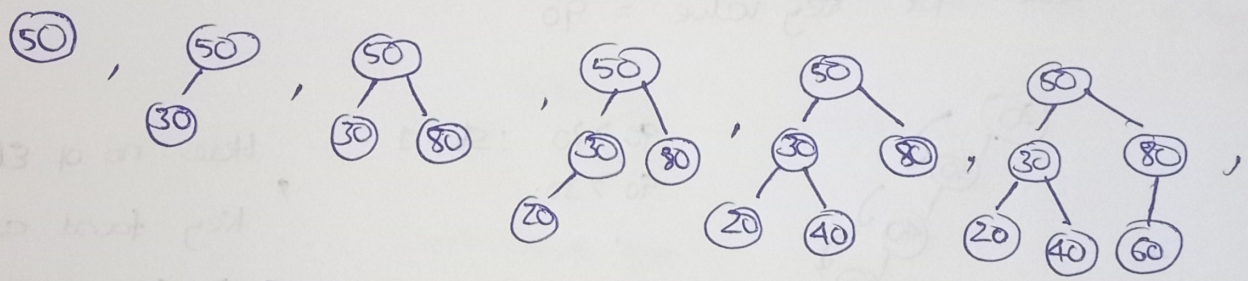
UNIT - V

Self Balanced Trees - AVL Trees & Priority Queues

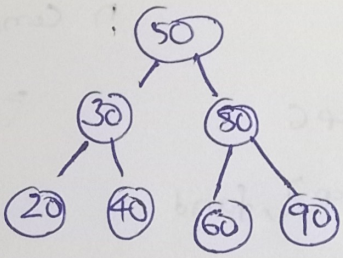
AVL Tree - AVL Tree is a self-balancing Binary Search Tree (BST) in which the difference between height of left sub tree and right sub tree is not more than one.

Before getting into the operations of AVL Tree, Let's see the Problem with Binary Search Tree

Construct BST for order of elements 50, 30, 80, 20, 40, 60, 90



Finally :



