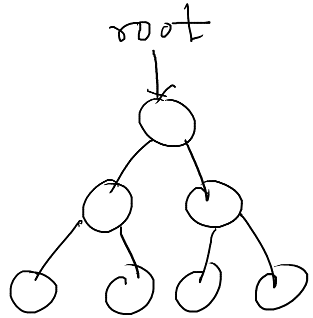
Binary Search Tree

h – height of BST
n – number of nodes in BST

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



h	n
-1	0
0	1
1	3
2	7
3	15

Add : $O(h)$ or $O(\log n)$

search : $O(h)$ or $O(\log n)$

Delete : $O(h)$ or $O(\log n)$

Traversal : $O(n)$

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

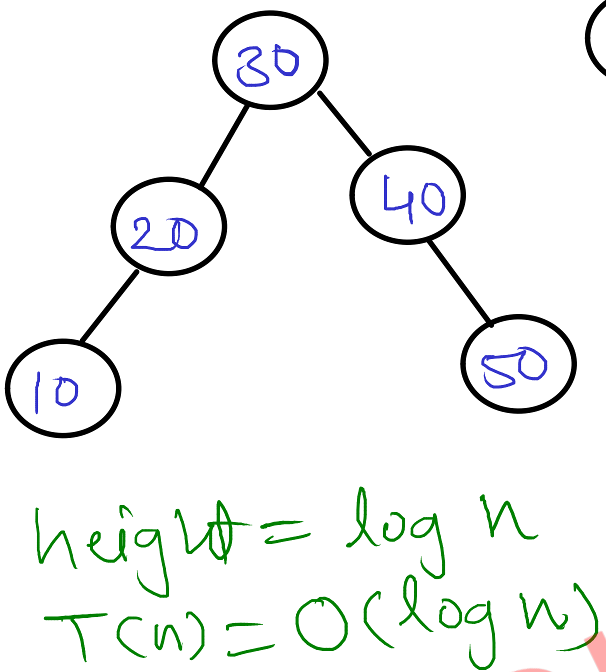
– In BST, time complexity of add, delete & search is dependent on height

$\text{Time} \propto h$

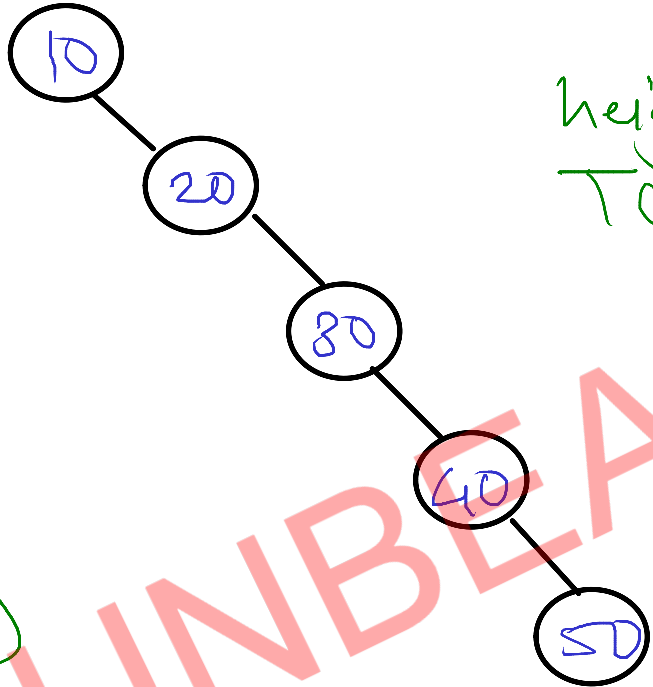
$\text{Time} \propto \frac{\log n}{\log 2}$

