Sunbeam Institute of Information Technology
Pune and Karad

Data structures and Algorithms

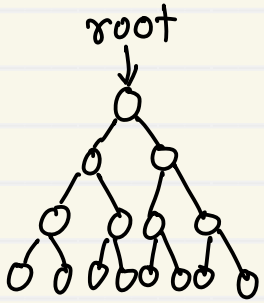
Trainer - Devendra Dhande

Email – devendra.dhande@sunbeaminfo.com

BST - Time complexity of operations

Capacity:
max number of nodes that can be added into a binary tree for give height.

height	Node
-1	0
0	1
1	3
2	7
3	15



h - height of binary tree
n - no. of elements in binary tree

$$n = 2^{h+1} - 1$$

$$n = 2^{h+1} - 1$$
$$2^h \approx n$$
$$\log 2^h = \log n$$
$$h \log 2 = \log n$$
$$h \approx \frac{\log n}{\log 2}$$

Time $\propto$ Height

Time $\propto \frac{\log n}{\log 2}$

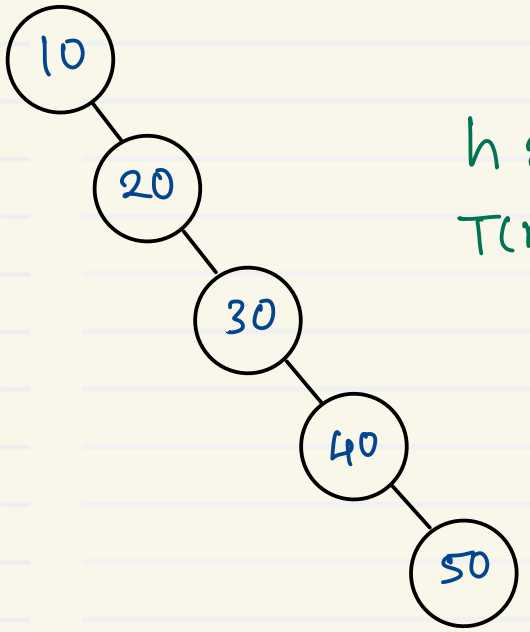
$$T(n) = O(\log n)$$

Add :	O(h)	O(log n)
Delete :	O(h)	O(log n)
Search:	O(h)	O(log n)
Traverse :		O(n)

Skewed Binary Tree

- In binary tree if only left or right links are used, tree grows only in one direction such tree is called as skewed binary tree
  - Left skewed binary tree
  - Right skewed binary tree
- Time complexity of any BST is O(h)
- Skewed BST have maximum height ie same as number of elements.
- Time complexity of searching is skewed BST is O(n)

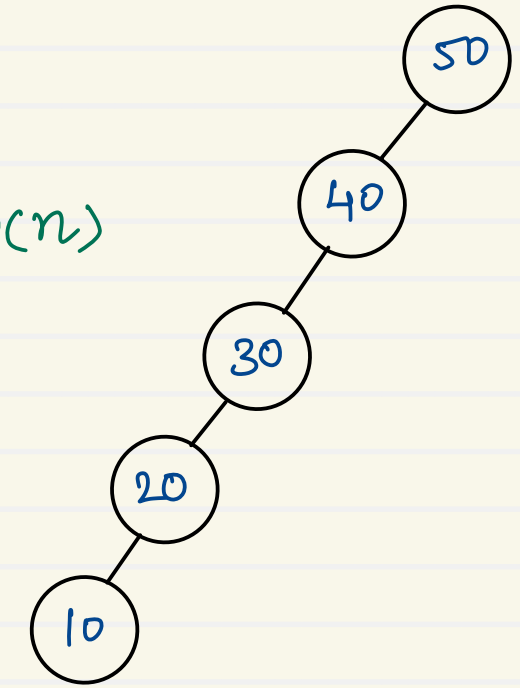
Keys : 10, 20, 30, 40, 50



$h \approx n$

$T(n) = O(n)$

Keys : 50, 40, 30, 20, 10



Balanced BST

- To speed up searching, height of BST should be minimum as possible
- If nodes in BST are arranged, so that its height is kept as less as possible, is called as Balanced BST

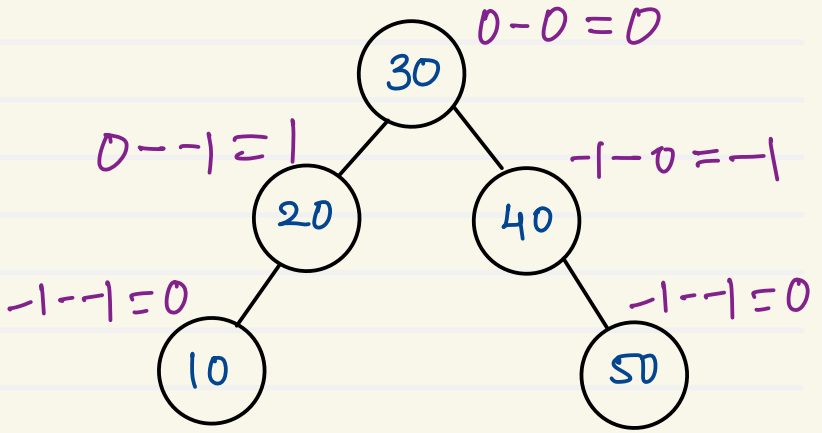
Types :
1. AVL Tree
2. Red Black Tree
3. Splay Tree

Balance factor = Height (left sub tree) - Height (right sub tree)