B.S.T 1

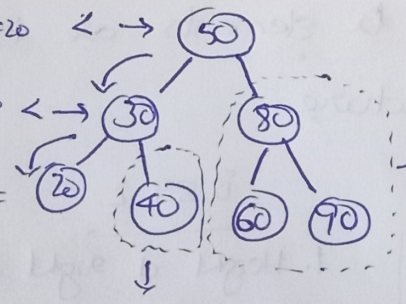
Now

Search for 20 in above B.S.T , key = 20

Step 1 ; key=20 <-> 50

Step 2 : key=20 <-> 30

Step 3 : key=20 = 20



Eliminated → for step 1

Element found

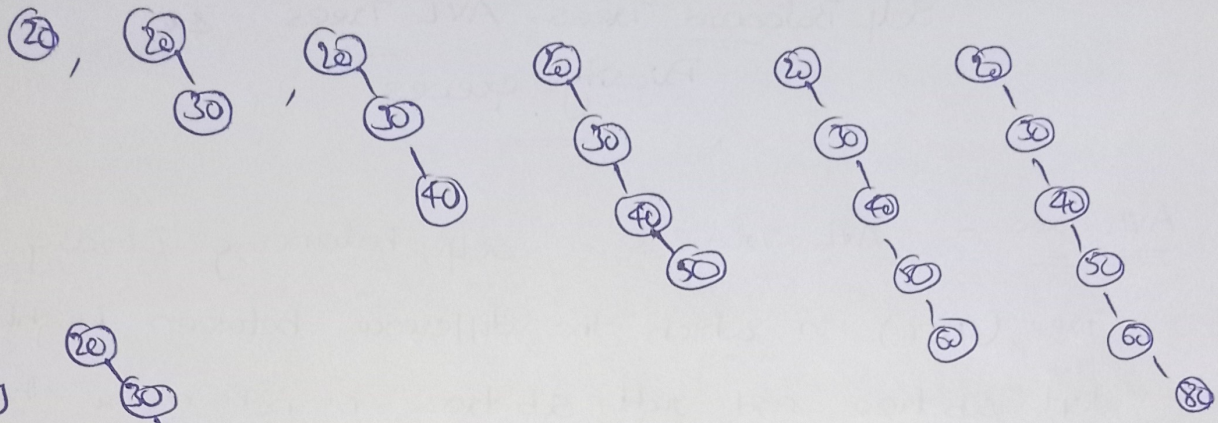
Eliminated after Step 2

for Elements = n

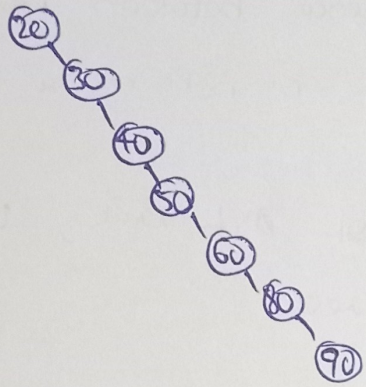
Time taken to search, key = $\log_2 n$

for n = 7 = 8, no of Comparisions done = $\log_2 8$ = 3

Construct BST for order of Elements 20, 30, 40, 50, 60, 80, 90

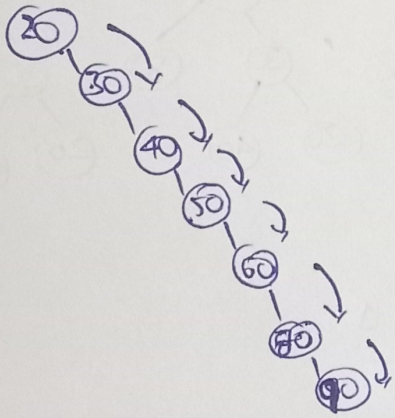


Finally



B.S.T 2

Now search for key value = 90



90 > 20 : Step 1
90 > 30
⋮
90 > 80 : Step 6
90 = 90 : Step 7, found

Here no of Elements = n
Key found after n comparisions
= O(n)

NOTE :

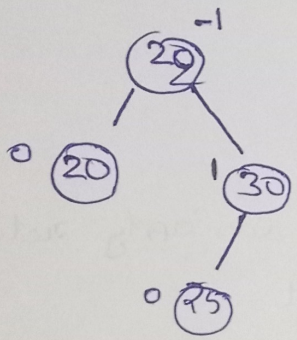
In the above two B.S.T, the Elements were same but the order in which the elements are inserted leads to different tree structure

B.S.T 1	B.S.T 2
1. Height of left and right subtree are same (balanced)	1. Height of right subtree is very high compared to left (known as Right skewed B.S.T)
2. Searching key takes less time = $\log_2 n$	2. Searching key takes longer time = O(n)

So to balance Binary Search Tree, Adelson-Velskii & Landis Proposed AVL Tree as Binary Search Tree which has balancing factor = {-1, 0, 1}

balancing factor = Height of Left Subtree − Height of Right Subtree

Ex:



