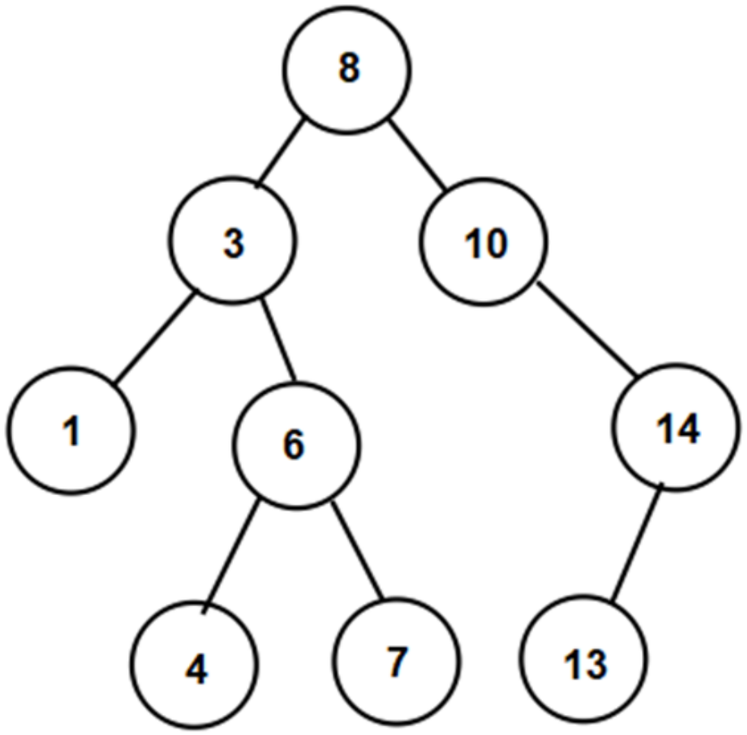
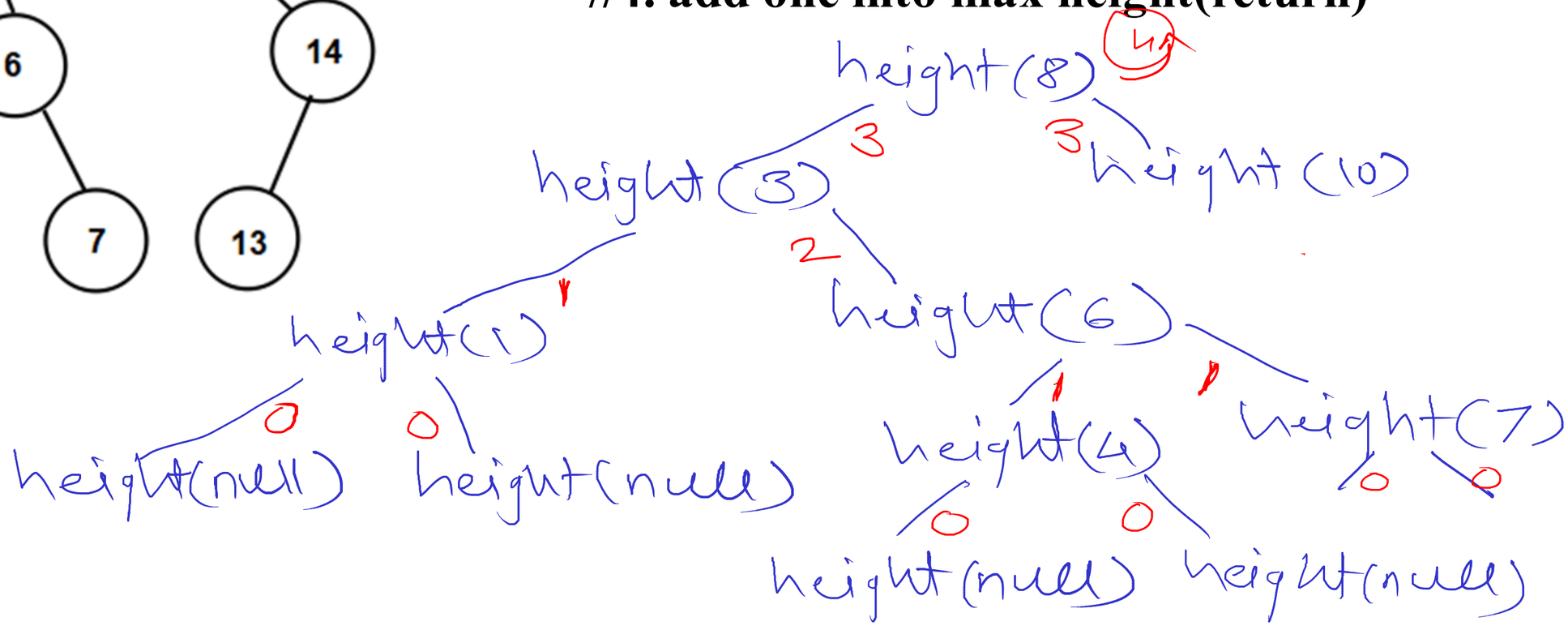
BST - Height



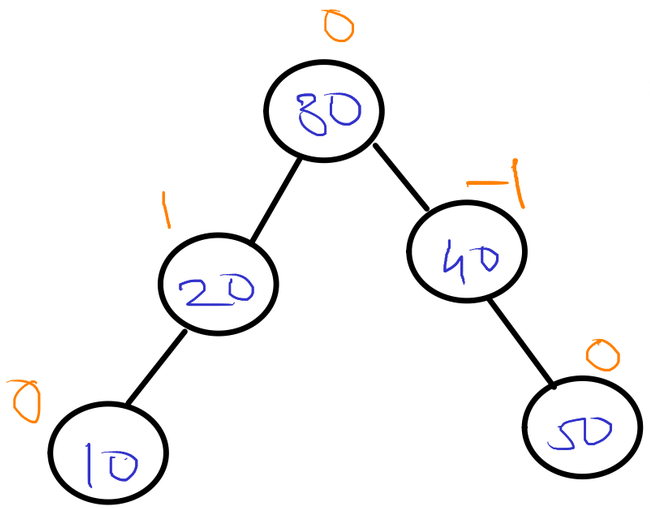
//0. if left or right sub tree is absent
//then return 0
//1. find height of left subtree
//2. find height of right subtree
//3. find max height
//4. add one into max height(return)



