Priority Queue and Sorting: Using Heaps

A **Priority Queue** is a data structure maintaining a set S of elements, each with a **key**. A **max-priority queue** supports the following operations:

- Insert(S, x): inserts the element x into the set S.
- Maximum(S): returns the element of S with the largest key.
- Extract-Max(S): removes and returns the elements of S with the largest key.
- Increase-Key(S, x, k): increase the value of element x's key to the new value k.

Applications of Priority Queue

Job Scheduler in Operating Systems.

- Each process is assigned a priority. (each process has an item in a priority queue).
- When a new process comes in, system will assign it a priority. (Insert into the priority queue).
- System picks the process with the highest priority to run. (Maximum).
- When a process terminates, system needs to remove it from the queue. (Extract-Max)
- Sometimes system needs increase or decrease the priority of certain process. (Increase-Key).

First implementation: Sorted Array

- Insert: Need search for the proper place to insert and some elements need be moved; worst case: $\Theta(n)$.

- Maximum: $\Theta(1)$

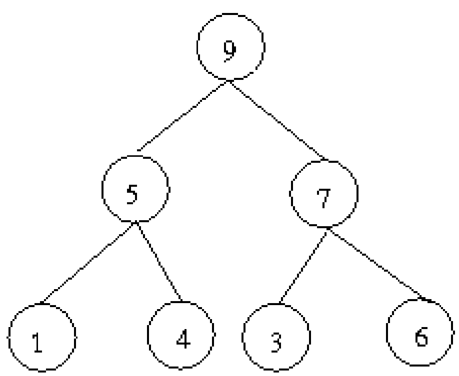
- Extract-Max: $\Theta(1)$

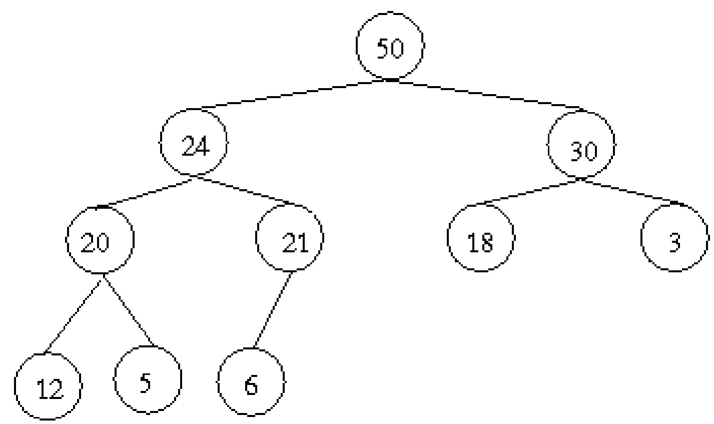
- Increase-Key: May need to find a new place to put; worst case: $\Theta(n)$

Heap

- **Heap** is a *complete binary tree*, namely, it's filled at all levels except at the lowest level (filled from left to right).

- (**Max-Heap property**) The value sorted in a node is greater than or equal to the values stored at its children .





Heap Representation using Arrays

Since there are no nodes at level l unless level $l-1$ is completely filled, a heap can be stored in an array level by level (beginning with the root), left to right within each level.

- The ROOT is always stored at A[1]

