

HEAPS

→ Time complexities of operations on different types of arrays

	insert	search	find min	delete min
unsorted Array	$O(1)$	$O(n)$	$O(n)$	$O(n)$
sorted Array	$O(n)$	$O(\log n)$	$O(1)$	$O(n)$
linked list	$O(1)$	$O(n)$	$O(n)$	$O(n+1) = O(n)$
notes				

If some problem needs, insert, find-min and delete-min to be optimal , then we have

HEAP - Data structure

	insert	find-min	delete min
min Heap	$O(\log n)$	$O(1)$	$O(\log n)$

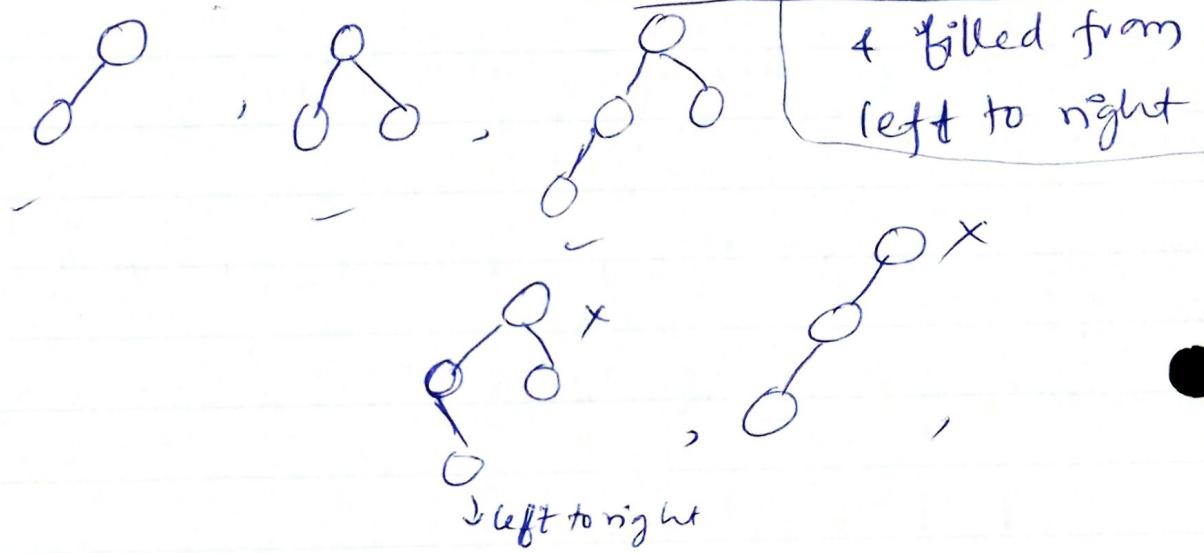
→ "Heap" is a "binary tree" or 3-ary con-n-ary tree

→ Binary Tree: Every node can have atmost '2' children

- Every Heap is a "almost complete binary tree"
- complete-B-T: All nodes are filled in the last level



- Almost -Complete B.T: leaves should be in either last (or) last but one "level" + filled from left to right



MAX HEAP: "ROOT", should be greater than left and right sub-tree. nodes

MIN-HEAP: "ROOT", should be less than left and right sub-tree nodes

HEAP IMPLEMENTATION

→ Heap implementation is done using the
"array"

