Unit - III

Search Trees

Binary Search Trees :- The binary search tree is a kind of binary tree in which the values at left subtree are less than the value of root node. And the values of right subtree are greater than the value of root node.

Left Subtree < Root node < Right Subtree

Operations on Binary Search Tree

Various operations can be performed on binary search tree are

1) Insertion of a node
2) Deletion of a node
3) Searching of a element
4) Display of binary tree

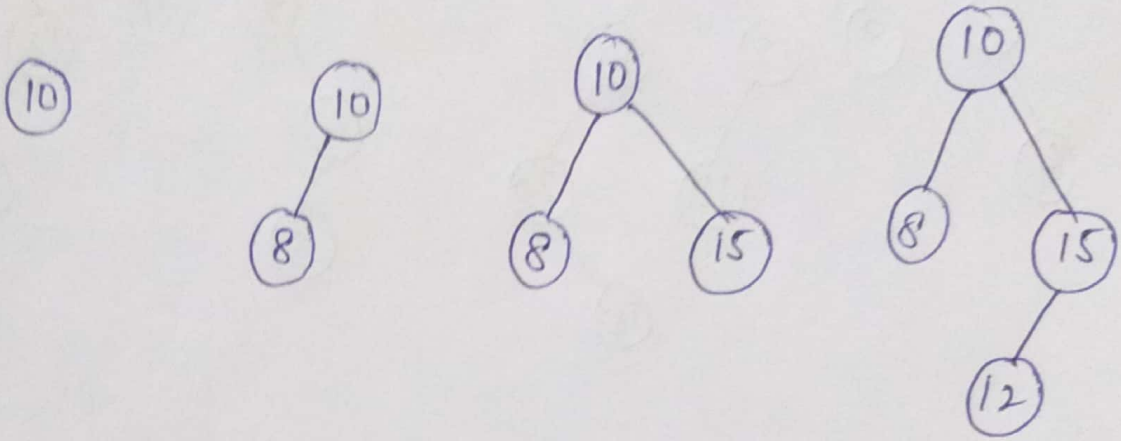
Insertion of a node in a binary tree

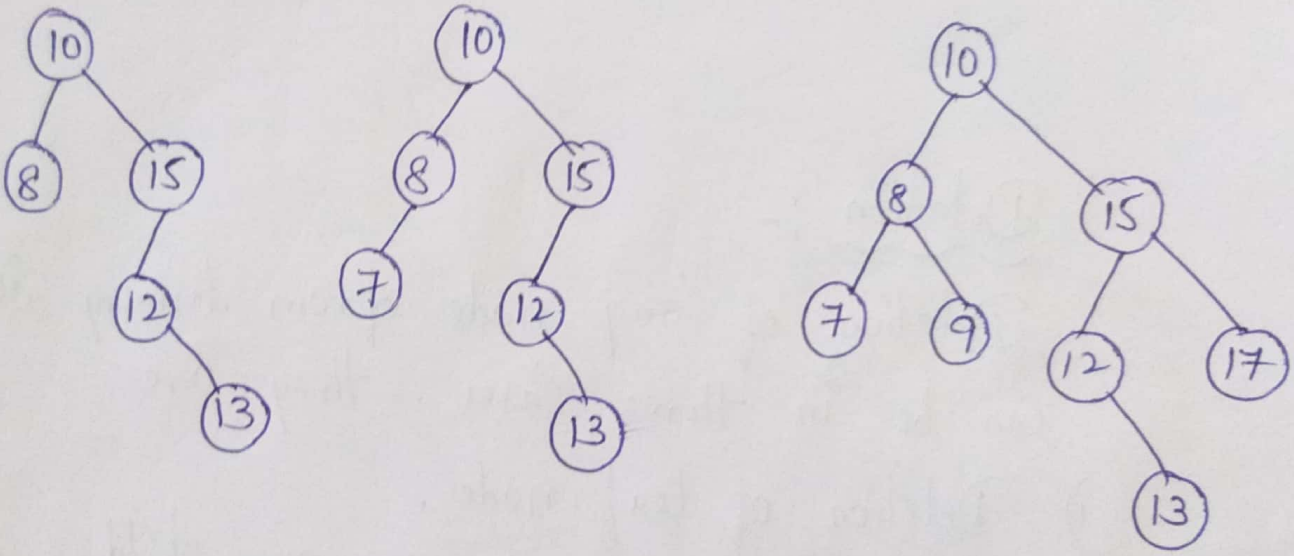
Algorithm

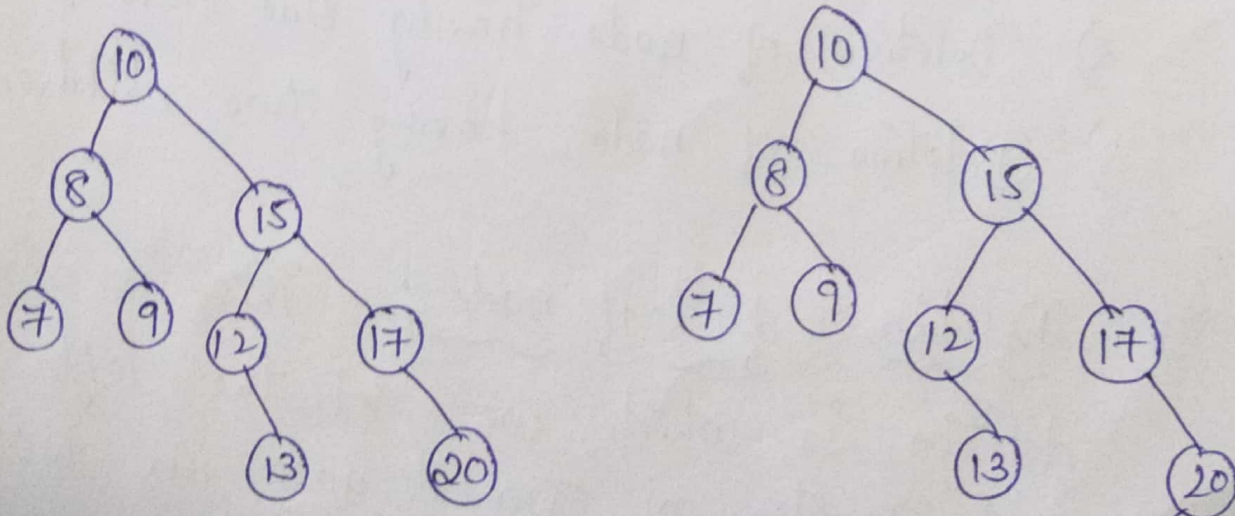
1) Read the value for the node which is to be created, and store it in a node called New.
2) Initially if (root != NULL) then root = New.
3) Again read the next value of node created in New.
4) If (New → value < root → value) then attach New node as a left child of root otherwise attach New node as a right child of root.
5) Repeat step 3 and 4 for constructing required binary search tree.

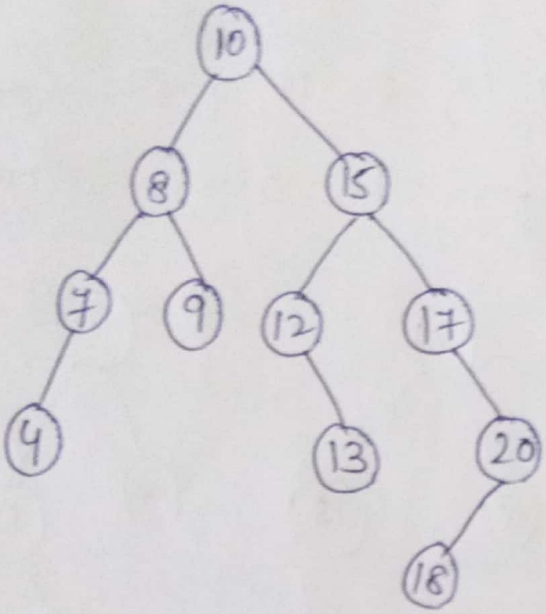
Insertion :-

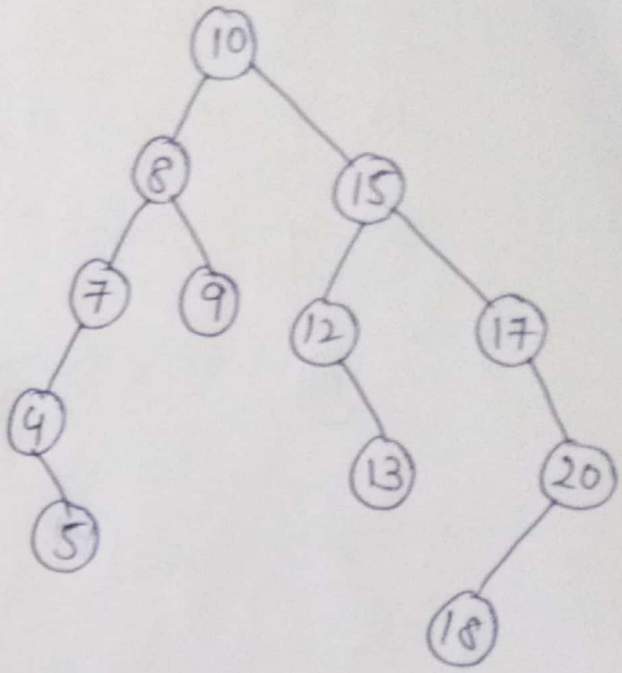
10, 8, 15, 12, 13, 7, 9, 17, 20, 18, 4, 5. Construct binary search tree for the above elements.