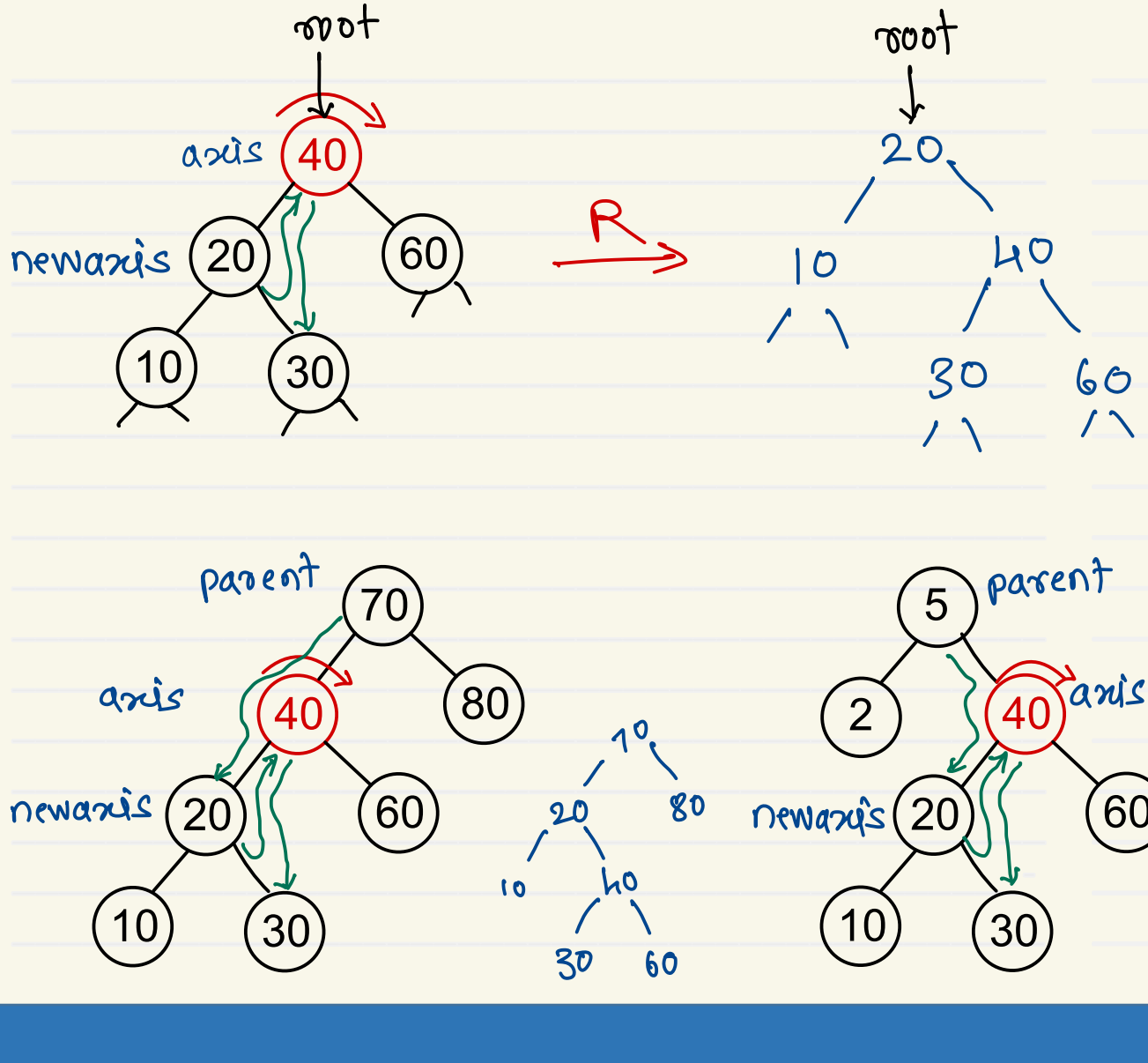
- tree is balanced if balance factors of all the nodes is either -1, 0 or +1
- balance factors = {-1, 0, +1}
- A tree can be balanced by applying series of left or right rotations on imbalance nodes

