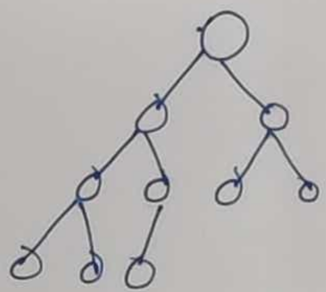
Heap

* A heap is a binary tree 'T' that satisfies two additional property.
  - → relational Property
  - → structural Property

Structural Prop:

It is a complete binary tree: → completely filled at each level, except possibly lower level.
→ Tree is filled from left to right



How to map tree in an array?? [Relational Property]

Array index starts from - 1.

A[ ].

for a node at index 'i'

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

$l(i)$ = left child of i

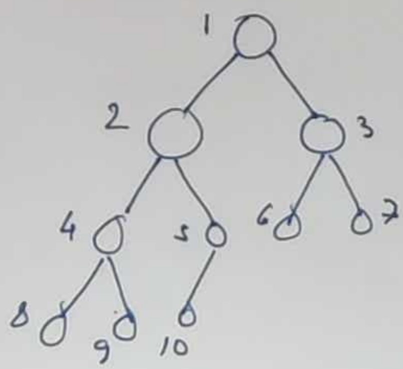
$r(i)$ = Right

$p(i)$ = Parent

$l(i) = 2i$

$r(i) = 2i+1$

$p(i) = \lfloor i/2 \rfloor$



Max-Heap : for node i $A[p(i)] \geq A[i]$

Min-Heap : —— $A[p(i)] \leq A[i]$

* No ordering among siblings.

Height of Heap:

Claim: $\log(n+1) - 1 \leq h \leq \log n$

Proof:

$$\sum_{i=0}^{h-1} 2^i + 1 \leq n \quad \sum_{i=0}^{h} 2^i$$

$$\boxed{\log(n+1) - 1 \leq h \leq \log n}$$

Height always $= \Theta(\log n)$

Maintaing the Heap Property :-

downheap (i)

max ← i ↗ last element of array.

if $l(i) \leq$ heap-size[A] & A[max] < A[l(i)]

  max ← l(i)

if $r(i) \leq$ heap-size[A] & A[max] < A[r(i)]

  max ← r(i)

if max ≠ i

  A[i] ← A[max]

  downheap(max)

$$\boxed{O(\log n)}$$

Building a heap

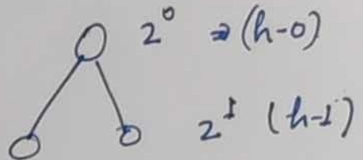
Buildheap(n).

for $i \leftarrow n$ down to 1
    downheap(i)

$\boxed{O(n \log n)}$

Correct analysis:

$$\sum_{i=0}^{h} i \cdot 2^i = (h+1)\, 2^{h+1} - 2^{h+2} + 2$$



We 2^i nodes which had gone through height $(h-i)$ in ~~the~~ worst case.

