UNIT-2

AVL Trees : Max height of AVL Tree, Insertions & Deletions. 2-3 Trees : insertion & deletion.

Binary Heaps :- creation Min & Max heap tree, Implementation of insert & delete in to heap.

AVL Tree :- Max height of AVL Tree is $1.44 \log n$ (or) $\log P$

→ AVL Trees are special kind of binary search trees.

→ In AVL trees, height of left subtree & right subtree of every node differs by at most one

→ AVL trees are also called as self-balancing binary search trees.

AVL tree can be defined as height-balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right subtree from that of its left subtree.

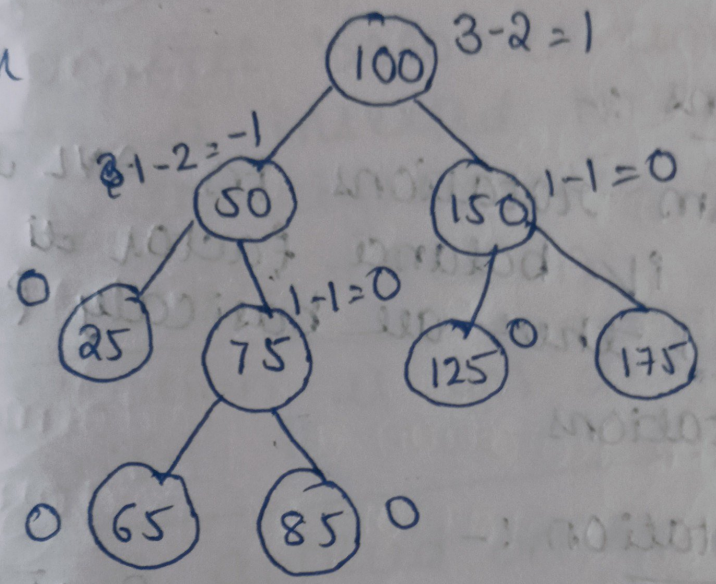
→ Tree is said to be balanced if balance factor of each node is in b/w -1 to +1 otherwise the tree will be unbalanced & need to be balanced

Balance factor (K) = height (left(K)) - height (right(K))

→ If balance factor of any node is 0 it means that the left subtree & right subtree contains equal height.

→ If balance factor of any node is -1 it means that the left subtree is one level lower than the right subtree.

Ex



Operations on AVL tree :-

Insertion :-

Insertion in AVL tree is performed in the same way as it is performed in a binary search tree. However it may lead to violation in the AVL tree property & therefore the tree may need balancing. The tree can be balanced by applying rotations.

Deletion :-

Deletion can also be performed in the same way as it is performed in a binary search tree. Deletion may also disturb the balance of the tree therefore various types of rotations are used to rebalance the tree.

AVL Rotations :-

We perform rotations in AVL tree only in case if balance factor is other than -1, 0, & 1. There are basically four types of rotations

1. **LL rotation :-**

   Inserted node is in left subtree of left subtree of A

2. **RR rotation :-**

   Inserted node is in right subtree of right subtree of A.

3. **LR rotation :-**

   Inserted node is in right subtree of left subtree of A

4. **RL rotation :-**

   Inserted node is in the left subtree or right subtree of A

where node A is the node whose balance factor is other than -1, 0, 1

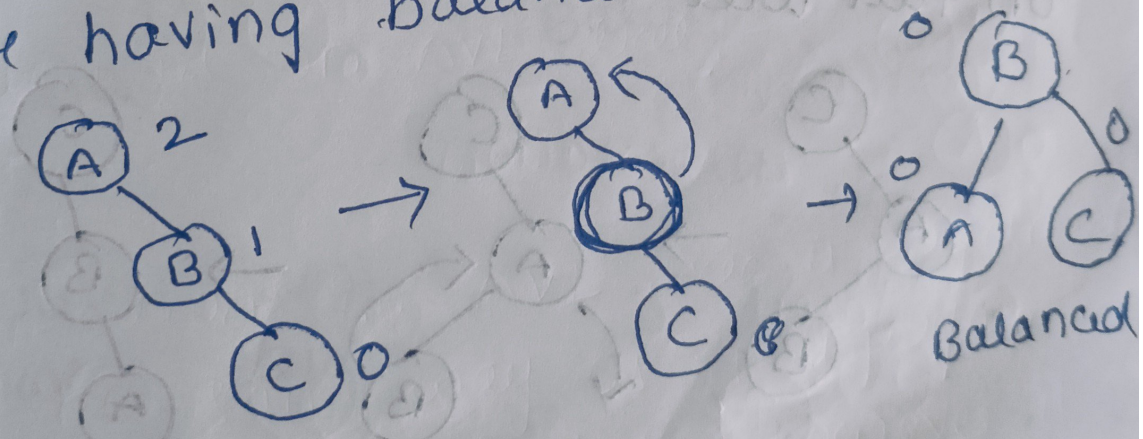
The first two rotations LL & RR are single rotations & the next two rotations LR & RL are double rotations for a tree to be unbalanced, min height must be at least 2,

1. RR rotation :-

when tree becomes unbalanced, due to node is inserted in the right subtree of the right subtree of A then we perform RR rotation

> RR Rotation is an anticlockwise rotation

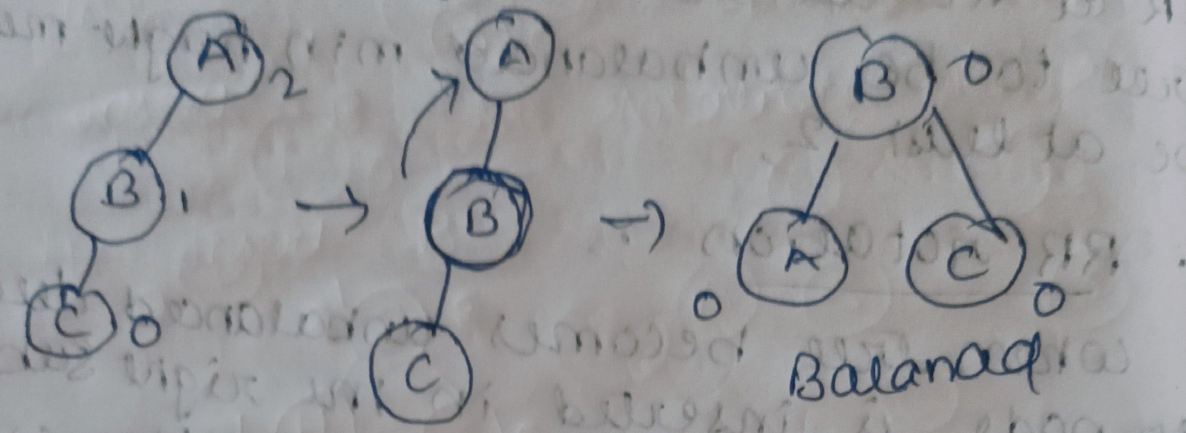
which is applied on the edge below a node having balance factor -2



2. LL rotation :-

when tree becomes unbalanced due to a node is inserted in to the left subtree of the left subtree of A then we perform LL rotation

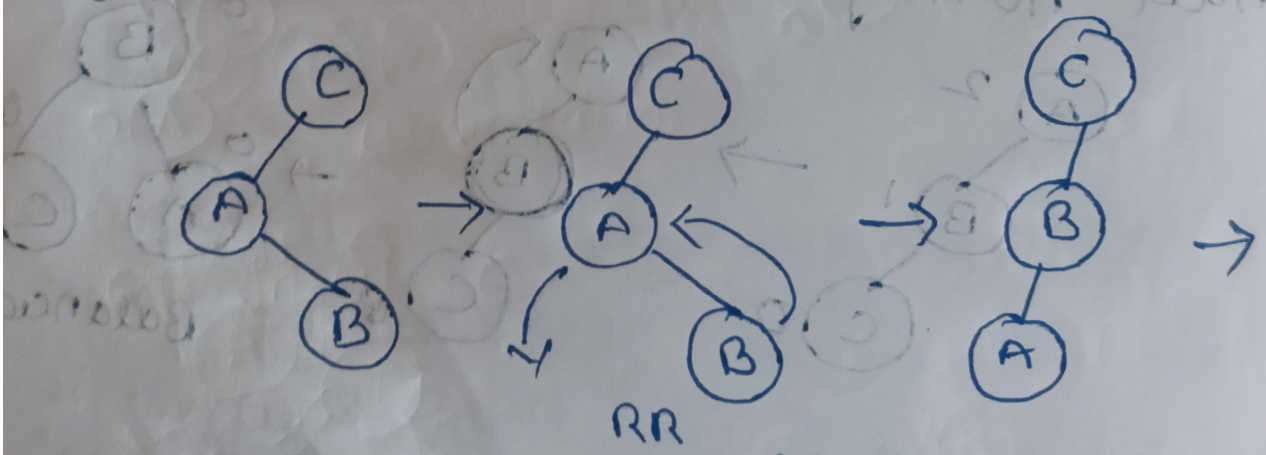
LL rotation is clockwise rotation which is applied on the edge below a node having balance factor 2

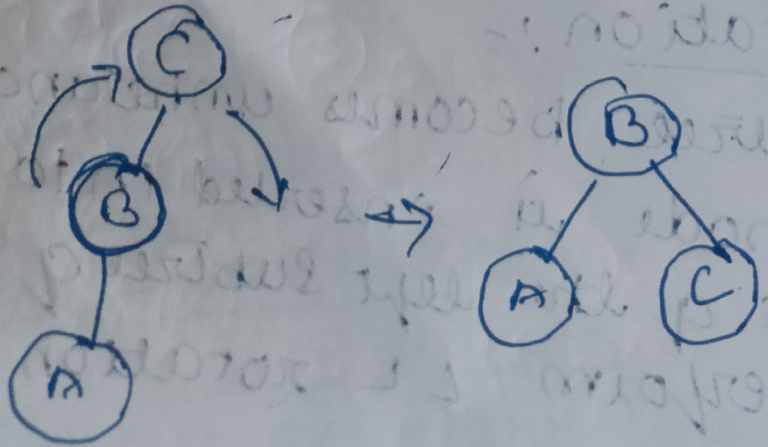


LR Rotation :-

LR rotation = RR rotation + LL Rotation

First, RR Rotation is performed on subtree & LL rotation is performed on full tree

