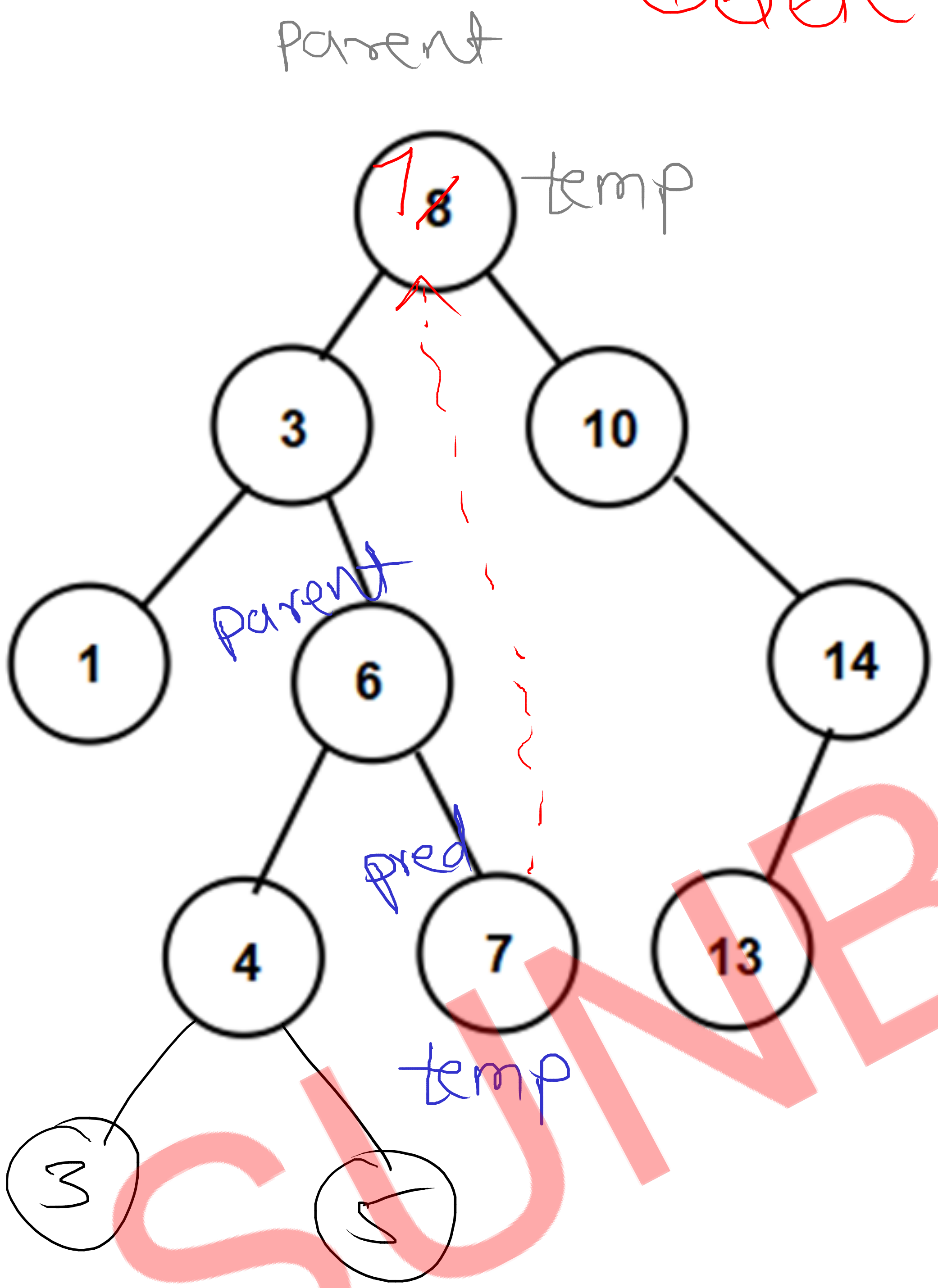
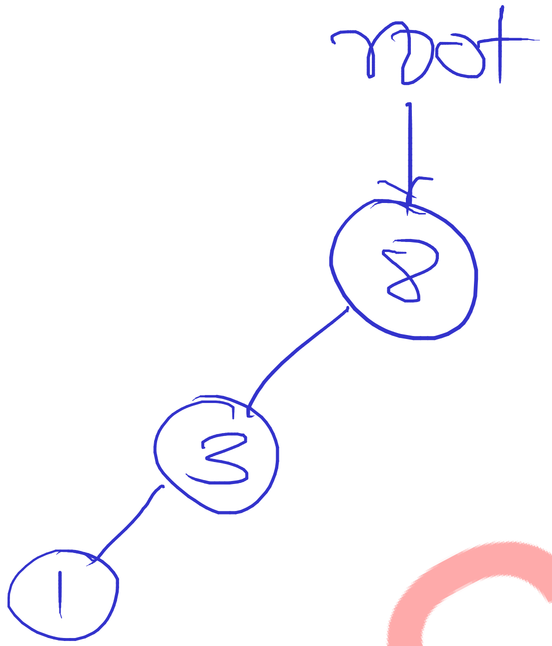
Delete Node - BST