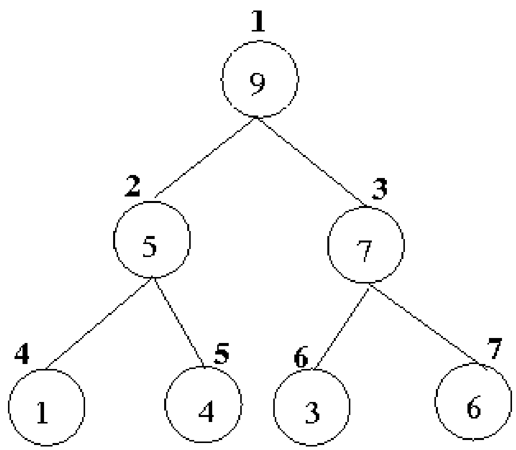
- PARENT$(i) = \lfloor i/2 \rfloor$

- LEFT-CHILD$(i) = 2i$

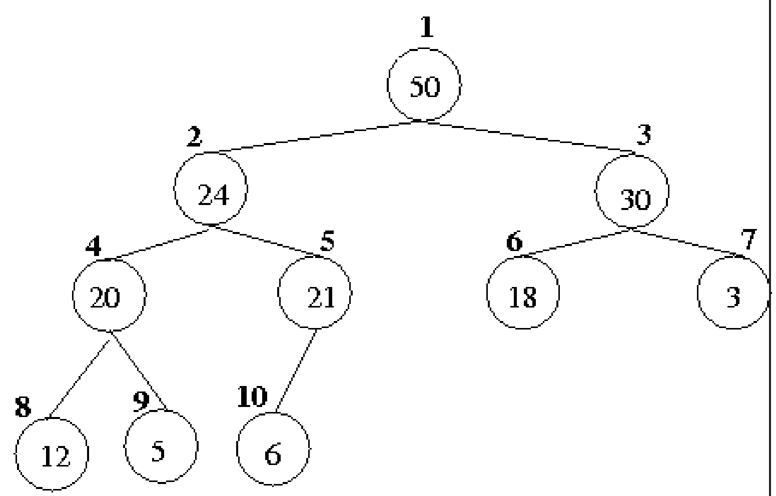
- RIGHT-CHILD$(i) = 2i + 1$

- Length[A]: number of elements in the array A;
  Heapsize[A]: number of elements in the heap stored within array A.

