

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:

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

Applications of Priority Queue

Job Scheduler in Operating Systems.

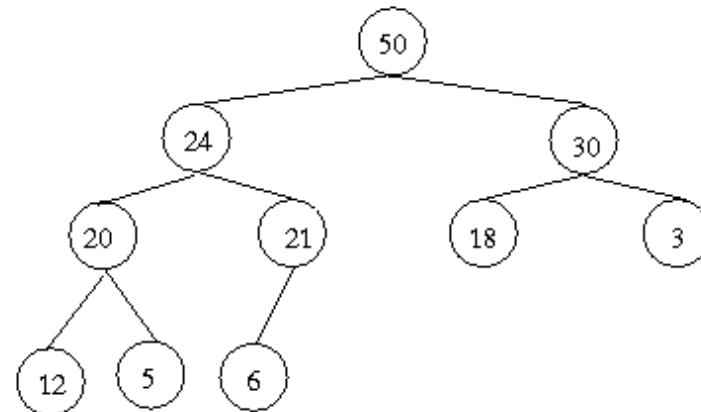
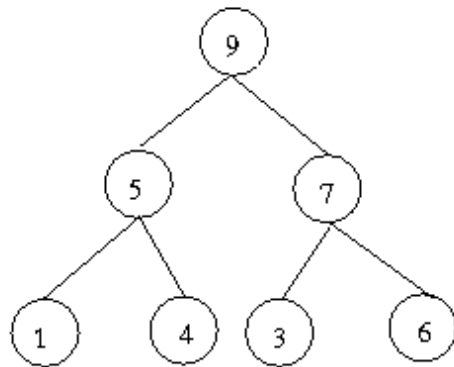
- Each process is assigned a priority. (each process has an item in a priority queue).
- When a new process comes in, system will assign it a priority. (Insert into the priority queue).
- System picks the process with the highest priority to run. (Maximum).
- When a process terminates, system needs to 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: $\Theta(n)$
- 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 stored in a node is greater than or equal to the values stored at its children

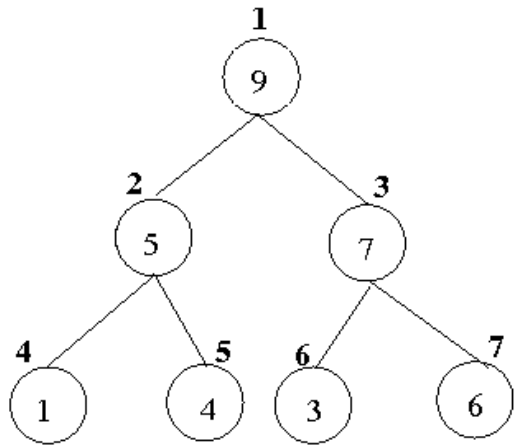


Heap Representation using Arrays

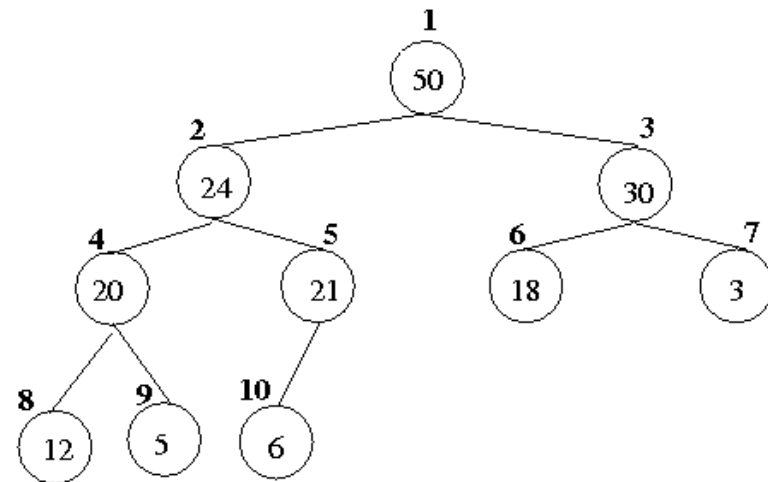
Since there are no nodes at level l unless level $l - 1$ is completely filled, a heap can be stored in an array level by level (beginning with the root), left to right within each level.