→ Node (20) and (25) have no childs

balancing factor = 0

→ Node 30 has one left child & no right child

balancing factor = 1 − 0 = 1

→ Node 25

height of left Subtree = 1

" right " = 2

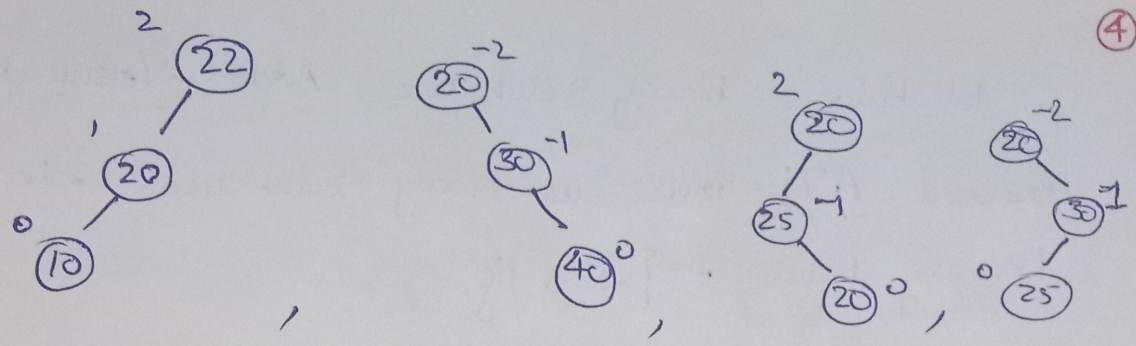
balancing factor = 1 − 2

= −1

For all Nodes balancing factor is in range of {1, 0, −1}

=> The above B.S.T is an AVL Tree ✓

Ex:

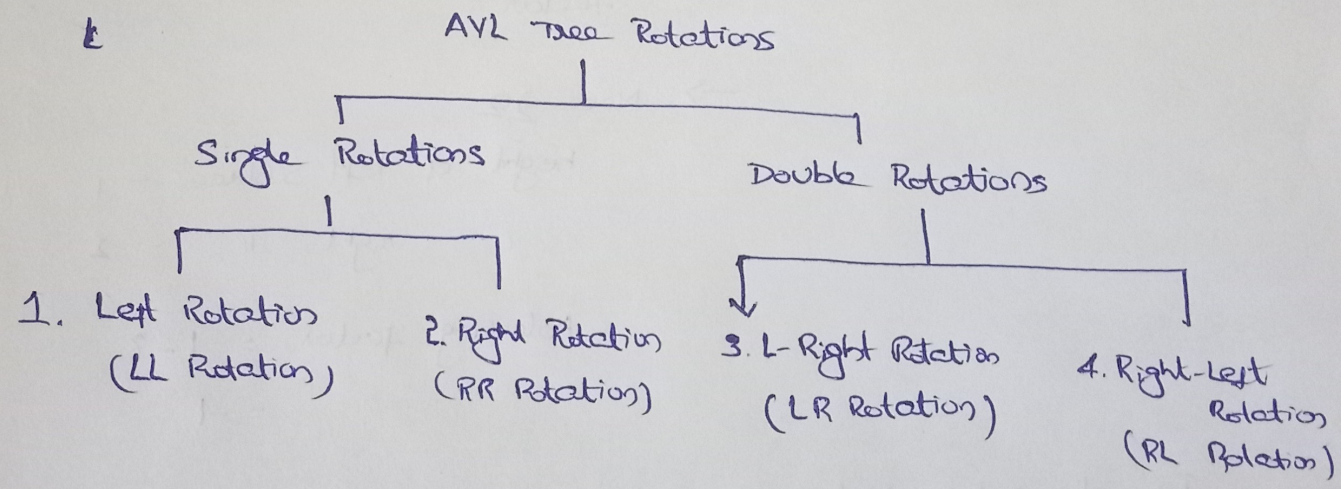


The above Tree are No AVL Tree, bcz at root node balancing factor is not in range of {1, 0, -1}

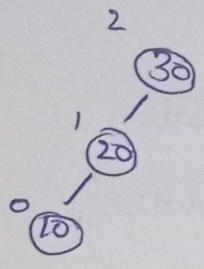
How to balance a B.S.T ?

