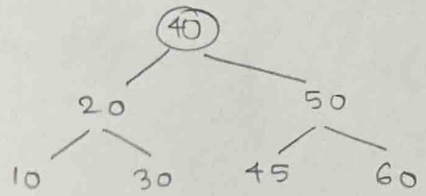
AVL TREES

(1) What is BST ?
(2) Drawbacks of BST
(3) How BST improve ?
(4) What is an AVL tree ?
(5) Rotations ?
(6) How to create AVL ?

BST



Useful for searching

O(height)

if h=3 : O(3)

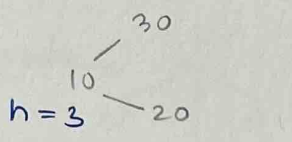
HEIGHT : min : log n
max : n

On given array (order of elements) we get diff. BST

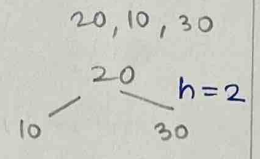
Drawback of BST

For same elements, but different order we get BST of different height

eg 30, 10, 20



20, 10, 30



HOW TO IMPROVE ?

Rotate the BST

→ rotations are done only on 3 nodes

AVL TREE

height balanced BST

Balance Factor = Height of left subtree − Height of right subtree

$bf = h_l - h_r = \{-1, 0, 1\}$

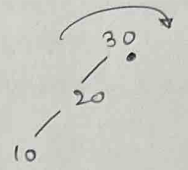
$|bf| = |h_l - h_r| \leq 1$

if $|bf| > 1$: imbalanced

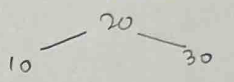
$|bf| <= 1$: balanced

ROTATION

*

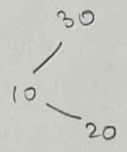


After rotation