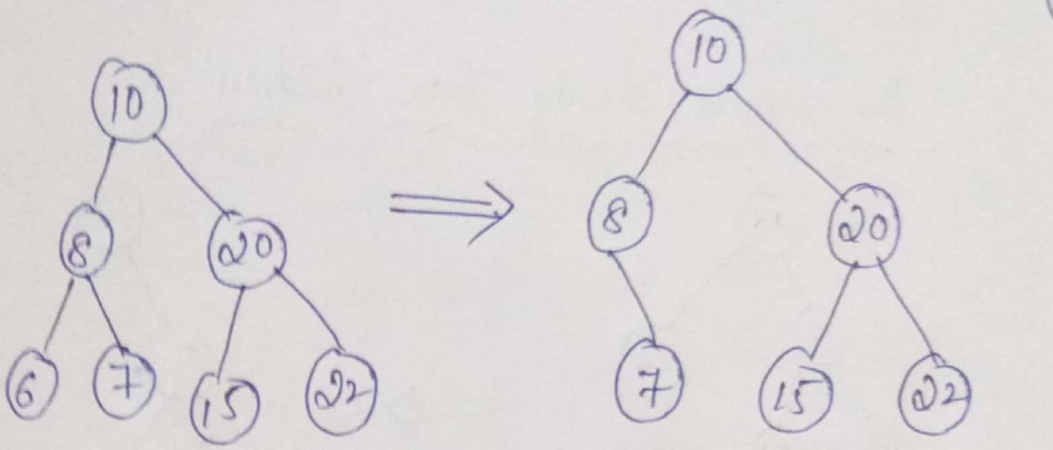


Deletion :-

Deletion of any node from binary search tree can be in three cases. they are

1) Deletion of leaf node.
2) Deletion of node having One child.
3) Deletion of node having two children.

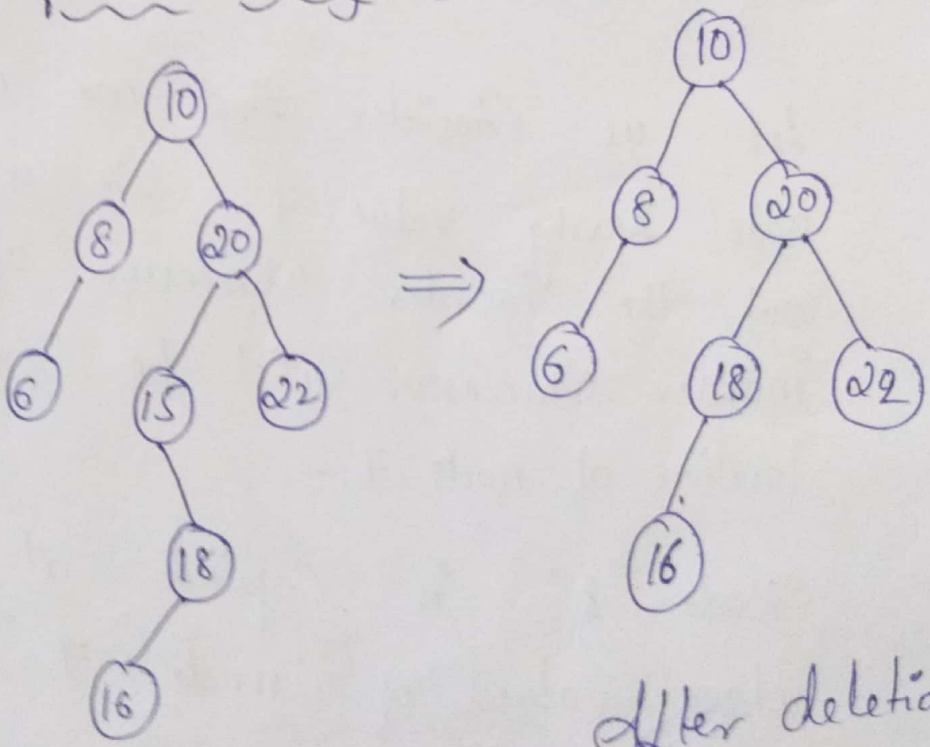
Deletion of leaf node :- This is simplest deletion, in which we set the left or right pointer of parent node as NULL.



Before deletion

After deletion

Deletion of node having one child :-

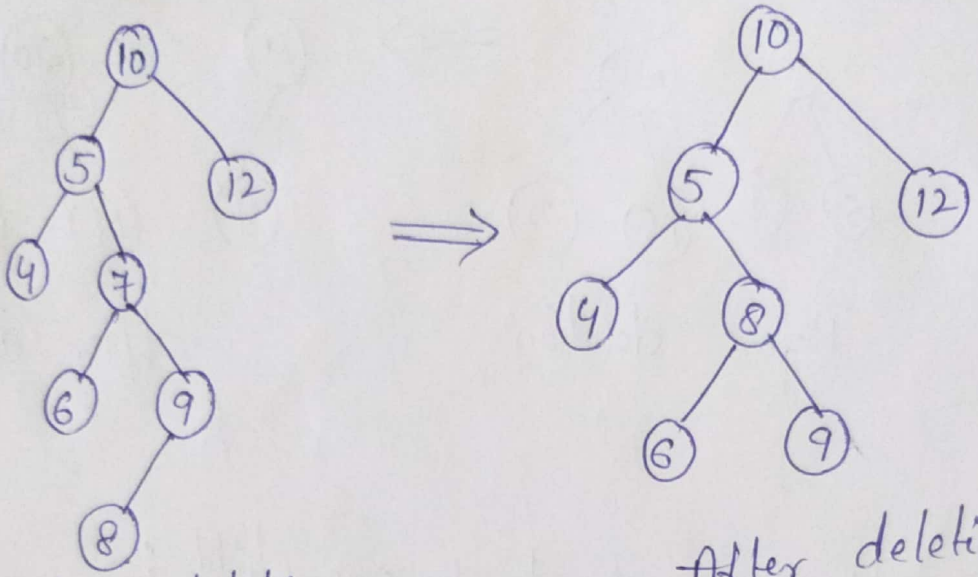


Before deletion

After deletion.

If we want to delete the node 15, then we Copy node 18 at place of 15.

The node having two children



Before deletion

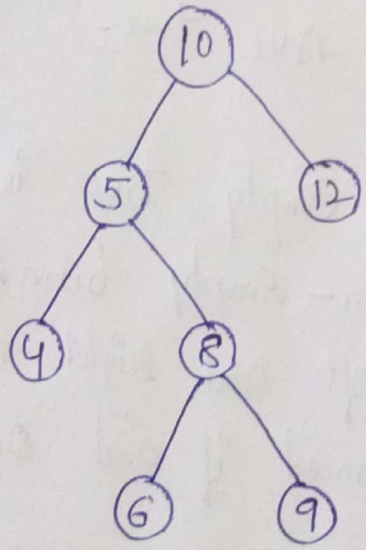
After deletion.

Let us Consider that we want to delete node having value 7. We will then find out the inorder Successor of node 7. The inorder Successor will be simply Copied at location of node 7.

Node "8" is Copied at the position where value of node "7". And set left pointer of 9 as Null. This Completes the deletion procedure.

Searching a node from binary Search Tree

In Searching, the node which we want to search is called a key node. The key node will be compared with each node starting from root node if value of key node is greater than current node then we search for it on right sub-branch otherwise on left sub-branch.



In the above tree, if we want to search for value 9. Then we will compare 9 with root node 10. As "9" is less than 10 we will search on left sub-branch. Now Compare 9 with 5, but 9 is greater than 5. So we will move on right sub-branch. Now Compare 9 with 8, but 9 is greater than 8. we move on right sub-branch.

AVL Trees

Adelsion Velski and Landis in 1962 introduced binary tree structure that is balanced with respect to height of subtrees.

The tree can be made balanced and because of this retrieval of any node can be done in $O(\log n)$ times, when n is total number of nodes. From the name of these scientists the tree is called AVL tree.

Definition :- An empty tree is height balanced if "T" is a non-empty binary tree with T_L and T_R as its left and right subtrees. The T is height balanced if and only if

i) T_L and T_R are height balanced

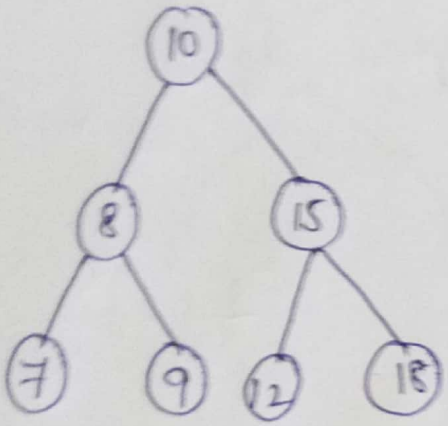
ii) $h_L - h_R <= 1$ where h_L and h_R are heights of T_L and T_R.

The idea of balancing a tree is obtained by calculating the balance factor of a tree.