- The ROOT is always stored at $A[1]$
- $\text{PARENT}(i) = \lfloor i/2 \rfloor$
- $\text{LEFT-CHILD}(i) = 2i$
- $\text{RIGHT-CHILD}(i) = 2i + 1$
- $\text{Length}[A]$: number of elements in the array A ;
 $\text{Heapsize}[A]$: 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
----	----	----	----	----	----	---	----	---	---

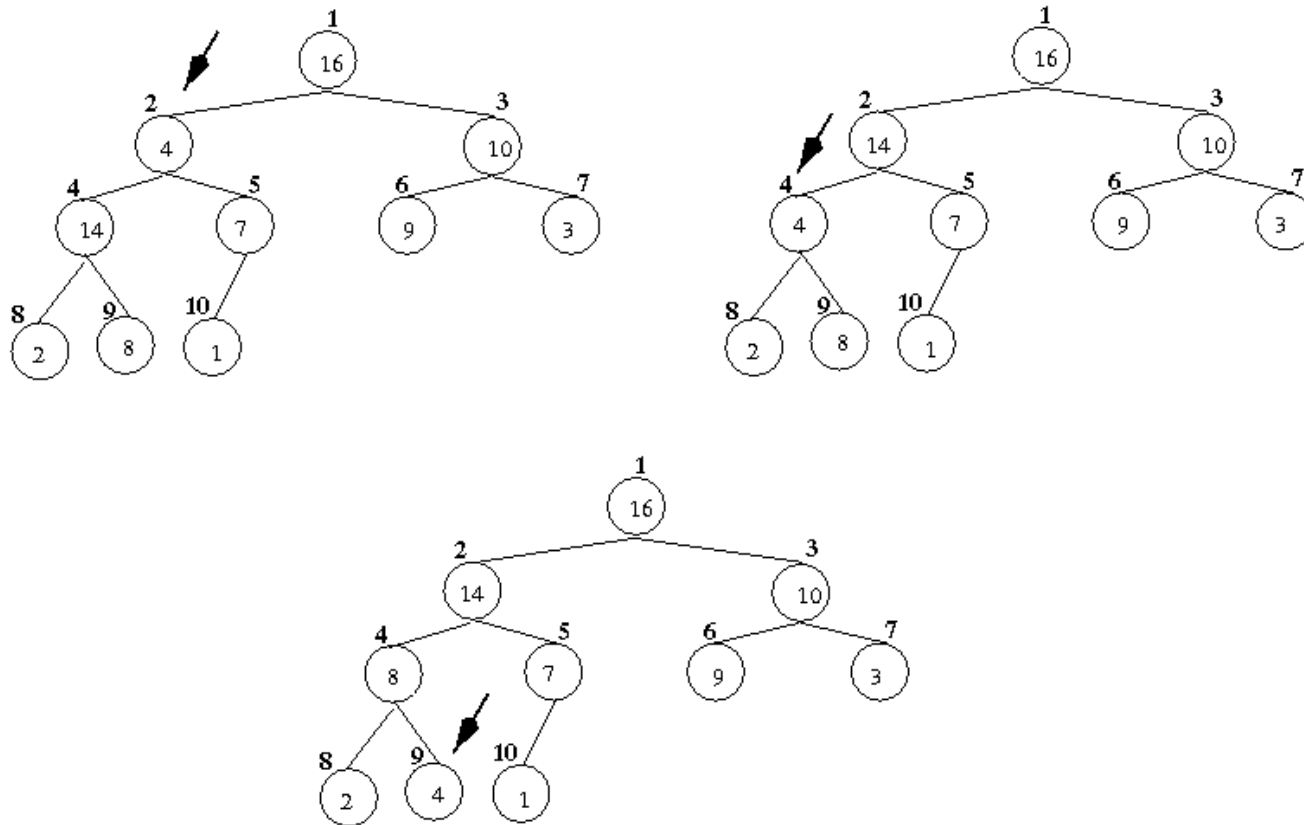
Maintain(Restore) the heap property

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

$\text{Max-Heapify}(A, i)$ // Restore the heap property for the tree rooted at i . When called, the left and right subtrees must be Max-Heaps.

```
if ( $A[i] \geq A[2i]$ ) AND  $A[i] \geq A[2i + 1]$ )
    return;
else
    let  $j$  be the largest child of  $i$ ;
    exchange  $A[i]$  and  $A[j]$ ;
    Max-Heapify( $A, j$ );
```

Max-Heapify



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.