Skewed BST

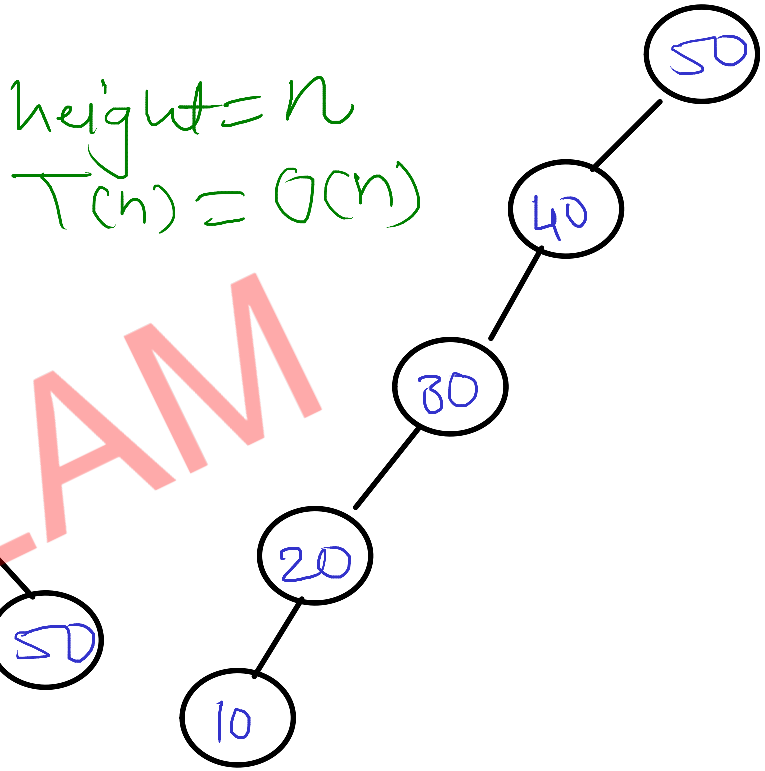
Keys : 30, 40, 20, 50, 10



Keys : 10, 20, 30, 40, 50



Key : 50, 40, 30, 20, 10



- if BST is growing in only one direction, then it is known as Skewed BST
- if BST is growing in only left direction, then it is known as left skewed BST
- if BST is growing in only right direction, then it is known as right skewed BST

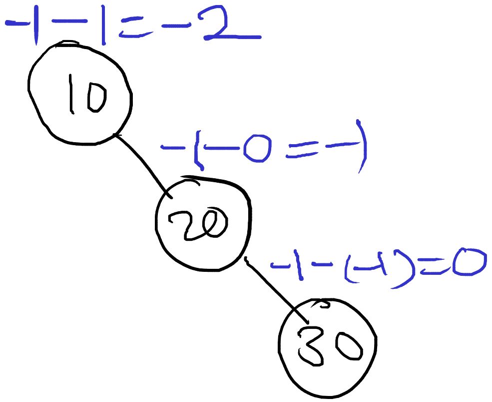
Balanced BST

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

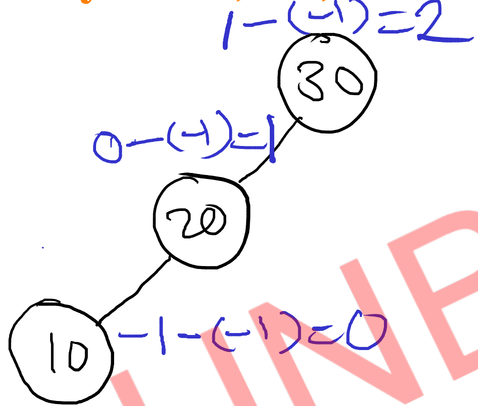
if balance factors of each node is either -1, 0 or +1
then such BST is called as Balanced BST

$\boxed{\text{height} - 2} \rightarrow n$

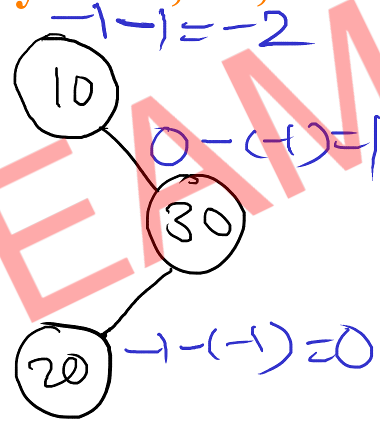
Keys : 10, 20, 30



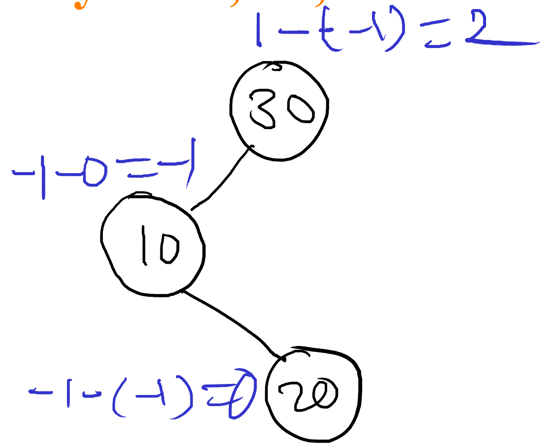
Keys : 30, 20, 10



Keys : 10, 30, 20



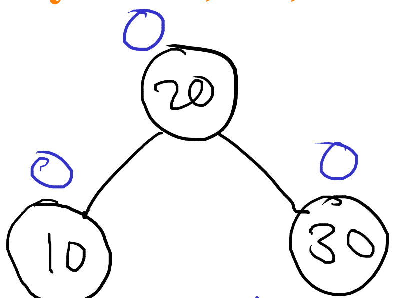
Keys : 30, 10, 20



$\boxed{\text{height} - 1} \rightarrow \log n$

Keys : 20, 10, 30

Keys : 20, 30, 10



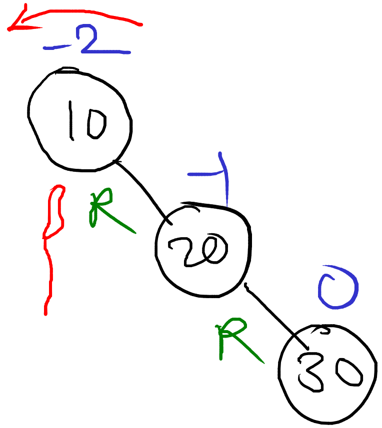
Balanced BST

Rotations

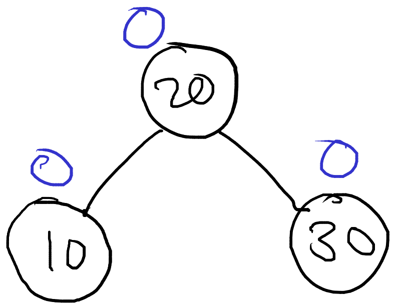
RR Imbalance	LL Imbalance	RL Imbalance	LR Imbalance
Keys : 10, 20, 30	Keys : 30, 20, 10	Keys : 10, 30, 20	Keys : 30, 10, 20