Definition of Balance factor

The balance factor BF(T) of a node in binary tree is defined to be $h_L - h_R$ where h_L and h_R are heights of left and right subtrees of T.

for any node in AVL tree the balance factor i.e BF(T) is -1, 0, or +1



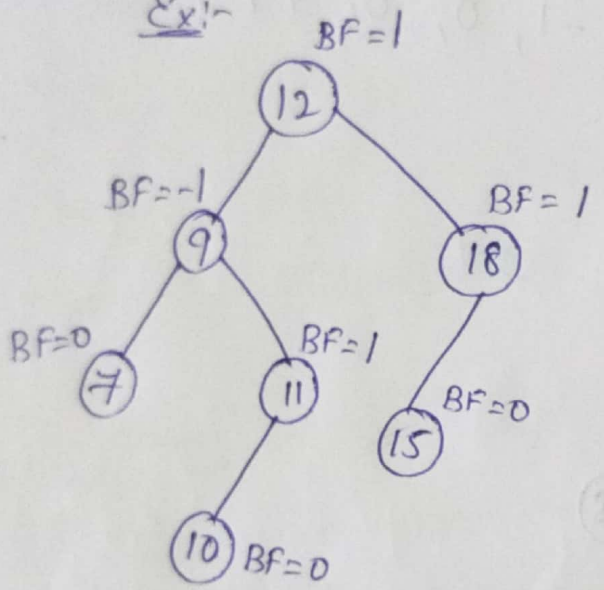
AVL Tree

Representation of AVL Trees

1) The AVL tree follows the binary search tree property. In fact AVL trees are basically binary search trees with balance factor as -1, 0, or +1.

2) After Insertion of any node in AVL tree if the balance factor of any node becomes other than -1, 0, or +1 then it is said that AVL property is violated.

Ex:-



Insertion :-

There are four different cases when rebalancing is required after insertion of new node.

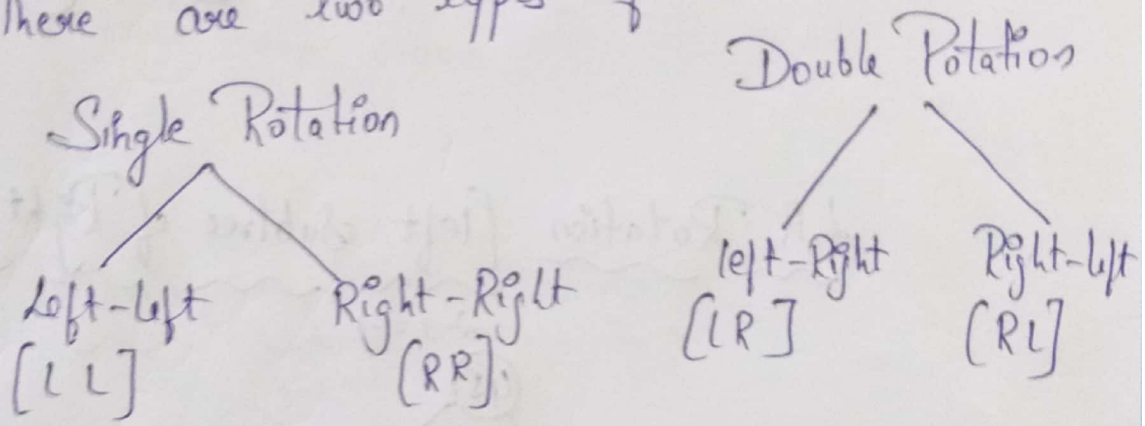
1) An Insertion of new node into left Subtree of left child [LL].

2) An Insertion of new node into right Subtree of left child [LR].

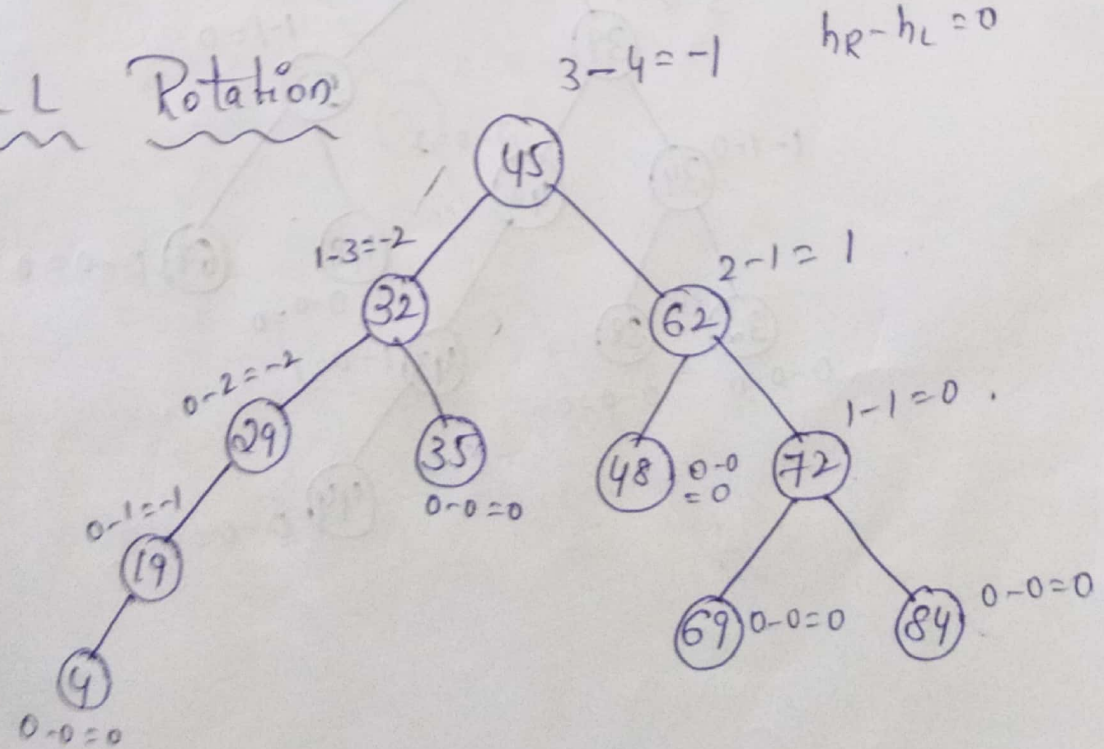
3) An insertion of new node into left subtree of right child [RL].

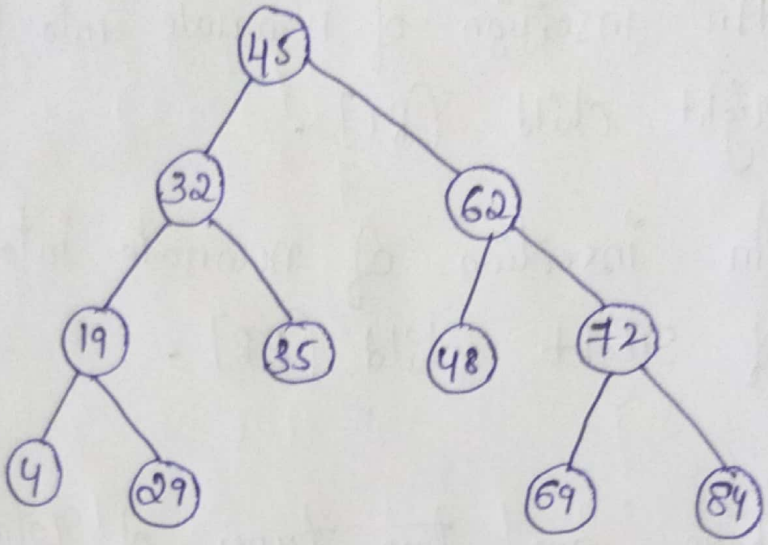
4) An insertion of new node into right subtree of right child [RR].

