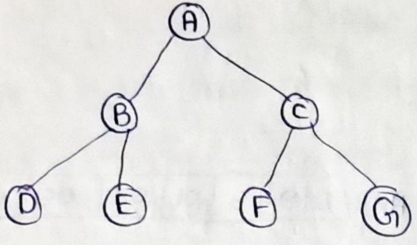
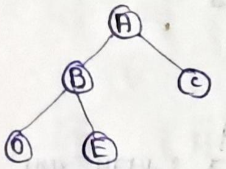
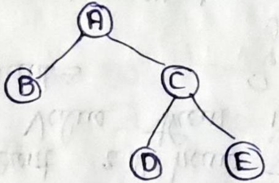
ARRAY REPRESENTATION OF B.T



T	A	B	C	D	E	F	G
	1	2	3	4	5	6	7

if a node is at index i
Left child at $2i$
Right child at $2i+2$
Parent node at $\left\lfloor \frac{i}{2} \right\rfloor$





T	A	B	C	D	E

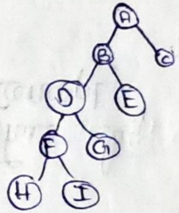
T	A	B	C	-	-	D	E

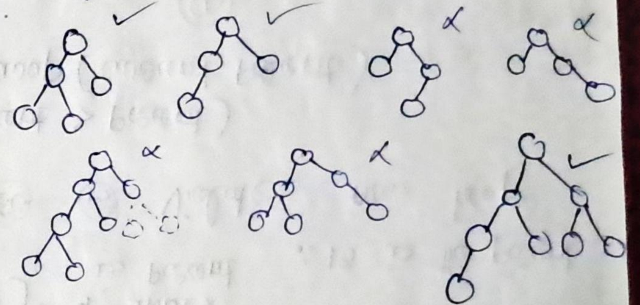
we fill level by level.

FULL & COMPLETE BINARY TREE

full binary tree ⇒ B.T with maximum no of nodes.
Complet B.T ⇒ Each node must have 0 or 2 nodes and are arranged as left as possible. → elements filled from left to right.

But, (another def) while doing array rep: there should be no gaps.





A B C D E F G H I ✗

A	B	C	D	E	-	-	F	G	-	-	-	-	-	-	H	I

✓ - COMPLET BT

The def. goes like Comp BT is a full BT until node h-1 and filled from as left as possible.

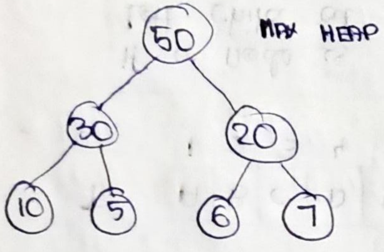
height of comp BT ⇒ log n. } min. !

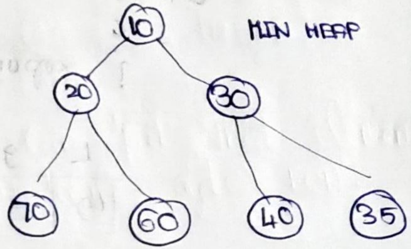
MAX & MIN HEAP :
+ HEAP :

We have 2 types of heap max & min.

MAX HEAP : Each parent will have the highest value than its descendants } for COMPLETE BT.

MIN HEAP : Each parent will have lower value than its decendands.



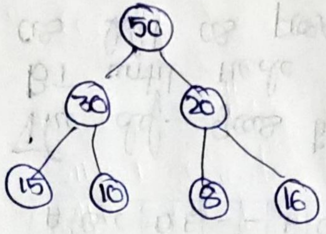


H	50	30	20	10	5	6	7

H	10	20	30	70	60	40	35

INSERTION IN HEAP

@ free space



H:	50	30	20	15	10	8	6	60
	1	2	3	4	5	6	7	8

END OF HEAP

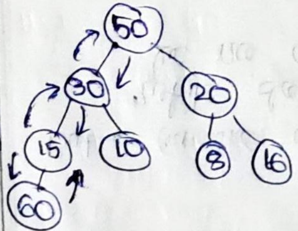
- The new element is inserted at END OF HEAP array. ie) it is a child of leftmost child.