RR Imbalance

Keys : 10, 20, 30

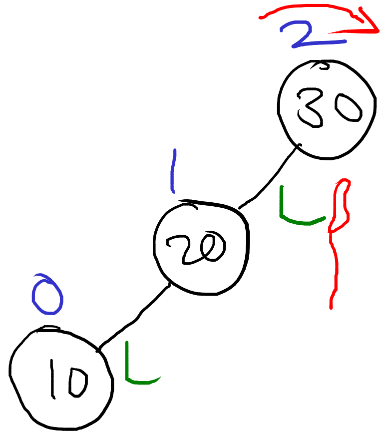


Left Rotation

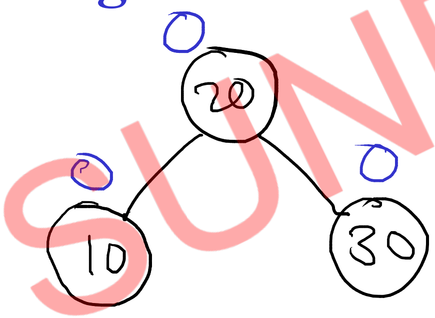


LL Imbalance

Keys : 30, 20, 10



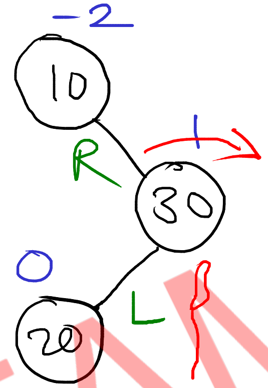
Right Rotation



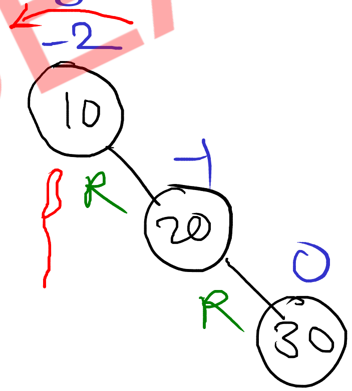
Single Rotation

RL Imbalance

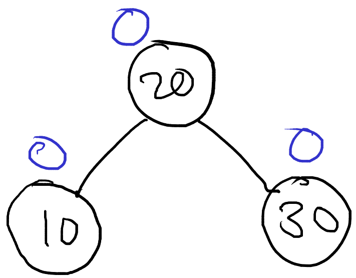
Keys : 10, 30, 20



Right Rotation

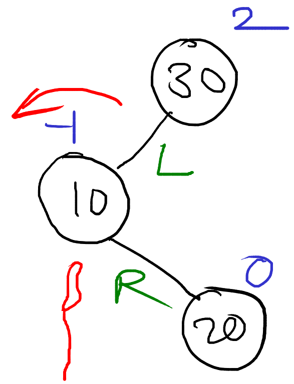


Left Rotation

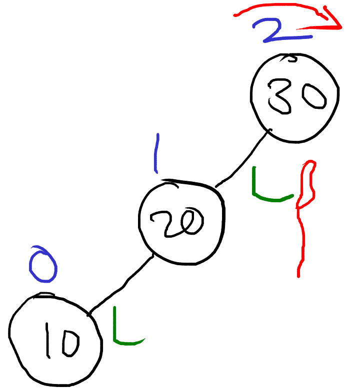


LR Imbalance

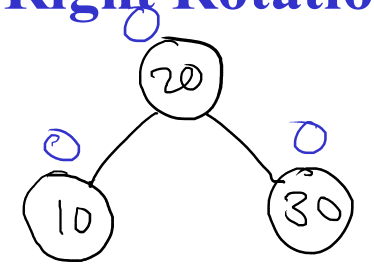