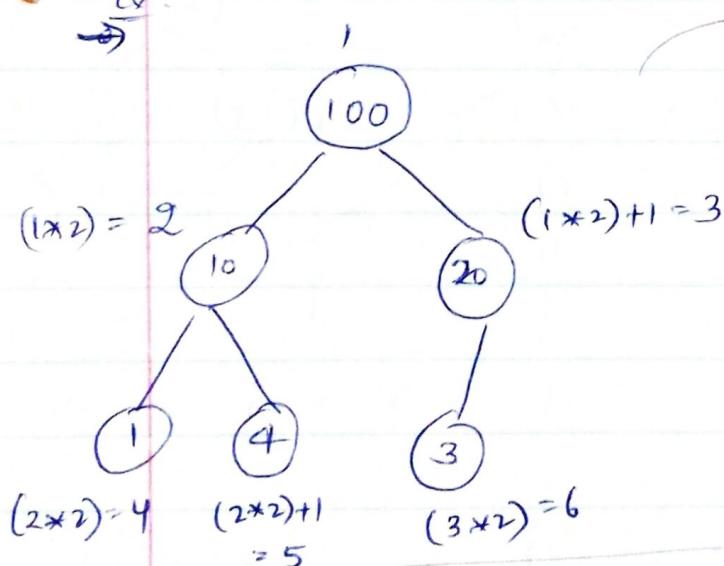
Ex: 8 Points

① "Root" will be placed at 1st element

② "Left child" = (index of root) * 2

③ "Right child" = ((index of root) * 2 + 1)

Ex:



Why heap complete or Almost complete B-T?

x) If you have complete (or) Almost complete

array will be completely filled (or) empty space (or) gaps is left between elements.

Finding elements :-

$$\textcircled{1} \text{ Left Child } (i) = 2 \times i^{\text{index}} \Rightarrow \text{ multiplying with '2' } \hookrightarrow \text{left shift}$$

$$\textcircled{2} \text{ Right Child } (i) = 2 \times i + 1$$

$$\textcircled{3} \text{ Parent } (i) = \lfloor i/2 \rfloor \Rightarrow \text{right shift } \Rightarrow \text{dividing by '2'}$$

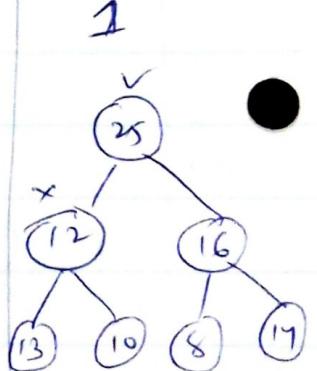
Ex:-

25, 12, 16, 13, 10, 8, 14

array length
↑
A.length

MAX-HEAP
no. of elements
↑ following heap property
A.heap size

7



14, 13, 8, 12, 10

array in descending order
will be a max-heap

5

5

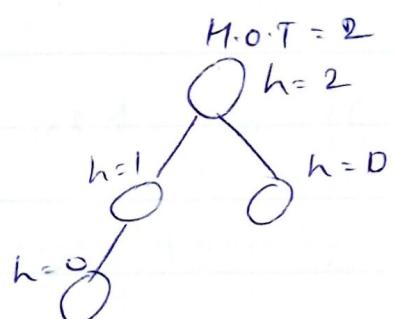
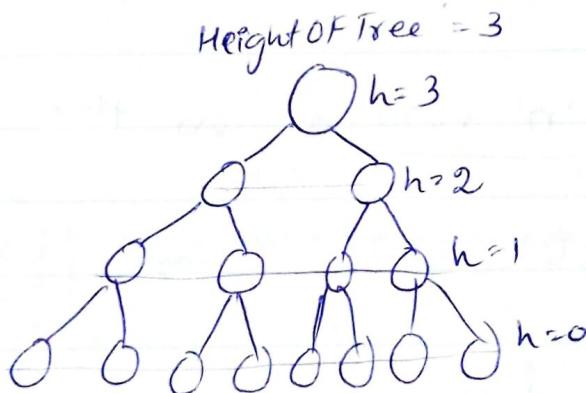
Array in ascending order
 \Rightarrow Min heap

→ If 'you' have to construct a heap, no need to sort, which takes $O(n \log n)$ time. Instead, we can construct a heap in $O(n)$ time complexity.

Complete Binary Tree Properties :-

- ① Height of a Node :- The longest possible no. of edges from that node to any leaf

Ex:-



- ② Height of a Tree :- = Height of the root
- ③ Given, height of the ^{complete binary} tree, what are maximum of Nodes in the Tree ?