There are two types of rotations

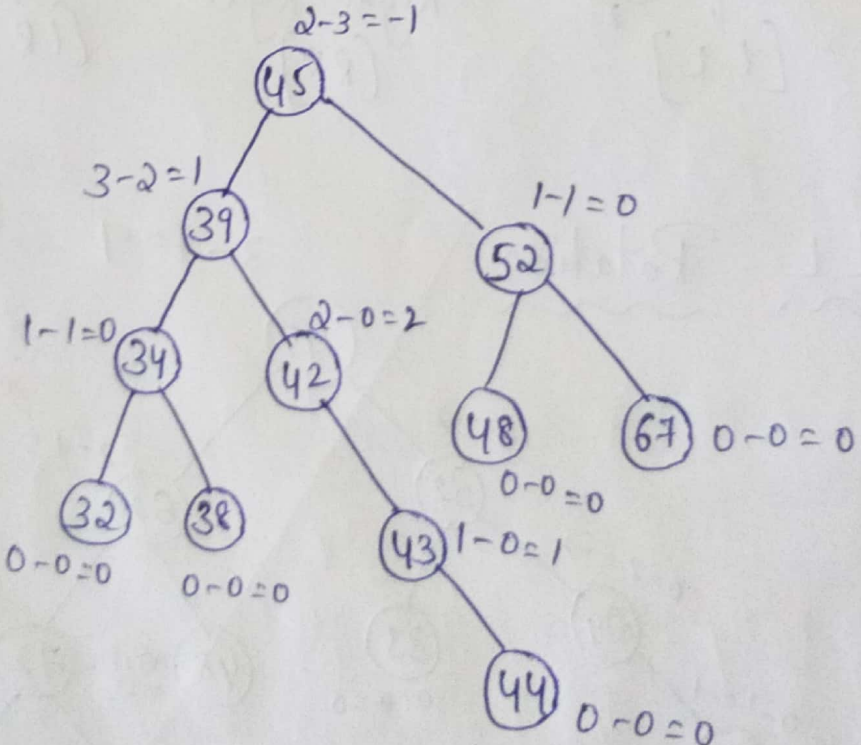


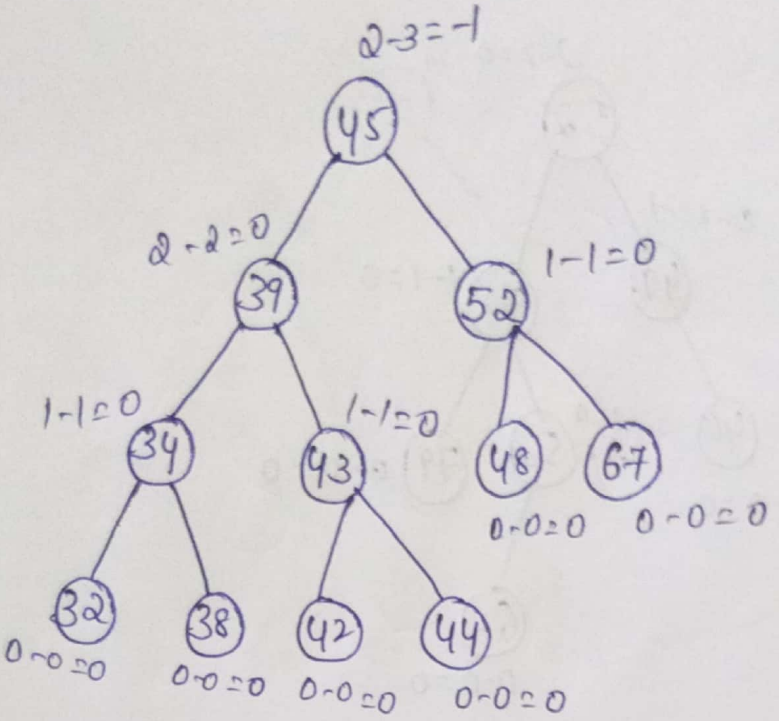
① LL Rotation



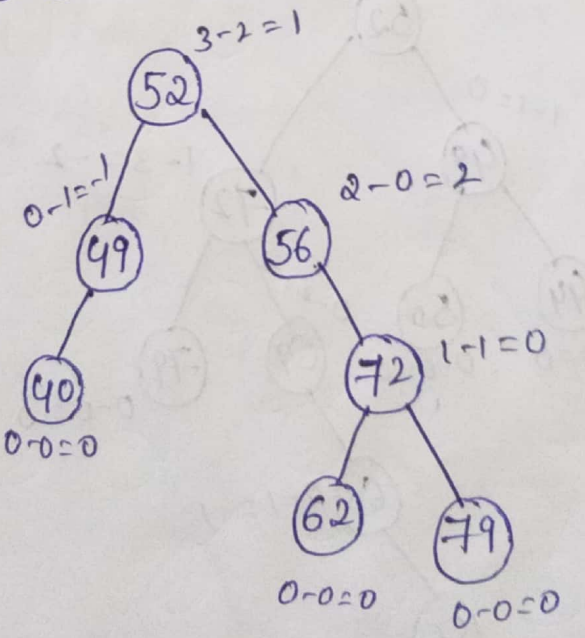


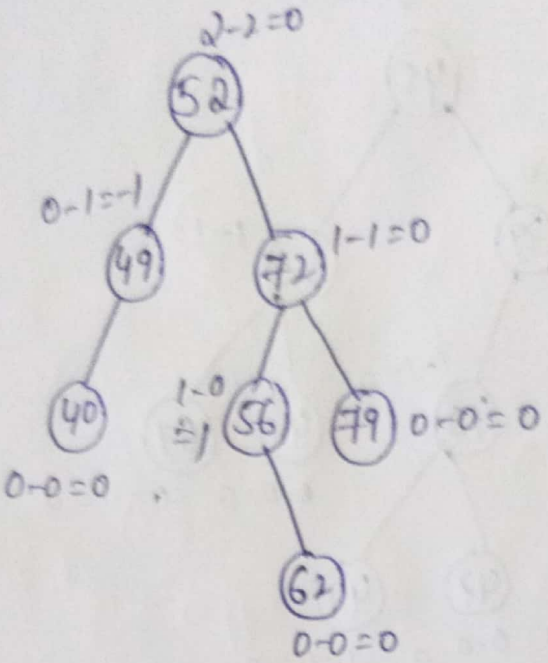
LR Rotation [left Subtree of Right child]



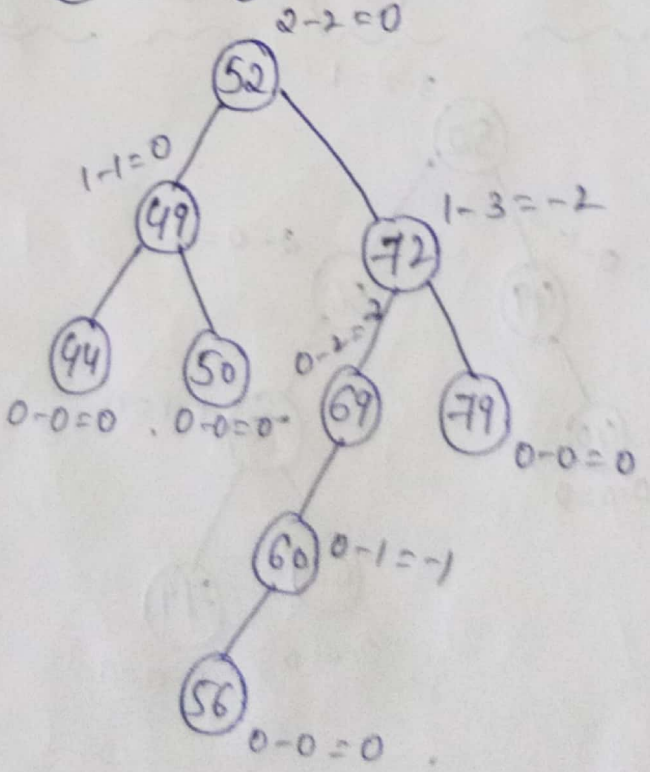


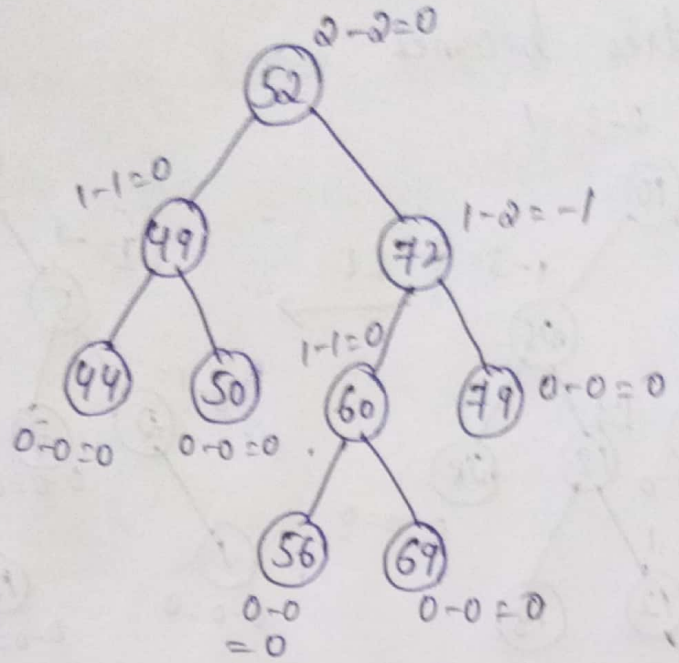
RR Rotation :- [Right Subtree right child] :-