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

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

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

$$\Rightarrow 2^h \leq n \leq 2^{h+1} - 1 \Rightarrow \lg n + 1 - 1 \leq h \leq \lg n$$

$$\text{Because } T(n) \leq h \Rightarrow T(n) = O(\lg n)$$

Build a Heap

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

Build-Max-Heap(A)

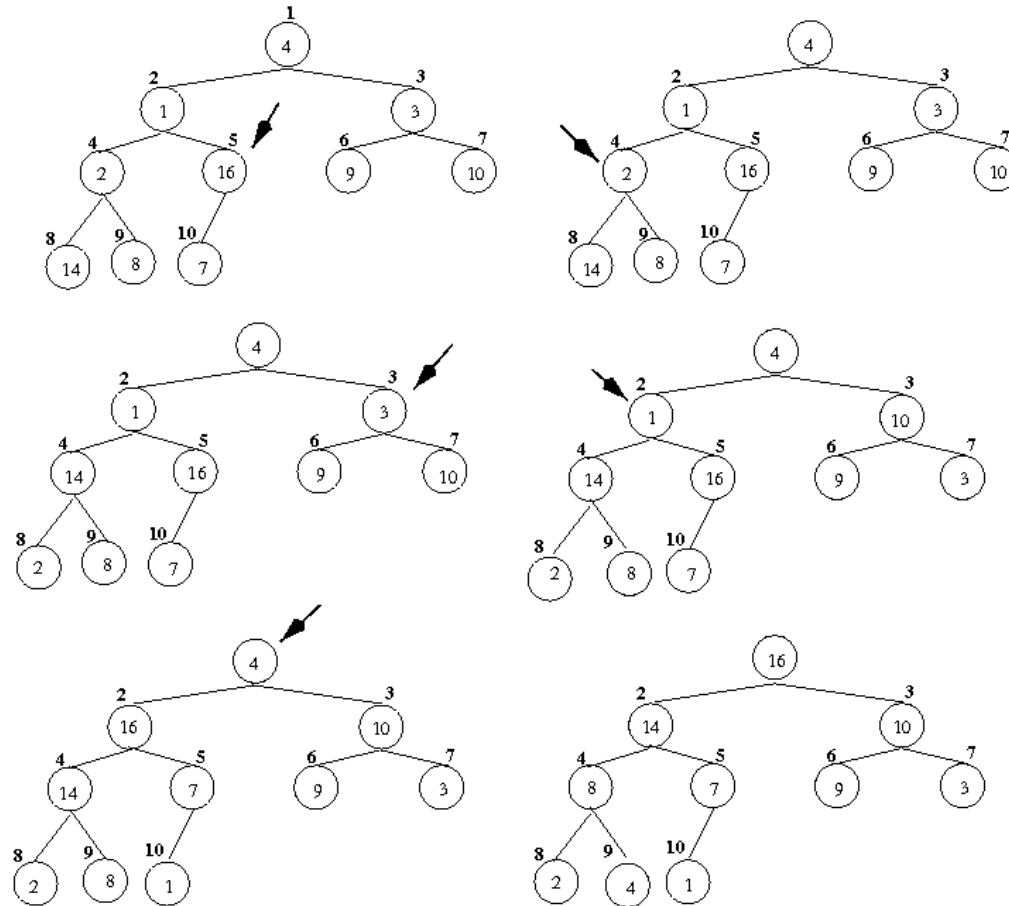
 Heapsize[A] = Length[A];

 for $i := \lfloor \text{Length}[A]/2 \rfloor$ downto 1

 Max-Heapify(A, i);

An example of Build-Max-Heap

4	1	3	2	16	9	10	14	8	7
---	---	---	---	----	---	----	----	---	---



Complexity for BUILD-Max-HEAP

- Assume that the binary tree is a full binary tree (the proof is slightly more complicated if the binary tree is not full):
- $T(n) = T(n/2) + T(n/2) + O(\lg n)$,
where the first $T(n/2)$ is to Build the left sub heap, the second is for right sub heap. The $O(\lg n)$ 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