When a tree is imbalanced, we can apply rotations on unbalanced node and balance it

Types of Rotations



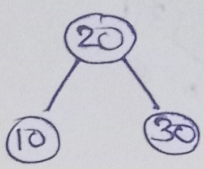
Case (i)



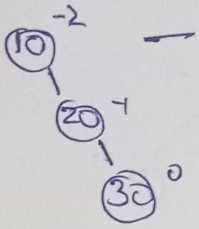
→ This node is imbalanced, bcz of Left Sub tree

Apply

RR Rotation



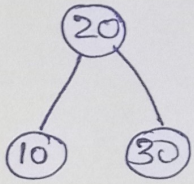
Case (ii)



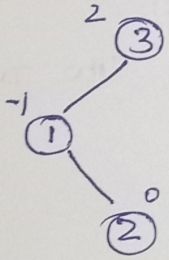
— This Node is imbalanced, bcz of right sub tree

Apply

LL Rotation



Case (iii)



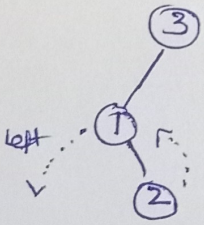
— The Node is imbalanced,

Apply LR Rotation

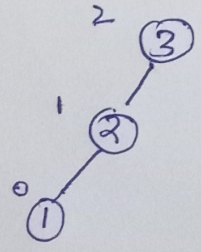
LL + RR Rotation

1 @ Node 1

2 @ Node 3

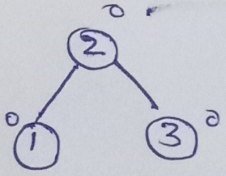


1. Left Rotation



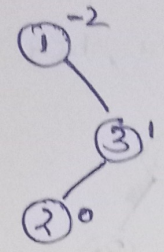
Apply

2. Right Rotation



Tree is balanced

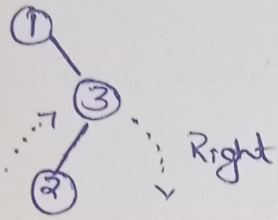
Case iv,



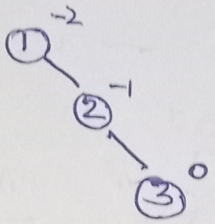
Apply RL Rotation

(RR + LL Rotation)

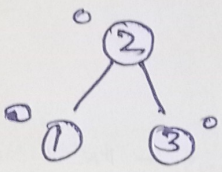
① ②



① Right Rotation RR



② Apply Left Rotation LL



The Tree is balanced

operations on AVL Tree

(i) Insertion

(ii) Search

(iii) Deletion

AVL Insertion

To insert a node in AVL Tree, follow the same Procedure of B.S.T insertion. After insertion if there is an imbalance, Apply AVL Rotations and balance the tree

Case 1, If there is an imbalance in left child of left sub tree then you Perform a right rotation

Case 2, If there is an imbalance in right child of right sub tree then you Perfor a left rotation

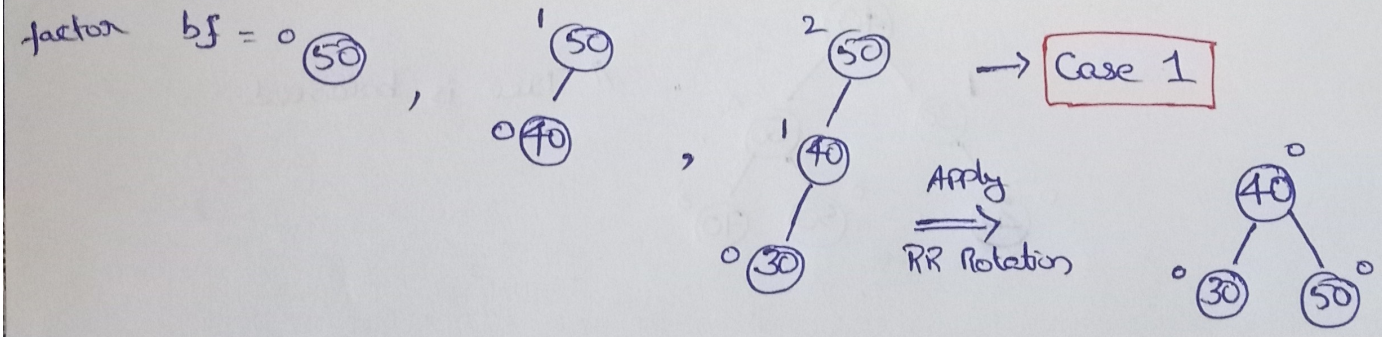
Case 3, If there is an imbalance in left child of right subtree, then Perform RL Rotation (RR + LL)