$$\text{max. no. of nodes: } (2^0 + 2^1 + 2^2 + \dots + 2^h)$$

$$= \frac{1(2^{h+1} - 1)}{2 - 1} = 2^{h+1} - 1$$

① max. no. of nodes of 3-ary tree = $3 + 3^1 + 3^2 + \dots$

$$= \frac{1}{3-1} (3^{h+1}-1)$$

$$= \frac{3^{h+1}-1}{2}$$

② max. no. of nodes for n-ary tree =

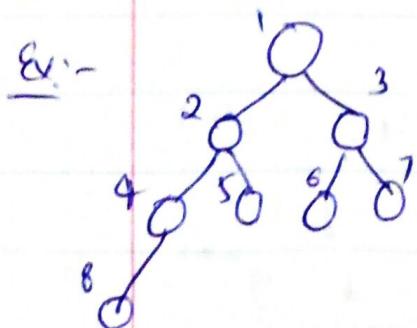
$$n^0 + n^1 + n^2 + n^3 + \dots + n^h$$

$$= \frac{1}{n-1} (n^{h+1}-1) = \frac{n^{h+1}-1}{n-1}$$

⇒ If we have 'n' nodes in the complete B.T
 (or) almost complete B.T, then '[log n]' is the height of the tree.

③ In a complete B.T (or) almost complete B.T, the range of leaves nodes start from index is

$$\left[\left\lfloor \frac{n}{2} \right\rfloor + 1 \text{ to } n \right]$$



$$\text{leaves} = [5, 6, 7, 8]$$

nodes ^{from}
₅ to 8

MAX-HEAPIFY:

MAX-HEAPIFY(A, i) :

{

$l = 2i$;

$r = 2i + 1$;

if ($l \leq A.\text{heap-size}$ and $A[l] > A[i]$)

 largest = l ;

else largest = i ;

if ($r \leq \text{heap-size}$ and $A[r] > A[\text{largest}]$)

 largest = r ;

if (largest $\neq i$)

 exchange $A[i]$ with $A[\text{largest}]$

 MAX-HEAPIFY($A, \text{largest}$)

}

- One way to create a Max-Heap is to sort elements in the descending order. But it takes $O(n \log n)$ time.

- ① Every leaf is a heap, as it is the maximum & it is the minimum and everything.
- ② So, Building blocks of heaps are leaf nodes,

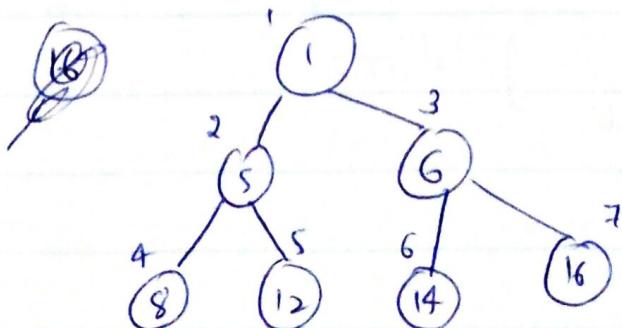
Ex:-

Consider arr = [1, 5, 6, 8, 12, 14, 16]

1. while constructing a heap, we consider that leaf nodes are heaps, as ~~it is them~~
2. We know that leaf nodes start from the indices $\lfloor \frac{n}{2} \rfloor + 1$

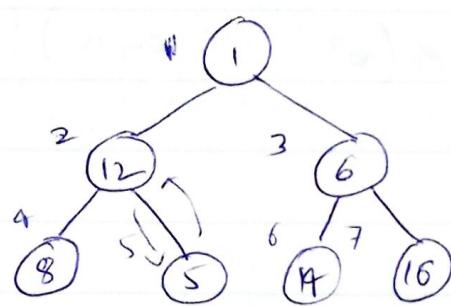
3. So, we will start heapifying from the largest non-leaf node index and moves towards root.

If you consider above example, C.B.T for is

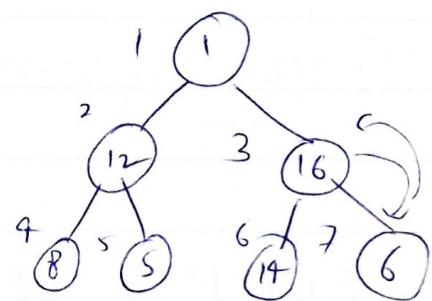


- ① leaf nodes start from, $\lceil \frac{7}{2} \rceil + 1 = 4$.
- ② So, largest non-leaf node = 3
- ③ Call the max-heapify on the node with index = 3.