nodes having balance factor other than {-1, 0, +1}

Keys : 30, 40, 20, 50, 10

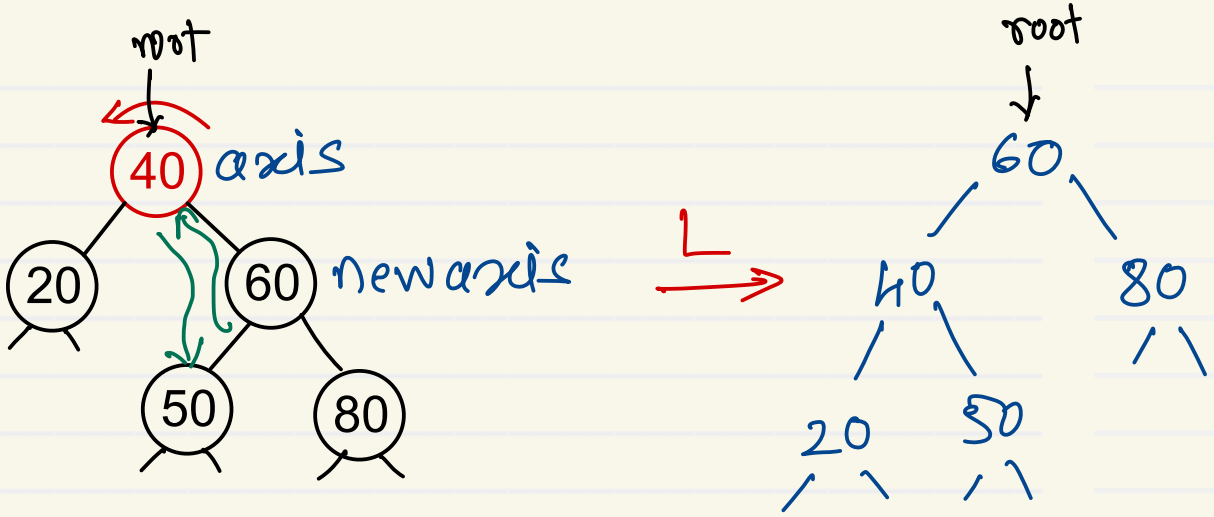


Right Rotation

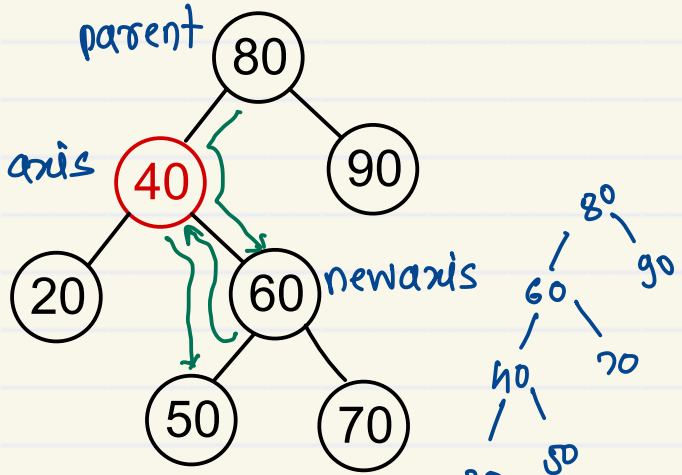


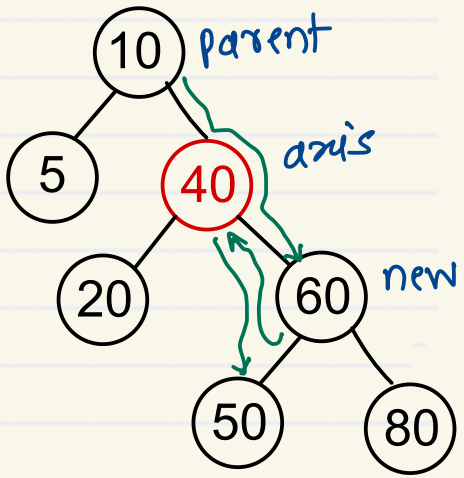

```
void rightRotation( axis, parent){
    newaxis = axis.left;
    axis.left = newaxis.right;
    newaxis.right = axis;
    if(axis == root)
        root = newaxis;
    else if( axis == parent.left)
        parent.left = newaxis;
    else if( axis == parent.right)
        parent.right = newaxis;
```

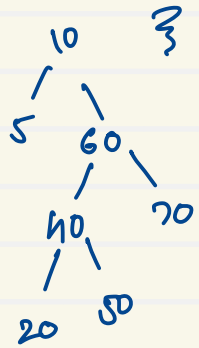
Left Rotation




```
void leftRotation(axis, parent){
    newaxis = axis.right;
    axis.right = newaxis.left;
    newaxis.left = axis;
    if(axis == root)
        root = newaxis;
    else if(axis == parent.left)
        parent.left = newaxis;
    else if(axis == parent.right)
        parent.right = newaxis;
```



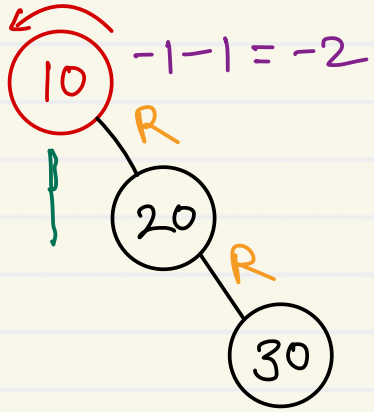




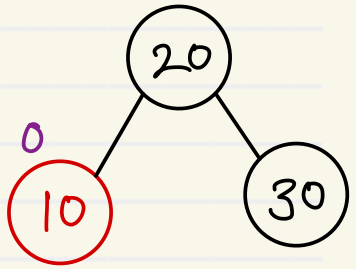
Rotation cases : Right-Right case (RR Imbalance)

↳ apply left rotation on imbalance node

Keys : 10, 20, 30



Left →

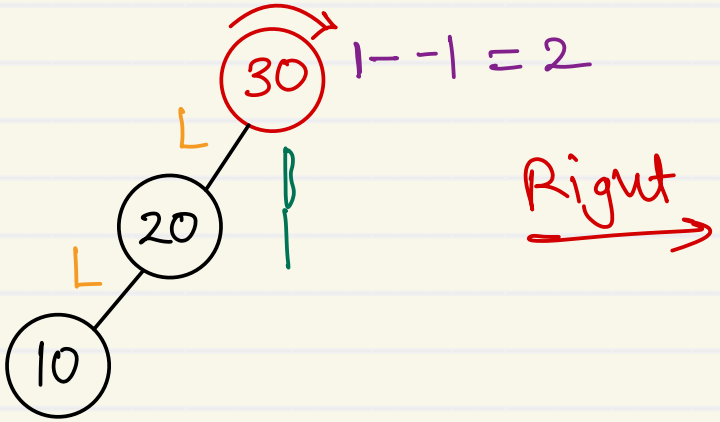


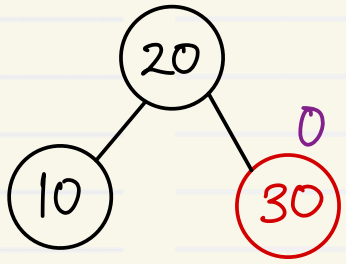

```
if( bf < -1 && value > trav.right.data) {
        leftRotation(trav, parent);
}
```