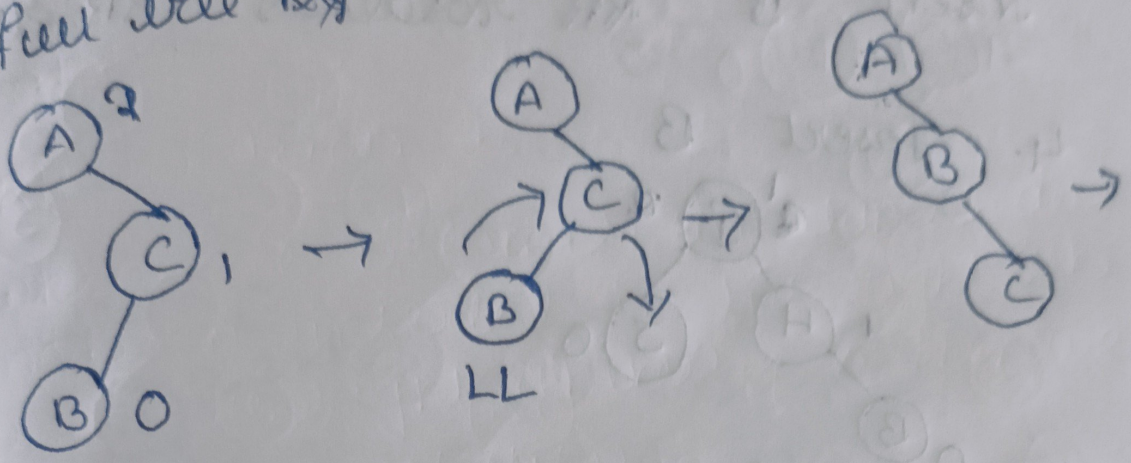


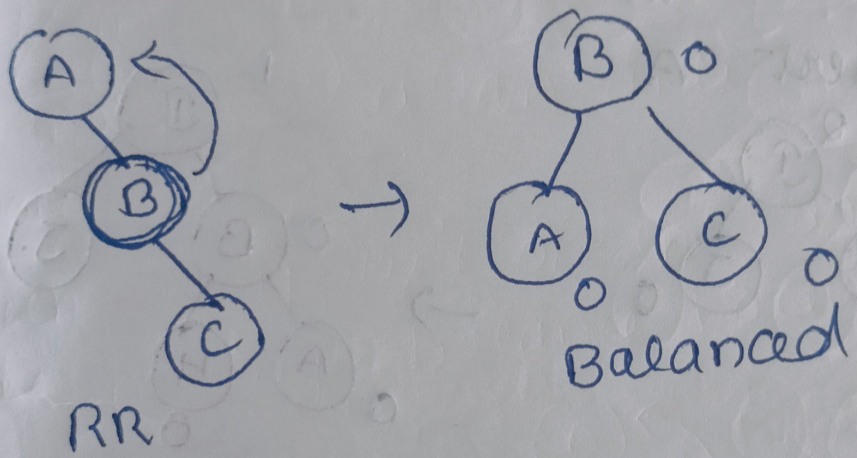


RL Rotation :-

RL rotation = LL rotation + RR rotation

First LL rotation is performed on subtree & then RR rotation is performed on full tree by