Height of node = Max(Height(left sub tree), Height(right sub tree)) + 1

Skewed BST

Keys : 30, 40, 20, 50, 10

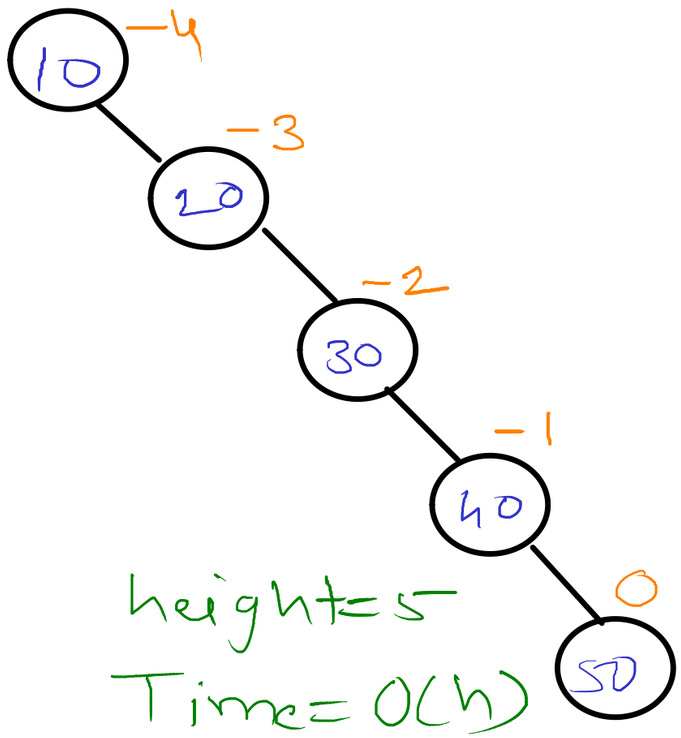


height = 3

Time = O(h)

Time = O(log n)

Keys : 10, 20, 30, 40, 50

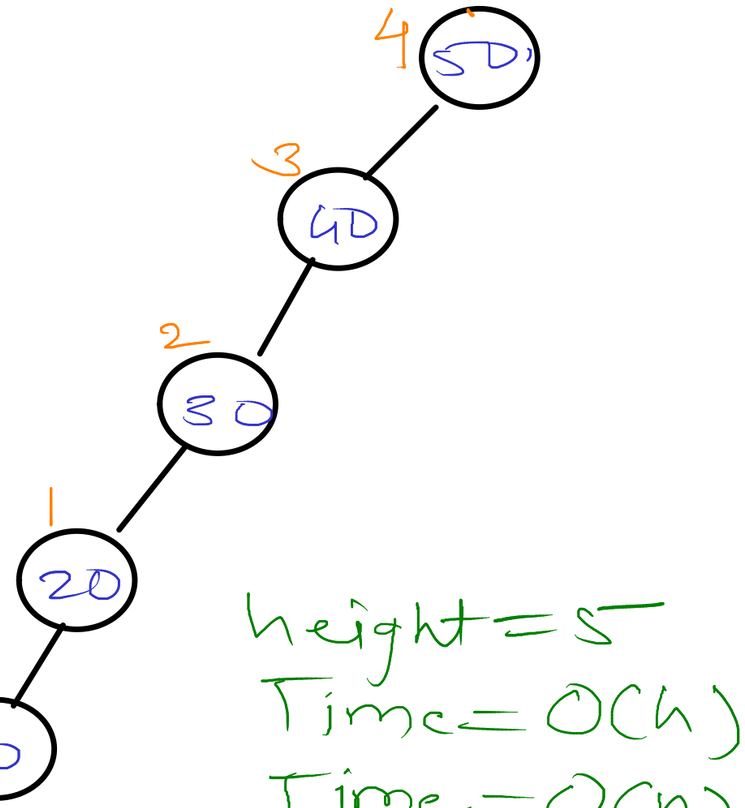


height = 5

Time = O(h)

Time = O(n)

Key : 50, 40, 30, 20, 10



height = 5

Time = O(h)

Time = O(n)

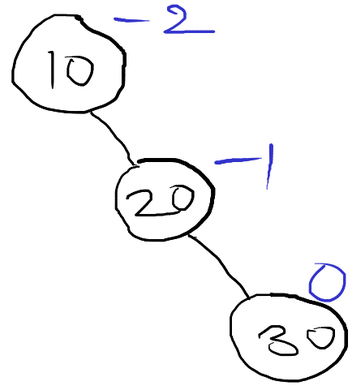
- **tree which grows in only one direction is called as skewed BST.**
- **if tree grows in left direction, it is called as left skewed BST.**
- **if tree grows in right direction, it is called as left skewed BST.**

Balanced BST

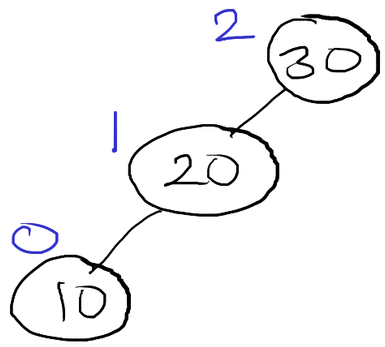
Balance Factor = height(left sub tree) - height(right sub tree)

- tres is balanced if balance factor of all the nodes is either -1, 0 or +1
- balance factor = {-1, 0, +1}