Total complexity

$$\sum_{i=0}^{h} 2^i\, O(h-i)$$

$$= O\left(h \sum_{i=0}^{h} 2^i - \sum_{i=0}^{h} i \cdot 2^i \right)$$

$$= O\left((h \cdot 2^{h+1} - h) - (h+1)\, 2^{h+1} + 2^{h+2} - 2 \right)$$

$$= O\left(2^{h+2} - 2^{h+1} - h - 2 \right)$$

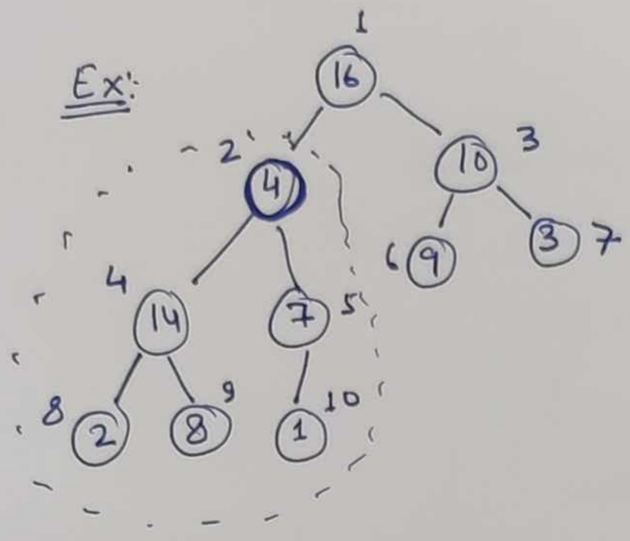
$$= O\left(2^{h+2} - h - 2 \right)$$

$$= O\left(2^h \right)$$

$$\boxed{T(n) = O(n)}$$

Maintaining the heap Property :

Ex:



downheap (2) or

Max-Heapify (2)

$\boxed{i=2}$

Pseudocode : downheap(i)

$max \leftarrow i$

if $l(i) \leq heapsize[A]$ & $A[max] < A[l(i)]$

$\quad max \leftarrow l(i)$

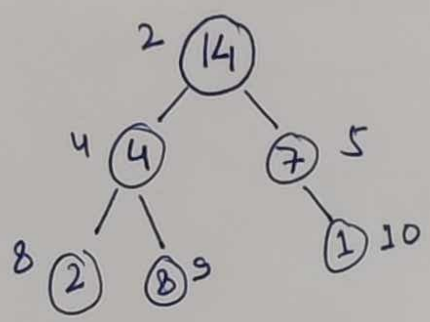
if $r(i) \leq heap\text{-}size[A]$ & $A[max]\ A[r(i)]$

$\quad max \leftarrow r(i)$

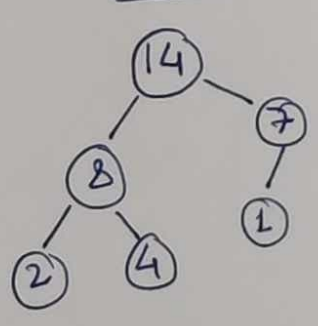
if $max \neq i$ (if any of above cond^n true)

$\quad A[i] \leftrightarrow A[max]$

$\quad$ downheap(max)



Now; $(i=4)$



Heap Sort:

Heap Sort (A) **Heap-size [A] = n : Given.**

Build-heap (A)

for $i \leftarrow n$ down to 2

  $A[1] \leftarrow A[i]$

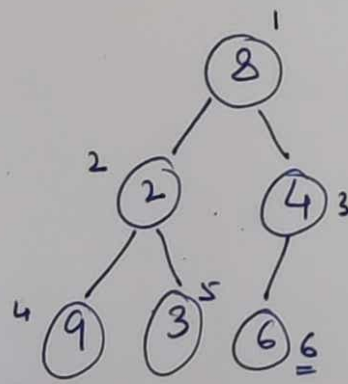
  heap-size [A] $\leftarrow i-1$

  downheap (1)

$$\boxed{\text{Complexity} : O(n \log n)}$$

Ex:-

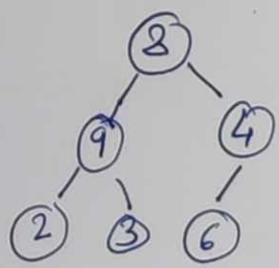
1	2	3	4	5	6
8	2	4	9	3	6



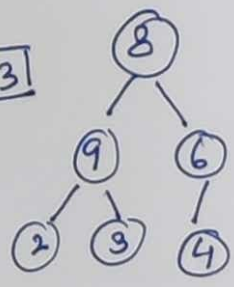
Now, call down heap (i).