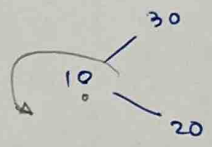
LL imbalance ⇒ LL rotation

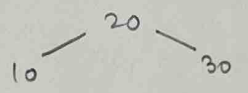
*



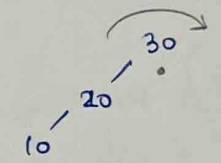
LR imbalance

2 step rotation

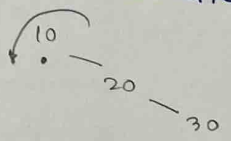


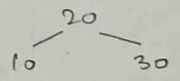


LR rotation

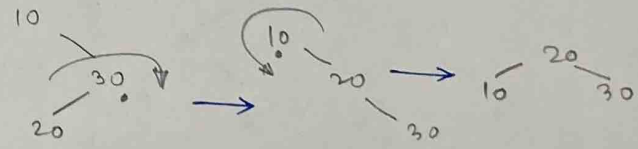


* RR rotation

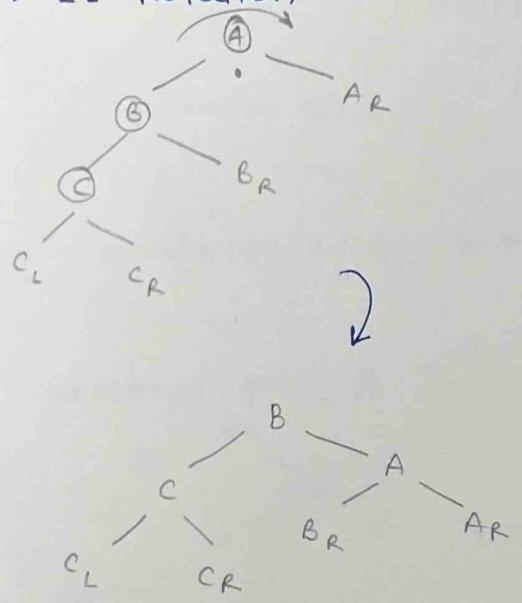




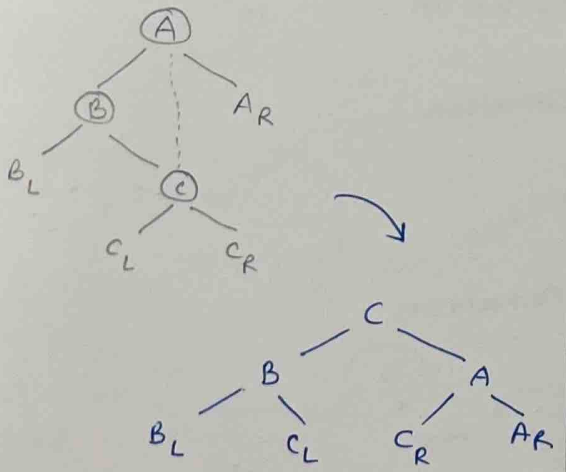
* RL rotation



* LL Rotation



* LR ROTATION

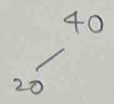


Q. How AVL tree is generated ?

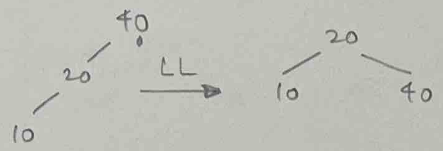
order : 40, 20, 10, 25, 30, 22, 50

(1) 40

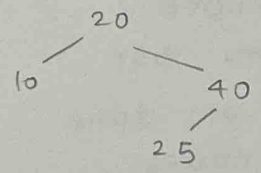
(2)



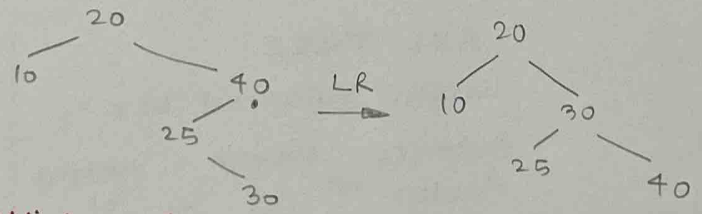
(3)



(4)

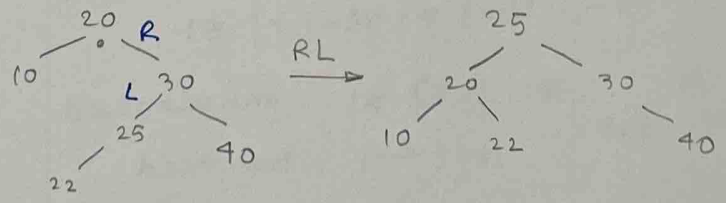


(5)

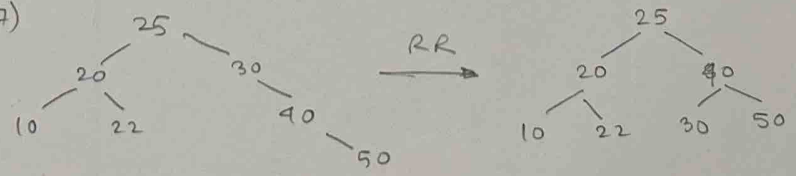


if multiple nodes are imbalance, start with lower one

(6)

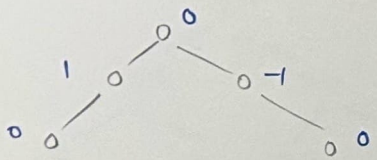


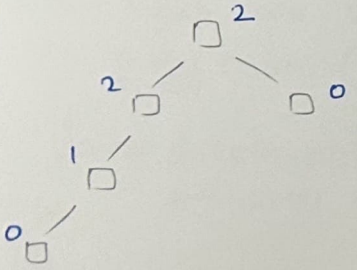
(7)