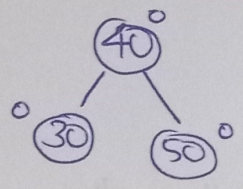
Case 4, If there is an imbalance in right child of left subtree, then Perform LR Rotation (LL + RR)

Construct AVL tree for following Elements

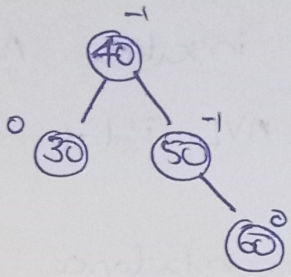
50, 40, 30, 60, 70, 20, 25, 80, 75

balancing factor bf =





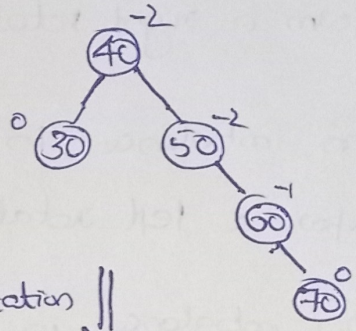
Insert next Element 60,



// Tree is balanced

No rotations required

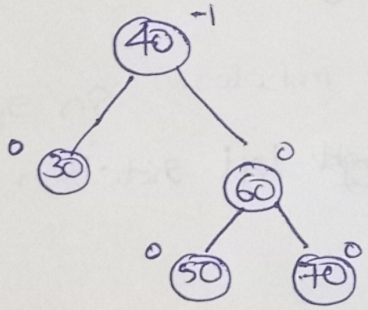
Insert next Element 70,



// Tree is imbalanced

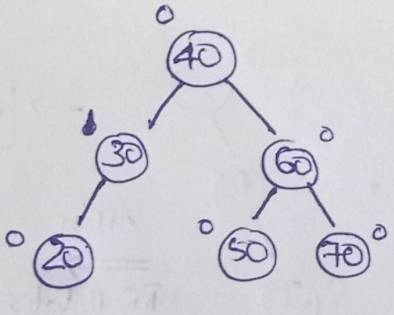
of Case 2.