Keys : 30, 10, 20



Left Rotation



Right Rotation

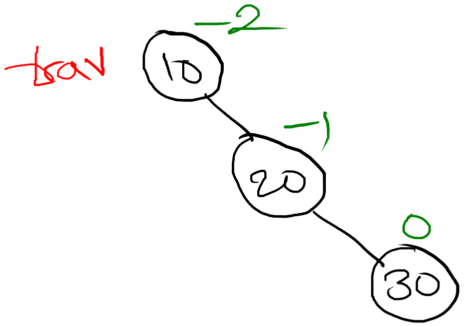


Double Rotation

Rotations

RR Imbalance

Keys : 10, 20, 30



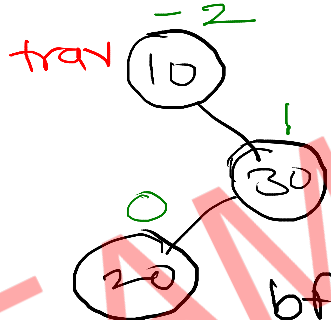
bf < -1

value > trav.right.data

30 > 20

RL Imbalance

Keys : 10, 30, 20



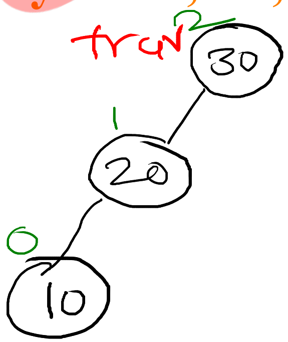
bf < -1

value < trav.right.data

20 < 30

LL Imbalance

Keys : 30, 20, 10



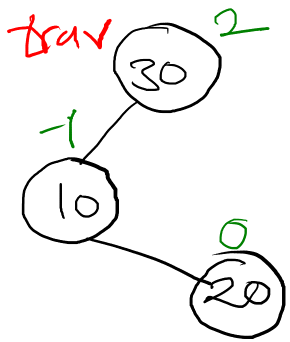
bf > 1

value < trav.left.data

10 < 20

LR Imbalance

Keys : 30, 10, 20

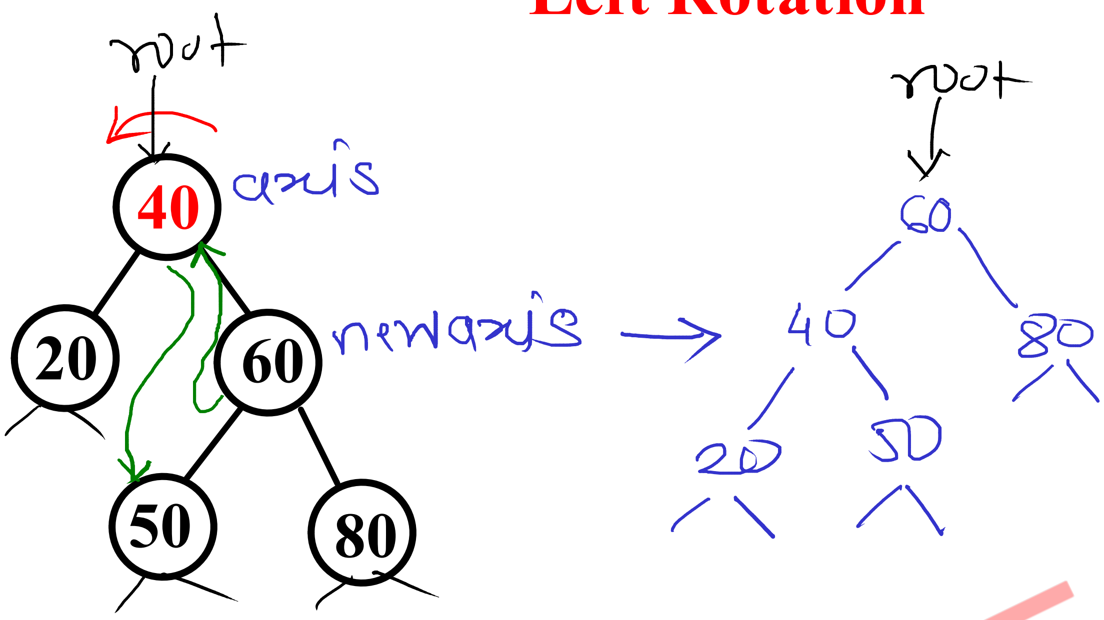