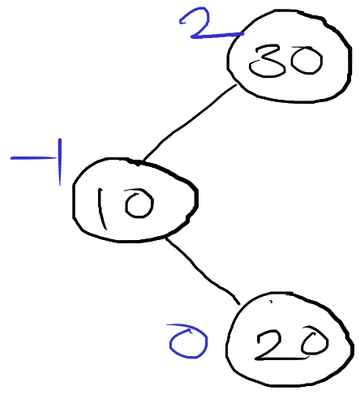
Keys : 10, 20 ,30



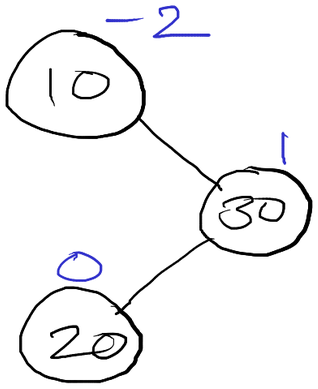
Keys : 30, 20, 10



Keys : 30, 10, 20

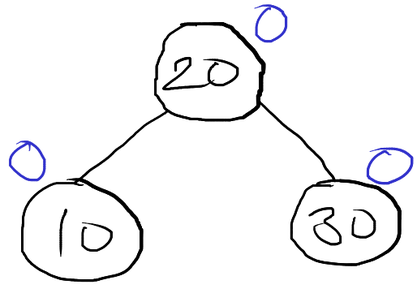


Keys : 10, 30, 20



Keys : 20, 10, 30

Keys : 20, 30, 10

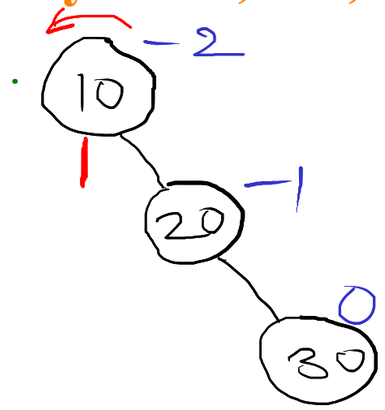


Balanced BST

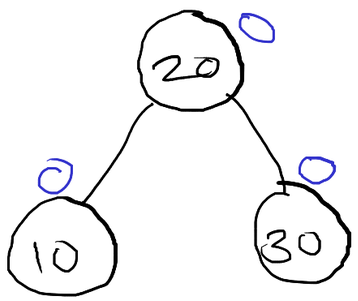
Rotations

RR Imbalance

Keys : 10, 20 ,30

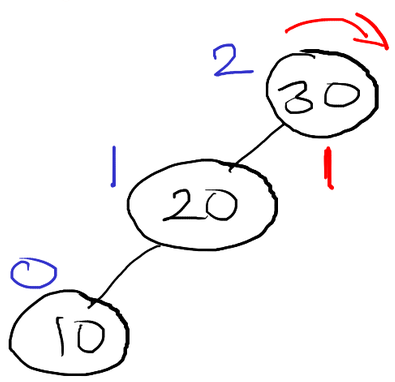


Left Rotation

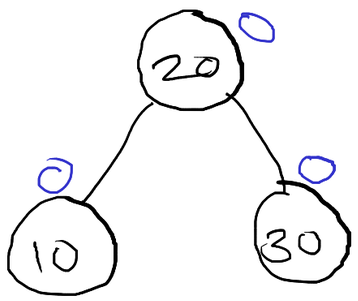


LL Imbalance

Keys : 30, 20, 10



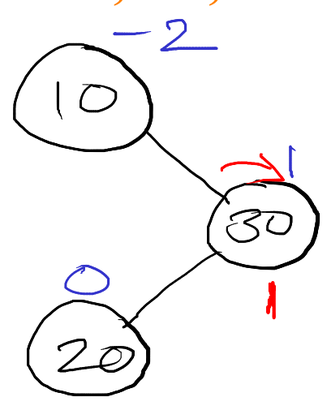
Right Rotation



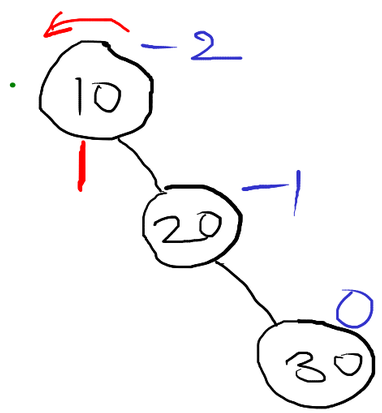
Single Rotation

RL Imbalance

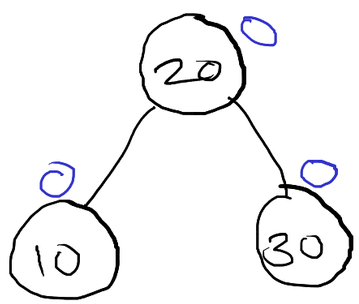
Keys : 10, 30, 20



Right Rotation

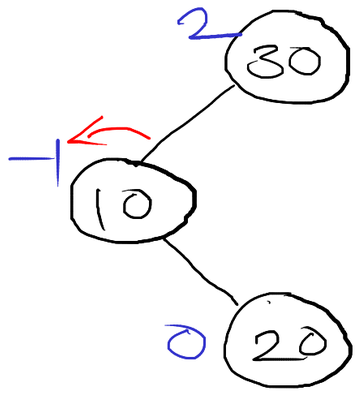


Left Rotation

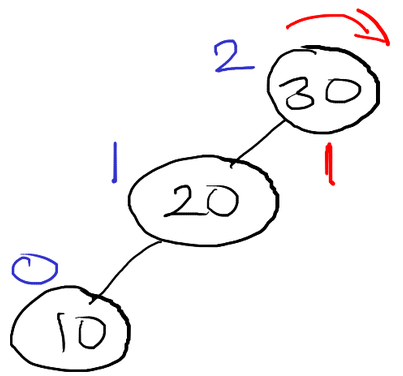


LR Imbalance

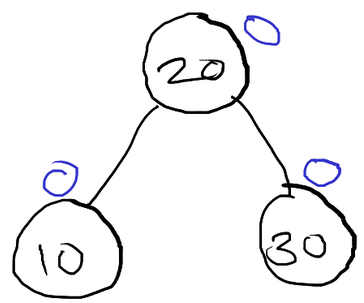
Keys : 30, 10, 20



Left Rotation



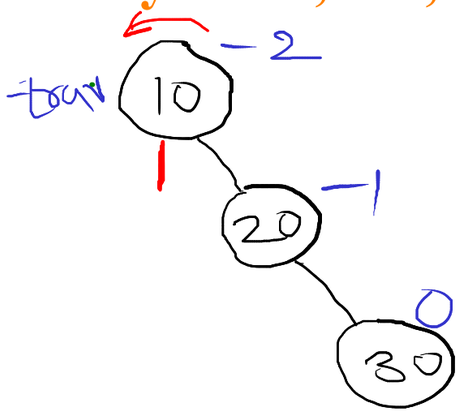