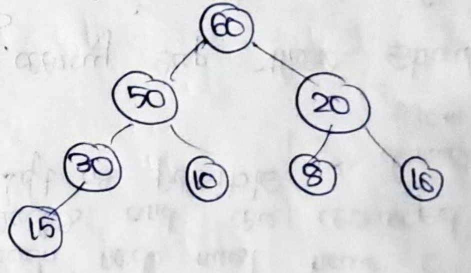
INSERT - 60

$\left\lfloor \frac{8}{2} \right\rfloor = 4$th INDEX is Parent. ∴ 15 is the parent.

Thus by adding 60 — 15 violates max heap concept.

```
while (current > parent)
    swap (current, parent).
```





Time taken for ins?

@ MAX how many swaps?

↳ ht. of tree swaps

WKt, CBT have $h = \log N$

∴ TC : $O(\log N)$ → worst case

↳ $O(1)$ to $O(\log N)$

* Insertion happened from leaf to the root.

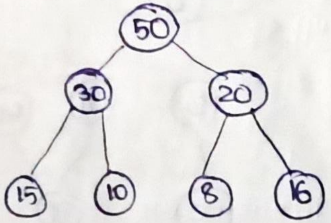
⊙ Direction of adjustment is upward ↗

DELETION IN HEAP DS

Del can happen oly at top.

Top of the heap is deleted.

THAT'S HOW IT IS DESIGNED



→ 50 should be del.

⊙ It should be ~~del~~ replaced by the rightmost leaf on the array rep.

H						
50	30	20	15	10	8	16

∴ 16 will take 50's position.

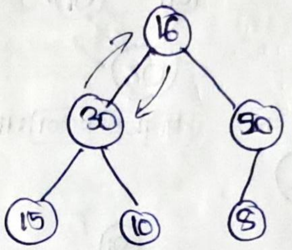
↑50

	30	20	15	10	8	16

16	30	20	15	10	8	

END OF HEAP.

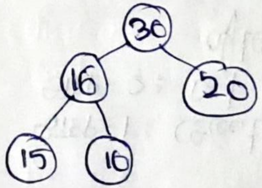
So now we have preserved Complete BT property.



} Complete BT but not Max heap.

⊛ While del, we go DOWNWARDS ↙ from ROOT → LEAF

```
While (gchild > current)   // gchild is the
    Swap(current, child)   * greatest child
                             among the 2
                             child.
```



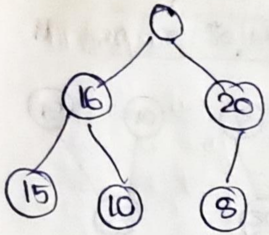
H :				
30	16	20	15	10

TC : $O(\log n)$, adjustment swaps log n times

So,

* del each time we get the largest no in the heap.

↳ SMALLEST @ MIN HEAP



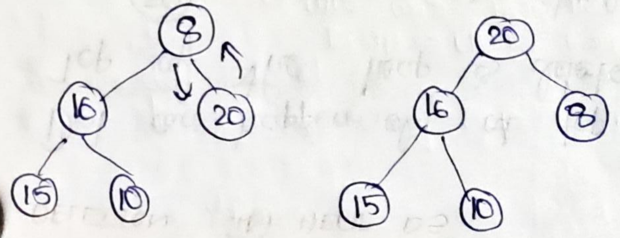
Del TOP.

30	16	20	15	10	8	50

EOH

8	16	20	15	10	30	50

EOH



* After deletion we can put that element after the space available after EOH.

* Swap element is the rightmost or the "element before EOH".

Then after that we swap until we reach max/min heap.

HEAP SORT

→ So the array space is not wasted and after N deletions we get sorted array.