BST - Add Node Recursive

```
void addNode(int value) {
    if(root == null)
        root = new Node(value)
    else
        addNode(root, value);

}
```



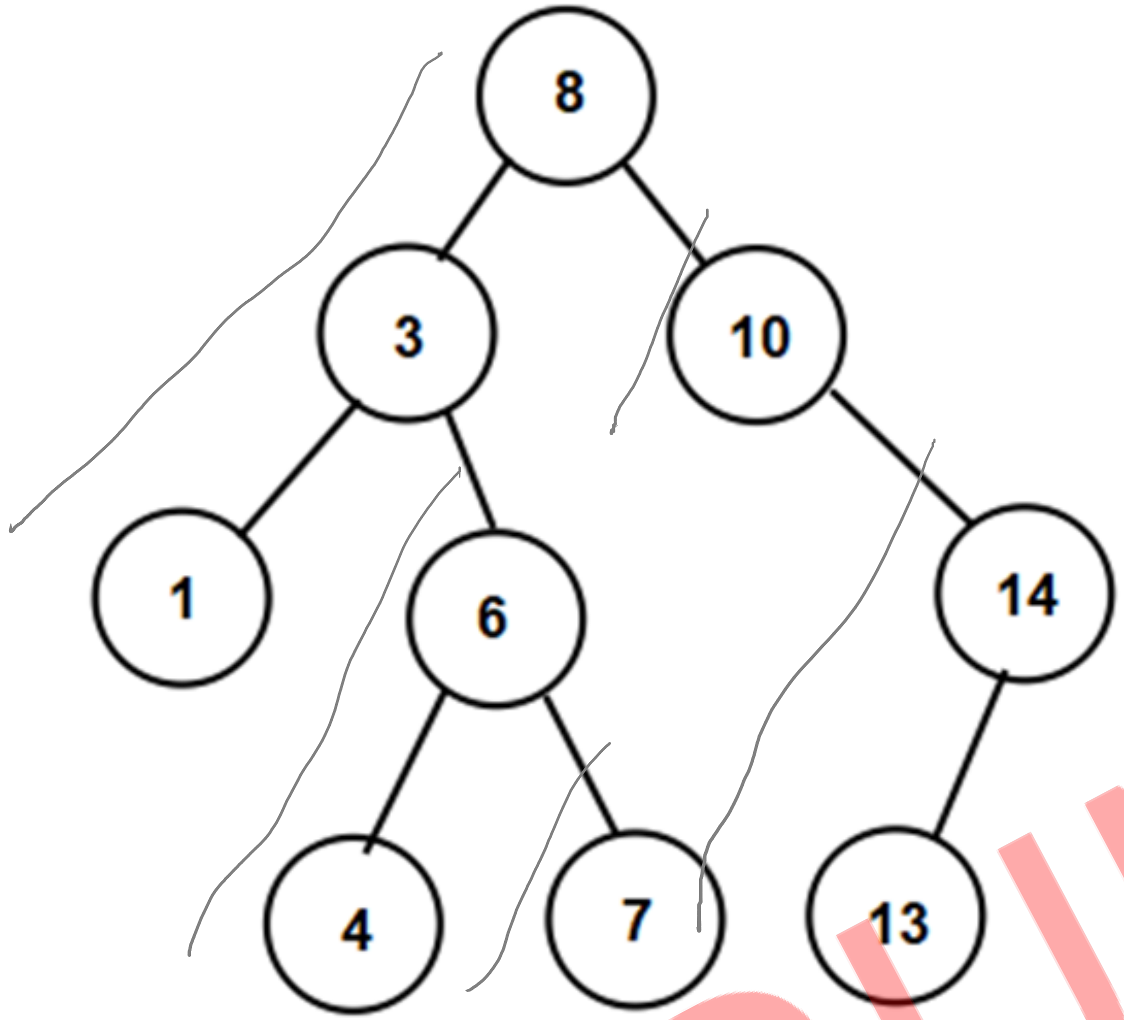
8, 3, 1

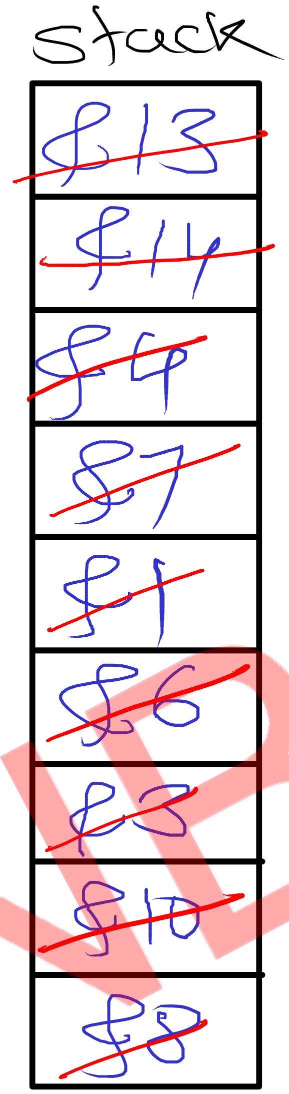
```
void addNode(Node trav, int value)
{
    if(value < trav.data) {
        if(trav.left == null)
            trav.left = new Node(value);
        else
            addNode(trav.left, value)
    } else {
        if(trav.right == null)
            trav.right = new Node(value);
        else
            addNode(trav.right, value)
    }
}
```