apply LL rotation at node 50



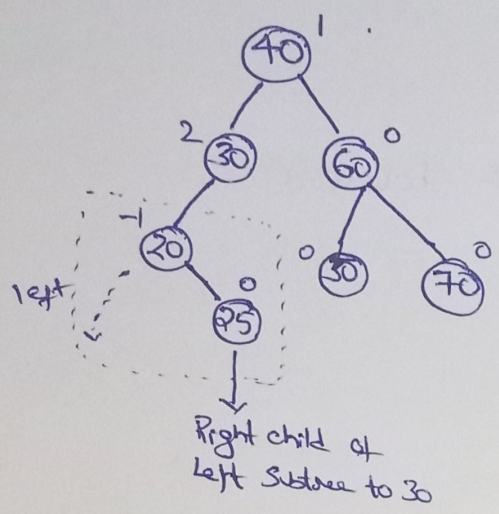
// Tree is balanced

Insert next Element 20,



// Tree is balanced

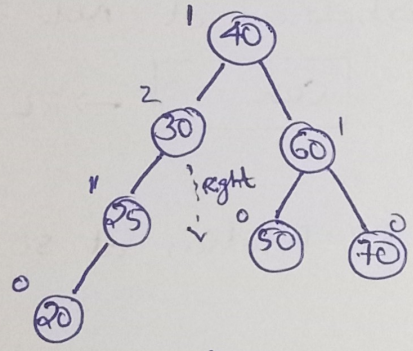
Insert next Element 25,



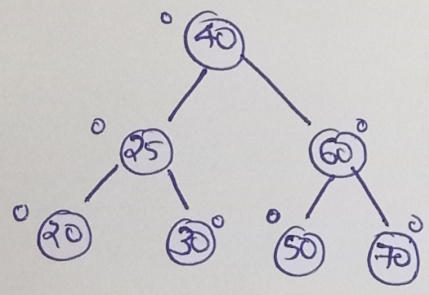
Tree is imbalanced at Node 30

Case 4 → L'R Rotation

① Apply Left Rotation at 20

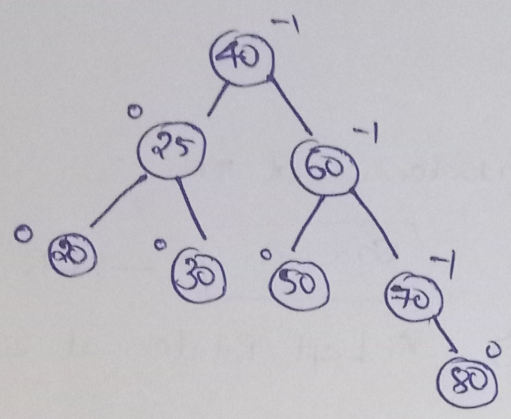


② Apply Right Rotation at 30



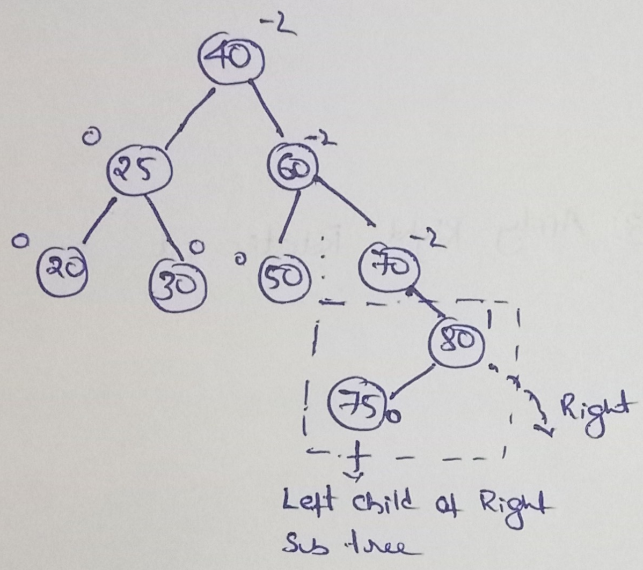
The Tree is balanced

Now next Element is 80



Tree is balanced

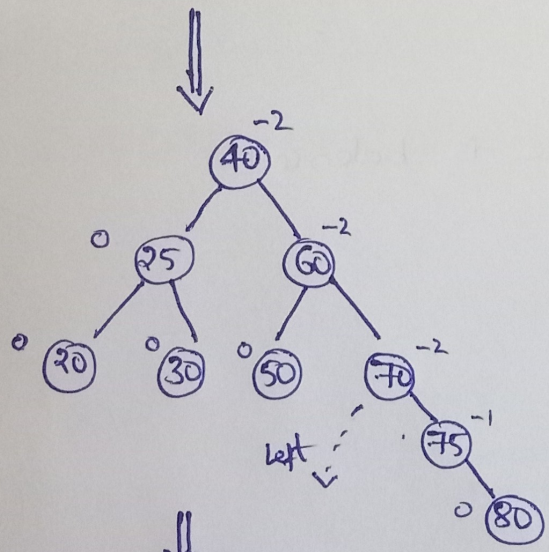
Next Element is 75,



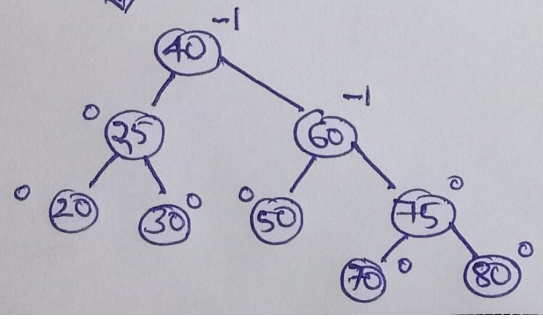
Tree is imbalanced at Node 70

Case 3 → RL Rotation

① Apply RR Rotation at 80



② Apply LL Rotation at 70



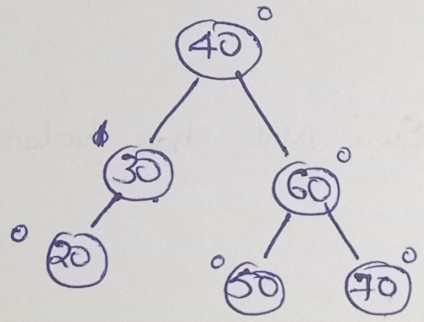
⇒ Tree is balanced

AVL Search

Search operation in AVL Tree is same as in B.S.T

1. If key value is less than Root, Search in Left Subtree
2. If key value is greater than Root, Search in Right Subtree