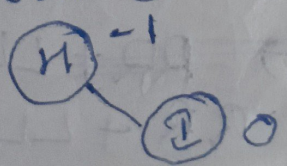
Ex:-

Construct an AVL Tree having the following elements

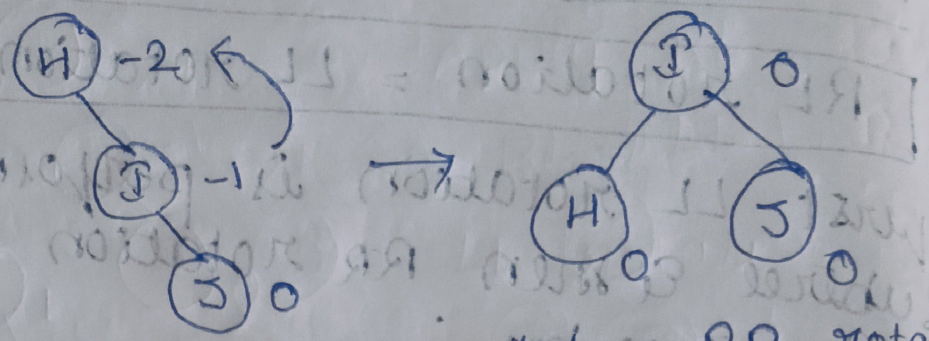
H, I, J, B, A, E, C, F, D, G, K, L

1. Insert H

2. Insert I

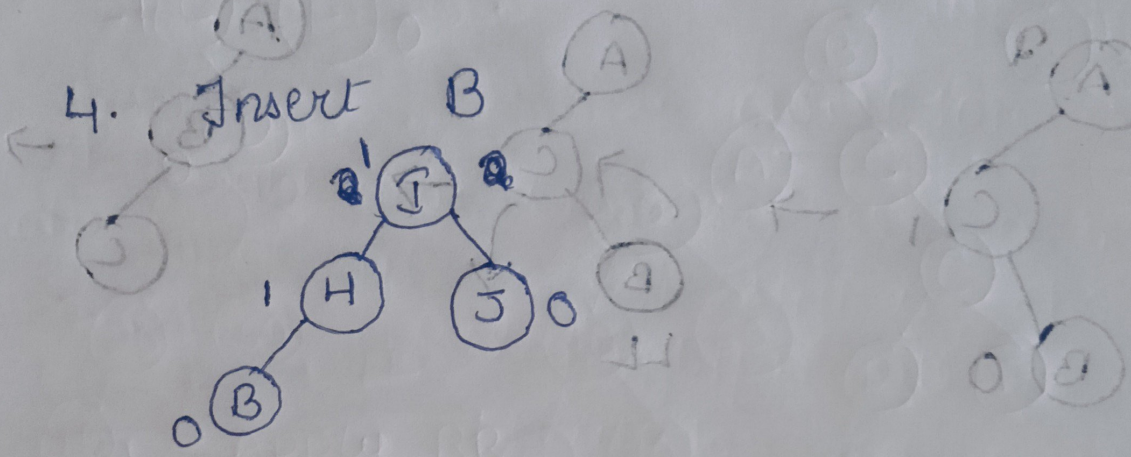


3. Insert J

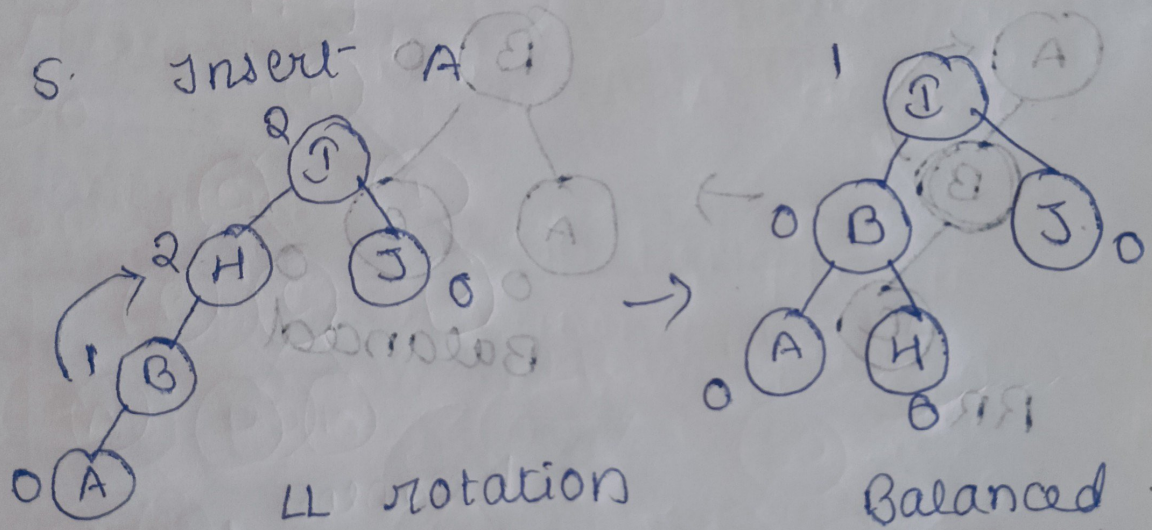


Insert J RR rotation BR rotation

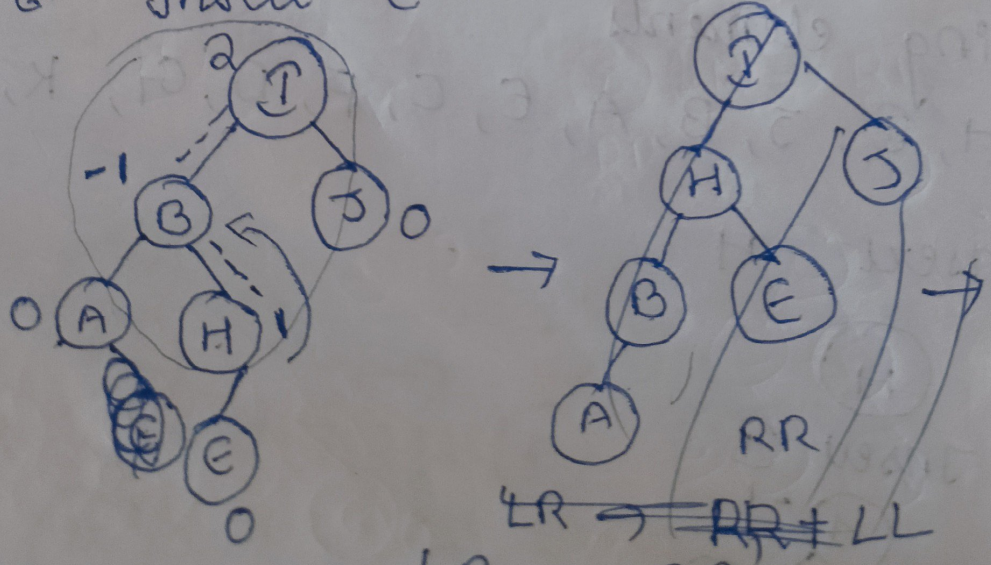
4. Insert B



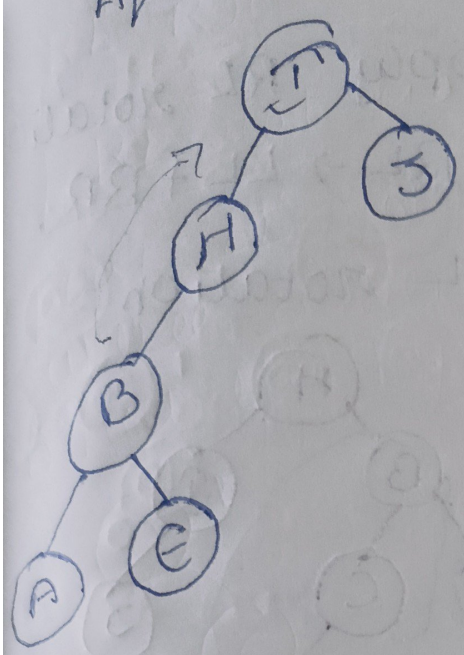
5. Insert A



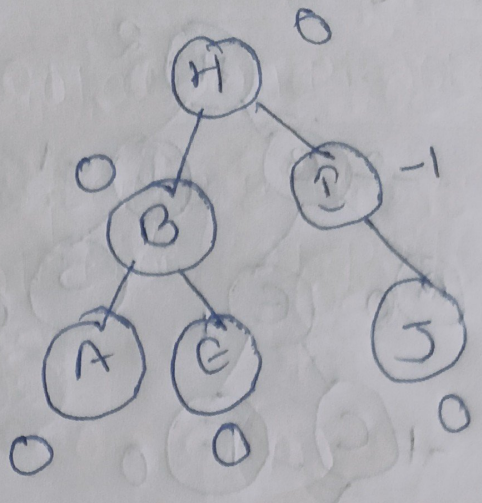
6. Insert E



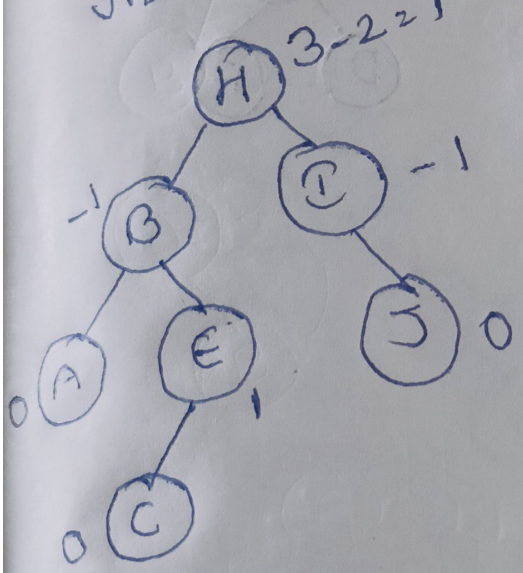
After RR



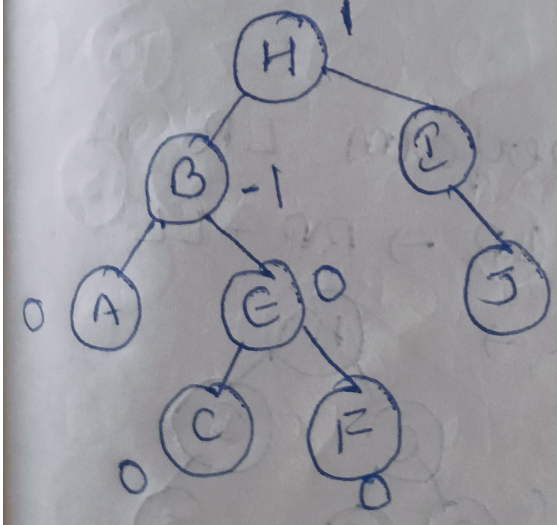