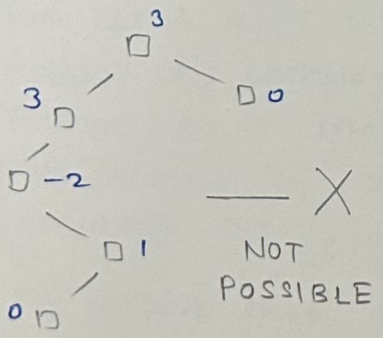


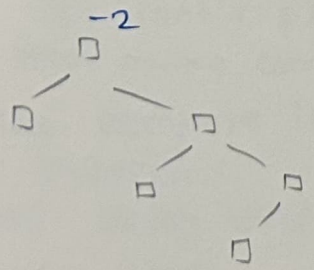
as soon as a node becomes imbalance, perform rotation dont wait for other keys to get inserted and then perform rotation

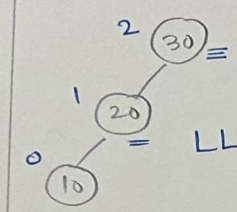
AVL



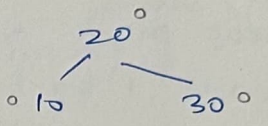








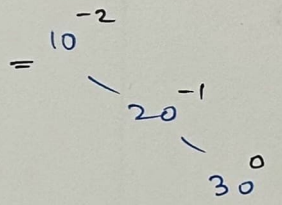
LL imbalance



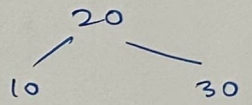
LL rotation
clockwise rotation

$\Longrightarrow$ we can also call it as RR

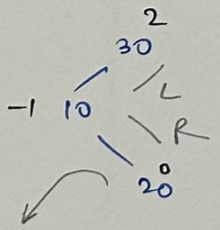
Rotation : performed on 3 nodes ONLY !