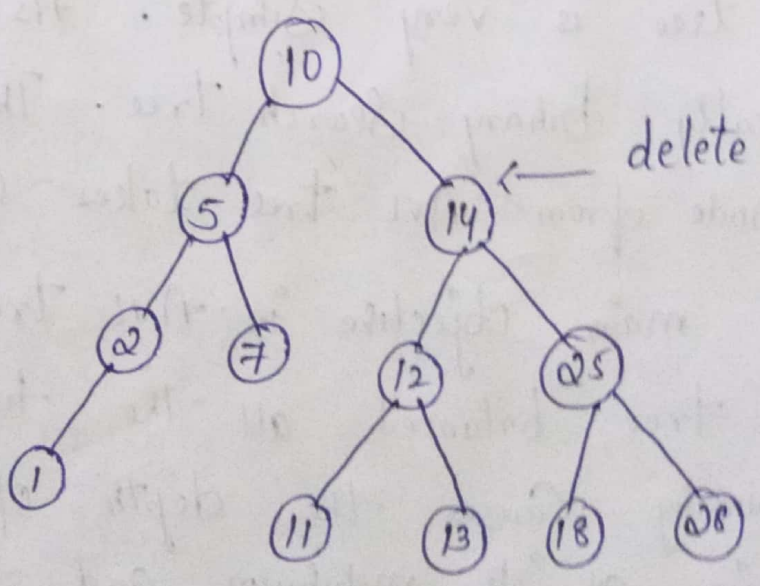


RL Rotation :- Right Subtree of left child :-

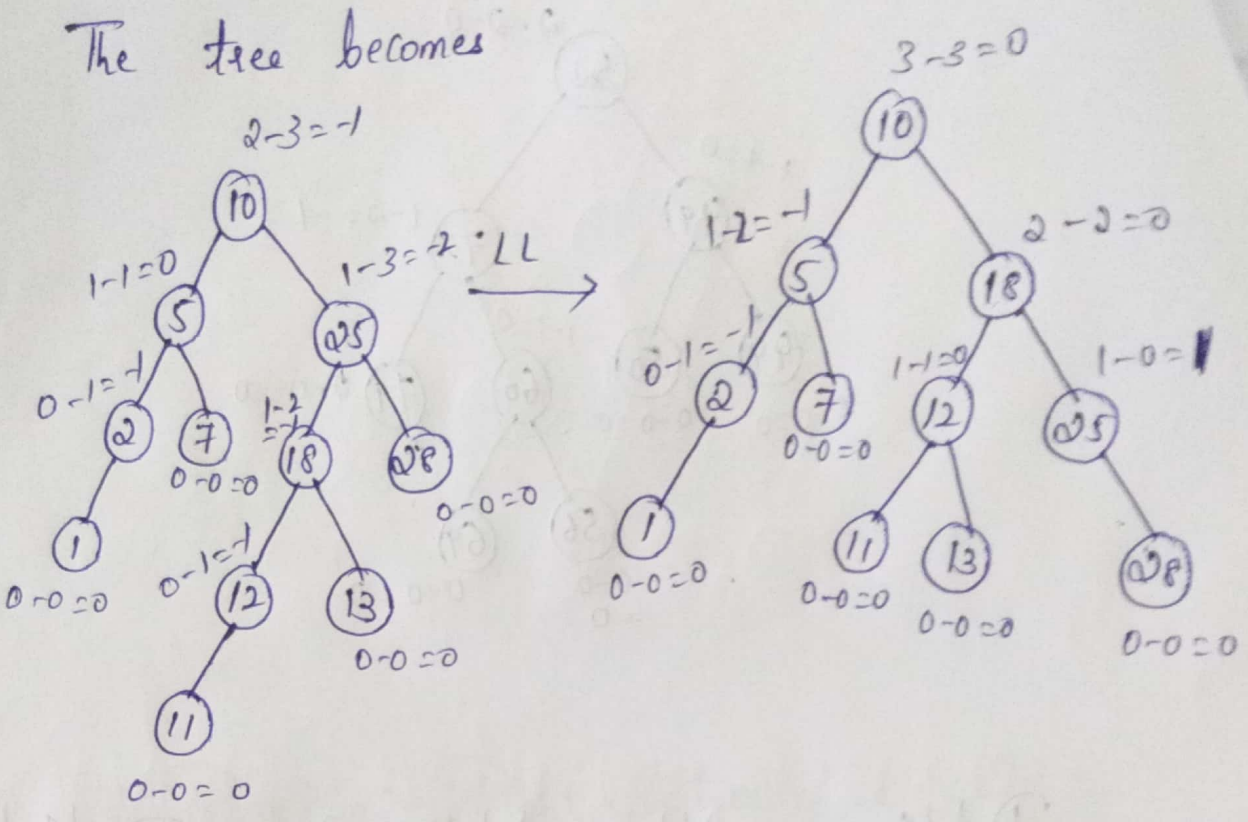




Deletion :- After deletion of any particular node from AVL tree, the tree has to be restructured in order to preserve AVL property. And various rotations need to be applied.



The tree becomes



Thus the node 14 gets deleted from AVL tree.

Searching :- The Searching of a node in an AVL tree is very simple. As AVL tree is basically binary Search tree. The Searching of a node from AVL tree takes $O(\log n)$ times. The main objective in AVL tree is to keep the tree balanced all the times. Such balancing causes the depth of the tree to remain at its minimum and therefore overall costs of search is reduced.

AVL trees or height balanced tree give the best performance for dynamic tables.

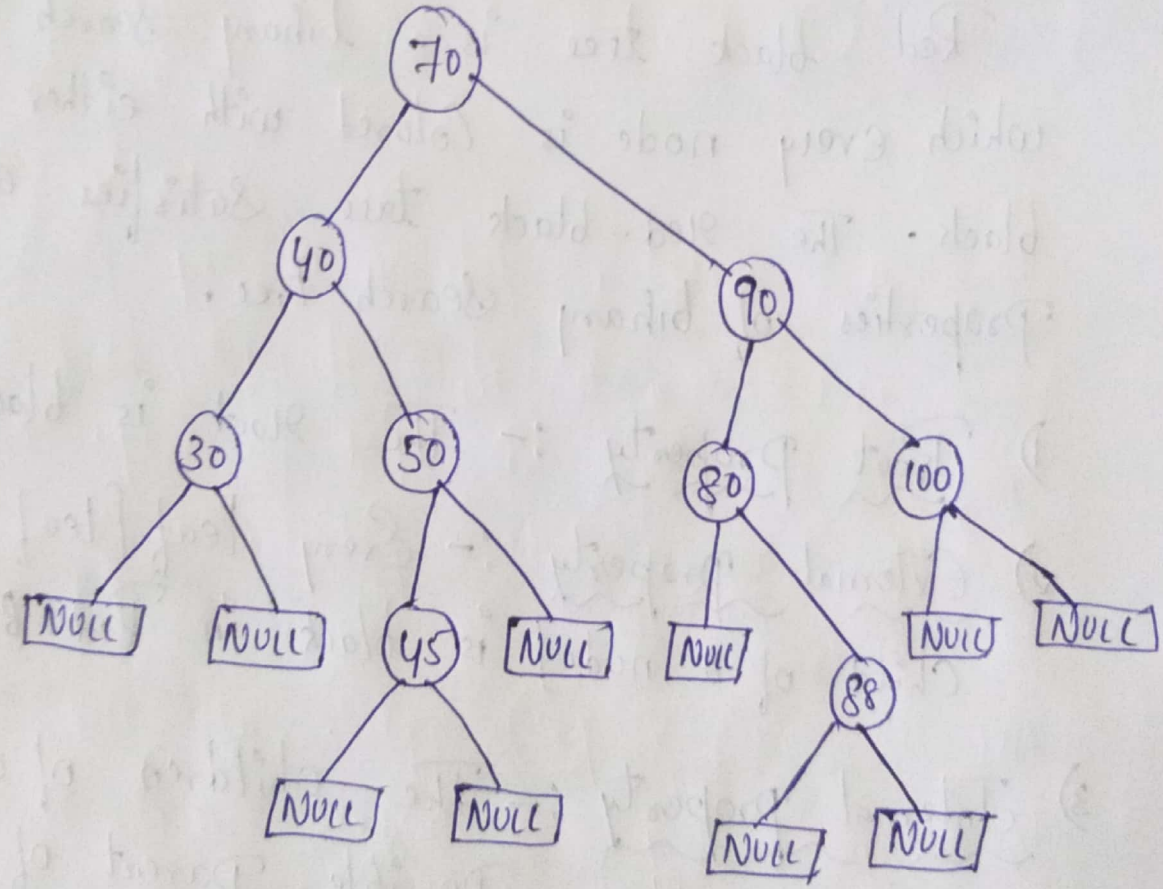
Red Black Trees :-

Red black tree is a binary search tree in which every node is colored with either red or black. The red-black tree satisfies all the properties of binary search tree.

1) Root Property :- The root is black.
2) External Property :- Every leaf [leaf is a NULL child of a node] is black in Red-Black tree.
3) Internal property :- The children of a red node are black. Hence possible parent of red node is a black node.
4) Depth Property :- All the leaves have the same black depth.

5) Path Property :- Every Simple path from root to leaf node Contains same number of black nodes.

Representation of Red-Black tree



1. It is a binary Search tree.
2. The root node is black.
3. The children of red node are black.
4. No root - to external node path has two Consecutive red nodes [e.g. 70-90-80-88-NULL].

(Cont...d)

Insertion operations

1) Recolor
2) Rotation
3) Rotation followed by Recolor.

Step 1: Check whether Tree is Empty

Step 2: If the tree is Empty then Insert the new node as Root node with Color Black and exit from the operation.

Step 3: If the tree is not Empty then insert the new node as leaf node with Color Red.

Step 4: If the Parent of the new node is Black then Exit from the operation.

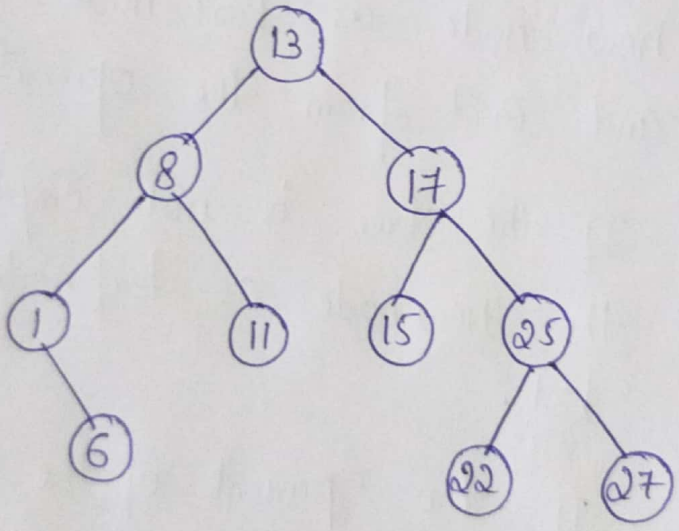
Step 5: If the parent of new node is Red then check the Color of Parent node's Sibling of new node.

Step 6: If it is Colored Black or NULL then make Suitable Rotation and Recolor it.

[Cont...d]

5. All the root to external node paths contain same number of black nodes [including root and external node]

for. Eg. Consider path 70-40-30-NULL and 70-90-80-88-NULL in both these paths 3 black nodes are there. Similarly other paths can be checked.