After LL



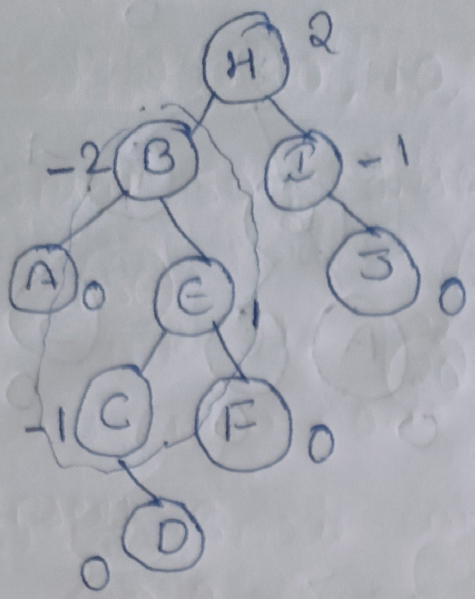
Insert C



Insert F



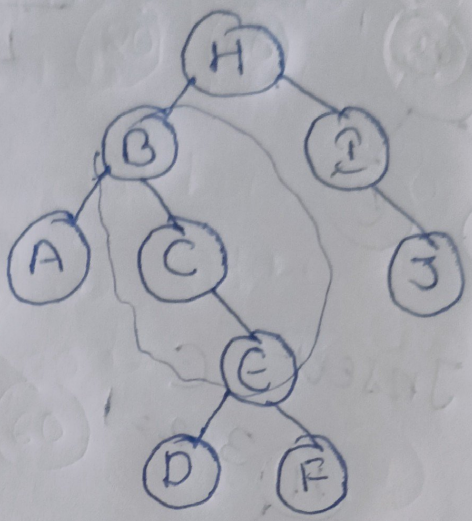
Insert D



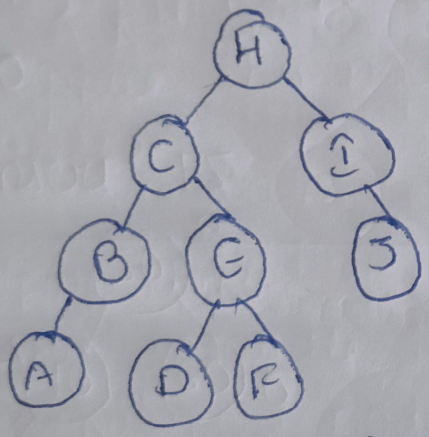
Apply RL rotation

RL → LL + RR

LL rotation

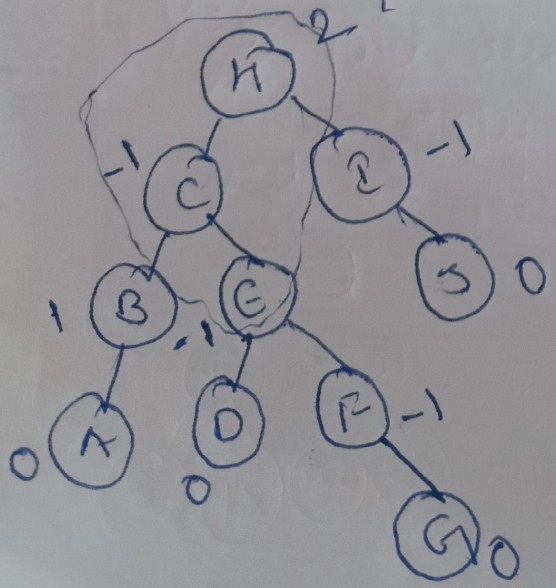


Apply RR



Balanced

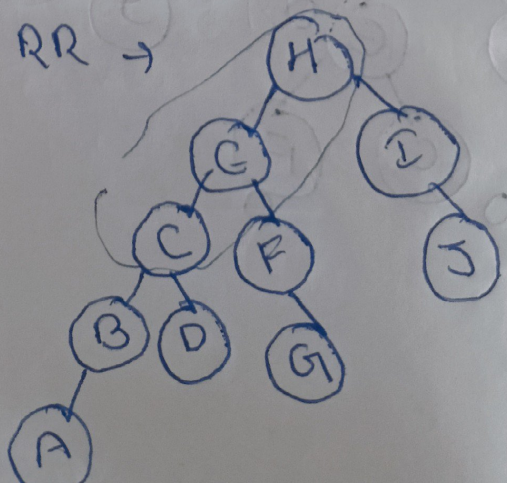
Insert G



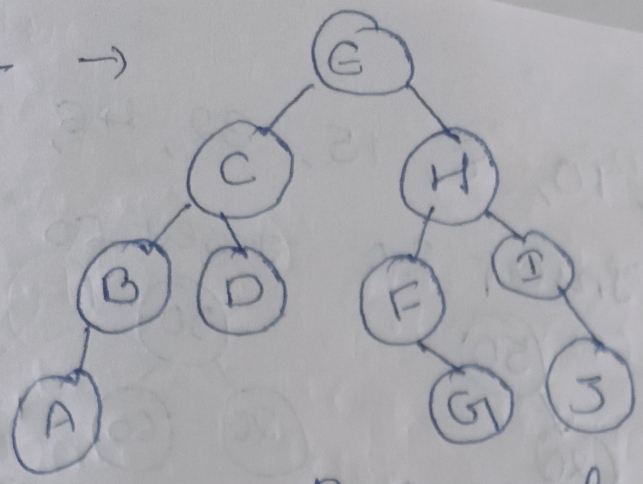
perform LR

LR → RR + LL

RR →

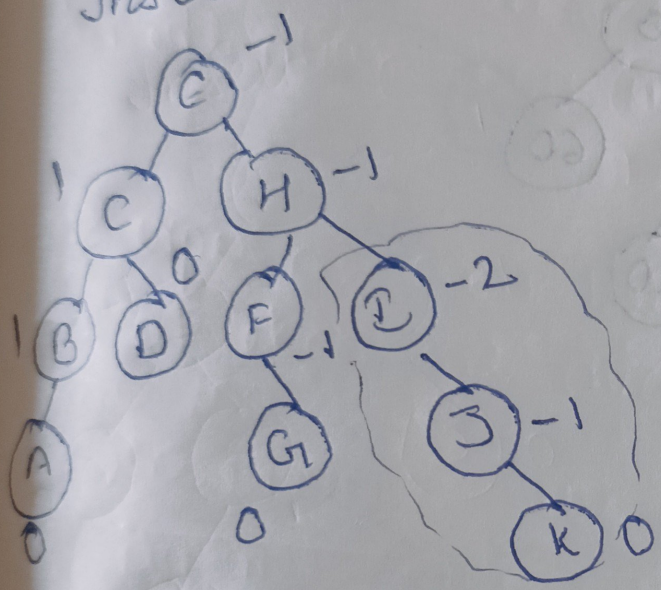


LL →

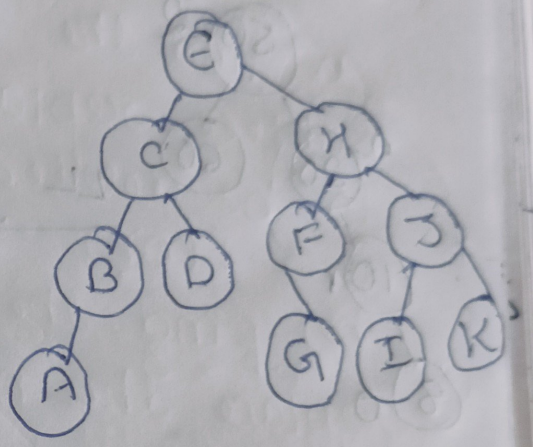


Balanced

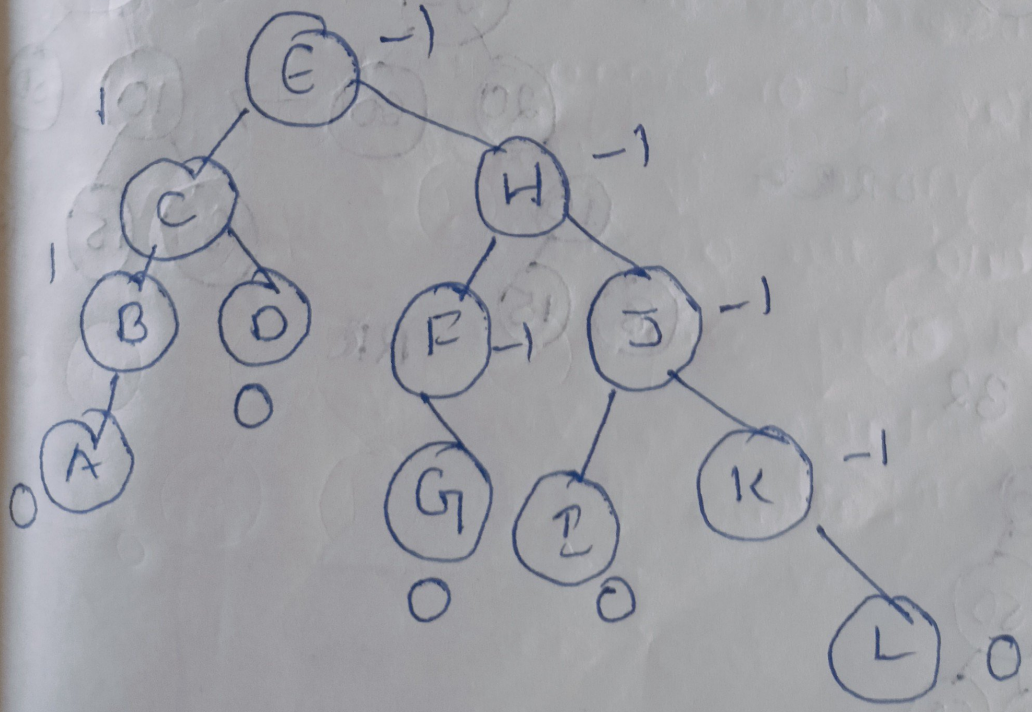
Insert K.