Right Rotation



Double Rotation

RR Imbalance

Keys : 10, 20 ,30

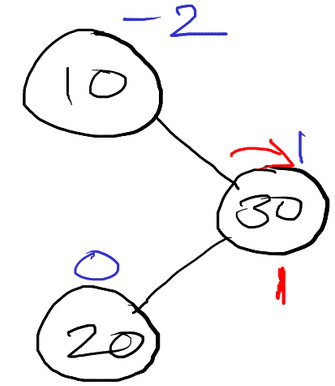


bf < -1

val > trav.righ-data

RL Imbalance

Keys : 10, 30, 20

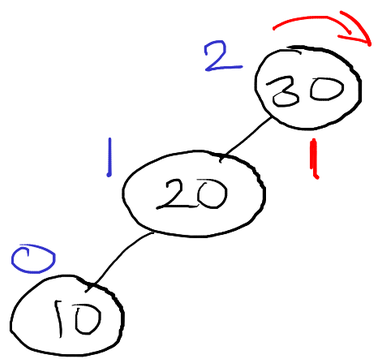


bf < -1

val < trav.right.data

LL Imbalance

Keys : 30, 20, 10

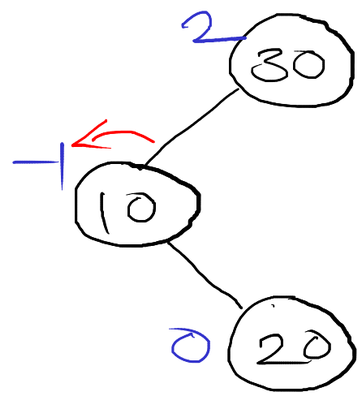


bf > 1

val < trav.left.data

LR Imbalance

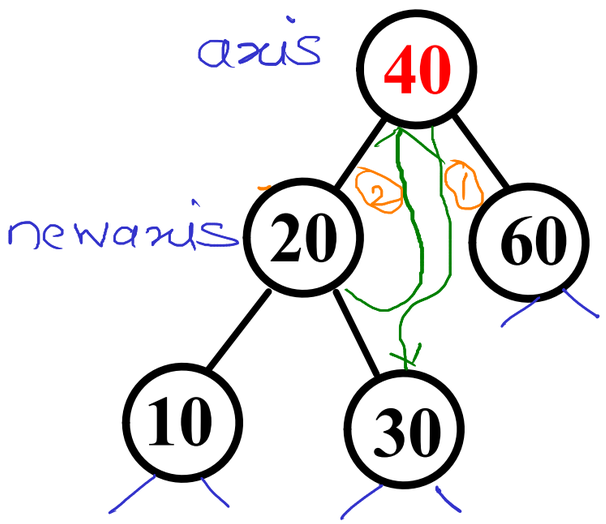
Keys : 30, 10, 20

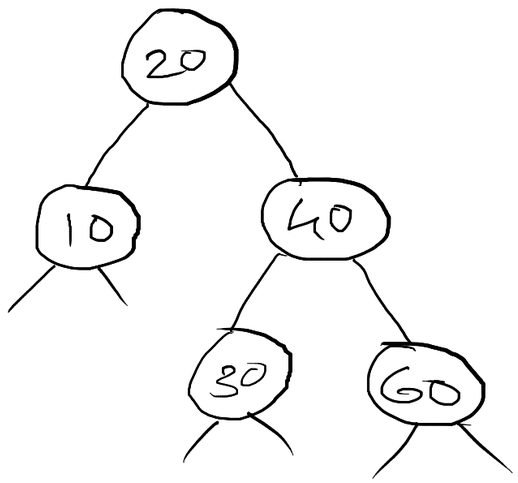


bf > 1

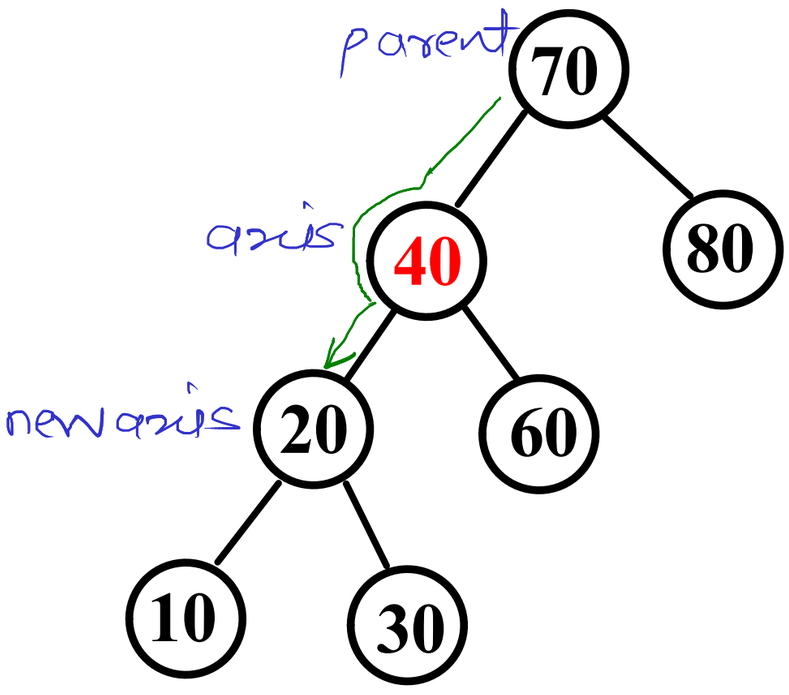
val > trav.left.data

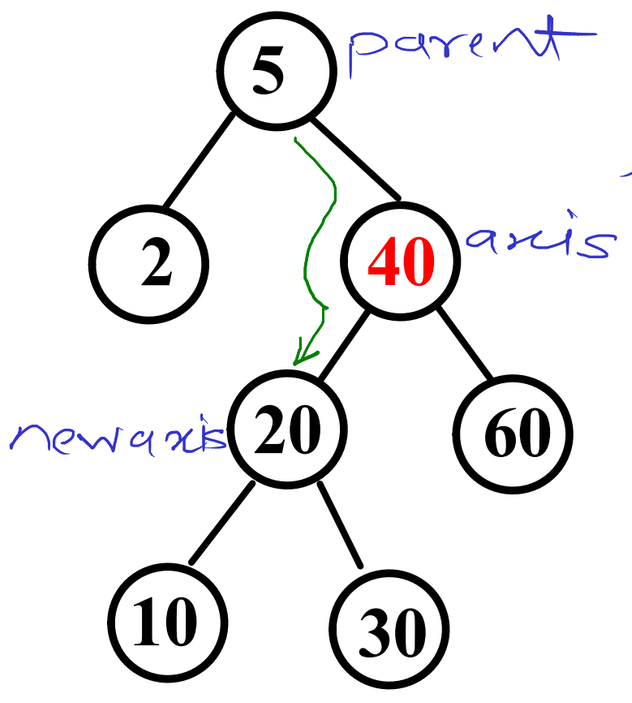
Right Rotation



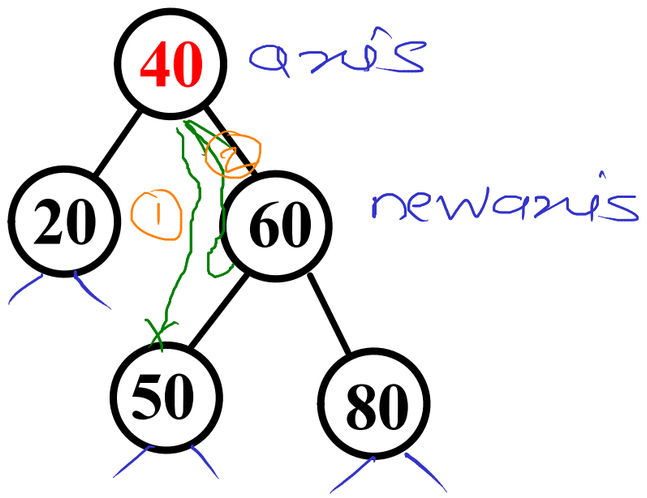


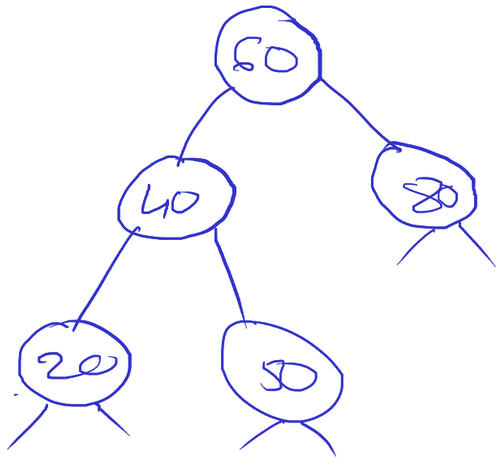

```
void rightRotation(axis, parent) {
    newaxis = axis.left
① axis.left = newaxis.right
② newaxis.right = axis
    if(axis == root)
        root = newaxis
    else if(axis == parent.left)
        parent.left = newaxis
    else if(axis == parent.right)
        parent.right = newaxis
}
```