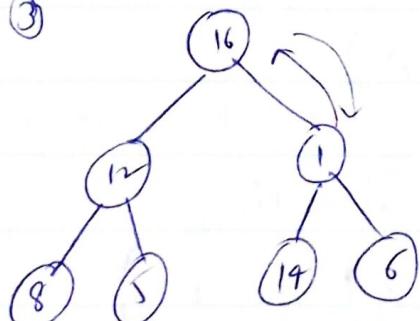
①



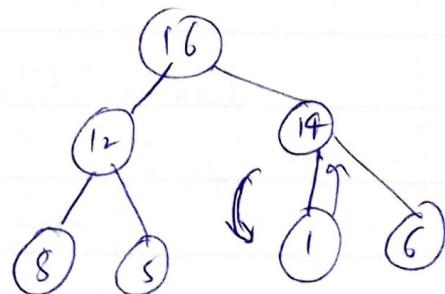
②



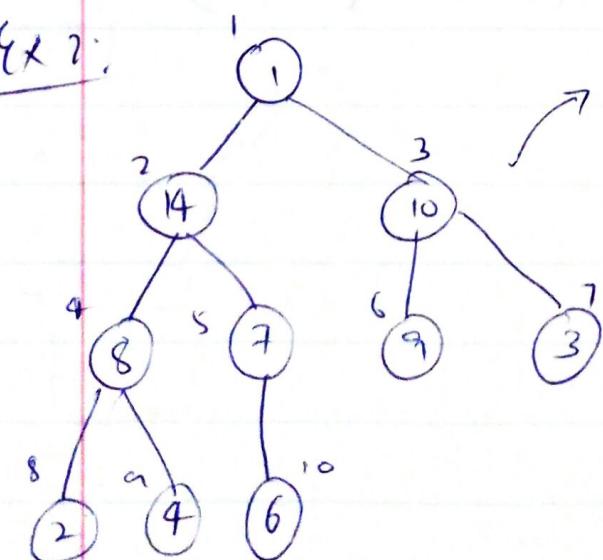
③



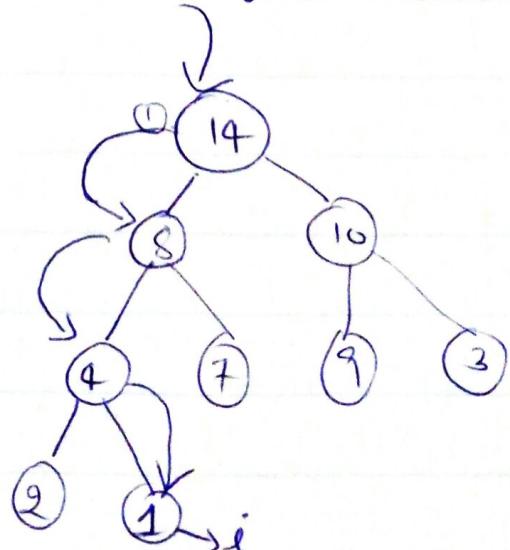
④



Ex 2:



\Rightarrow max-heapify(1)



Time Complexity:

1. In order to do one swapping, we need to do "2" comparisons at each level and at max " $\log n$ " levels, so

$$T(n) = 2 * \log n = O(\log n)$$

[Space Complexity = ~~O(n)~~ . $O(\log n)$]

BUILD-MAX-HEAP

Build-Max-Heap (A)

{
 $A.\text{heapsize} = A.\text{length}$

 for ($i = \lfloor A.\text{length}/2 \rfloor$ to 1)

 MAX-HEAPIFY (A, i)