bf > 1

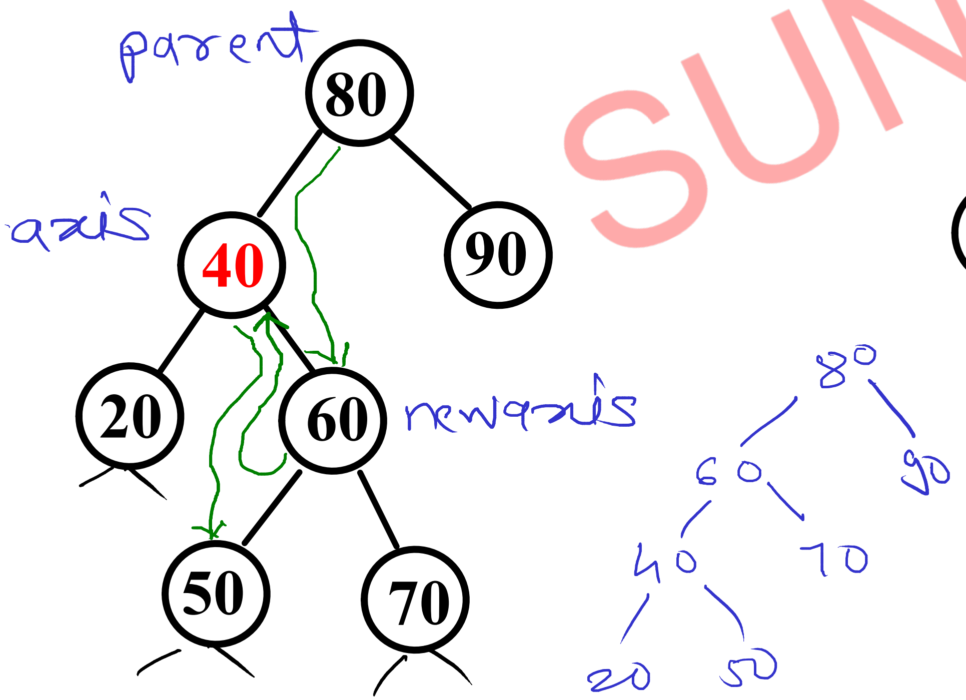
value > trav.left.data

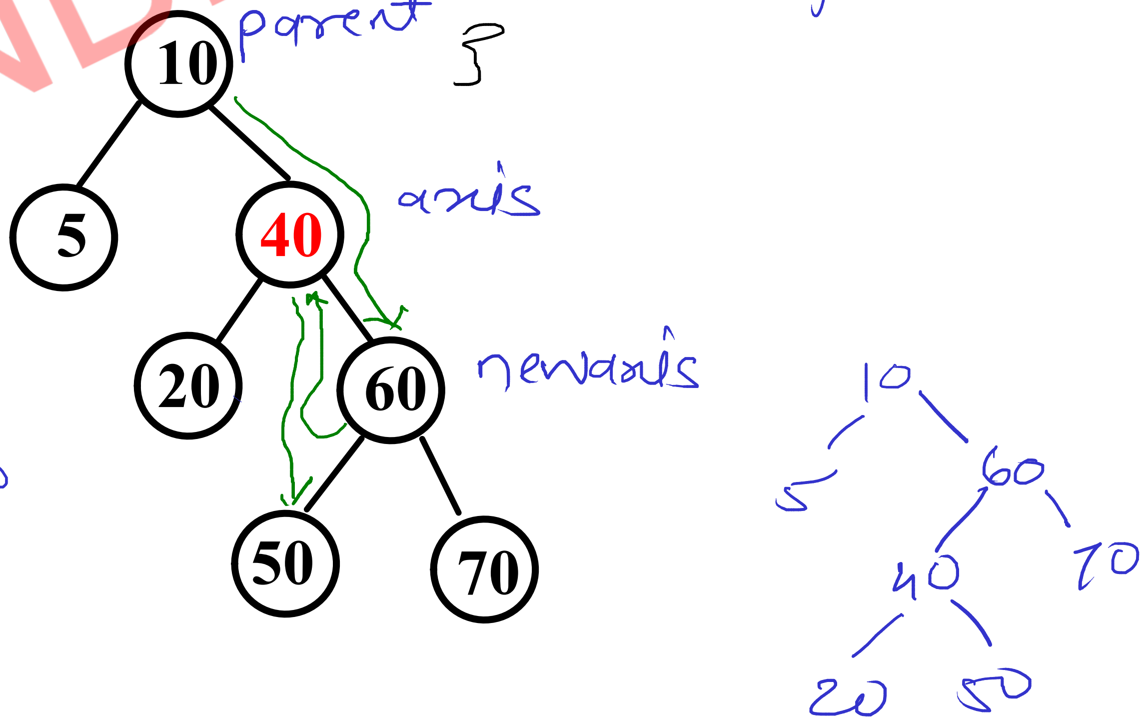
20 > 10

Left Rotation

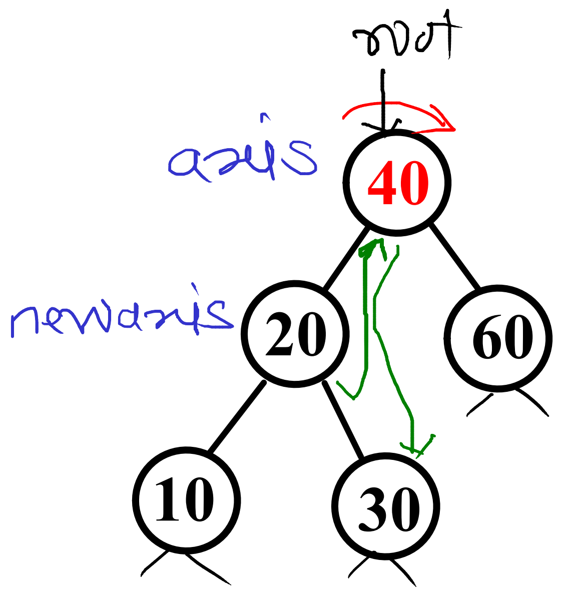


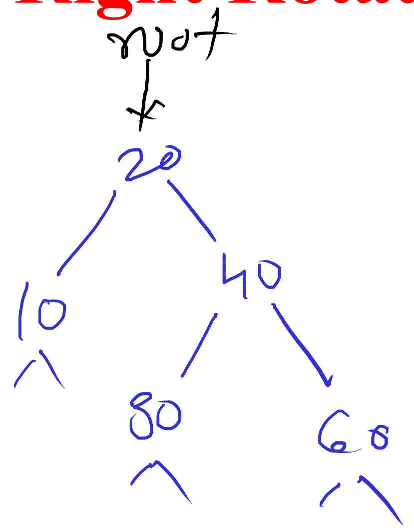

```
leftRotation(axis, parent) {
    newaxis = axis.right
    axis.right = newaxis.left
    newaxis.left = axis
    if (axis == root)
        root = newaxis;
    else if (axis == parent.left)
        parent.left = newaxis;
    else if (axis == parent.right)
        parent.right = newaxis;
}
```