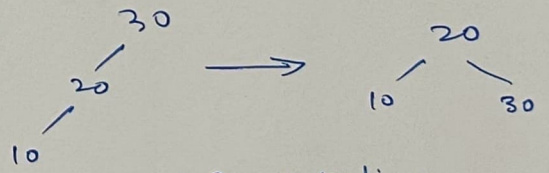
RR imbalance



RR rotation
or anti-clockwise rotation
or LL

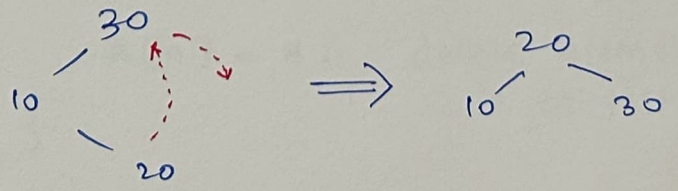


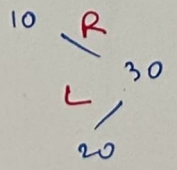
LR imbalance



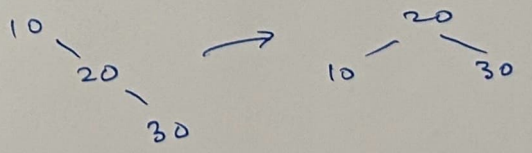
LR rotation

or





RL imbalance



$n = 3$

10 20 30

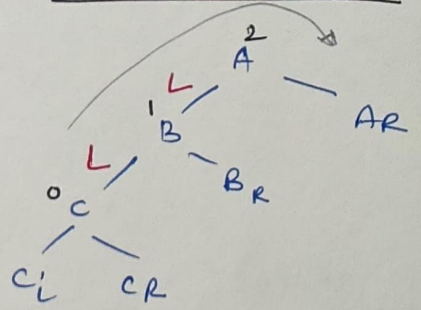
Possible BST $= {}^{2n}C_{n-1}$

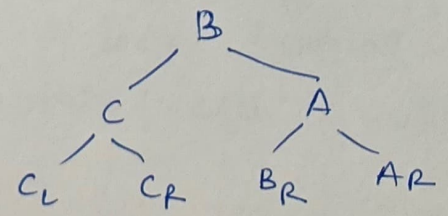
$${}^{6}C_{2} = \frac{6!}{4!\,2!}$$

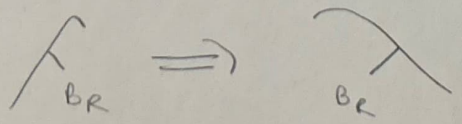
$= 5$

For AVL, we want only 1 BST that's height balanced

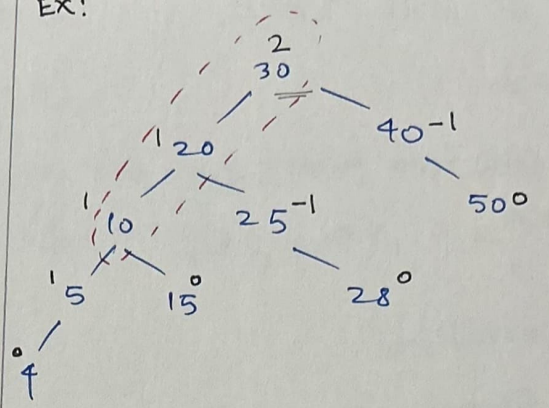
LL rotation





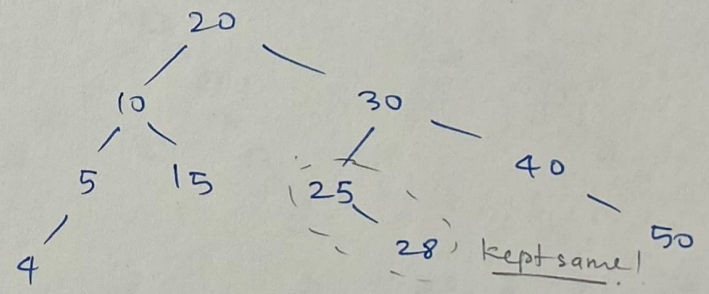


Ex:

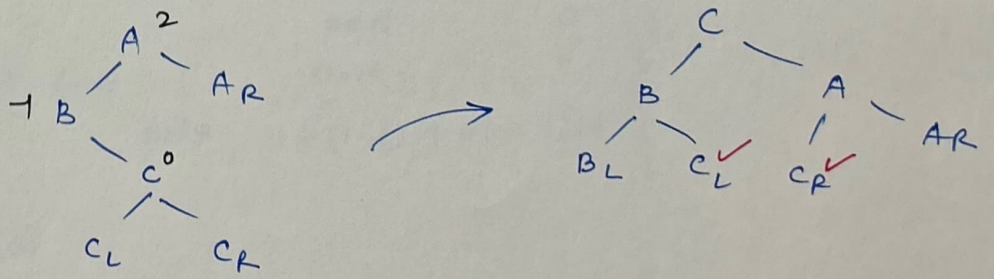


last insertion 4

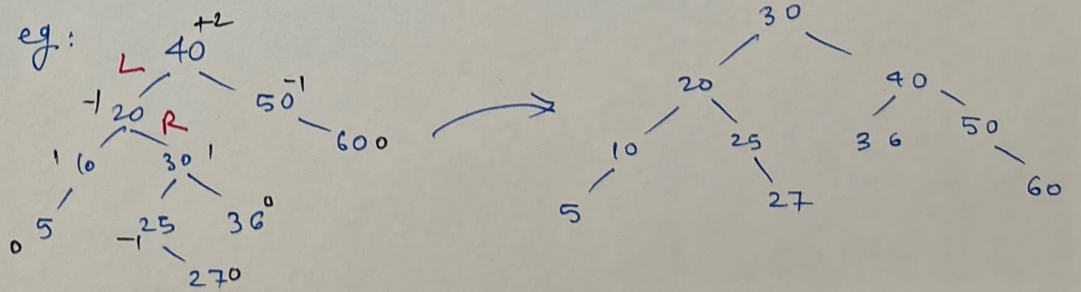
so, 30 became imbalance



LR ROTATION



eg:



HEIGHT ANALYSIS OF AVL TREE

height from 1 (assumption for formulas)

if height is given, find

min. nodes, $n =$ look into table

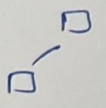
max. nodes, $n = 2^{h} - 1$ (From BT)

"Formula come from observation"

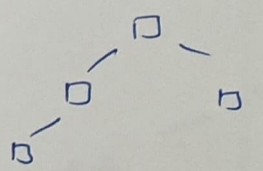
we assume height from 1

h=1 (MIN) n=1

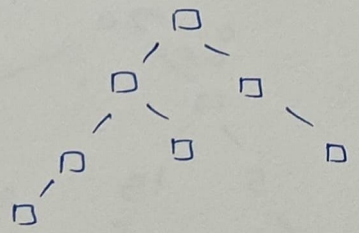
h=2 n=2



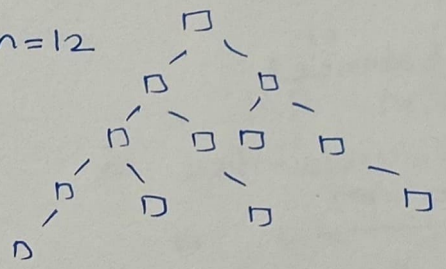
h=3 n=4



h=4 n=7



h=5 n=12



h	1	2	3	4	5	6	7
N(h)	1	2	4	7	12	(7+12+1) = 20	20+12+1 = 33

(MIN)

$$N(h) = \begin{cases} 0 & h = 0 \\ 1 & h = 0 \\ N(h-2) + N(h-1) + 1 & \text{else} \end{cases}$$

$\Longrightarrow$ Similar to fibonacci series

$\Downarrow$

Balanced series

So, we say AVL is also balanced

if 'N' nodes are given, find

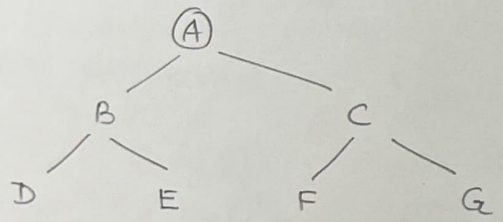
min. height $h = \log_2(n+1)$

max height $h =$ look in the table

$\approx 1.44 \log_2(n+2)$

HEAP

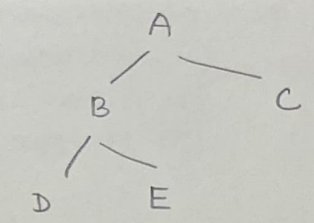
(1) Array representation of BT
(2) Complete BT
(3) Heap
(4) Insert and Delete
(5) Heapsort
(6) Heapify
(7) Priority Queue



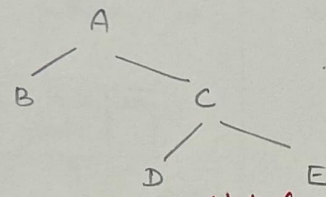
T:	A	B	C	D	E	F	G
	1	2	3	4	5	6	7

if a node is at index i
its left child is at : $2i$
right child : $2i+1$
parent : $\lfloor \frac{i}{2} \rfloor$