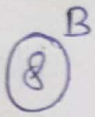


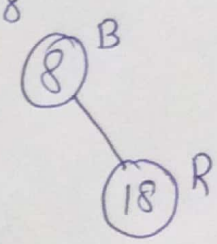
Applications

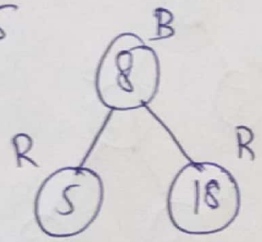
Completely fair Scheduler used in Current Linux kernel and epoll System Call implementation uses Red-black Trees.

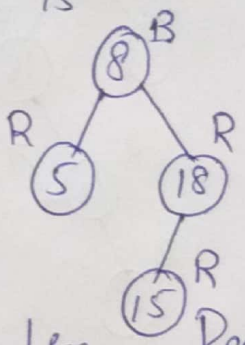
Step 7: If 'H' is Colored Red then perform Recolor, Repeat the same until tree becomes Red-black Tree.

8, 18, 5, 15, 17, 25, 40, 80

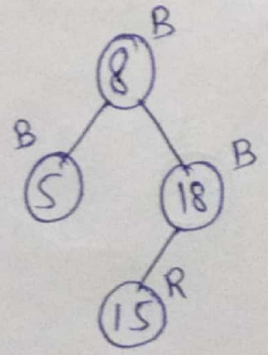
Insert 8, 

Insert 18


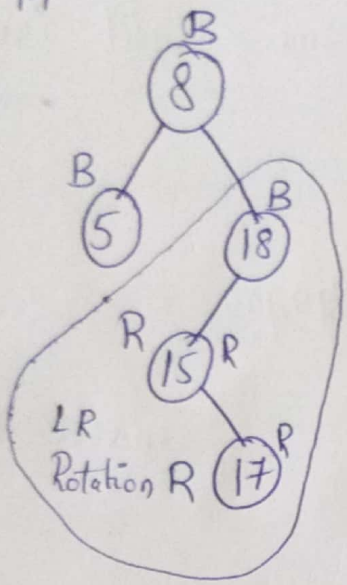
Insert 5


Insert 15


Here there are two Consecutive Red nodes 18, 15. The new nodes Parent Sibling Color is Red and Parents Parent is root node. So we use "Recolor" to make it Red-Black Tree.



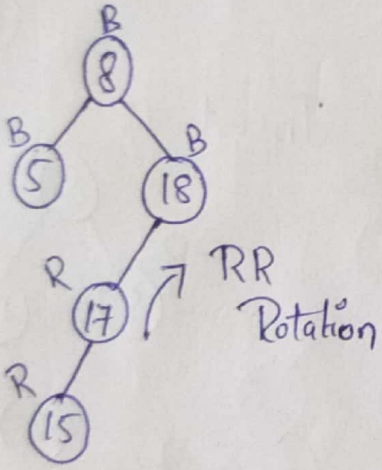
Insert "17"

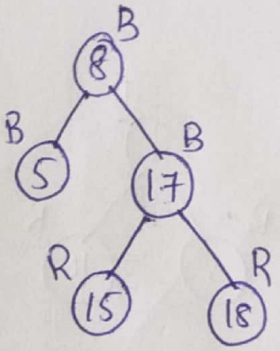


Here there are two Consecutive Red nodes [15 & 17]

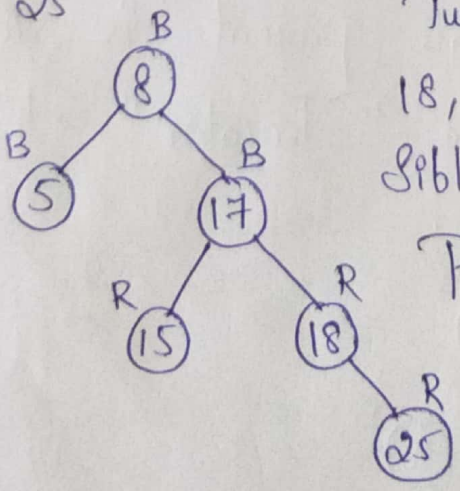
The new nodes Parent Sibling is NULL. So we need Rotation.

Here we need left and Right Rotation and Recolor





Insert "25"

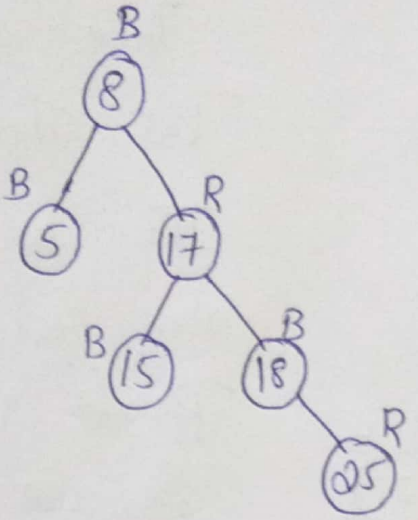


Two Consecutive Red nodes 18, 25. The new nodes Parent Sibling is Red Color and Parents Parent is not Root node.

So we use Recolor and Recheck.

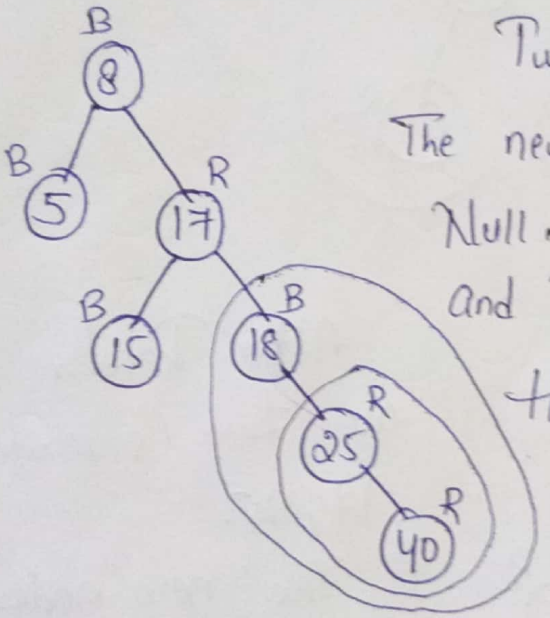
[Cont..d]

[Cont.. d]

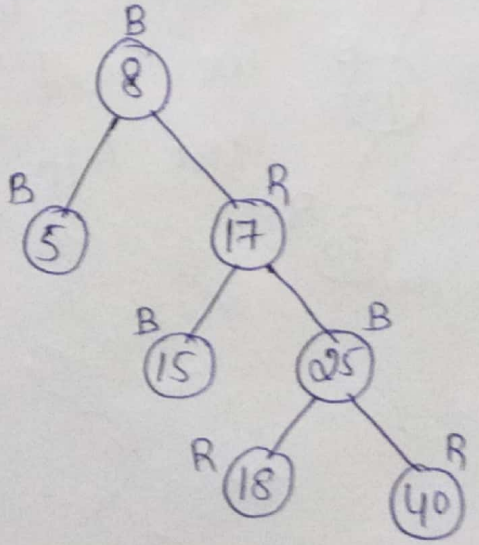


after Recoloring

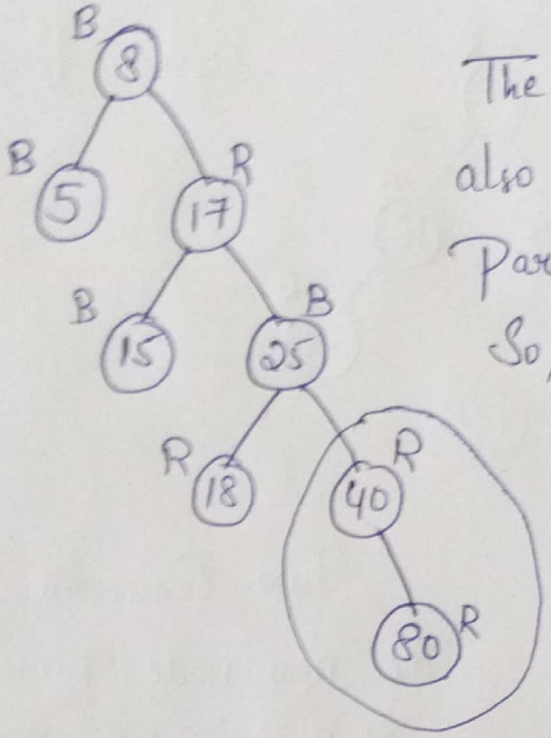
Insert "40"



Two Consecutive Red node [25 & 40]
The new node Parent's Sibling is Null. So we need a Rotation and Recolor.
Here we use LL Rotation.