Rotation cases : Left-Left case (LL Imbalance)

↳ apply right rotation on imbalance node

Keys : 30, 20, 10



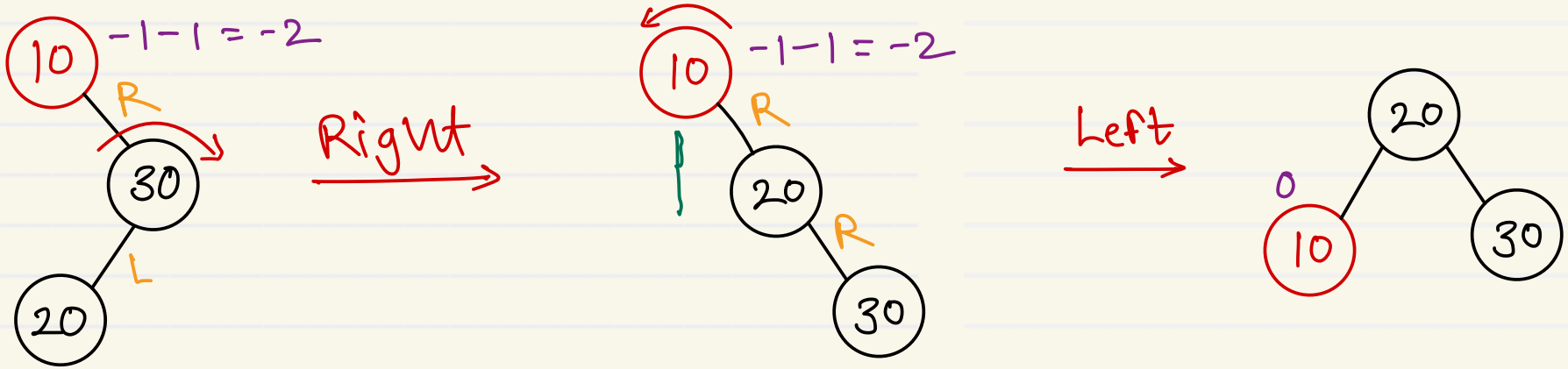



```
if( bf > +1 && value < trav.left.data){
        rightRotation(trav, parent);
}
```

Rotation cases : Right-Left case (RL Imbalance)

↳ first apply right rotation on right of imbalance node
second apply left rotation on imbalance node

Keys : 10, 30, 20

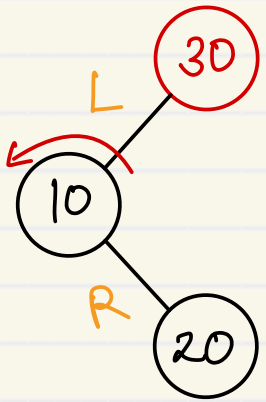



```
if(bf < -1 && value < trav.right.data){
    rightRotation(trav.right, trav)
    leftRotation(trav, parent)
}
```

Rotation cases : Left-Right case (LR Imbalance)

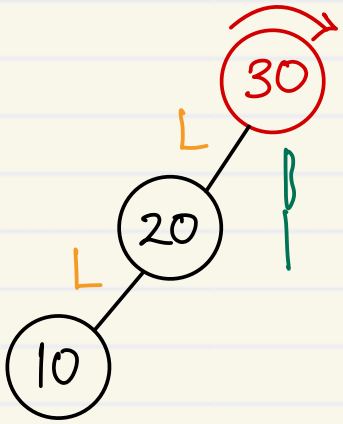
↳ first apply left rotation on left of imbalance node
second apply right rotation on imbalance node

Keys : 30, 10, 20



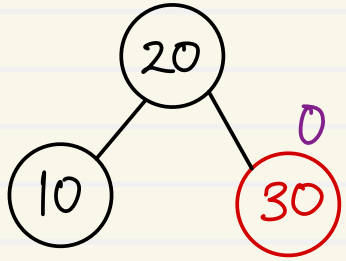
1 - -1 = 2

Left



1 - -1 = 2

Right

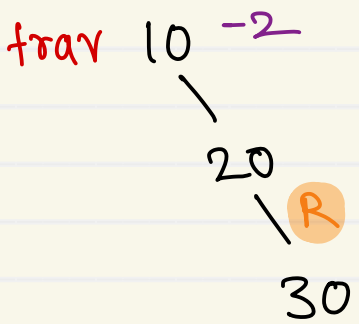



```
if( bf > +1 && value > trav.left.data){
    leftRotation(trav.left, trav);
    rightRotation(trav, parent);
}
```

