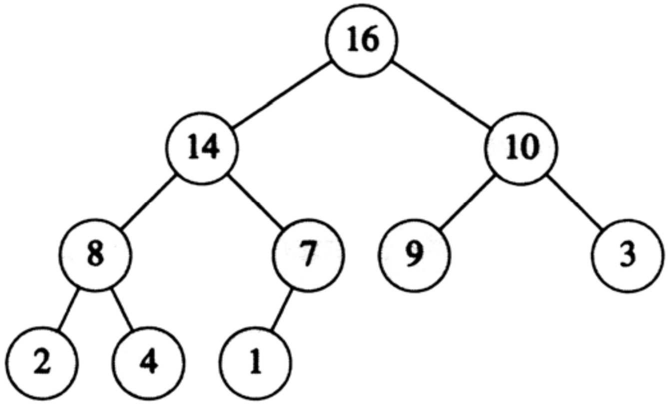
Priority queue*:** a data structure for maintaining a set *A* of elements, each has a value called a ***key. It should support the following operations.

Insert(*A*, *x*):
Maximum(*A*): (return)
Extract-Max(*A*): (return and remove)
Increase-Key(A, a, k): increase *a*'s key to larger key *k*, where *a* is an element of *A*

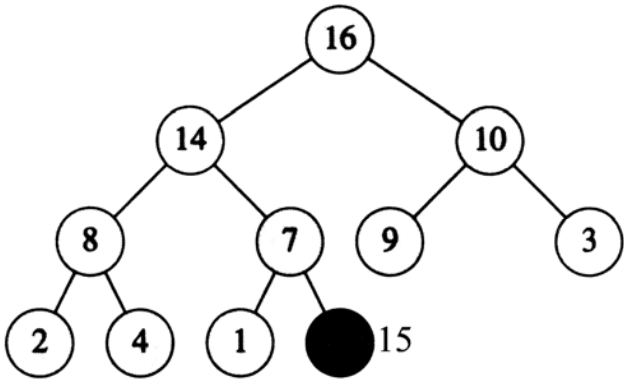
Applications: Job scheduling on a shared computer based upon "priorities."

Implement a priority queue by a heap:

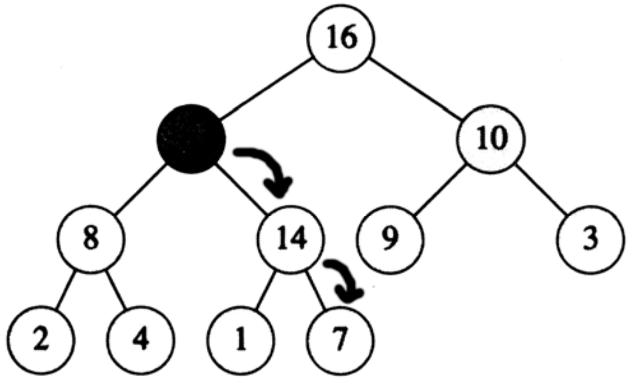
Insert(A, x): $O(\lg n)$ time



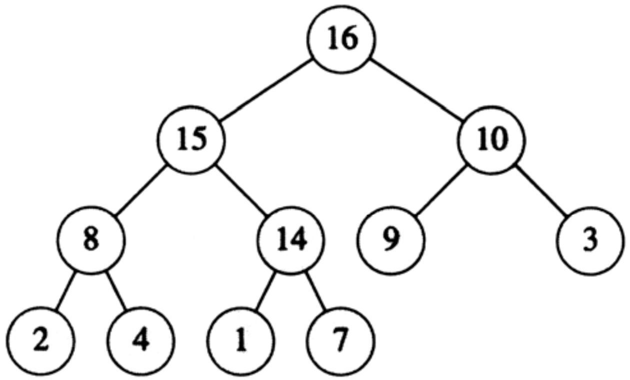
(a) *Insert*(A, 15)



(b)



(c)



(d)

Insert(A, x)

```
heap-size(A) ← heap-size(A)+1
i ← heap-size(A)
while i>1 and A[Parent(i)]<x
        do  A(i) ← A[Parent(i)]
            i ← Parent(i)
A[i] ← x
```

Increase-Key(A, i, k): $O(\lg n)$ time
(similar to *Insert*)

Maximum(*A*): $O(1)$ time

Extract-Max(*A*): $O(\lg n)$ time

Step 1: Exchange $A[1]$ and $A[\textit{heap-size}]$
Step 2: $\textit{heap-size} \leftarrow \textit{heap-size} - 1$
Step 3: *Heapify*(*A*, 1)
Step 4: return $A[\textit{heap-size} + 1]$

Homework: Ex. 6.2-5, 6.5-9, Prob. 6-2, 6-3