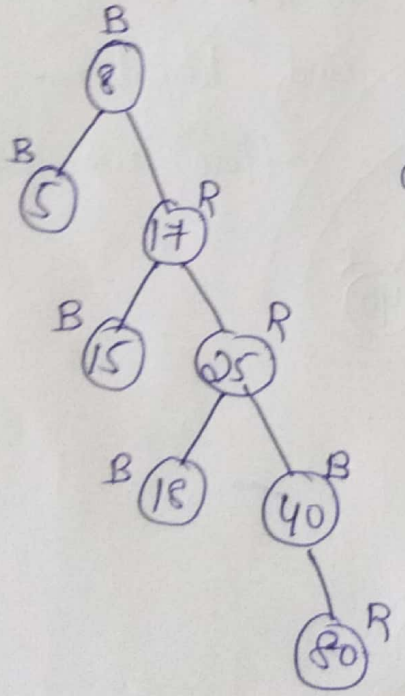


Insert '80"



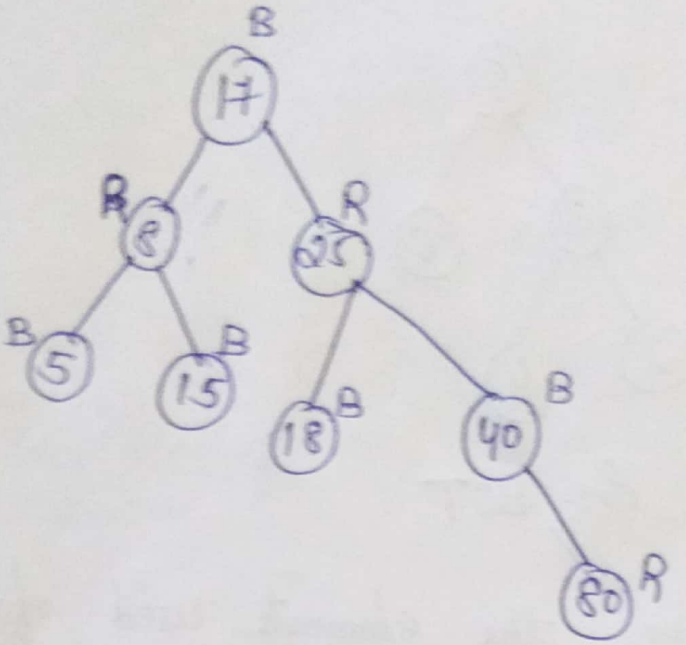
There are two Consecutive Red Color 40, 80.

The new nodes Parent Sibli also red color and Parent's Parent is not Root node. So, we use Recolor and Recheck.



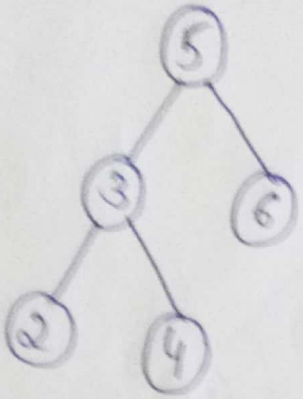
After Recolor again there are two Consecutive Red nodes 17, 25.

The new nodes Parent Sibling Color is black. So, we need Rotation. We use left Rotation & Recolor.



Splay Trees

1) Self Adjusted Binary Search Tree
2) In which every operation on Element rearranges the tree, so that the element is placed at the root position of the tree.
3) Every operation is performed at the root of the tree.
4) All the operations in Splay tree are involved with a Common operation called "Splaying".



Ex:- BST

whatever the elements used frequently and closer to the root node.

Time Complexity O(log N).

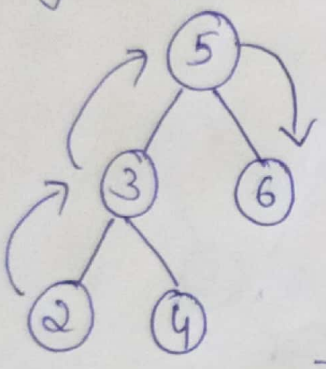
Splay an Element → from Current position to the Root node.

Rotations

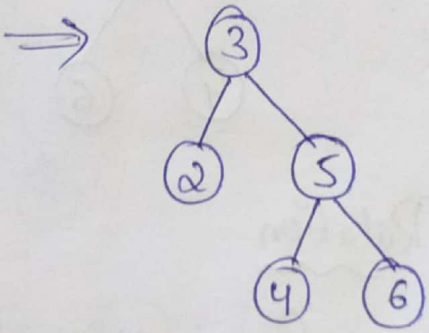
1) Zig Rotation
2) Zag Rotation
3) Zig-Zig Rotation
4) Zag-Zag Rotation
5) Zig-Zag Rotation
6) Zag-Zig Rotation

1) Zig Rotation Single Right Rotation [1 step]

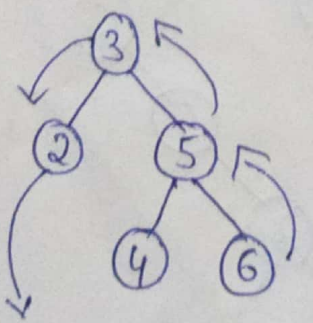
These are basically applied rotations in Splay trees. "Zig" means left and "Zag" means right. The root node moves from Current to right position.



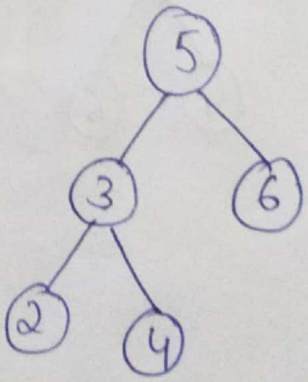
Splay (3) process of moving the "3" to the Root node.



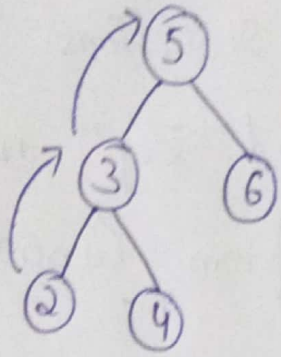
2) Zag Rotation :- Every node moves the Current position to left side.



Splay (5)

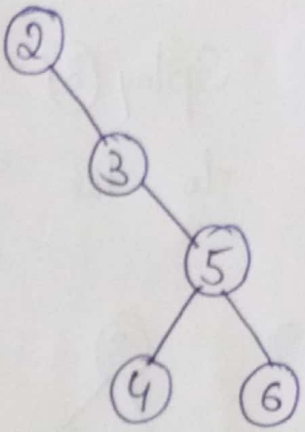


3) Zig - Zig Rotation

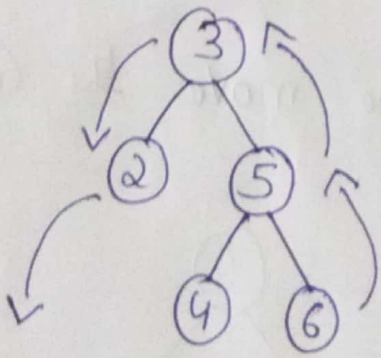


Every Nodes moves to right Position to its Current position.

Splay (2) then

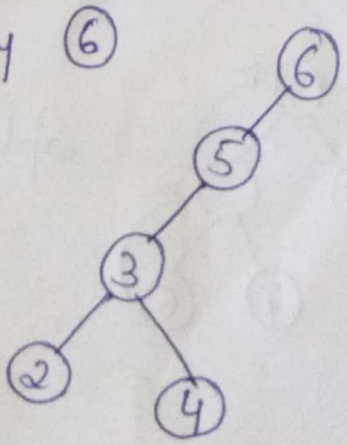


4) Zag - Zag Rotation

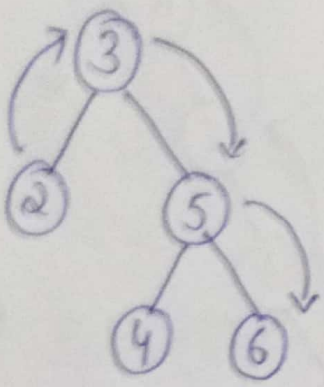


Every Nodes moves to left Position to its Current position.

Splay (6)

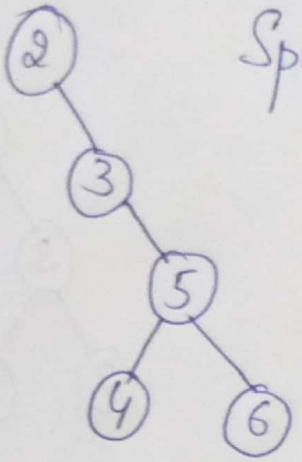


5) Zig - Zag Rotation



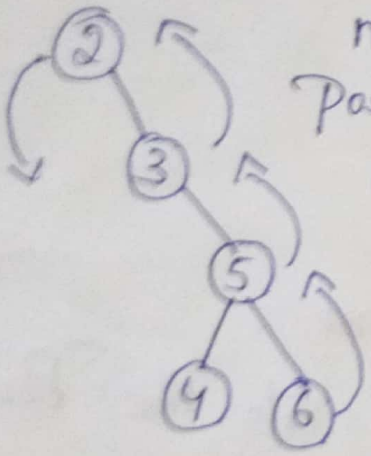
Zig.

Splay (2)

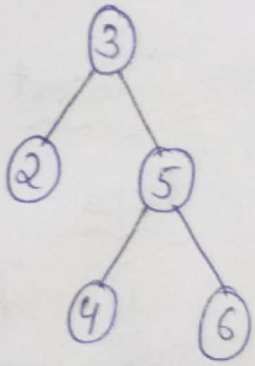


Splay (3)

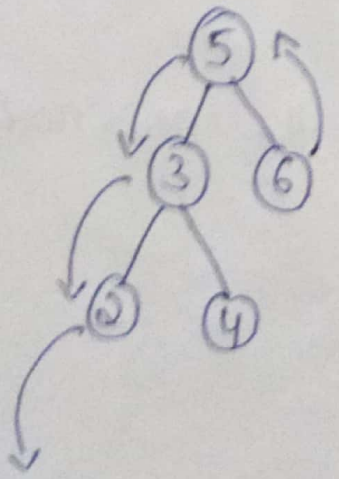
Zag.



moving one left Position from its Current position

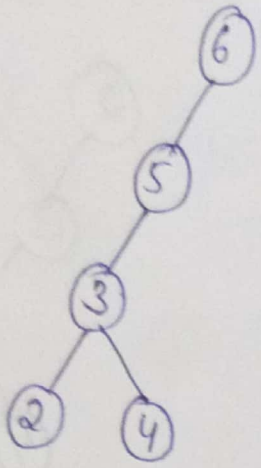


6) Zag - Zig Rotation

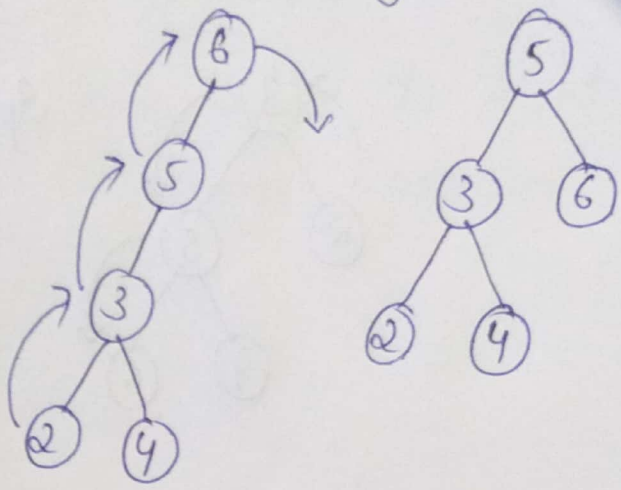


first perform Zag and next perform Zig Rotation.