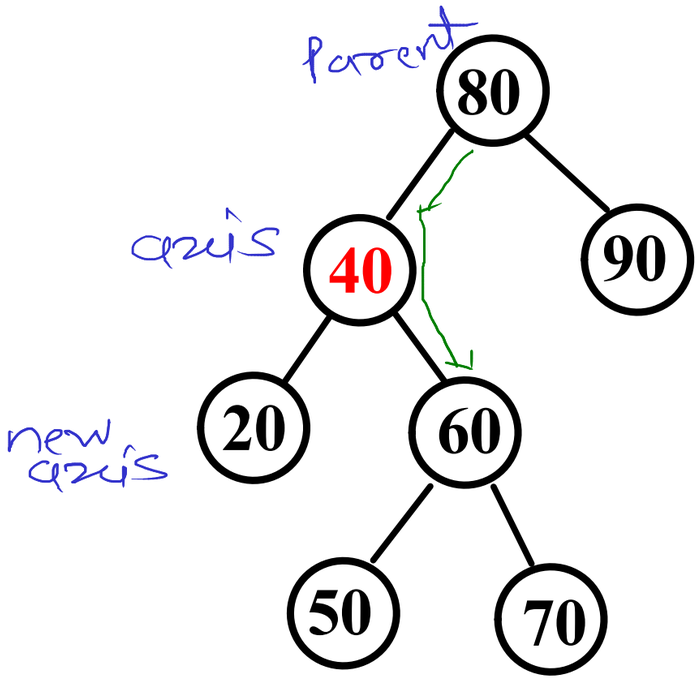


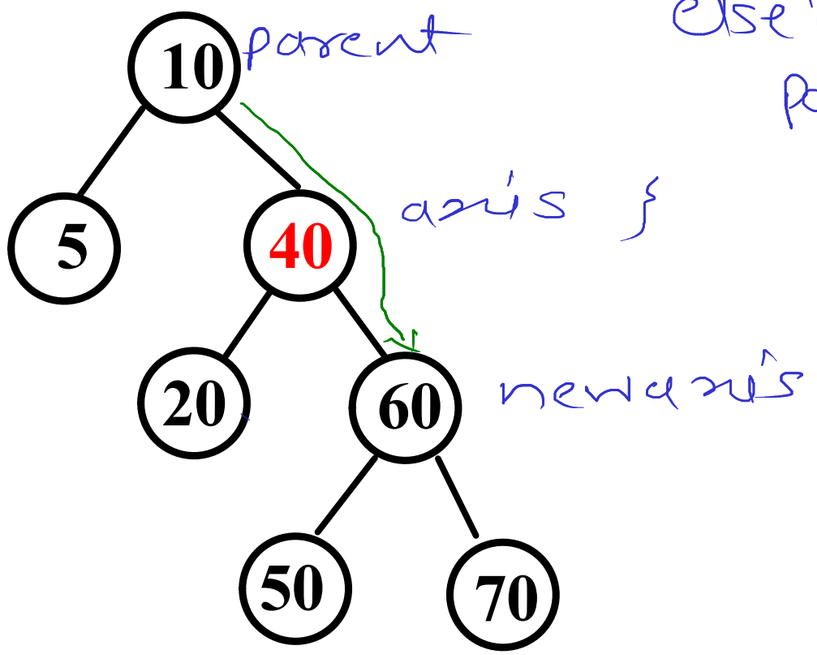
Left Rotation






```
void leftRotation(axis, parent){
    newaxis = axis.right
① axis.right = newaxis.left
② newaxis.left = axis
    if(axis == root)
        root = newaxis
    else if(axis == parent.left)
        parent.left = newaxis
    else if(axis == parent.right)
        parent.right = newaxis
```