$\lfloor 3.5 \rfloor = 3$



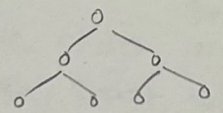
T:	A	B	C	D	E
	1	2	3	4	5



Not Complete BT

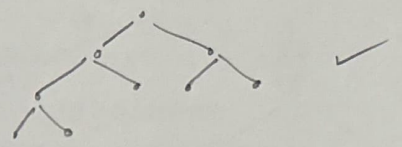
T:	A	B	C	-	-	D	E
	1	2	3	4	5	6	7

FULL BINARY TREE



It is BT with all possible nodes $(2^{h+1} - 1)$

COMPLETE BT



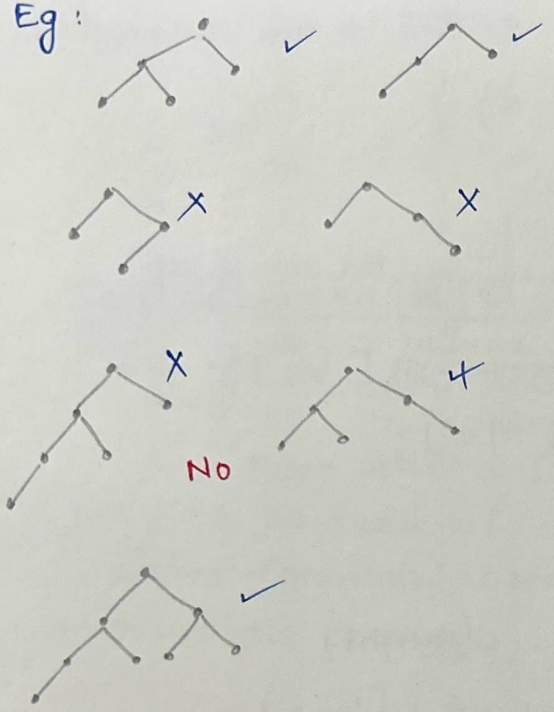
When represent this in array, there should not be any empty gaps (gaps in between elements)

⇒ ALL full BT are complete BT

A Complete BT of height 'h' is a full BT upto height h-1 and then elements starts filling left to right.

Height of a complete BT is $\log n$.

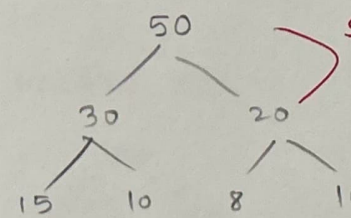
Eg :



HEAP

It is a complete BT

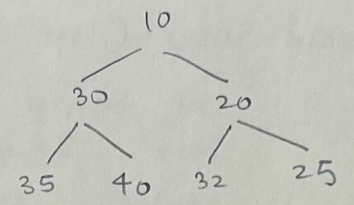
Max Heap :-



	1	2	3	4	5	6	7
H :	50	30	20	15	10	8	16

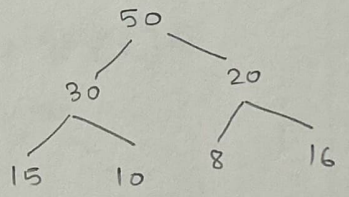
duplicates are ALLOWED

MIN HEAP



	1	2	3	4	5	6	7
H :	10	30	20	35	40	32	25

INSERT



Now we want to insert element 60 in this max-heap

initially :

H : 50 30 20 15 10 8 16

then

H : 50 30 20 15 10 8 16 60

But now it doesnt forms heap

Check its greater than its parent, if so, swap

Flow :-

	1	2	3	4	5	6	7	8
H :	50	30	20	15	10	8	16	60

then

H : 50 30 20 ~~15~~ 10 8 16 15

60

swap

then :

50 60 20 30 10 8 16 15

then :

60 50 20 30 10 8 16 15

O (height)

= O (log n)

∴

O(1) to O (log n)

- Direction of adjustment is upwards.

HEAP ⟹ Complete BT

No space in between (among) elements of array.

↳ duplicates are allowed

Height = $\log n$

:- While inserting element in array, always compare with parent, if needed swap, and increase heap size

if $i = 9$

parent = 4 ie $\lfloor \frac{9}{2} \rfloor$

for indexing that starts with 1.