3. If key value is Equal to root, Print Element found and terminate
4. Print Element Not found if node is not Equal to key and node has no child Element

Ex:



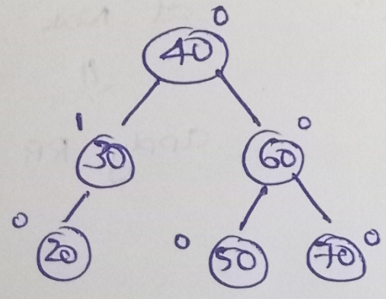
Search key = 60

① Root = 40, key = 60

key > Root, goto Right subtree

② New Root = 60, key = 60

key = = Root, Element found



Search key = 10

① Root = 40, key = 10

key < Root, goto Left subtree

② Root = 30, key = 10

key < Root, goto left subtree

③ Root = 20, key = 10

key < Root, goto left subtree

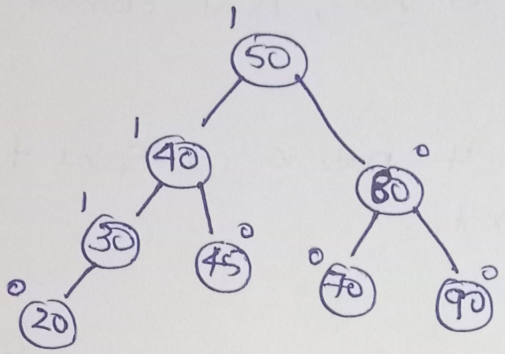
④ child doesn't Exist, NULL

=> Print Element Not Found

AVL Tree Deletion

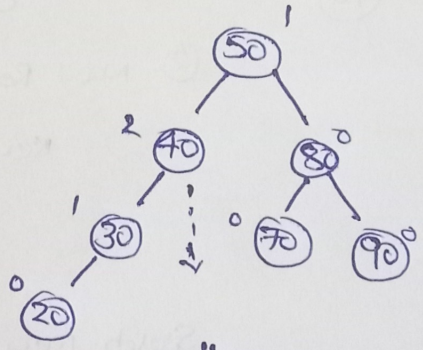
Deletion in AVL Tree is same as in Binary Search After deleting Node, if tree is imbalanced apply required Rotation and balance the Tree

Ex:-



At Present Tree is balanced Every Node has balancing factor under {-1, 0, 1}

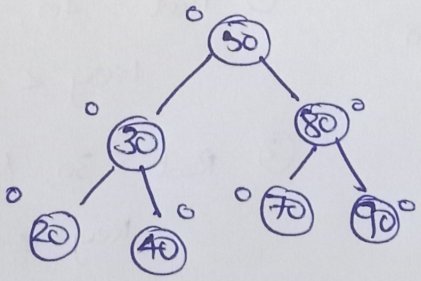
Now Delete (45)



Tree is imbalanced at Node 40

⇓

apply RR Rotation



Now Tree is balanced