i=6, ok
i=5, ok
i=4, ok

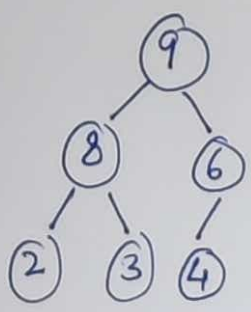
$\boxed{i=2}$



$\boxed{i=3}$



$\boxed{i=1}$



Application of Heap

1. Heap Maximum

```
Heap-maximum(A)
    return A[1]
```

$\Theta(1)$

2. Heap-Extract-Max : Remove max element of heap and again heapify

Heap-extract-Max(A)   Heap-size[A] = n

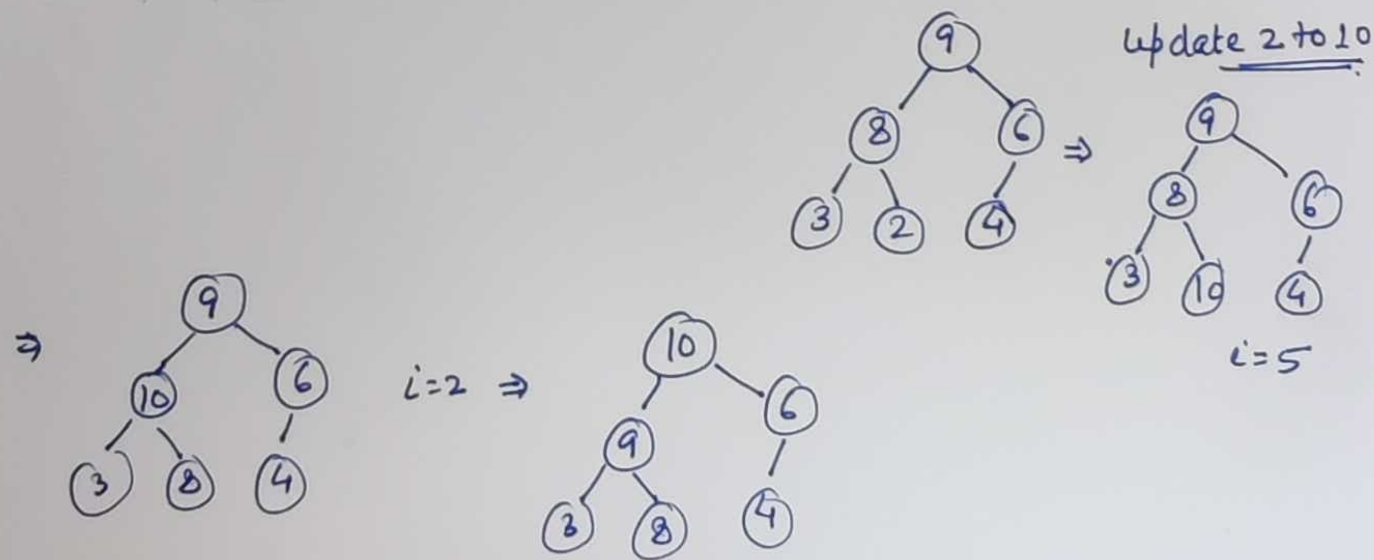
```
max ← A[1]
A[1] ← A[n]
n ← n-1
downheap(1)
```

$\boxed{O(\log n)}$

3. Heap-Increase-Key (A, i, key) : Increase the value of A[i] to Key, where A[i] < Key

$O(\log n)$

```
A[i] ← key
While i > 1 & A[p(i)] < A[i]
        A[i] ⟷ A[p(i)]
        i ← p(i)
```



(4) Max-Heap-Insert (A, key)

$n \leftarrow n+1$
downheap(n) $\Big]$ $O(\log n)$

(5) Max-Heap-Delete(A, ~~key~~ i)

If ~~key~~ i is last leaf node, ok
else exchange i with last leaf and Heapify

```
if i = n
    A[i] ← Nil
else
    A[i] ← A[n]
    A[n] ← Nil
    n ← n-1
    down heap(i)
```

$O(\log n)$