LL →



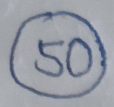
Insert L



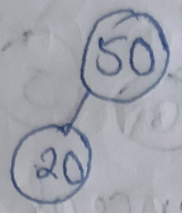
Example: 2

50 20 60, 10, 8, 15, 32, 46, 11, 48

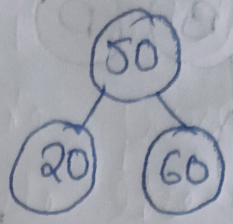
Insert 50



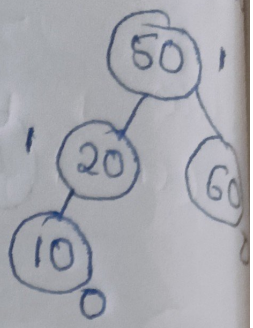
Insert 20



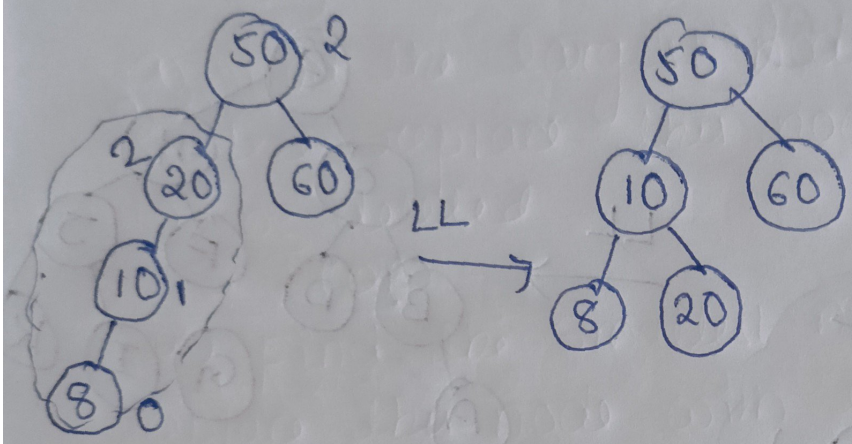
Insert 60



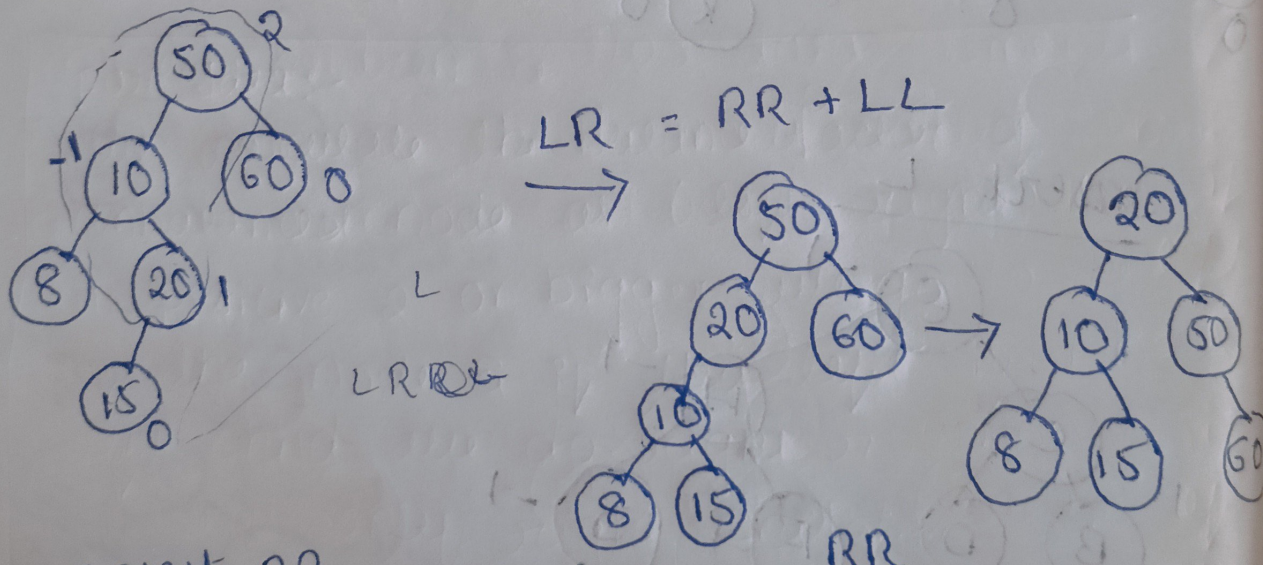
Insert



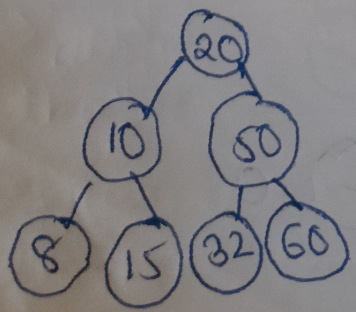
Insert 8



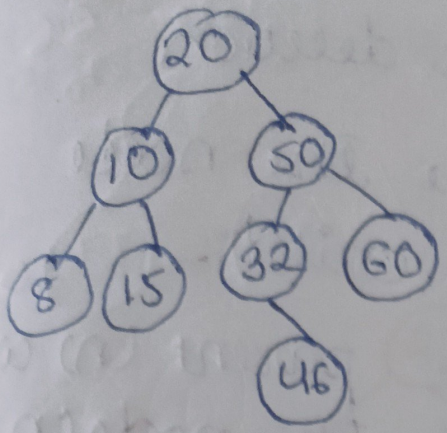
Insert 15



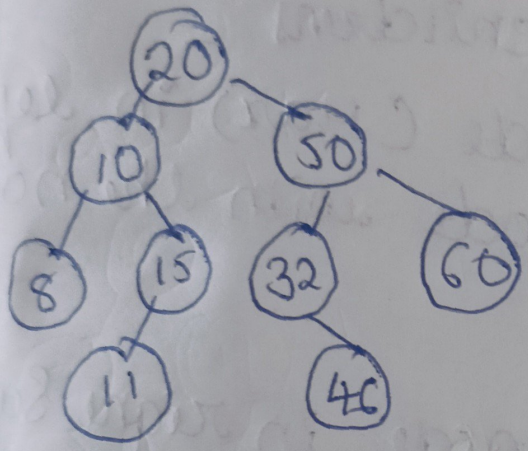
Insert 32



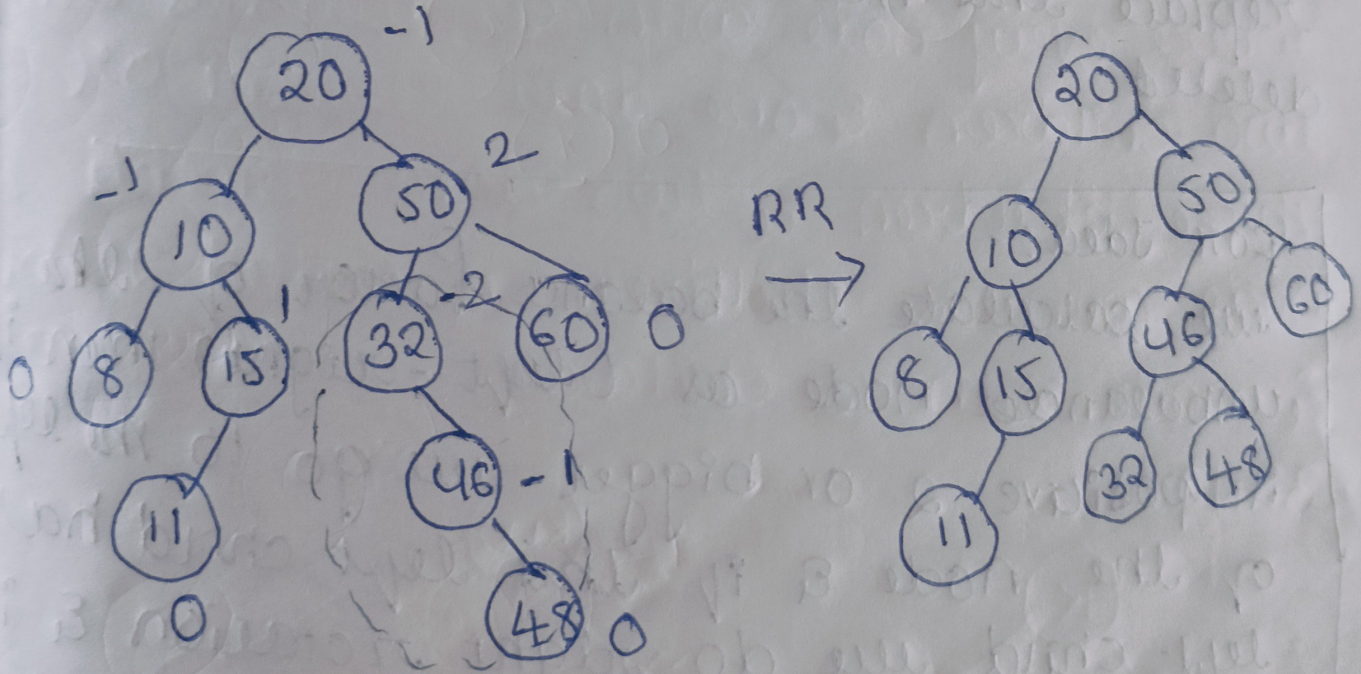
Insert 46



Insert 11



Insert 48



Deletion :-

3 different use cases for deletion

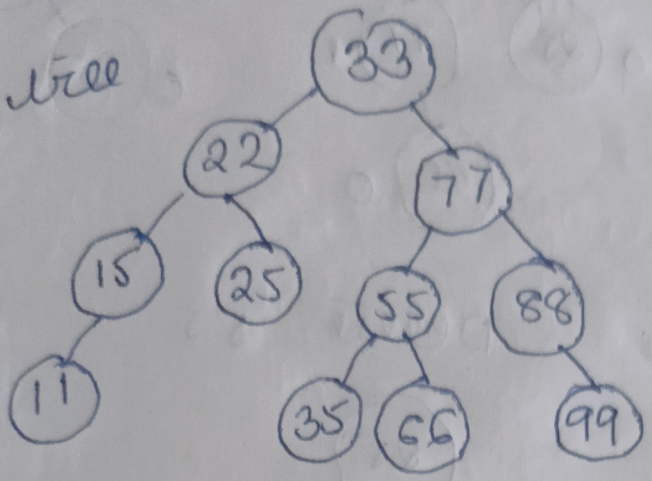
1) for leaf node → delete the node
2) for node(n) with 1 child -
   link the parent node parent(n) with this child node & delete the node(n)
3) for node with 2 childrens -
   a) find the largest node (Max) in left subtree replace this node with the node to be deleted.
   (or)
   b) Find the smallest node in right subtree replace this node with the node to be deleted

---

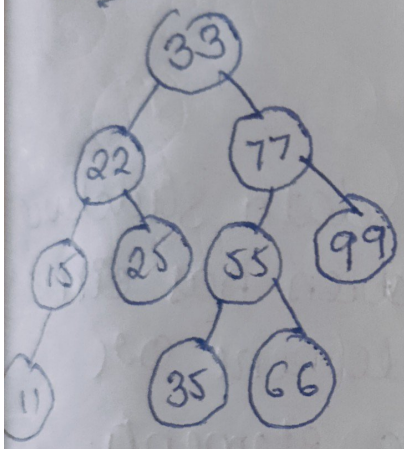
For Idea:-
we calculate the balance factor of the unbalanced node as (left - right) then if it positive 2 or bigger we go to the left of the node & if this left child has left child we do right rotation & if it has only right child we do left right rotation & vise versa for the negative balance factor

Ex:-
Delete 88, 99, 22, 33, 11, 15, 35, 66, 88, 77, 55

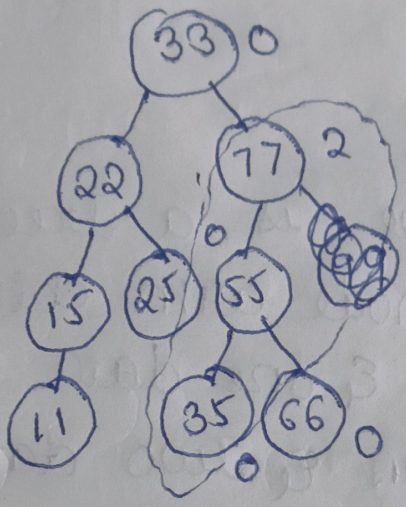
AVL tree



Delete 88

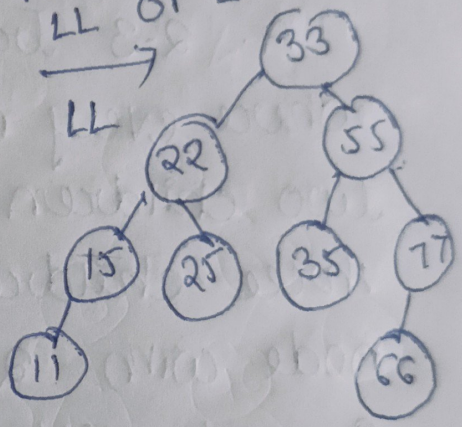


Delete 99

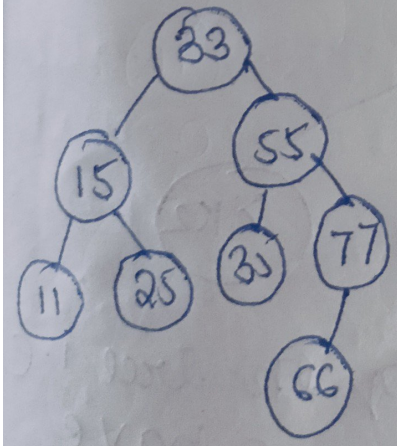


here we can perform LL or LR rotation.

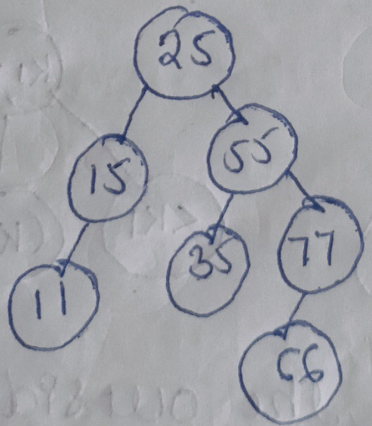
LL



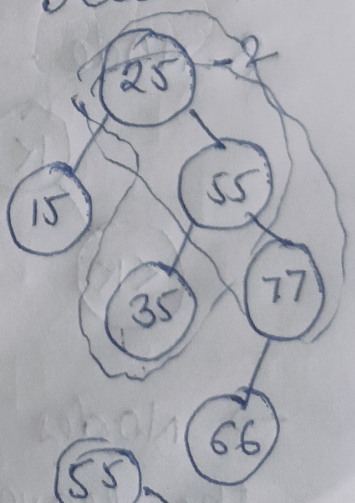
Delete 22



Delete 33

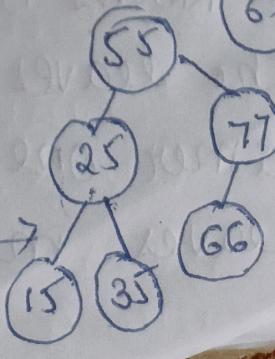


Delete 11

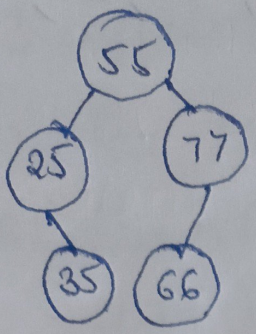


for balancing we can perform RR or RL

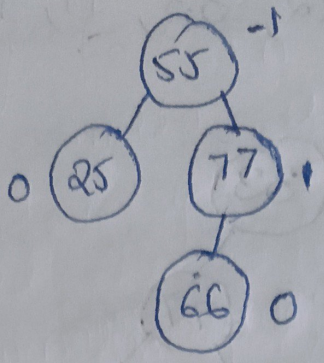
RR



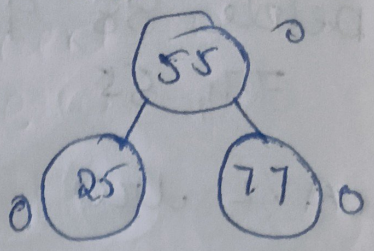
Delete 15



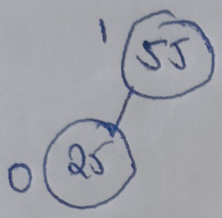
Delete 35



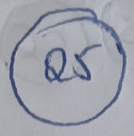
Delete 66



Delete 77

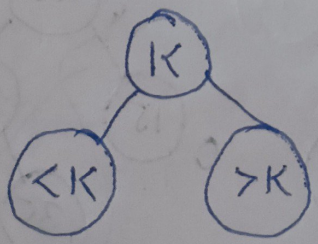


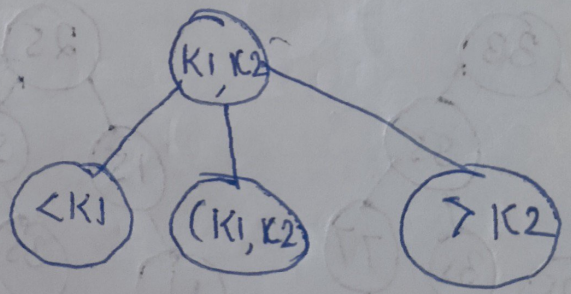
Delete 55



2-3 Trees :-

A 2-3 Tree is a tree data structure where every node with children has either two children & one data element or three children & two data elements. A node with 2 children is called 2-NODE & node with 3 children is called a 3-NODE.