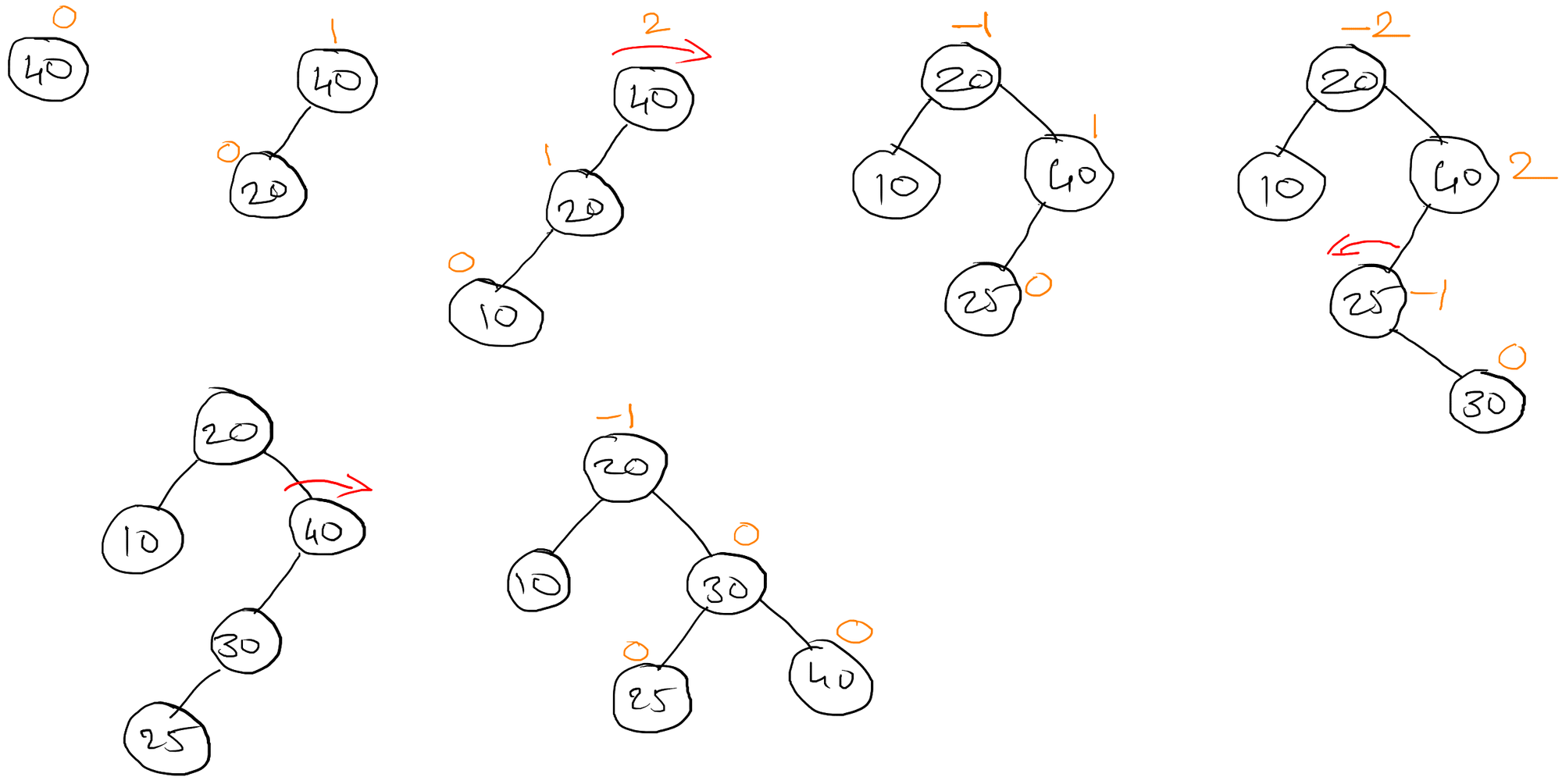


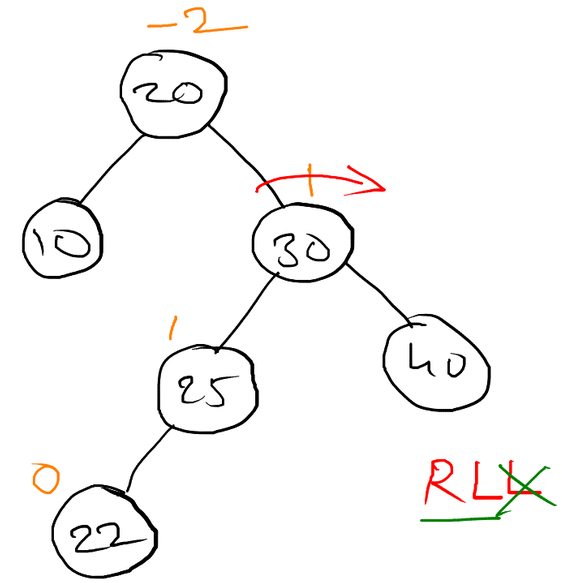


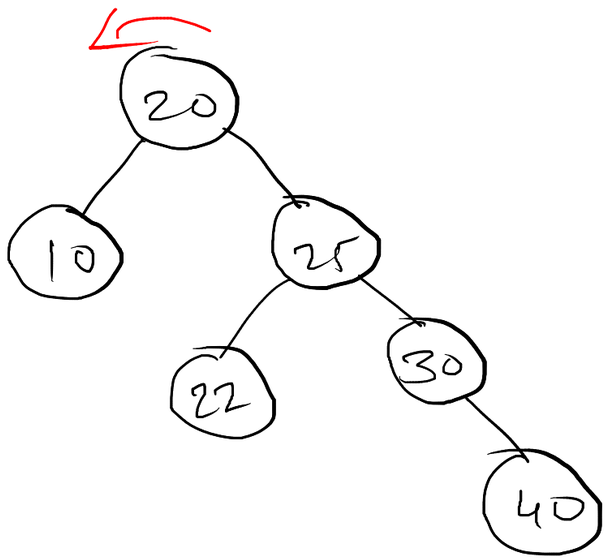
AVL Tree

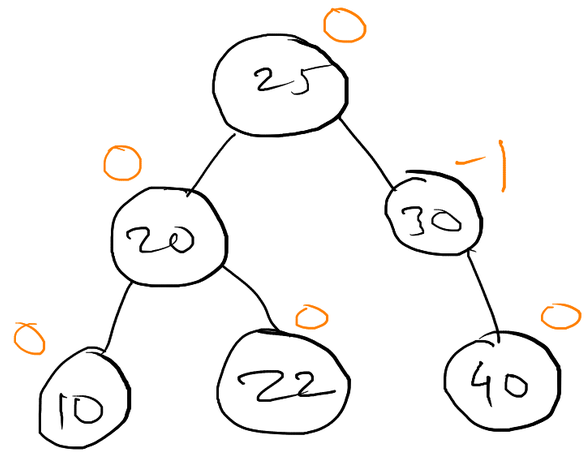
- Self balancing binary Search Tree
- on every insertion and deletion of node, tree is balanced
- All operation on AVL tree are perfromed in O(log n) time
- Balance factor of all nodes is either -1, 0 or +1