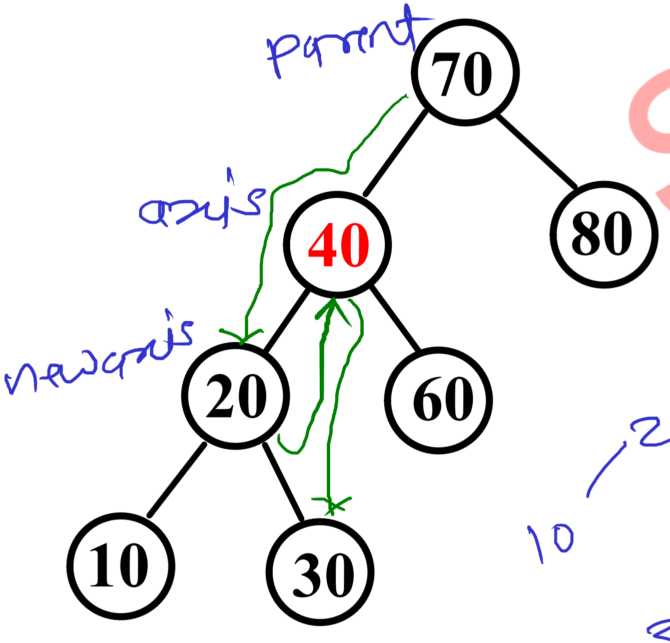


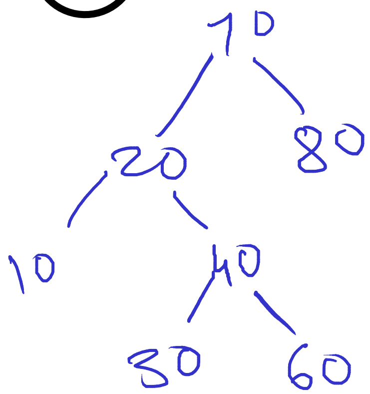
Right Rotation

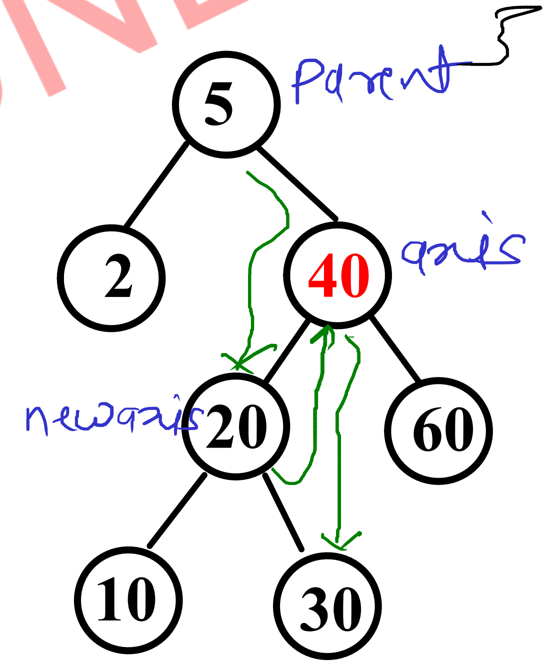


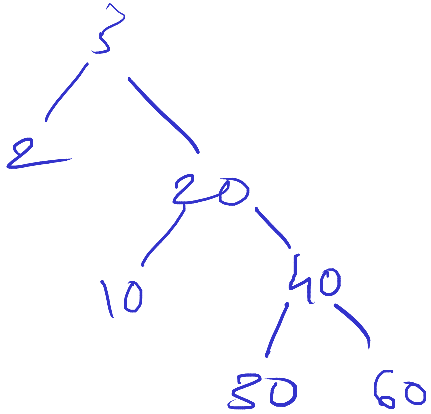



```
rightRotation(axis, parent) {
    newaxis = axis.left
    axis.left = newaxis.right
    newaxis.right = axis
    if(axis == root)
        root = newaxis;
    else if(axis == parent.left)
        parent.left = newaxis;
    else if(axis == parent.right)
        parent.right = newaxis;
}
```