Examples of Heap Representations



9	5	7	1	4	3	6



50	24	30	20	21	18	3	12	5	6

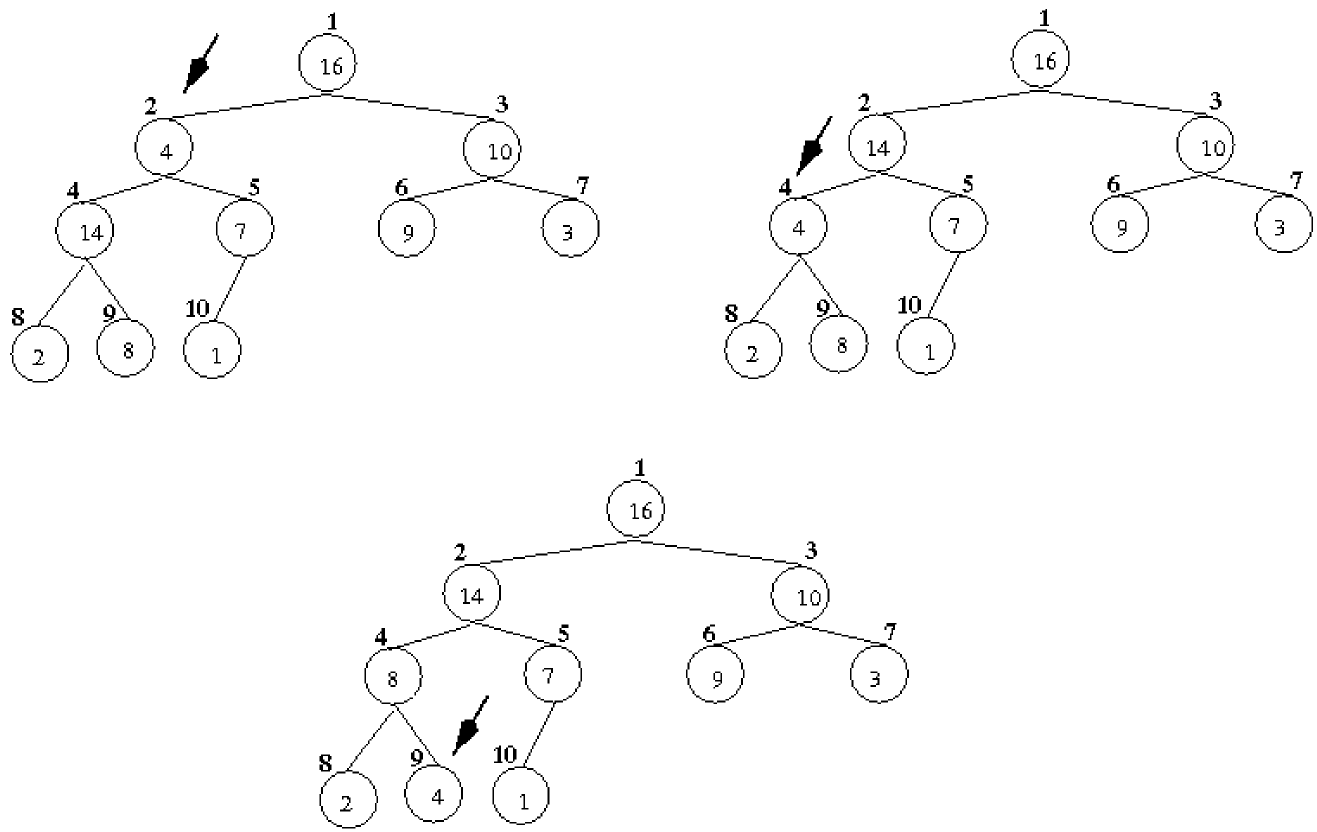
Maintain(Restore) the heap property

If A[i]'s left subtree and right substree are Max-Heaps, but A[i] violates the heap property, i.e., A[i] is smaller than its children, Max-Heapify(A, i) is called to let A[i] "float down" in the max-heap so that the subtree rooted at index i becomes a Max-Heap.

Max-Heapify(A, i)

```
max := i;
if (2i ≤ Heapsize[A] AND A[max] < A[2i])
        max := 2i;
if (2i + 1 ≤ Heapsize[A] AND A[max] < A[2i + 1])
        max := 2i + 1;
if (i ≠ max)
        exchange A[i] and A[max];
        Max-Heapify(A, max);
```

Max-Heapify



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

Complexity for MAX-HEAPIFY

- Comparing A[i] with A[$2i$] and A[$2i+1$] takes $\Theta(1)$ time.
- The children's subtree each has size at most $2n/3$ — the worst case occurs when the last row of the tree is exactly half full.

$$T(n) \leq T(2n/3) + \Theta(1)$$

$$\Rightarrow T(n) = O(lgn) \text{ (Using Master Method)}$$

- Another approach: the height of a complete binary tree with n elements is $\Theta(lgn)$. Why is that?

  Because for a complete binary tree with height h , it has at most $2^{h+1}-1$ and at least $2^h-1+1=2^h$ nodes.
  $\Rightarrow 2^h \leq n \leq 2^{h+1}-1 \Rightarrow \lg(n+1)-1 \leq h \leq lgn$