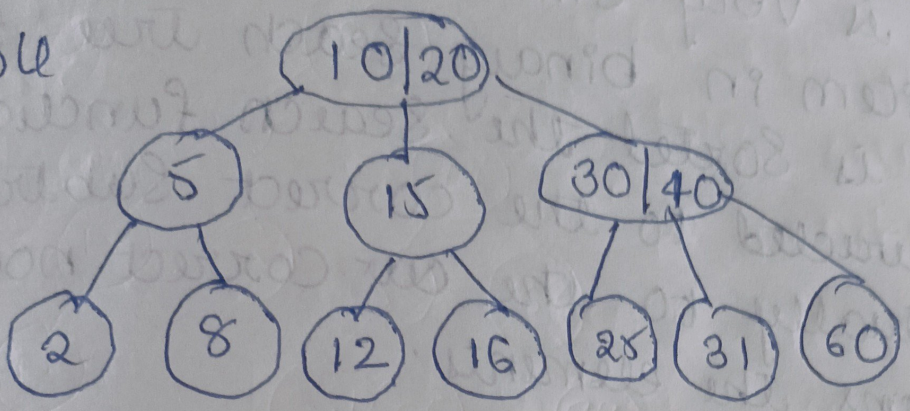




→ Nodes on the outside of the tree i.e the leaves have no children & have either one or two data elements. All leaves must be on the same level

so 2-3 tree is always height balanced
A 2-3 tree is a special case of a B-tree of order 3

Example



properties :-

→ Every internal node in the tree is a 2 node or a 3 node i.e it has either one value or two value.

→ A node with one value is either a leaf node or has Exactly two children values in left subtree < value in node < values in right subtree.

→ A node with two values is either a leaf node or has exactly 3 children it cannot have 2 children values in left sub tree < first value in node < values in middle subtree < second value in node < value in right subtree

→ All leaf nodes are at same level

Searching for an element :-

Searching for an element in a 2-3 tree is very similar as searching for an item in binary search tree. Since the tree is sorted the search function will be directed to the correct subtree & eventually to the correct node which contains the element.

Let d be the data element we want to search for in the 2-3 tree T. The base cases are as follows

→ If the tree T is empty then d is not in the tree & we return false
→ If the current node contains value which is equal to d we return true
→ If we reach the root node & it does not contain d we return false

recursive cases for 3 Node as follows

→ If d is less than the left value of the current node we now consider T to be left subtree of current node
→ Else if d is greater than left value & less than right value of current node we consider T to be middle subtree of current node
→ Else if d is greater than the right value of current node we consider T

to be right subtree of current node

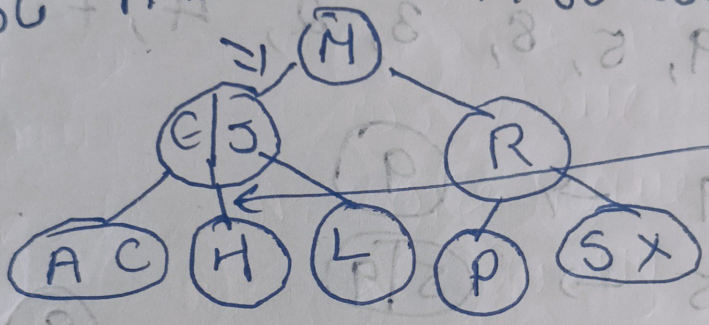
Recursive cases for 2 node

It is same as binary search tree as is as follows

→ If d is less than the value of the current node we consider T to be the left subtree of current node.

→ If d is greater than the right value of current node, we consider T to be right subtree of current node.

Example n is less than M so look to the left



: H is b/w E & J so look in the middle.

Inserting an element :-

The insertion operation takes care of balanced property of 2-3 tree the insertion algorithm in to 2-3 tree is quite different from binary search tree

Algorithm :-

→ If the tree is Empty create a node & put value in to node.

→ otherwise find the leaf node where the value belongs

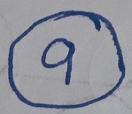
→ If the leaf node has only one value put the new value in to the node.

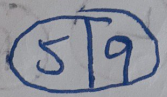
→ If the leaf node has more than two values split the node & promote the median of the three values to parent.

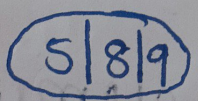
→ If the parent then has 3 values continue to split & promote, forming a new root node if necessary.

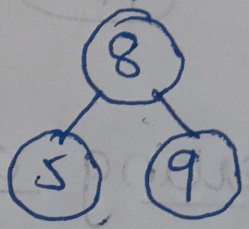
Example:

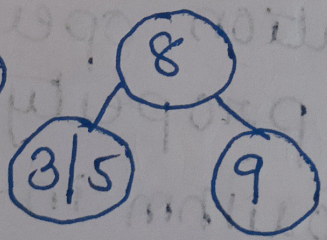
Insert 9, 5, 8, 3, 2, 4, 7

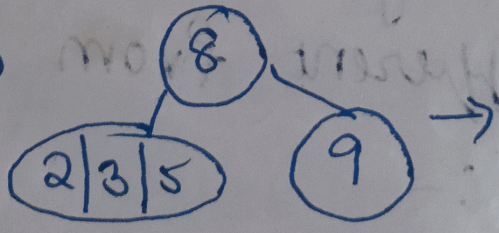
Insert 9 → 

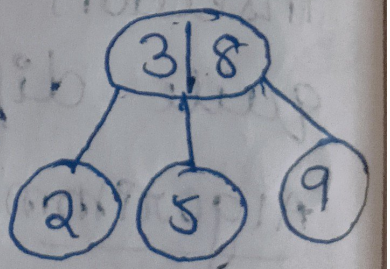
Insert 5 → 

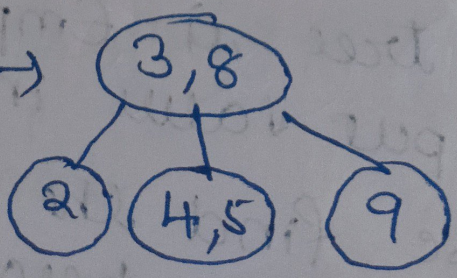
Insert 8 → 

→ 

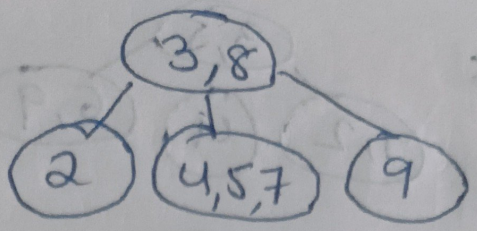
Insert 3 → 

Insert 2 → 

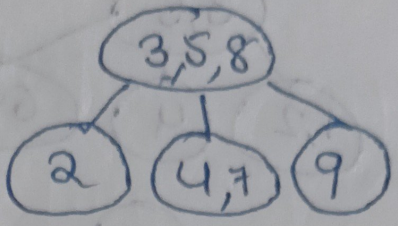
→ 

Insert 4 → 

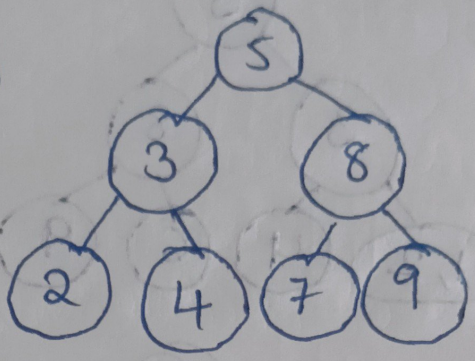
Insert 7 →



→



→



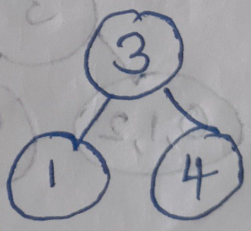
Example:

3, 1, 4, 5, 9, 2, 6, 8, 7, 0

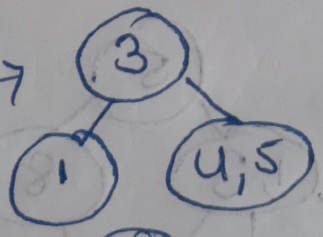
Insert 3 → 3

Insert 1 → 1,3

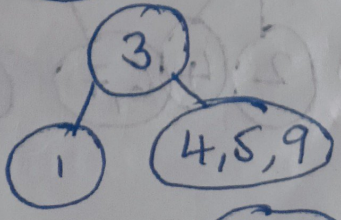
Insert 4 → 1,3,4 →



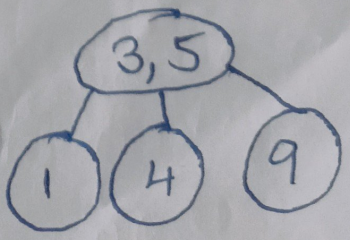
Insert 5 →



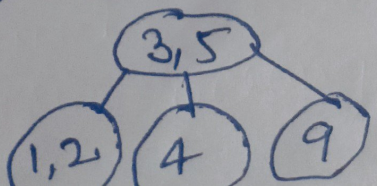
Insert 9 →



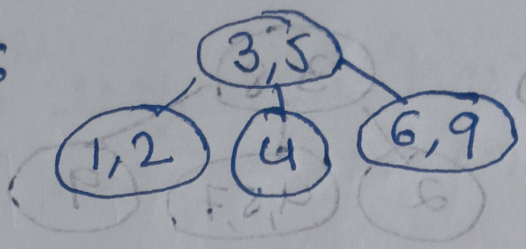
→



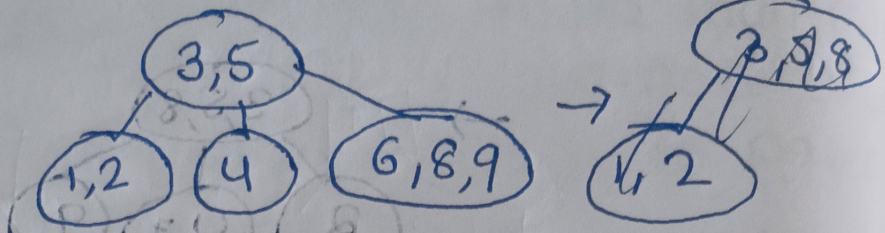
Insert 2 →



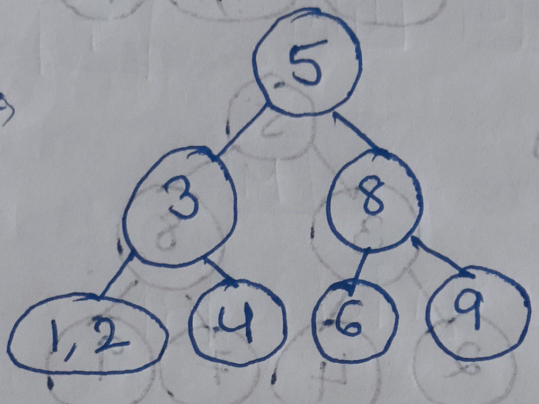
Insert 6



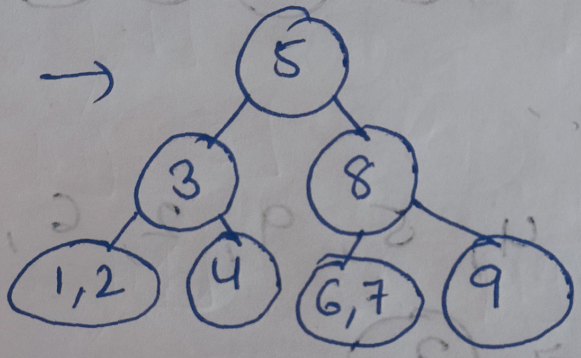
Insert 8



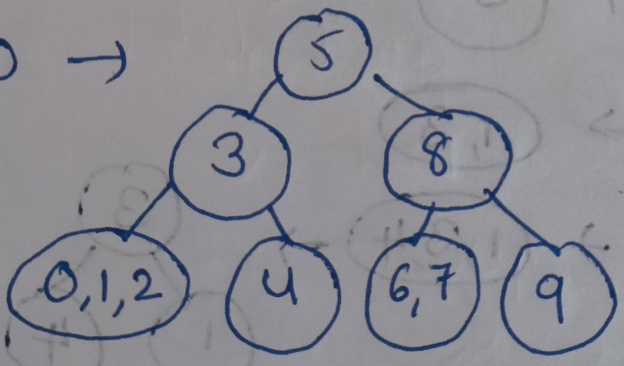
→



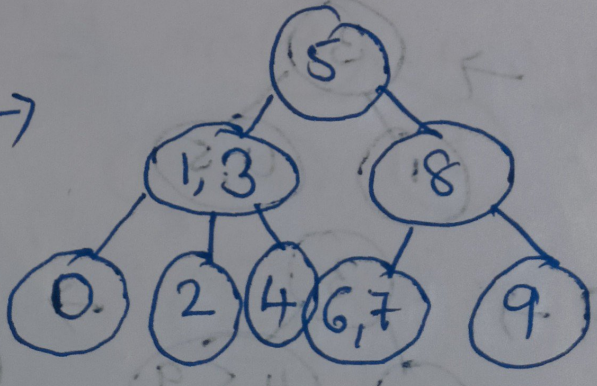
Insert 7 →



Insert 0 →

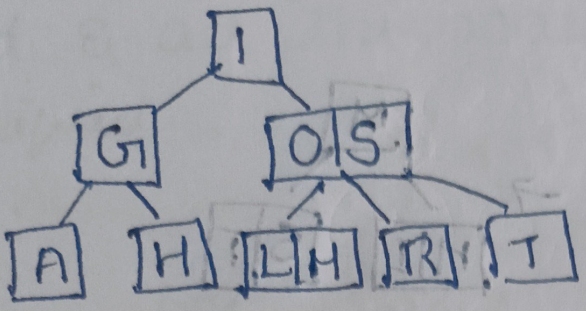


→

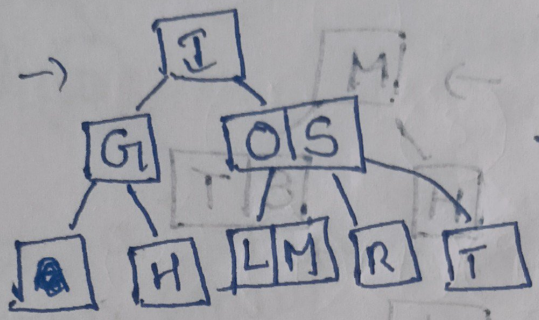


Deletion

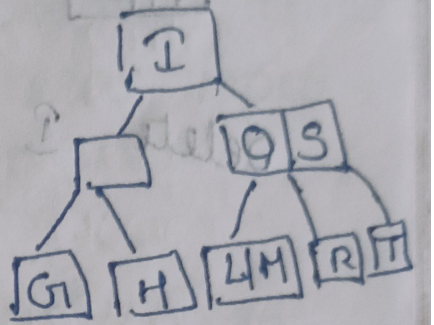
A, L, G O R I T H M S



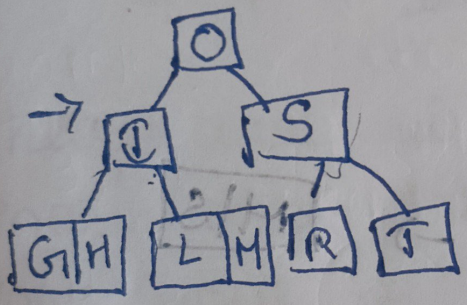
Delete A →



→



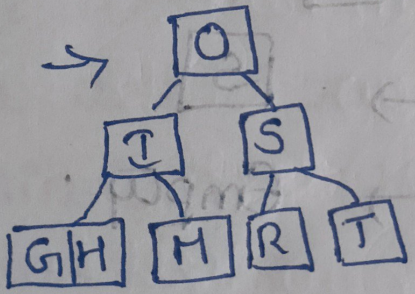
→

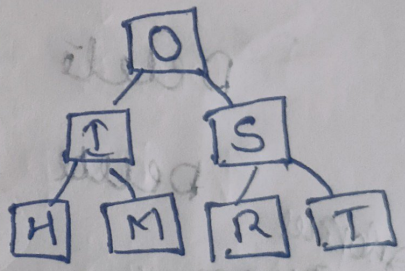


If the i am in a leaf simply delete
→ If not in leaf swap it with its inorder predecessor ~~then delete it from leaf~~

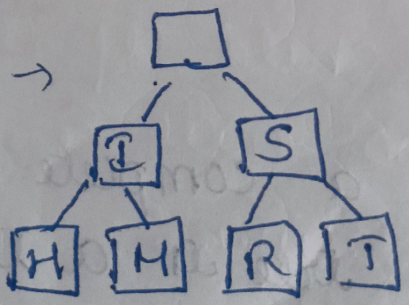
Delete G

Delete L →

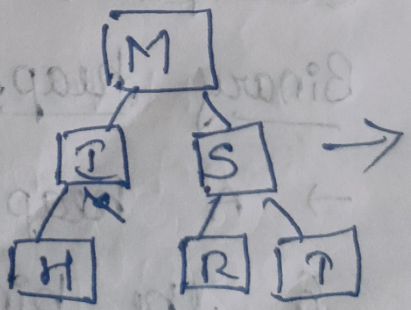




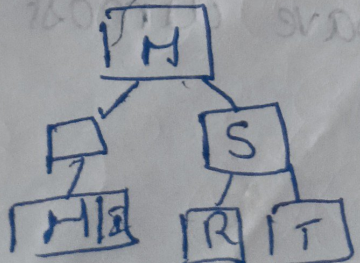
Delete O →



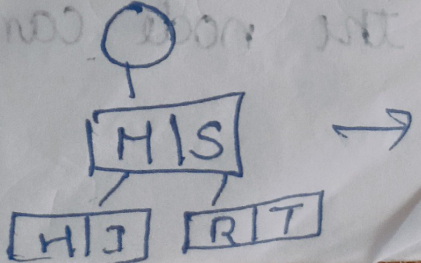
→



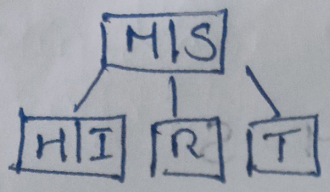
→



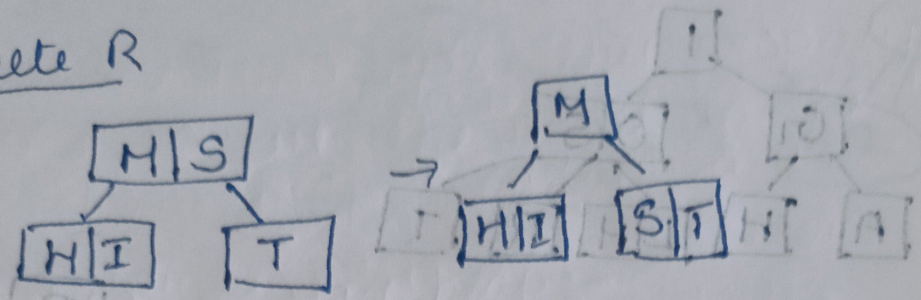
→

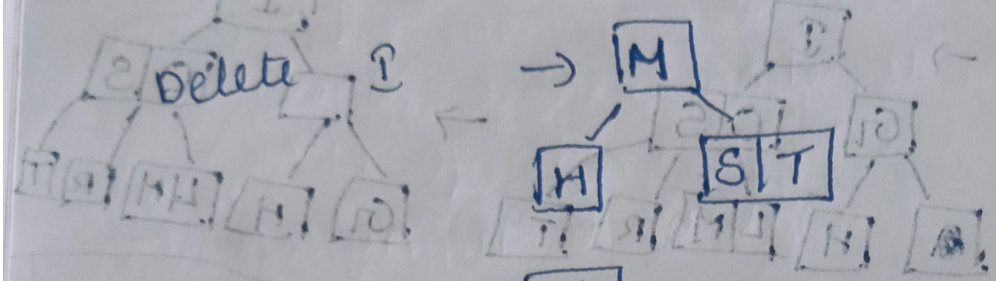


→

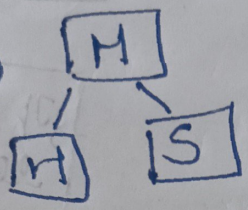


Delete R

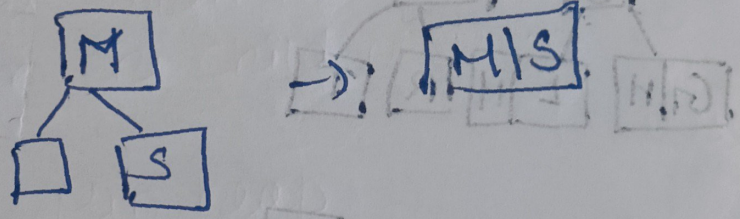




Delete T →



Delete H →



Delete M → S

Delete S → Empty.