Balance factor = {-1, 0, +1}

Bf < -1

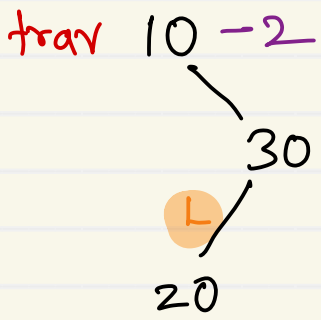
RR



value = 30

value > trav.right.data

RL

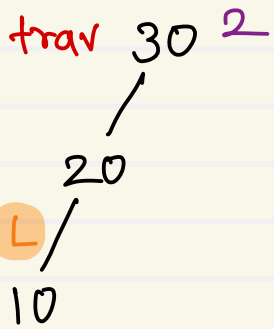


value = 20

value < trav.right.data

Bf > +1

LL



value = 10

value < trav.left.data

LR

value = 20

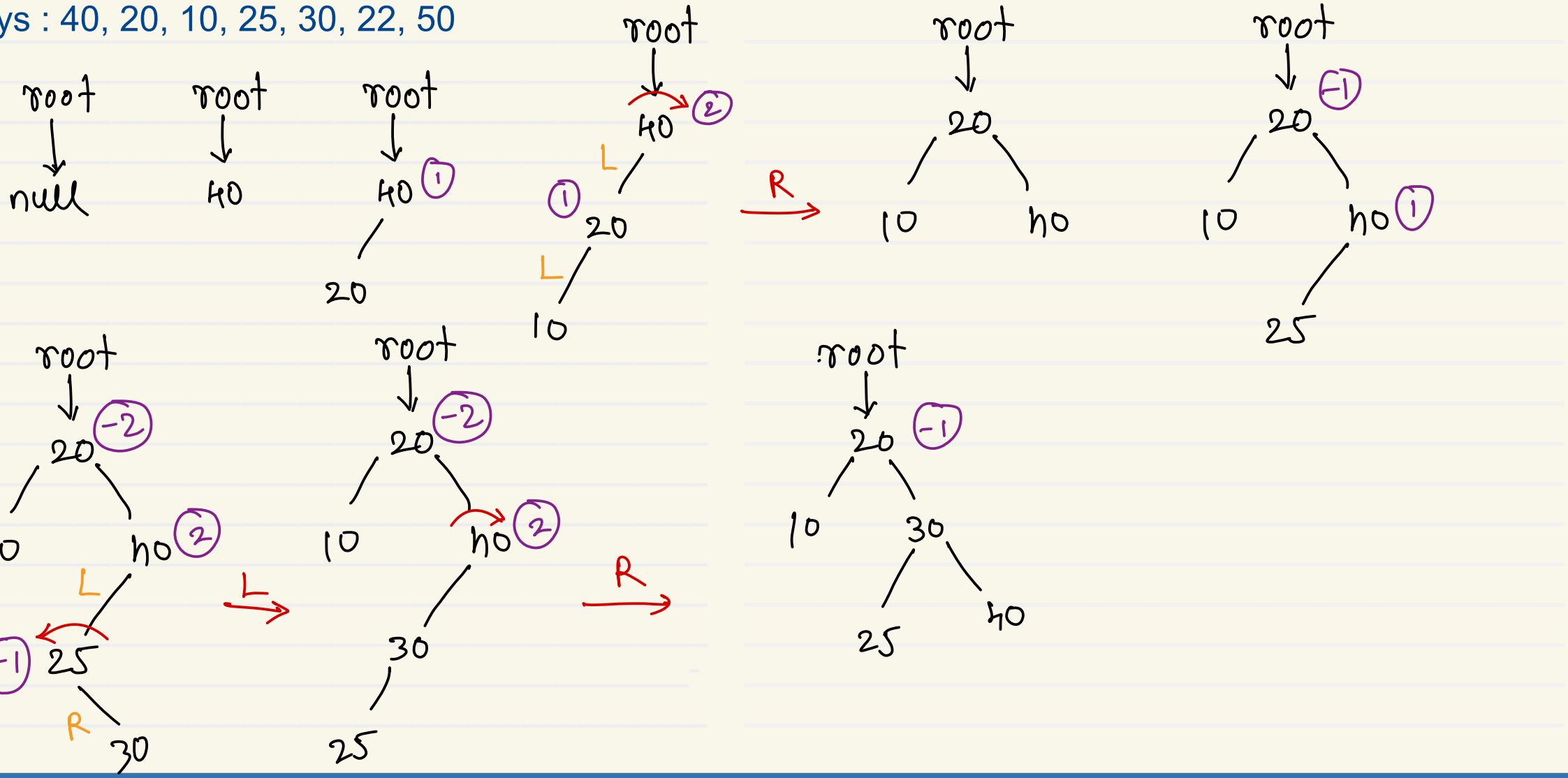
value > trav.left.data

AVL Tree

- self balancing binary search tree
- on every insertion and deletion of a node, tree is getting balanced by applying rotations on imbalance nodes
- The difference between heights of left and right sub trees can not be more than one for all nodes
- Balance factors of all the nodes are either -1 , 0 or +1
- All operations of AVL tree are performed in O(log n) time complexity