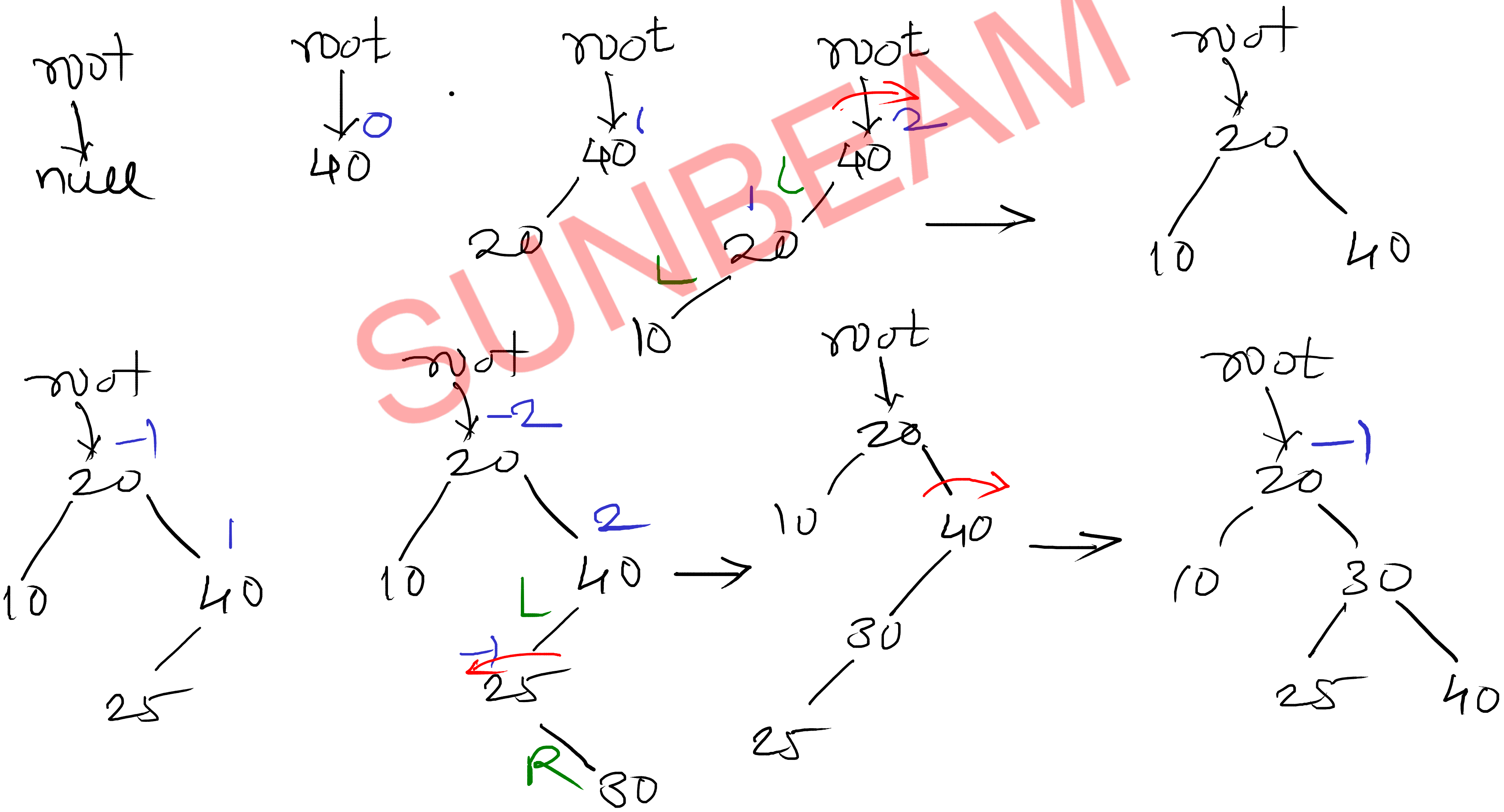




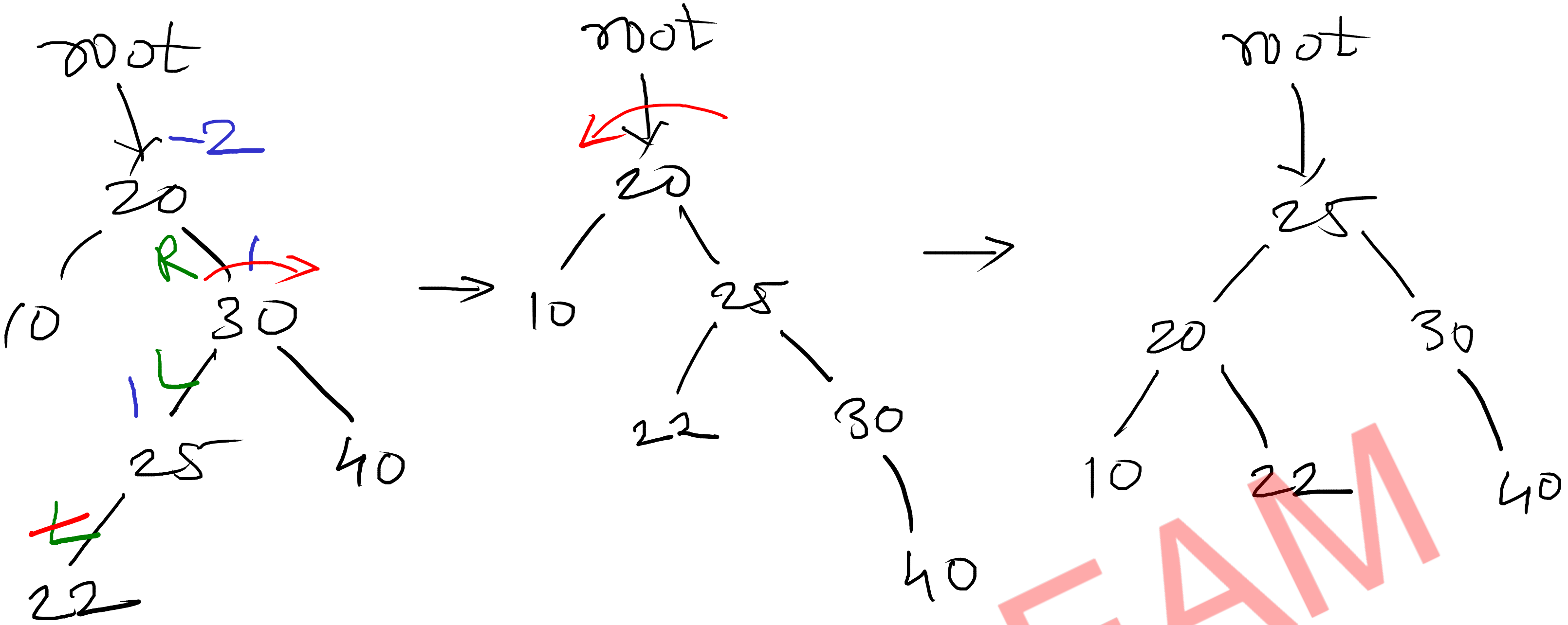
AVL - Tree

- AVL tree is a self balancing binary Search Tree
- on every insert and delete, tree is balanced(by performing rotations)
- most of the operations are performed into O(log n)
- balanced factore = {-1, 0, +1}

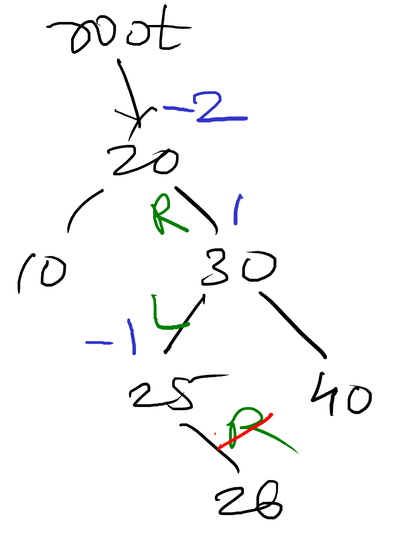
Keys : 40, 20, 10, 25, 30, 22, 50



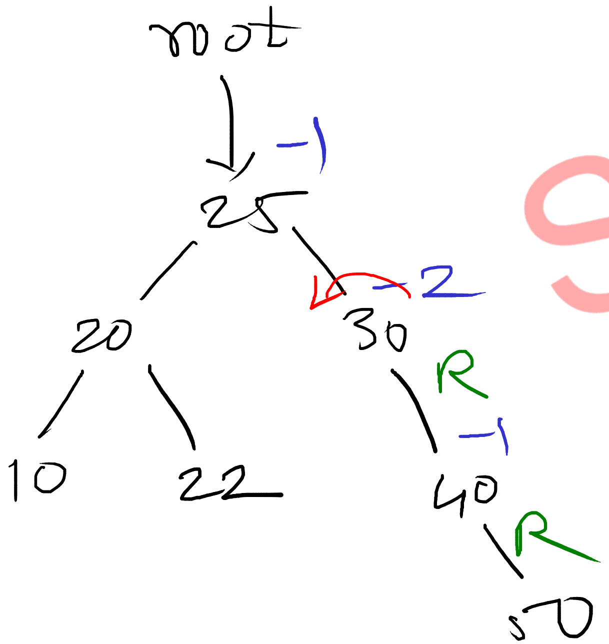
0-2=-2



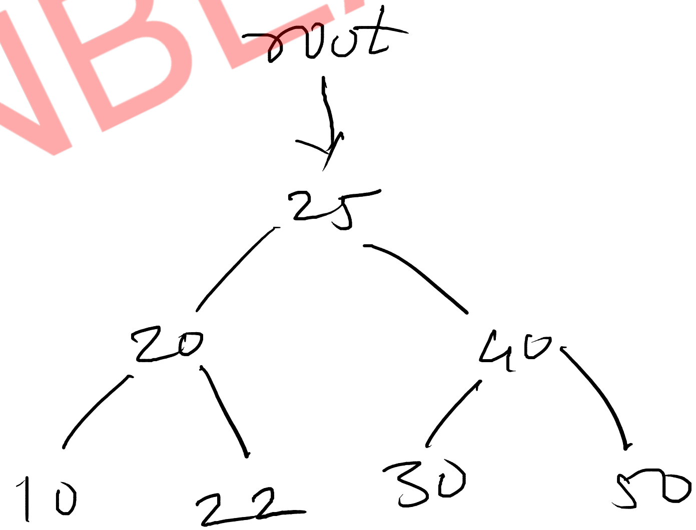
root
-2
20
10
R
1
30
1
L
25
40
L
22
root
20
10
25
22
30
40
root
25
20
30
10
22
40



root
-2
20
10
R
1
30
-1
L
25
40
R
28



root
-1
25
20
-2
30
R
10
22
-1
40
R
50



root
25
20
40
10
22
30
50

Almost Complete Binary Tree

(ACBT)

- binary tree
- all leaf nodes should be at level h or h-1
- ACBT is filled level by level, we always go on next level when previous is completely filled
- nodes of last level (h) should left aligned as possible as