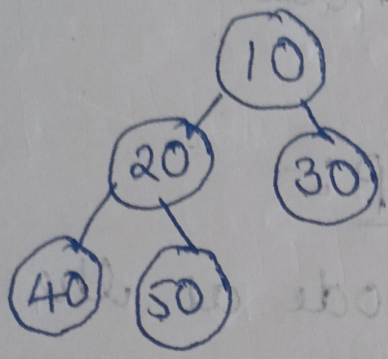
Examples:

Binary heaps :-

→ A heap is a complete binary tree & the binary tree is a tree in which the node can have utmost two children.

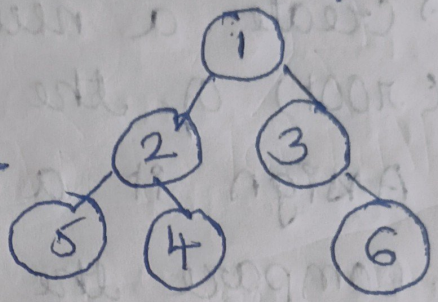
A complete binary tree is a binary tree in which all the levels except the last level i.e leaf node should be completely filled & all the nodes should be left-justified.

Ex:



→ All the Internal nodes are completly filled except leaf node.

Not a complete binary tree



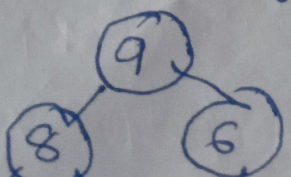
Heap tree is a special balanced binary tree data structure where the root node is compared with its children & arrange accordingly.

There are two types of heap

→ Min heap
→ Max heap

Max Heap :-

heaps where the parent node is greater than equal to (>=) its children are called max heaps. This property is applied to every node of the tree



Min heaps :-

heaps where the parent node is less than or equal to (<=) its children are called Min heaps. This property is applied to all the nodes in the tree.

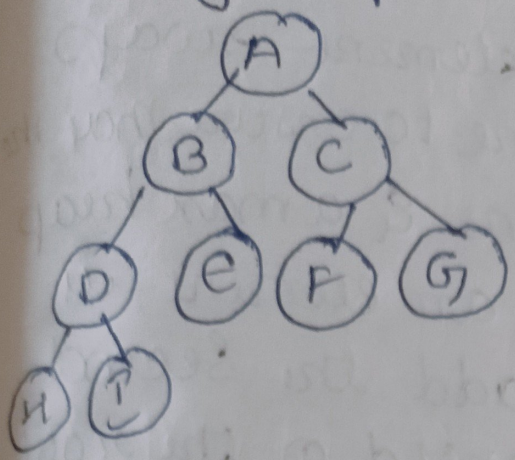
Creating a Max heap :-

→ Create a new node at the beginning (root) of the heap
→ Assign it a value.
→ Compare the value of child node with the parent node.
→ Swap nodes when the parent node is smaller than the child node.

Creating a Min heap :-

→ Create a new child at the end of heap (last level).
→ Add the new key to that node
→ Move the child up until you reach the root node & the heap property is satisfied.

Array representation of Binary Tree



case 1:

A	B	C	D	E	F	G	H	I
0	1	2	3	4	5	6	7	8

case 2:

	A	B	C	D	E	F	G	H	I
1	2	3	4	5	6	7	8	9	

case 1 :- If a node is at ith index
left child would be $(2*i)+1$
Right " " " $(2*i)+2$
parent would " $\left\lfloor \frac{(i-1)}{2} \right\rfloor$

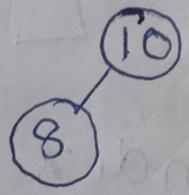
case 2 :-
node is at ith index
left child at = $(2*i)$
right child at = $(2*i)+1$
parent at = $\lfloor i/2 \rfloor$

Insertion in heap

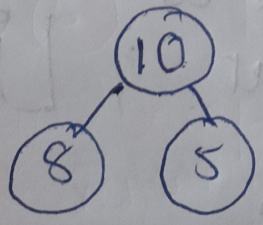
Consider we have an array with elements 10, 8, 5, 15, 6 in it to build a max heap using the given elements

Step 1. :-
Add first element 10 in the tree. It is the root element in the tree

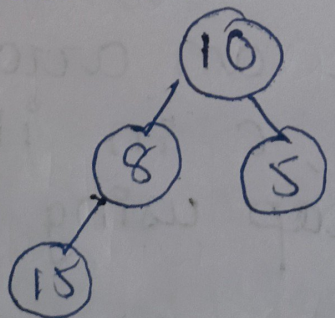
2. Add the next element in the Tree. Compare it with parent element. if it is greater than its parent element swap their positions. It is done to ensure that the tree follows heap conditions & a max heap is maintained each time an element is added. here we should add the second element because a heap is filled from left to right. NO swapping occurs here because 10 is greater than 8.



Repeat the above given step with the next element 5

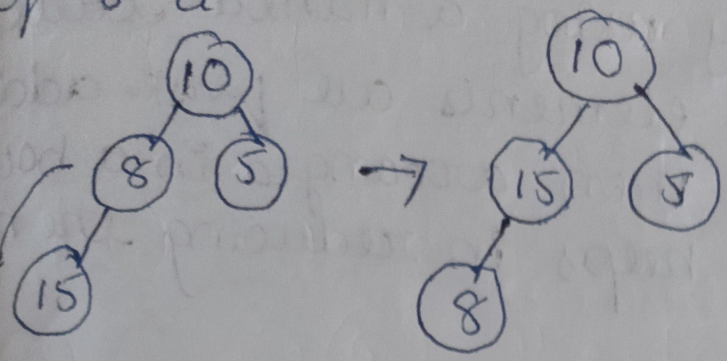


Add the next element 15 in tree.

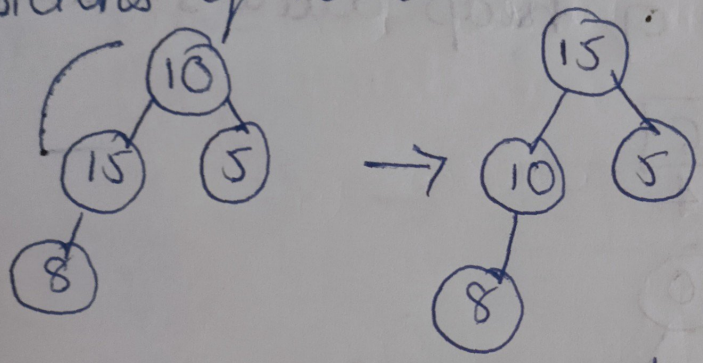


Now since 15 is greater than its parent element & this tree is not a max heap anymore to make it a

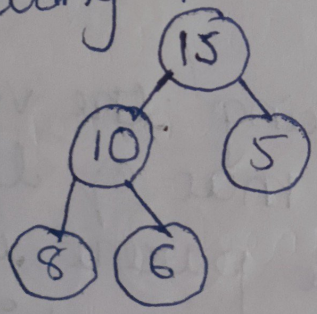
heap again we will swap the positions of 8 & 15



Again the obtained tree is not a max-heap since 15 is greater than its parent element 10 we will again swap the positions of 10 & 15



Add the last element 6 in the heap by comparing it with the parent.



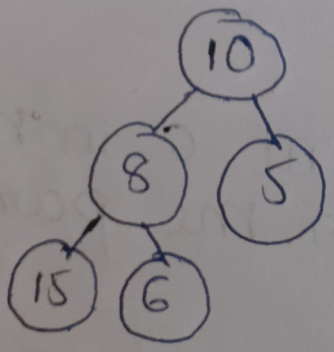
one thing to note here is that a comparison is done each time an element is added. the no of comparisons also depends on the height of the tree. In the above case, a total of 5 comparisons were made. this results in time complexity of O(n log n)

But we can reduce the no. of comparisons by using a method called heapify where elements are first added in to the tree then arranged in a bottom up fashion it helps in reducing the no of comparisons

heapify :-

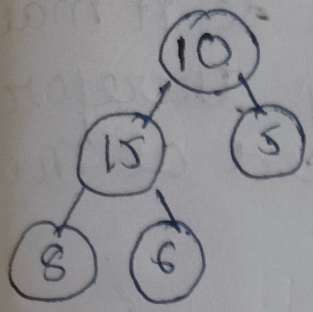
It is the process of rearranging the elements to form a tree that maintains the properties of heap data structure.

10	8	5	15	6
0	1	2	3	4



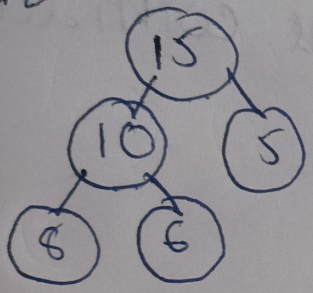
Start from comparing the values of children nodes with that of the parent if the value of the parent is smaller than the values of the children, swap it swapping is done with a larger of two children this process is repeated until every node satisfy properties of a max heap.

here we start comparing 8 with 15 & 6 Now since 15 is greater than 8 we will swap their positions



→ |10|15|5|8|6|

Again property of max heap is not satisfied since 15 is greater than 10 Therefore we will once again perform the above step

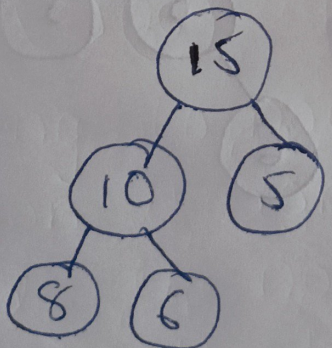


→ |15|10|5|8|6|

here time complexity is equal to height of tree i.e O(log n)

Deletion from Heap :-

let us consider max heap



Alg :-

→ replace the root or element to be deleted by the last element

→ delete the last element from the heap.

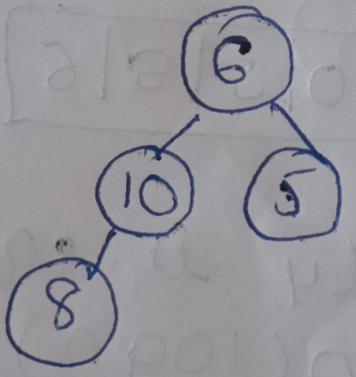
→ Since the last element is now placed at position of the root node so it may not follow heap property therefore heapify the last node placed at the position of root.

Ex: delete 15 from the above tree

process :-

The last element is 6

replace the last element with root



step 2 : heapify root