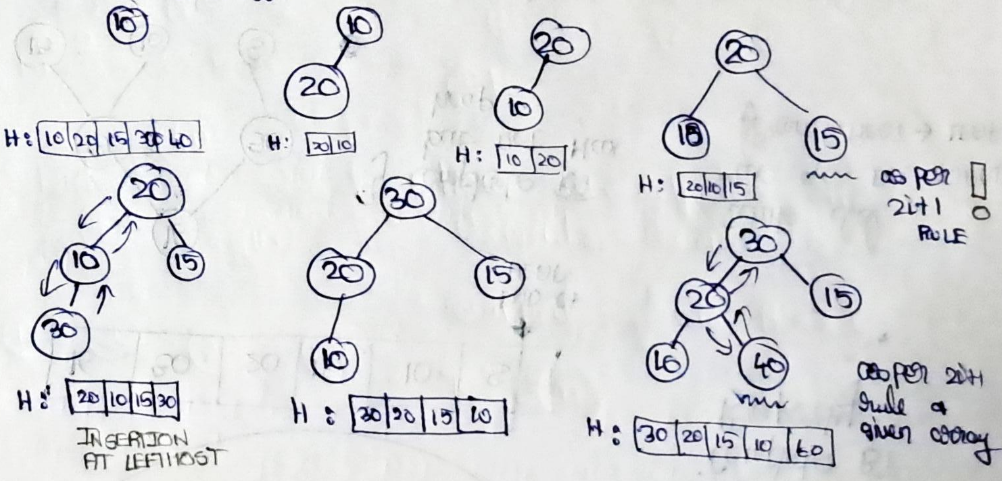
This is HEAP SORT.

STEP-1 : Create heap

STEP-2 : Del heap and store it after EOH.

10	20	15	30	40

CREATE HEAP DS:



H: 10 | 20 | 15 | 30 | 40

H: 20 | 10

H: 10 | 20

H: 20 | 10 | 15

as per 2i+1 rule

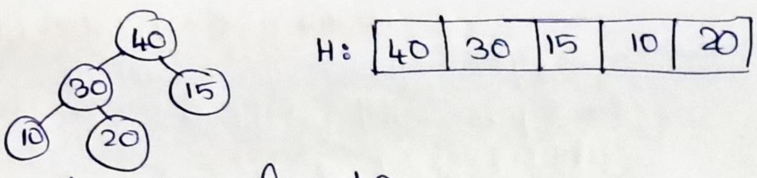
H: 20 | 10 | 15 | 30

INSERTION AT LEFTMOST

H: 30 | 20 | 15 | 10

H: 30 | 20 | 15 | 10 | 60

as per 2i+1 rule & given array



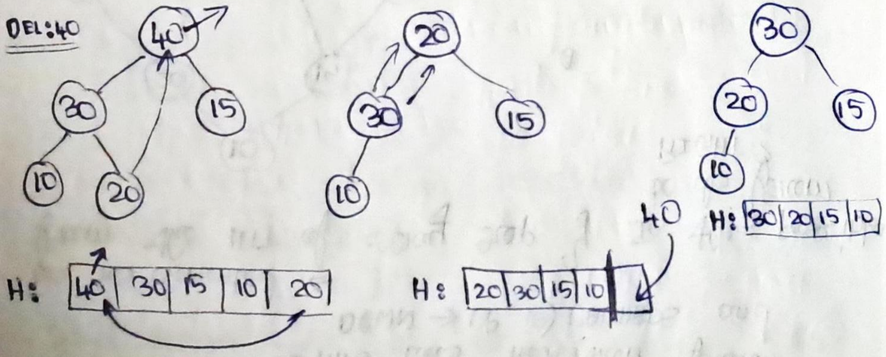
We have n elements.

WKt, TC : O(logn) for single insertion

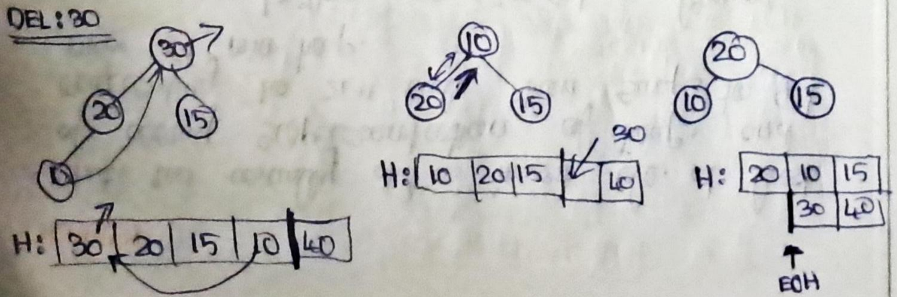
∴ TC for creating a BST : O(nlogn)

DELETION & SORTING :