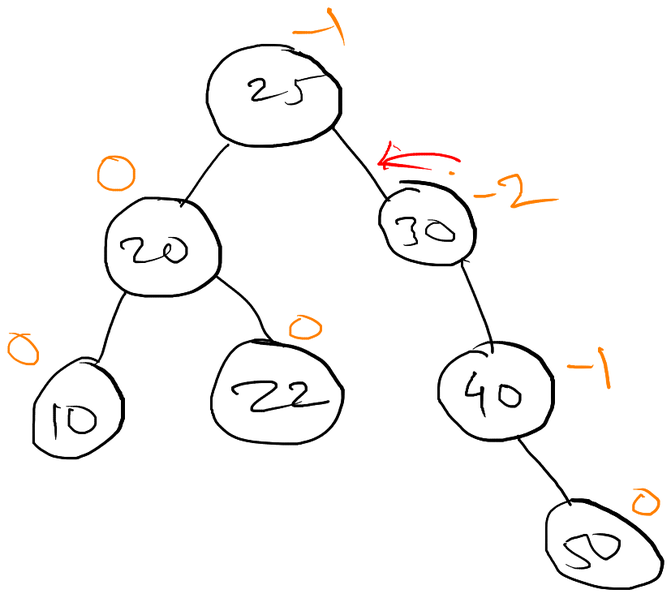
Keys : 40, 20, 10, 25, 30, 22, 50

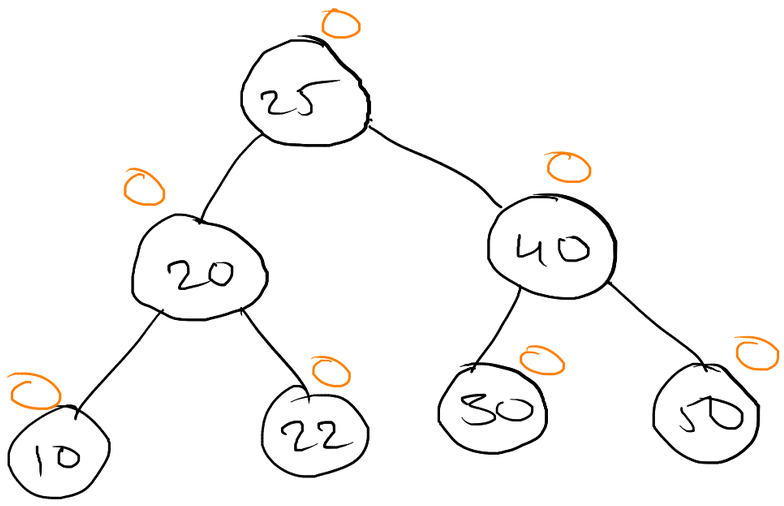



-2
20
10
30
+
25
40
0
22
RL

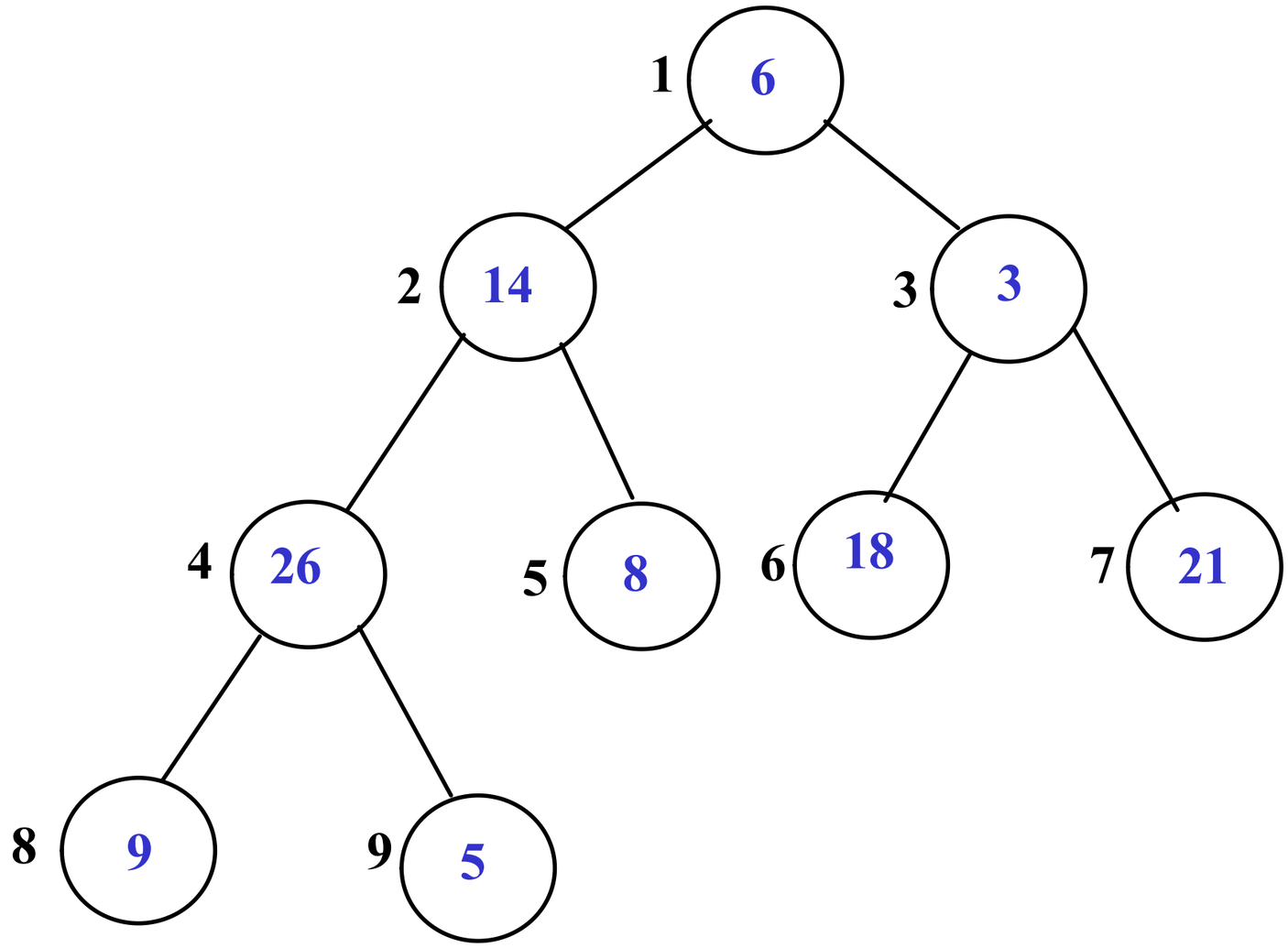

20
10
25
22
30
40


0
25
0
20
-1
30
0
0
0
10
22
40


-1
25
0
20
-2
30
0
0
0
10
22
40
-1
0
50


0
25
0
0
20
40
0
0
0
0
10
22
30
50

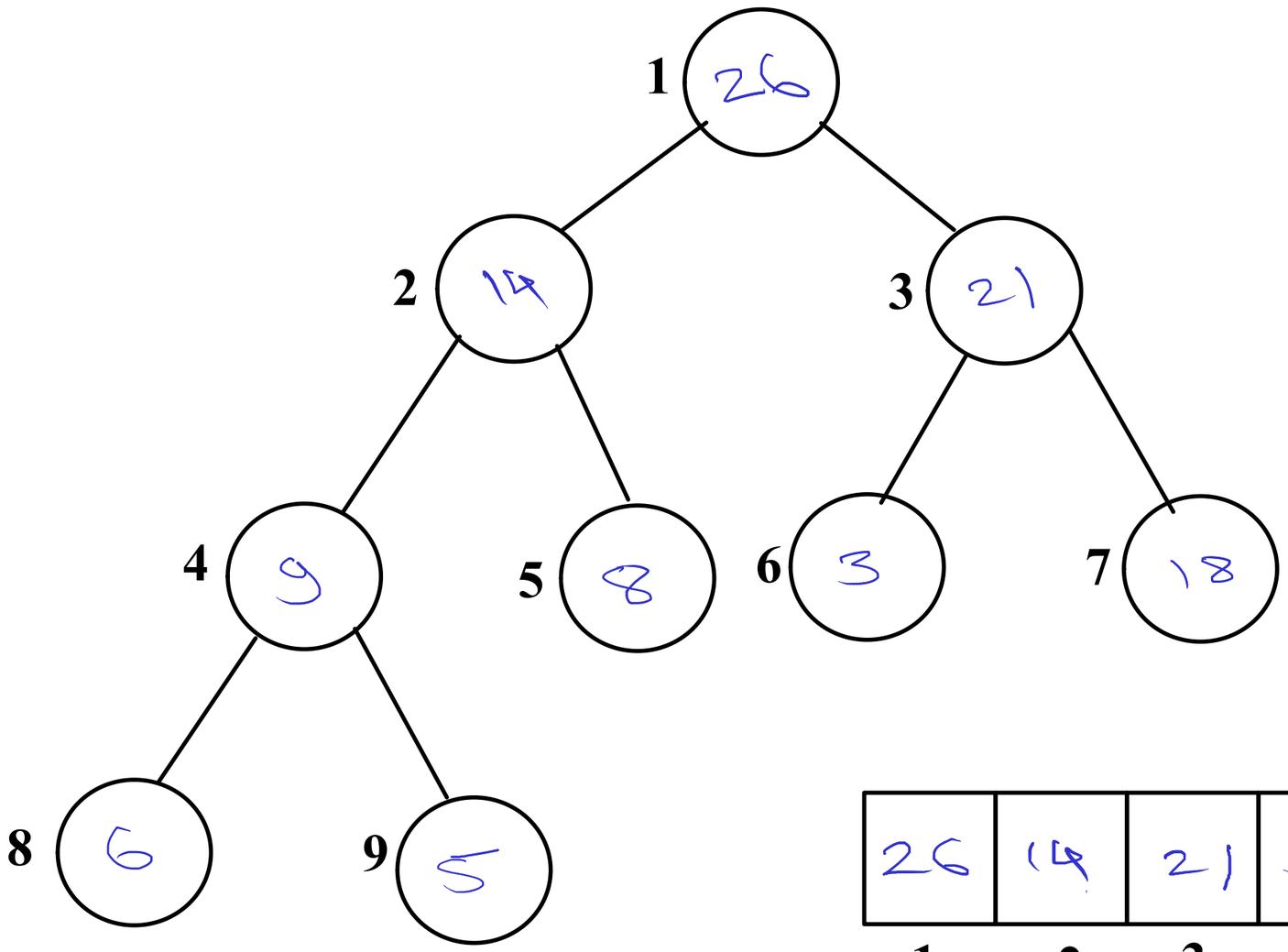
Almost Complete Binary Tree



6	14	3	26	8	18	21	9	5
1	2	3	4	5	6	7	8	9

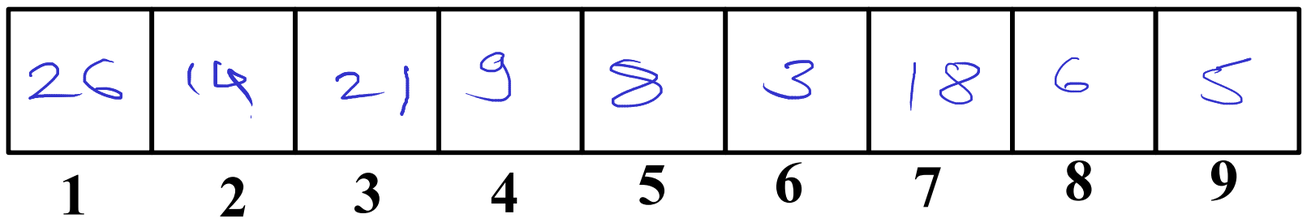