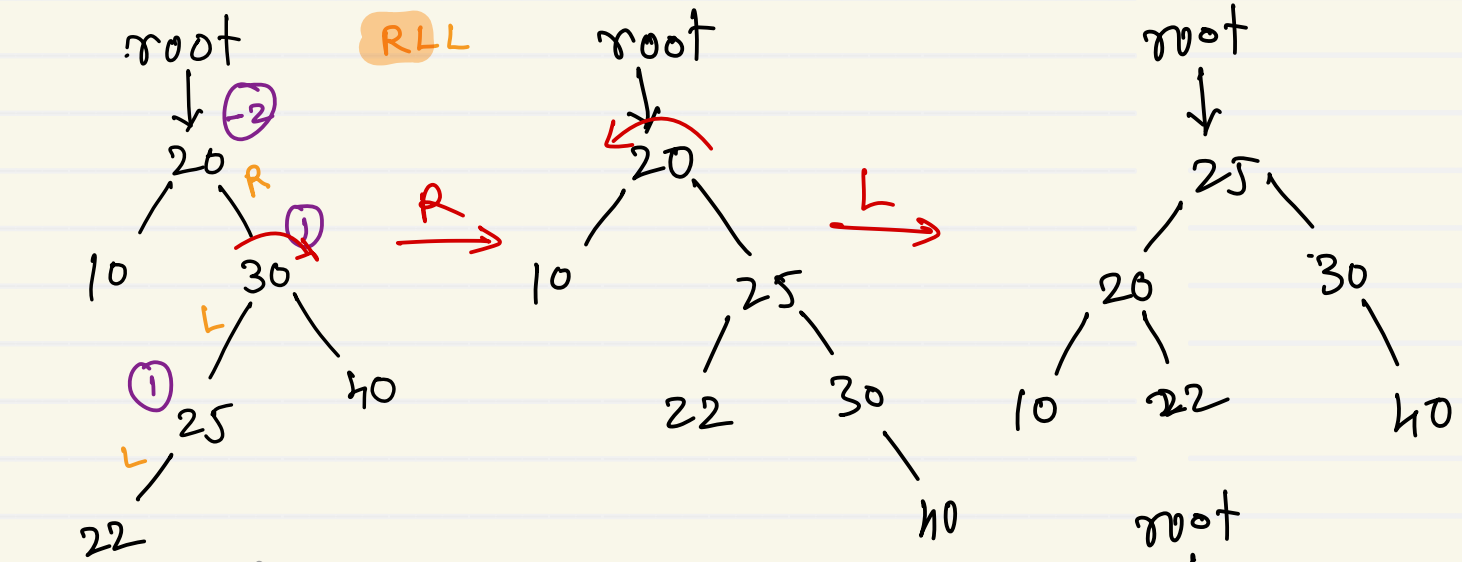
AVL Tree - Add Node

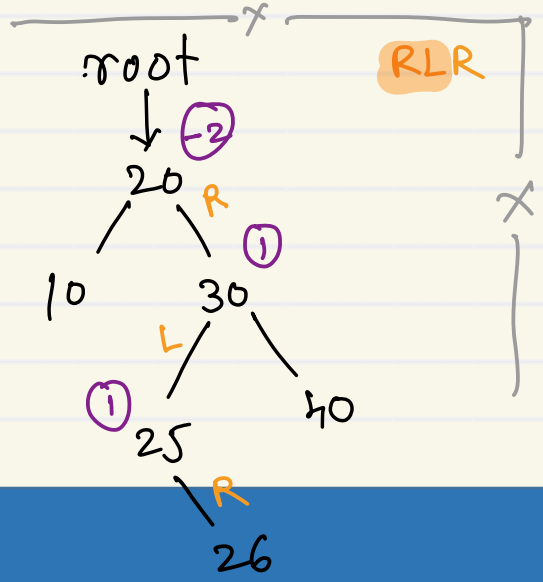
Keys : 40, 20, 10, 25, 30, 22, 50

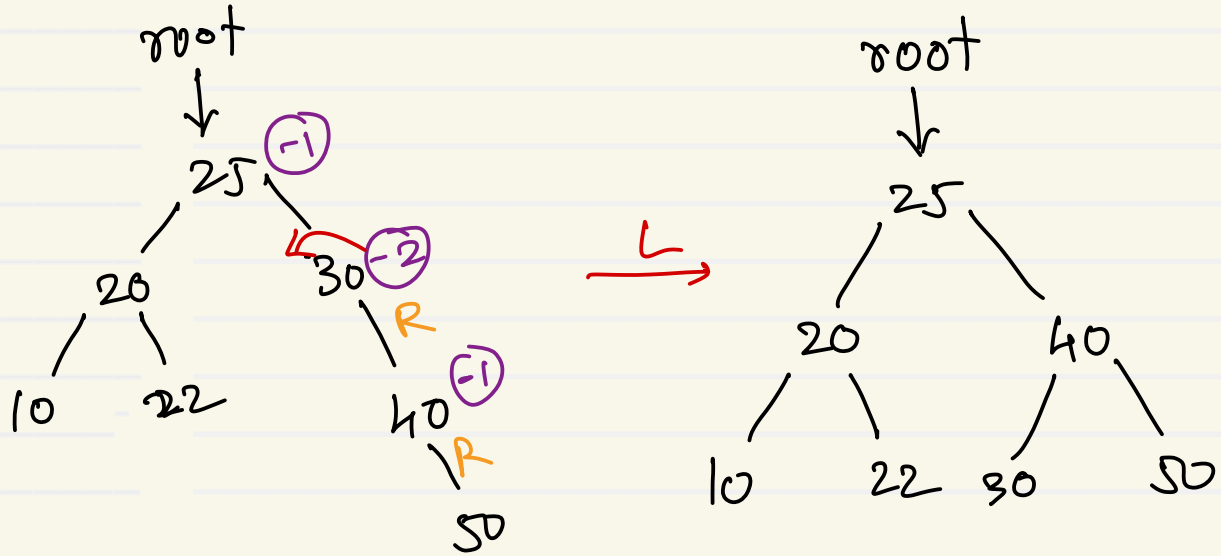


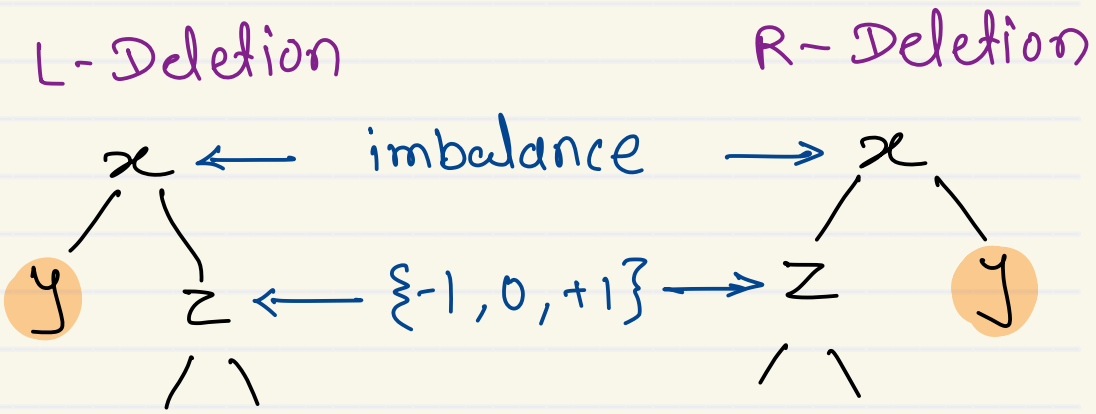
AVL Tree - Add Node

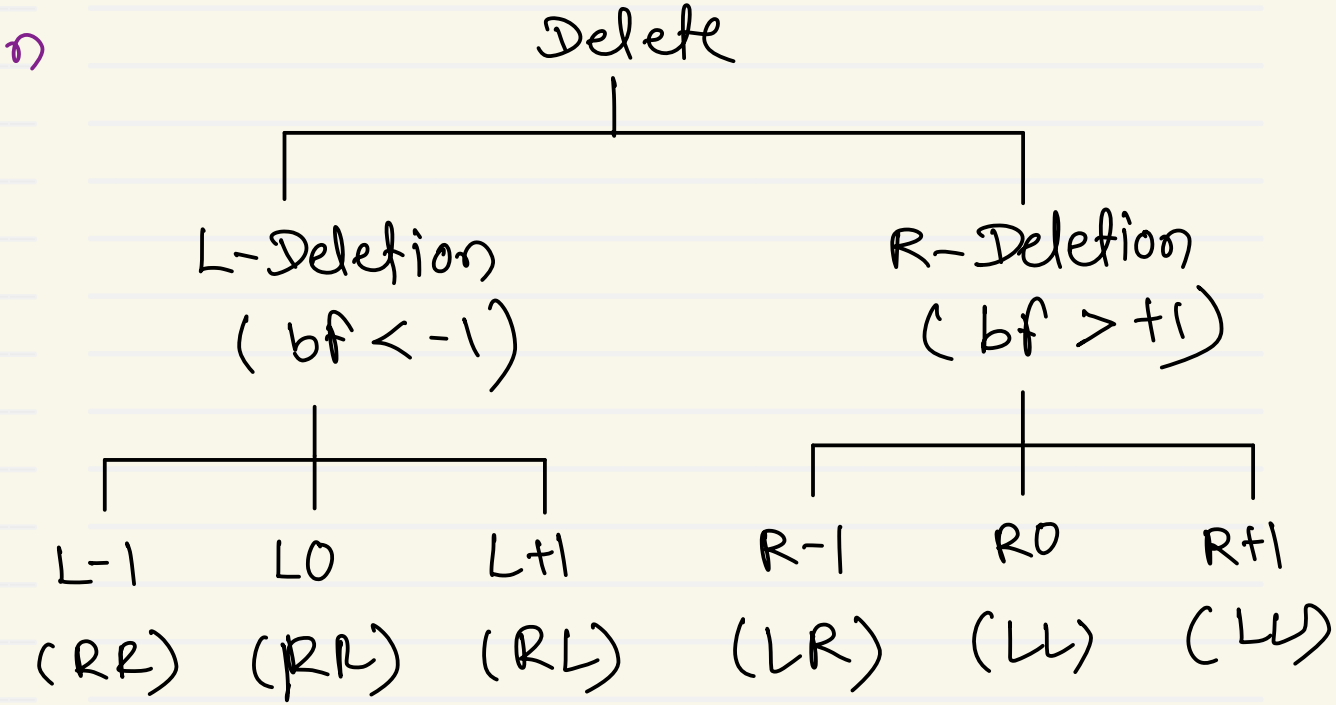