}

Time Complexity Analysis :-

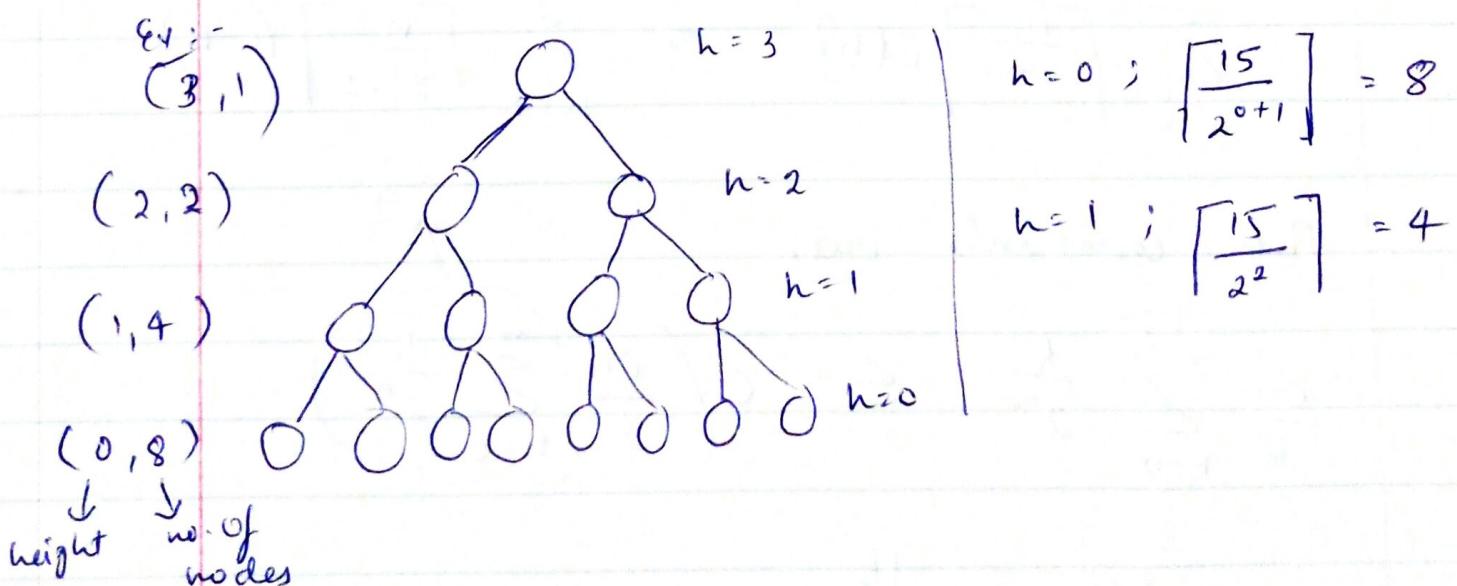
① We might think, if we call heapify on a node, it takes $O(\log n)$ time and as we have ' n ' nodes, time complexity would be $O(n \log n)$.

BUT, we can prove that, time complexity to build a heap is $O(n)$.

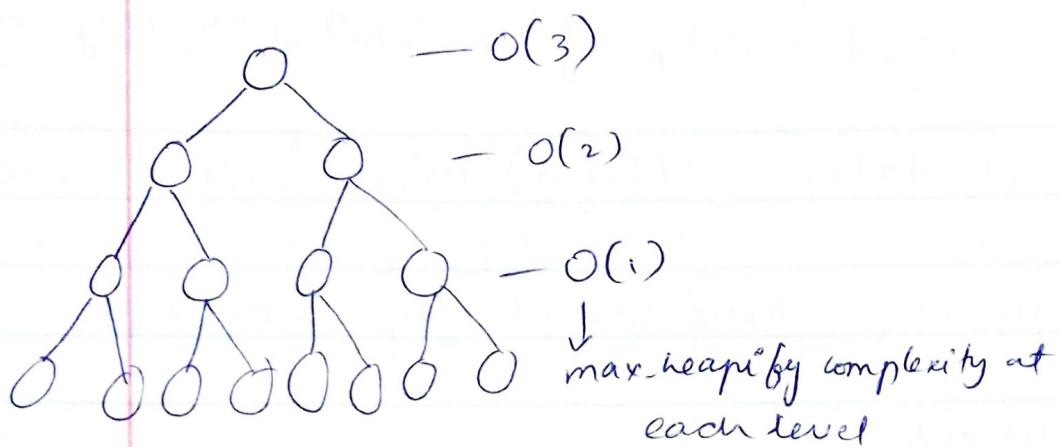
Ex:- In a complete B-T, for the no. of nodes of

a tree at height ' h ' =

$$\left\lceil \frac{n}{2^{h+1}} \right\rceil$$



So, if we apply max-heapify at each level



→ Work done, at height 'h', to apply max-heapify

$$= \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h)$$

→ In a tree, height can range from $0, \log n$

$$= \sum_{h=0}^{\log n} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) \Rightarrow \sum_{h=0}^{\log n} \left\lceil \frac{n}{2^{h+2}} \right\rceil (c \cdot h)$$