PSEUDOCODE:

n = new element

```
void insert(int A[], int n) {
    int temp = 0;
    int i = n;
    temp = A[n];
    while (i>1 && temp (>) A[i/2]) {
        A[i] = A[i/2];
        i = i/2;
    }
    A[i] = temp;
}
```

for min heap " < "

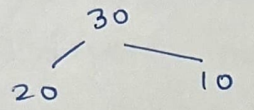
O(height) = $O(\log_2 n)$

start i = n

i = i/2

so, $O(\log_2 n)$

CREATING A HEAP



30	20	10	40	15	22	25

part of heap

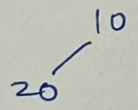
include each element in heap from RHS to LHS

So, no need of extra (another) array.

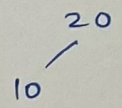
→ inplace change

(1) insert 10 : 10

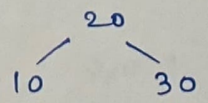
(2) insert 20 :



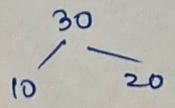
Now, swap :



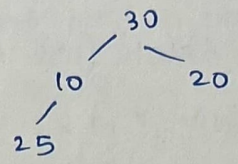
(3) Insert 30



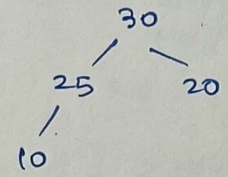
swap :



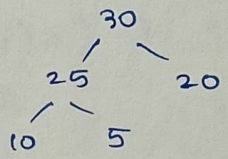
(4) Insert 25



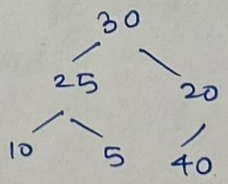
swap



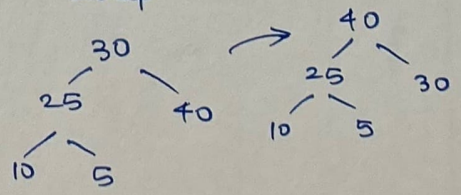
(5) Insert 5



(6) Insert 40

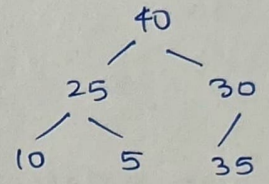


swap

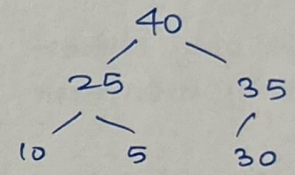


always compare with parent !

(7) insert 35



swap



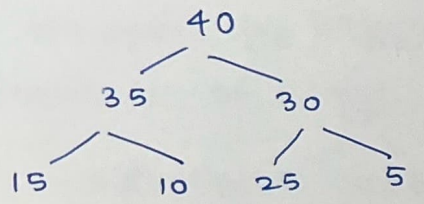
we assume first element to be root !

pseudocode :-

```
void createheap() {
    int A[] = {0, 10, 20, 30 .. }
    int i;
    for (i=2, i<=7, i++) {
        insert(A, A[i]);
    }
}
```

O(nlogn)

DELETE IN HEAP



In heap, we can delete only root. (max or min)

↳ delete only highest priority element

We cant after deletion, adjust by comparing children, as it may not result in a complete BT

(i) Delete root

(ii) Put last ele of array at root's place and then make comparisons!

Now compare it with its children.

Find which child is greater, then swap

pseudocode

```
void delete(int A[], int n) {        // n → size (last index)
    int x, i, j;
    x = A[n];
    A[1] = A[n];
    i = 1
    j = 2*i;
    while (j < n-1) {
        if (A[j+1] > A[j]) {
            j = j+1;
        }
        if (A[i] < A[j]) {
            swap(A[i], A[j]);
            i = j;
            j = 2*j;
        }
        else {
            break;
        }
    }
    A[n] = x;        // delete element and place it as last pos.
}
```

HEAPSORT

1) create heap of 'n' element $\Longrightarrow n \log n$

2) delete node 1 by 1 $\Longrightarrow n \log n$

So, $T = 2n \log n$

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