Keys : 40, 20, 10, 25, 30, 22, 50



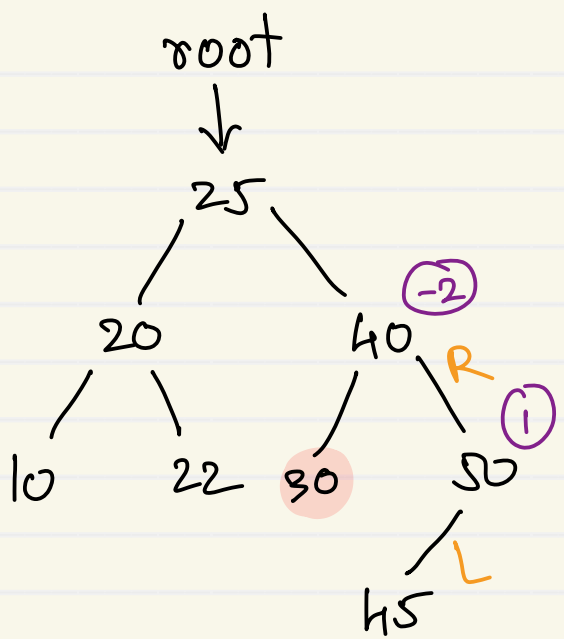




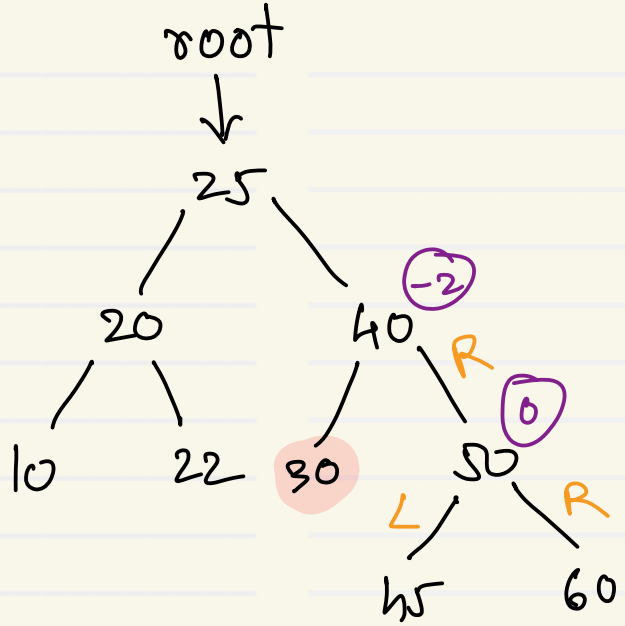

L-Deletion
R-Deletion
x
imbalance
x
y
z
{-1, 0, +1}
z
y


Delete
L-Deletion
(bf < -1)
R-Deletion
(bf > +1)
L-1
L0
L+1
R-1
R0
R+1
(RR)
(RR)
(RL)
(LR)
(LL)
(LL)