Pull constants out,

$$= \frac{cn}{2} \sum_{h=0}^{\log n} \frac{h}{2^h} \leq O\left(\frac{cn}{2} \cdot \left(\sum_{h=0}^{\infty} \frac{h}{2^h}\right)\right)$$

\uparrow

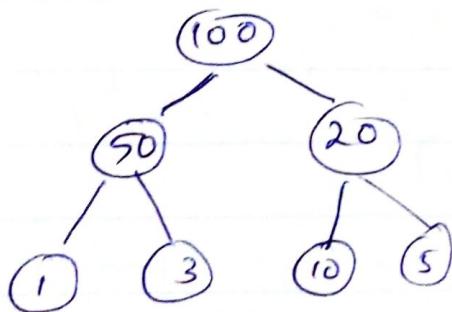
$O(n)$.

Given a max-heap, delete maximum from it, ?

To delete maximum element from heap,

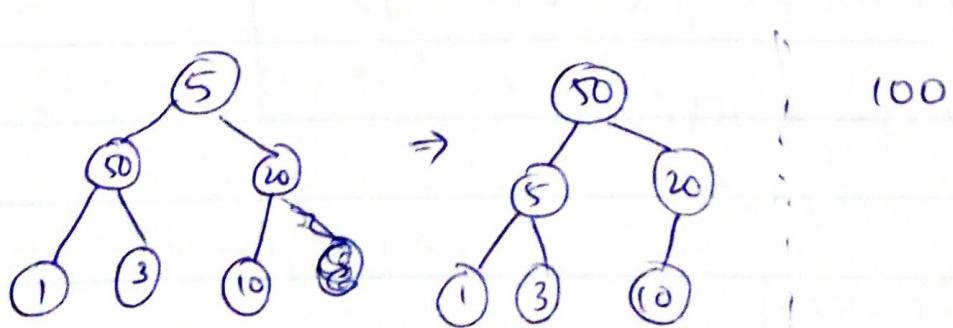
- Take the root node i.e. maximum
- Replace
- Swap the maximum i.e. root node with last element of the heap
- Call the max-heapify function on the root node (It'll again make the entire tree to heap)

Ex:- 100, 50, 20, 1, 3, 10, 5



Step 1 :-

Take '100' out and replace with '5'



Step 2 :- If you continue this for 'n' times, you get the elements in the descending order

Pseudo Code:-

```
Heap-Extract-Max (A)
{
    if A.heap-size < 1
        error "heap underflow"
    max = A[1]
    A[1] = A[A.heap-size]
    A.heap-size = A.heap-size - 1
    MAX-HEAPIFY (A, 1)
    return max
}
```

→ Time Complexity: $O(\log n)$

② Increase Key Value in a Heap:

Heap-Increase-Key (A, i, key):

```
{ if (key < A[i])
    error
```

Time complexity
 $= O(\log n)$.

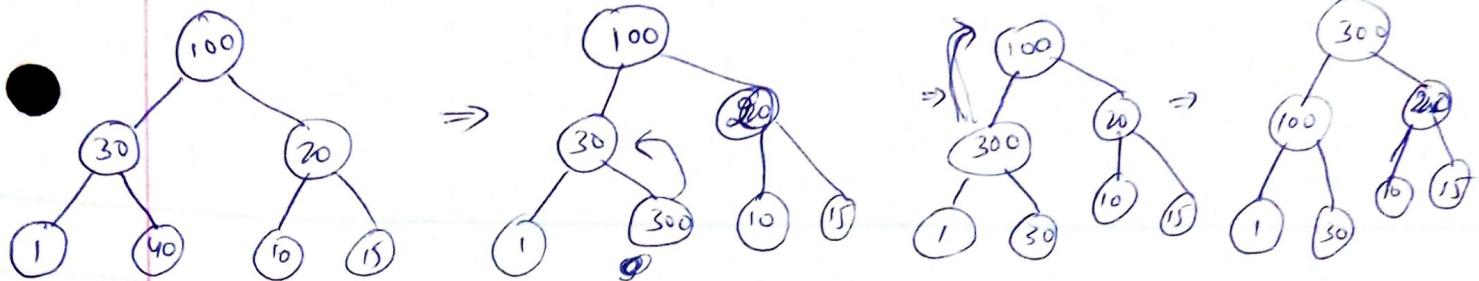
$A[i] = \text{key}$

while ($i > 1$ and $A[i/2] < A[i]$)

exchange $A[i]$ and $A[i/2]$

$i = i/2$

heap-increase-key (A, 5, 300)