  Because $T(n) \leq h \Rightarrow T(n) = O(lgn)$

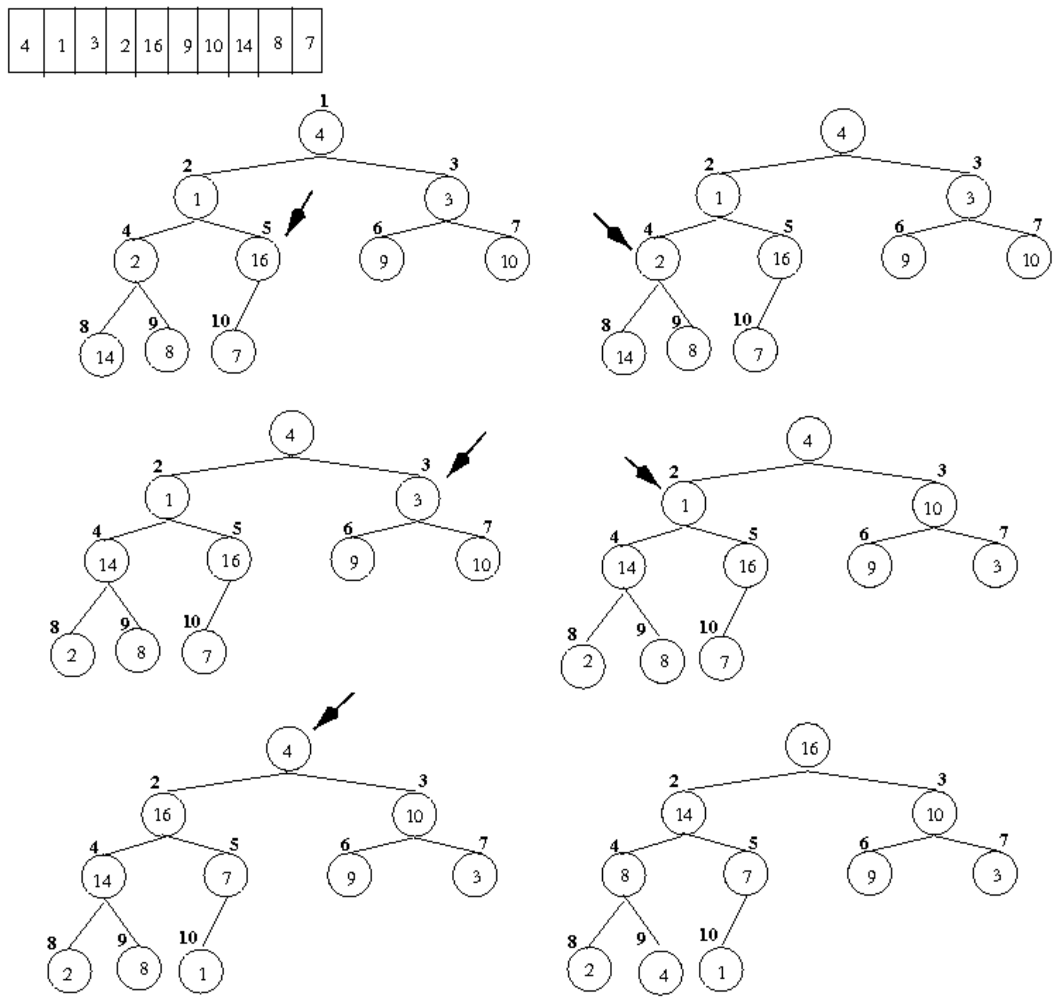
Build a Heap

Use Max-Heapify in a bottom-up fashion to convert A[1..n] to a Max-Heap.

Build-Max-Heap(A)

Heapsize[A] = Length[A];
for $i := \lfloor Length[A]/2 \rfloor$ downto 1
Max-Heapify(A, i);

An example of Build-Max-Heap



Complexity for BUILD-Max-HEAP

- Assume that the binary tree is a full binary tree (the proof is slighly more complicated if the binary tree is not full):

- $T(n) = T(n/2) + T(n/2) + O(lgn)$,
  where the first $T(n/2)$ is to Build the left sub heap, the second is for right sub heap. The $O(lgn)$ is the complexity for Max-Heapify(A, 1), which makes the whole tree a heap.

- Based on Master-Method case 1, $T(n) = \Theta(n)$.

Sort using Heaps – HeapSort

HEAPSORT(A)