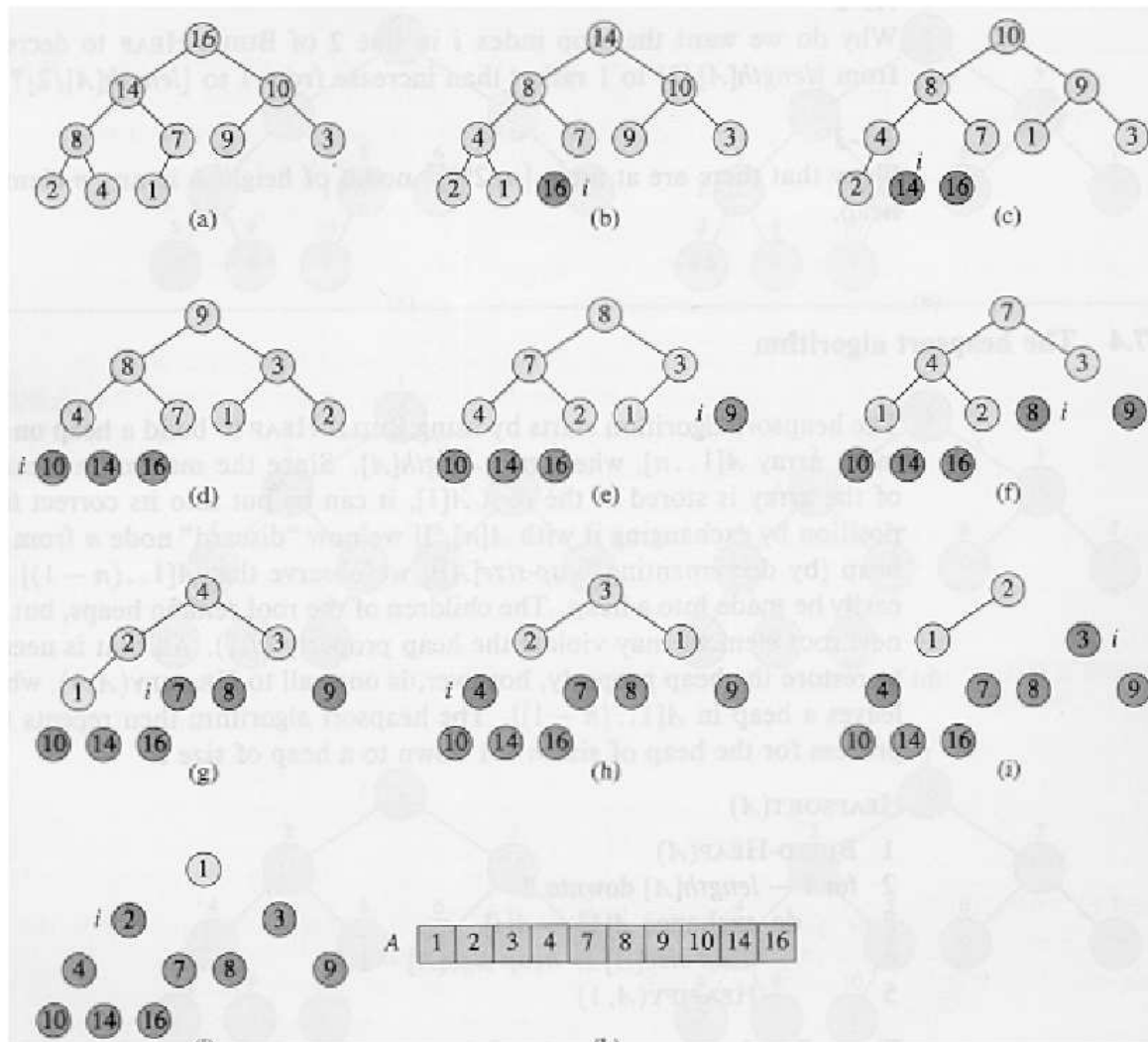
HEAPSORT(A)

```
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);      —  $O(\lg n)$ 
```

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: $\Theta(n) + O(n \lg n) = O(n \lg n)$.

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: $O(h) = O(\lg n)$.

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 $A[\text{parent}(i)] < A[i]$) **DO**

 exchange $A[i]$ and $A[\text{parent}(i)]$;

$i := \text{parent}(i)$;

-- Complexity: $O(h) = O(\lg n)$.

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

$\text{Heapsize}[A] := \text{Heapsize}[A] + 1$;

$A[\text{Heapsize}[A]] := -\infty$;

 Increase-Key(A, $\text{Heapsize}[A]$, key);

-- Complexity: $O(h) = O(\lg n)$.

Summary: Complexities using Heaps

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