→ Decreasing a Key?

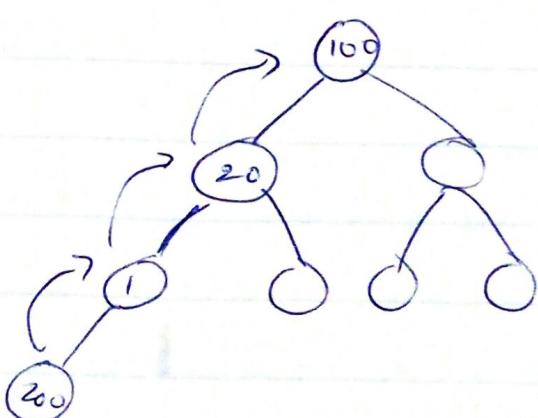
- Decrease a Key
- Call max-heapify on the key.

$$T(n) = O(\log n),$$

→ Insertion in^{MAX.} Heap:

- Insert element at the end
- compare it with 'parent'; if parent > key , swap ; ~~loop~~
- Continue till $i \geq 0$.

$$T(n) = O(\log n)$$



	find max	delete max	insert	increase key	decrease key
max-heap	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$

	find min	delete random element	search random element
no + optimal	$O(n)$	$O(n)$	$O(n)$

④ HEAP-SORT :-

Heap Sort (A)

```

BUILD-MAX-HEAP (A)
for (i=A.length down to 2)
    exchange A[i] with A[1]
    A.heapSize = A.heapSize - 1
    MAX-HEAPIFY (A, 1)
  
```

y

STEPS:-

- Build HEAP
- Take maximum i.e $a[0]$ and put it at the end of the array & Reduce the size of the array(heapSize) by "1".
- Call max-heapify on "first" element.

Heap Sort:

Time Complexity:

- Always max-heapify is applied at root.

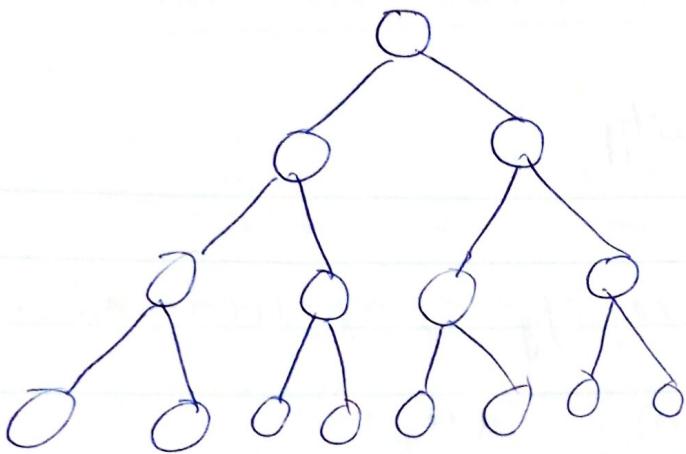
$$\text{So, } T(n) = O(\log n)$$

- If 'n' elements are there, then,

$$T(n) = O(n \log n)$$

→ we cannot argue, as we have done in "BUILD HEAP", because, As we are swapping first element with last element (mostly leaf), and running MAX-HEAPIFY on root. We will be having "log n" as the height of the tree, till (or) atleast $n/2$ times (~~leaves~~ leaves in a C.B.T of 'n' nodes).

$$\Rightarrow \text{So, } T(n) = O\left(\frac{n}{2} \log n\right) \Rightarrow O(n \log n)$$



} $n/2$ nodes

↓
i.e., " $n/2$ " times tree
height will be "log n".