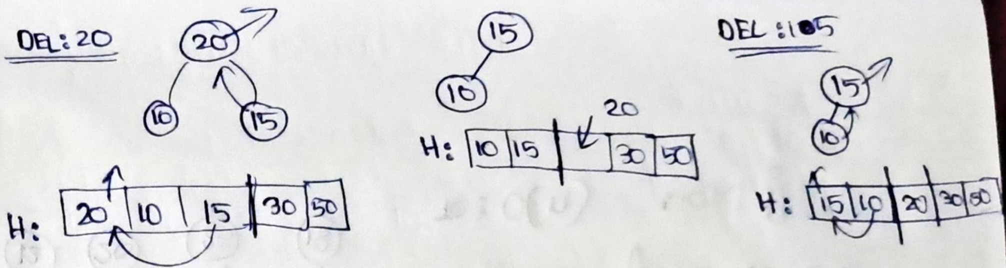
DEL:40



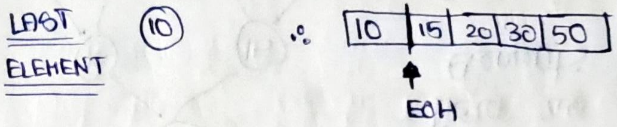
DEL:30



DEL:20 DEL:15



LAST ELEMENT



} Thus we got a sorted array.

WKt, del TC : O(logn)

Thus del of n element ⇒ TC : O(nlogn)

↳ Coz n elements.

∴ Heap sort takes nlogn time.

What is heapify ?

↓

It is also a process of creating a HEAP.

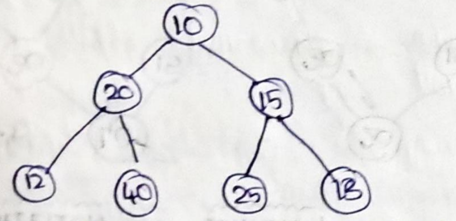
HEAPIFY

* What we usually did is insert ele. at last of array representation of heap. and according to 2i+1 rule then swaps to get max or min heap.

New one is from Right of EOH

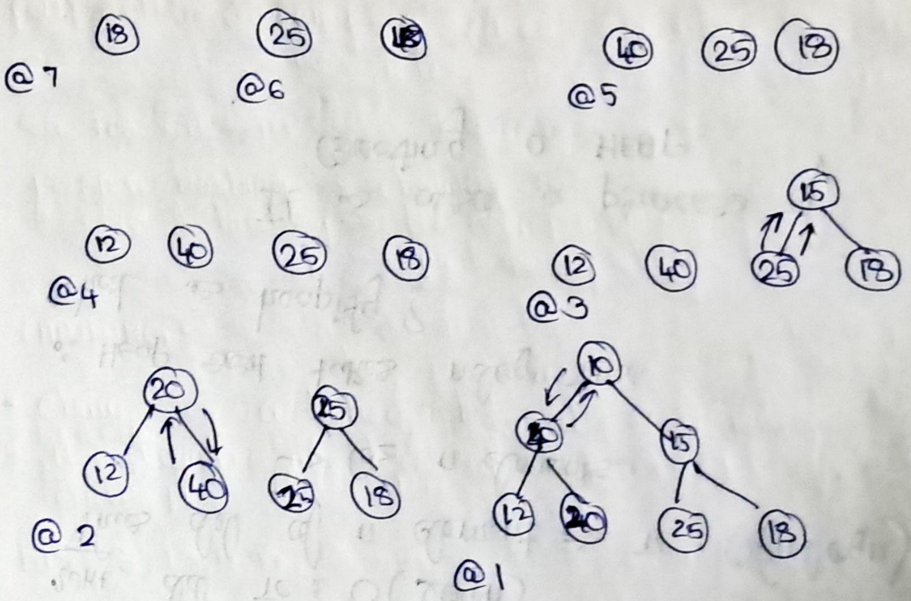
↳ This was insertion from DOWN → UP ie) UPWARDS and

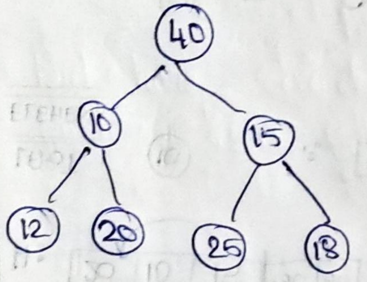
We ins elemented ↵ from the LEFT of array rep. } Is this possible to do from RIGHT ?



H:	10	20	15	40	25	18
	1	2	3 4	5	6	7

|| if no parent create node
if child connect the childs





Here we adjusted the elements downwards. ~ del

* scanned from R → L

T : O(n) WOW !