Create Max Heap

6 14 3 26 8 18 21 9 5

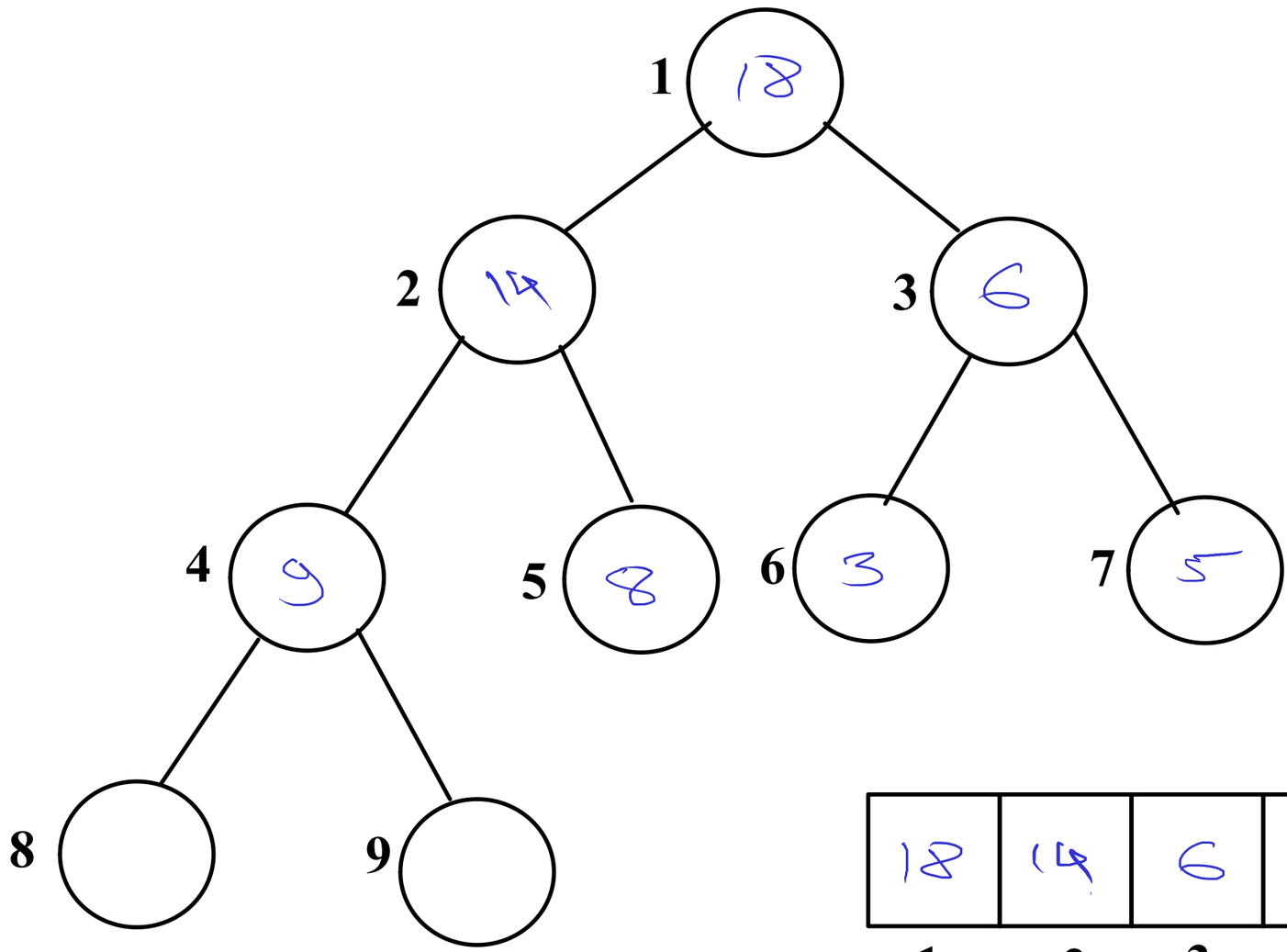


To add single element into heap, need O(log n) time.

To add n elements into heap, need O(n log n) time



Delete Max Heap



max = 21

To delete single element from heap need O(log n) time

To delete n elements from heap needs, O(n log n) time.

18	14	6	9	8	3	5	21	26
1	2	3	4	5	6	7	8	9

Heap Sort Time Complexity

1) Create heap – n log n
2) Delete heap – n log n

2n log n

T(n) = O(n log n)