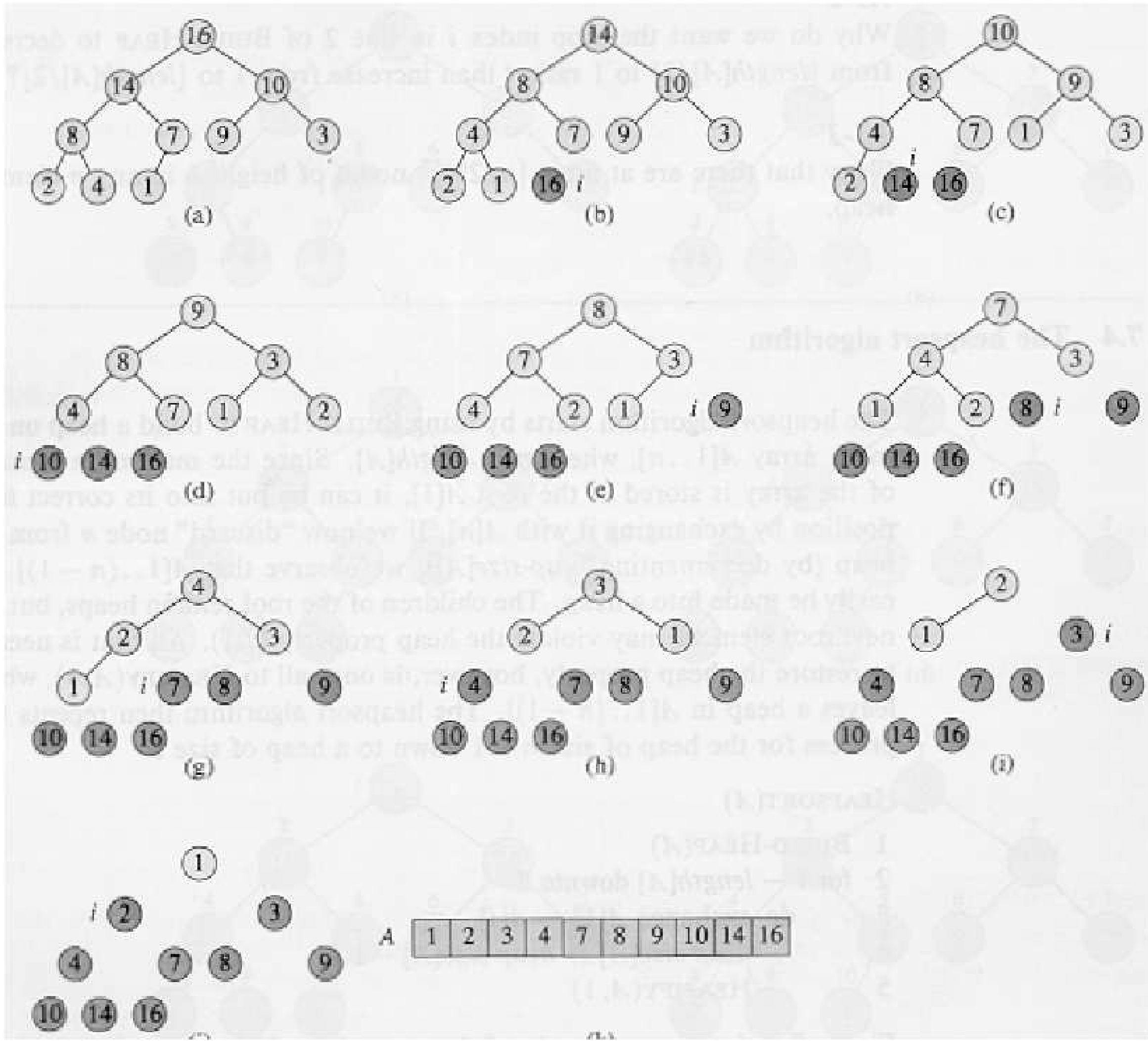
Build-Max-Heap(A); —— $\Theta(n)$
for i := n to 2
  exchange A[1] and A[i];
  Heapsize[A] := Heapsize[A] - 1;
  Max-Heapify (A, 1); —— $O(lgn)$

Comments:

- A[1..Heapsize[A]] are the elements currently in the heap. When elements are removed from the heap one by one, Heapsize[A] is decremented. Length[A] does not change.

- Complexity: $\Theta(n) + O(nlgn) = O(nlgn)$.

Examples of HeapSort



Priority Queue Operations: using Heaps

Maximum(A)

return A[1];

−− Complexity: $\Theta(1)$

Extract-Max(A) // Remove and return the max.

MAX := A[1];
Exchange A[1] and A[Heapsize[A]];
Heapsize := Heapsize - 1;
Max-Heapify(A, 1);
return MAX;

−− Complexity: $O(h) = O(lgn)$.

Priority Queue Operations, Cont'd

Increase-Key(A, i, key) // Increase the value of A[i] to key.
// assume key is bigger than A[i].

```
A[i] := key;
while (i > 1 AND A[parent(i)] < A[i]) DO
    exchange A[i] and A[parent(i)];
    i := parent(i);
```

$--$ Complexity: $O(h) = O(lgn)$.

Insert(A, key) // Insert key into A

```
Heapsize[A] := Heapsize[A] + 1;
A[Heapsize[A]] := −∞;
Increase-Key(A, Heapsize[A], key);
```

$--$ Complexity: $O(h) = O(lgn)$.

Summary:Complexities using Heaps

Operation	Worst Case
Max-Heapify	$O(lgn)$
Build-Max-Heap	$O(n)$
Heap-Sort	$O(nlgn)$
Maximum	$\Theta(1)$
Extract-Max	$O(lgn)$
Insert	$O(lgn)$
Increase-Key	$O(lgn)$