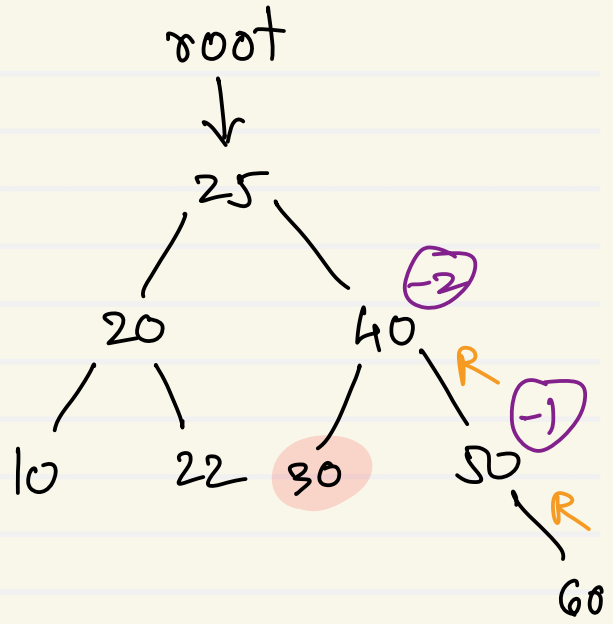
AVL Tree - Delete Node



L+1
(RL)

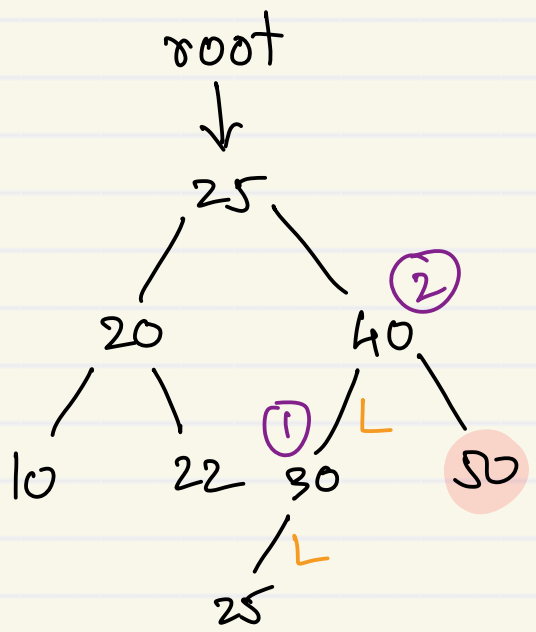


LO
(RR)



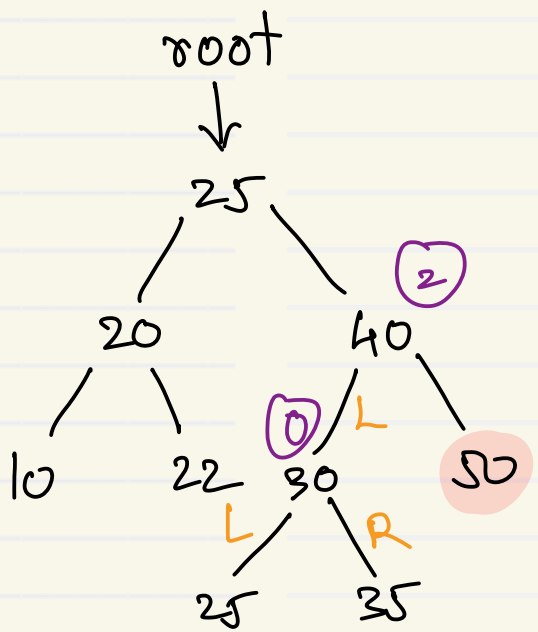
L-1
(RR)

AVL Tree - Delete Node



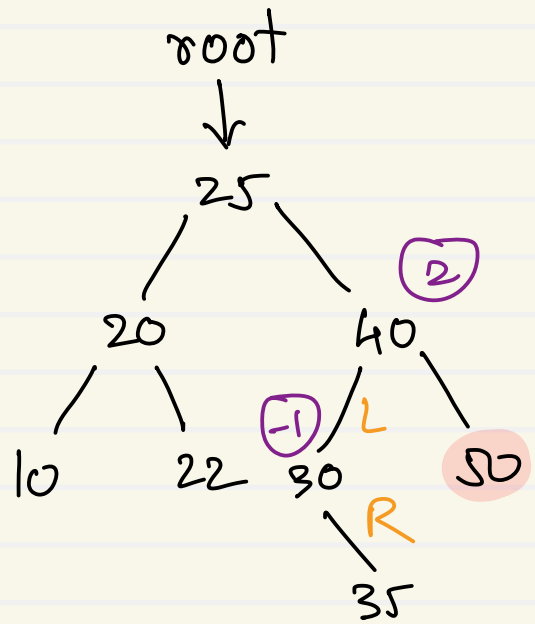
R+1

(LL)



RO

(LL)



R-1

(LR)