Splay (6)



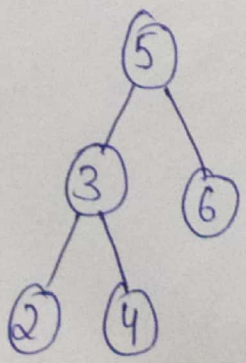
Splay (5) Zig Rotation



Insert operations

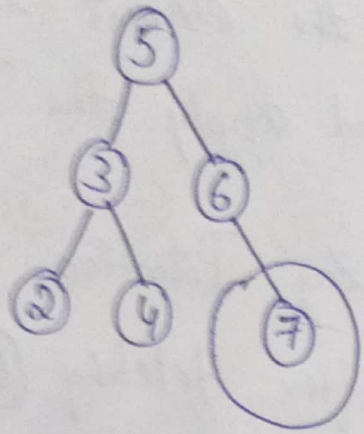
Steps

1) Tree is Empty or not.
2) Insert the newnode into the Root node.
3) Tree is not Empty. then insert the new nodes as a leaf node using binary Search Tree.
4) After Insert, Splay the new node.

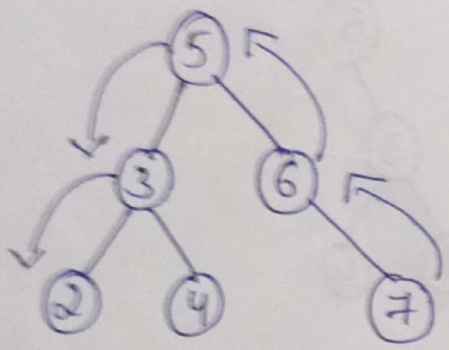


(Cont.. d]

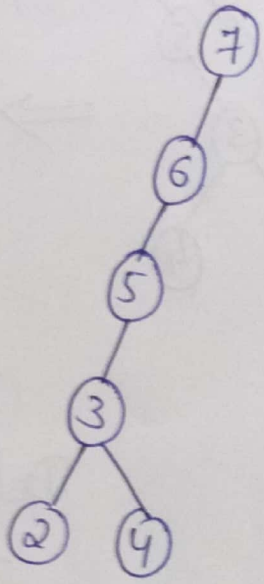
Insert 7



Splay (7) from its Current position to root node. Zag-Zag Rotation



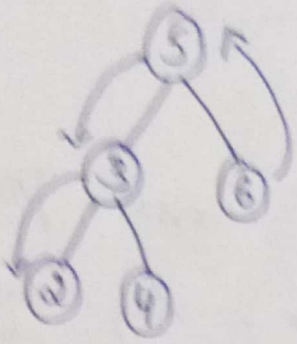
Zag-Zag Rotation



Delete operations

* Similar to BST
* We need the Splay tree.
* Then the move the element to root and delete the element.

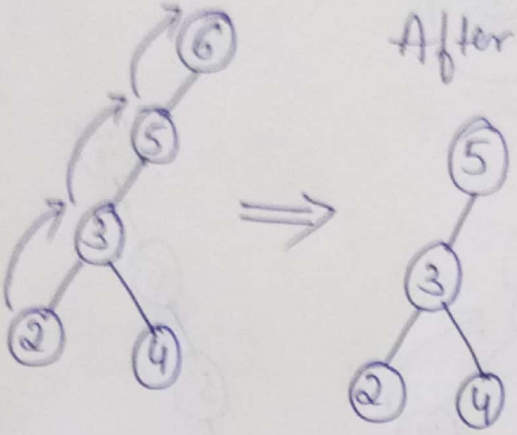
Ex:- Delete node (6)



move the element to the root node and apply the Rotation.

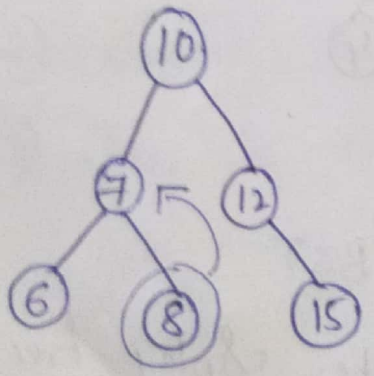
Splay (6) Zag Rotation

After deleting (6)



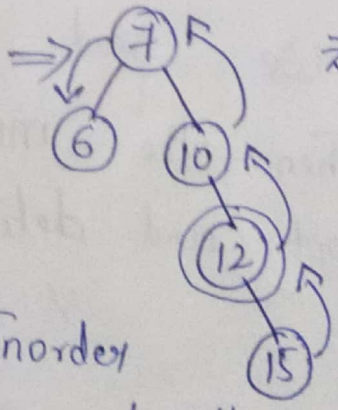
Ex:-

Delete node (8)

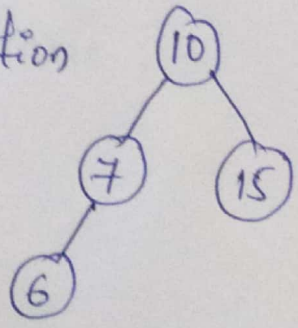


Inorder Successor. Then delete 8

Apply Zag Rotation.



Zag Rotation

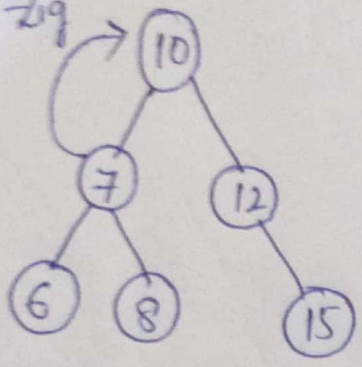


If delete 12, Inorder Successor will move to the deleted element place.

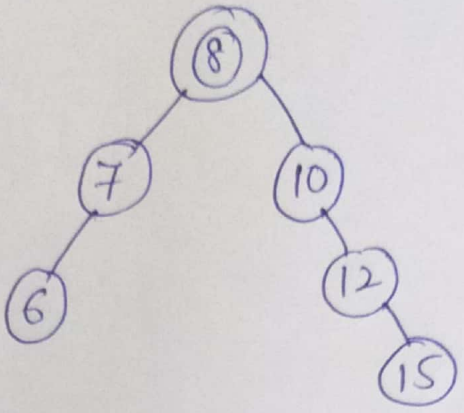
[Cont..d]

[Cont...d]

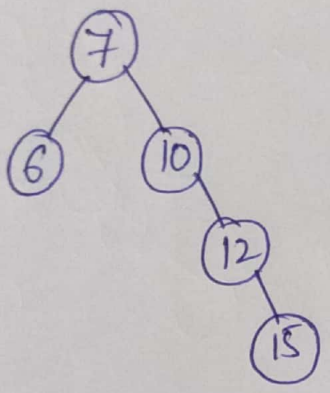
Zig



Delete 8



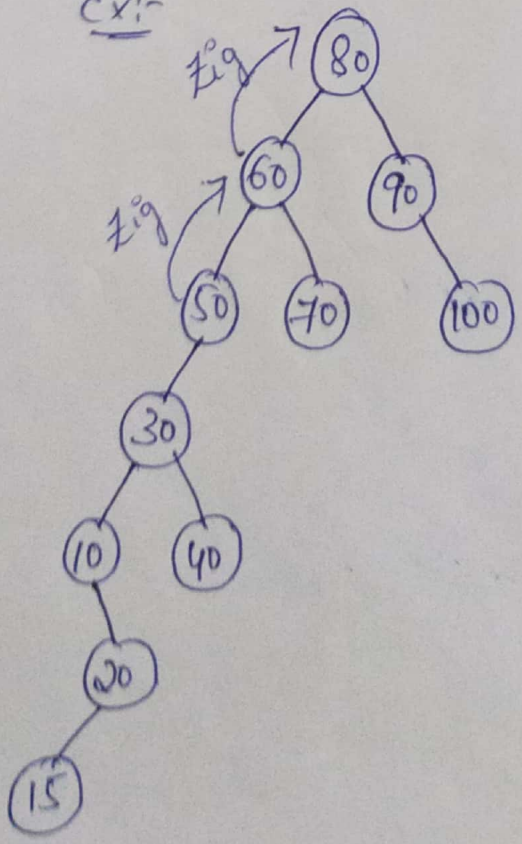
Splay (8) and delete



Ex:-

Zig

Zig



Delete 30