BST- Binary Search Recursive

```
Node binarySearchRec(int key, Node trav) {
    if( trav == null)
        return null;
    if(key == trav.data)
        return trav;
    else if(key < trav.data)
        return binarySearchRec(key, trav.left);
    else
        return binarySearchRec(key, trav.right);
}
```

BST - DFS (Depth First Search)

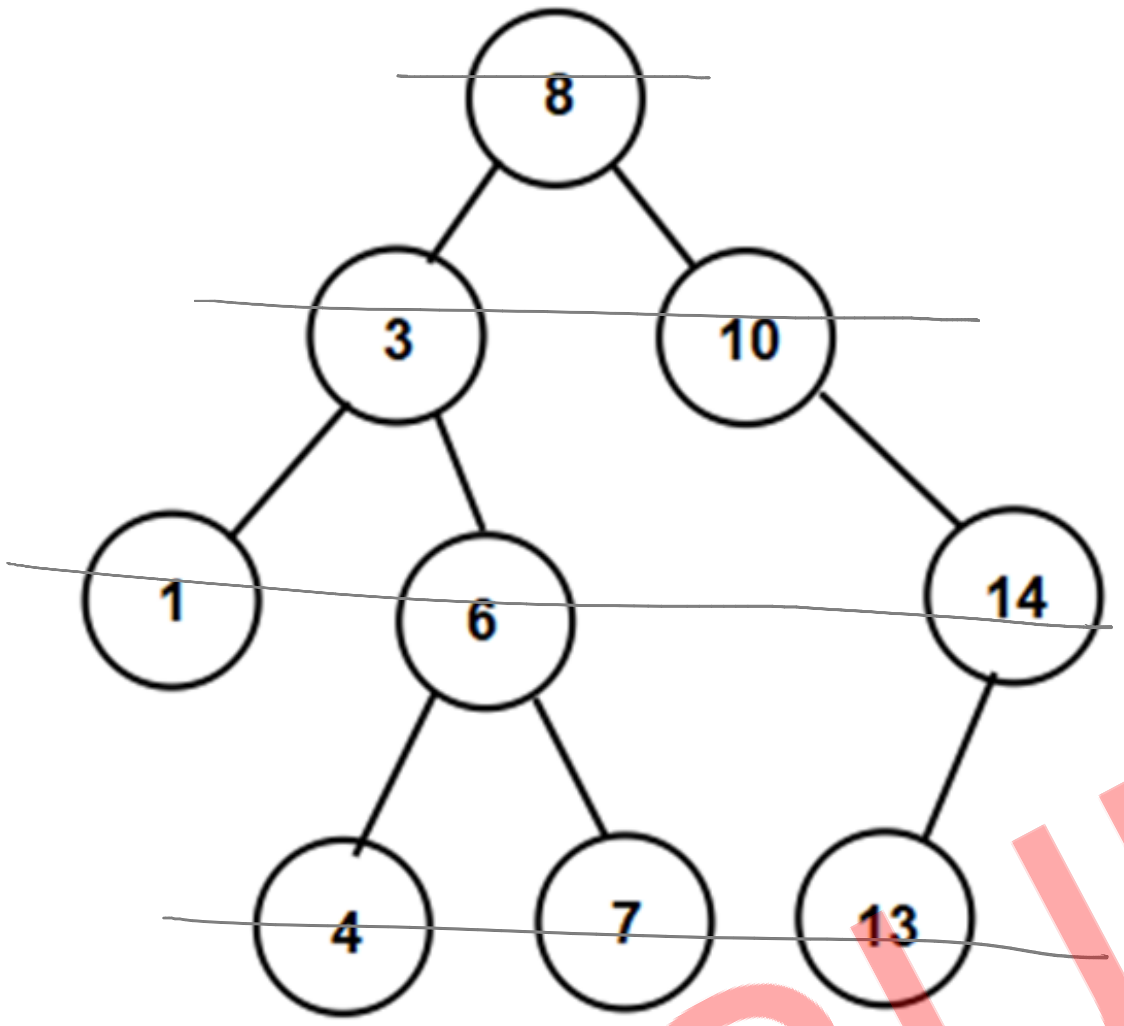




//1. push root on stack
//2. pop one node from stack
//3. visit(print) node
//4. if right exist, push it on stack
//5. if left exist, push it on stack
//6. while stack is not empty
//repeat ste 2 to 5

Traversal:
8, 3, 1, 6, 4, 7, 10, 14, 13

BST - BFS (Bredth First Search)



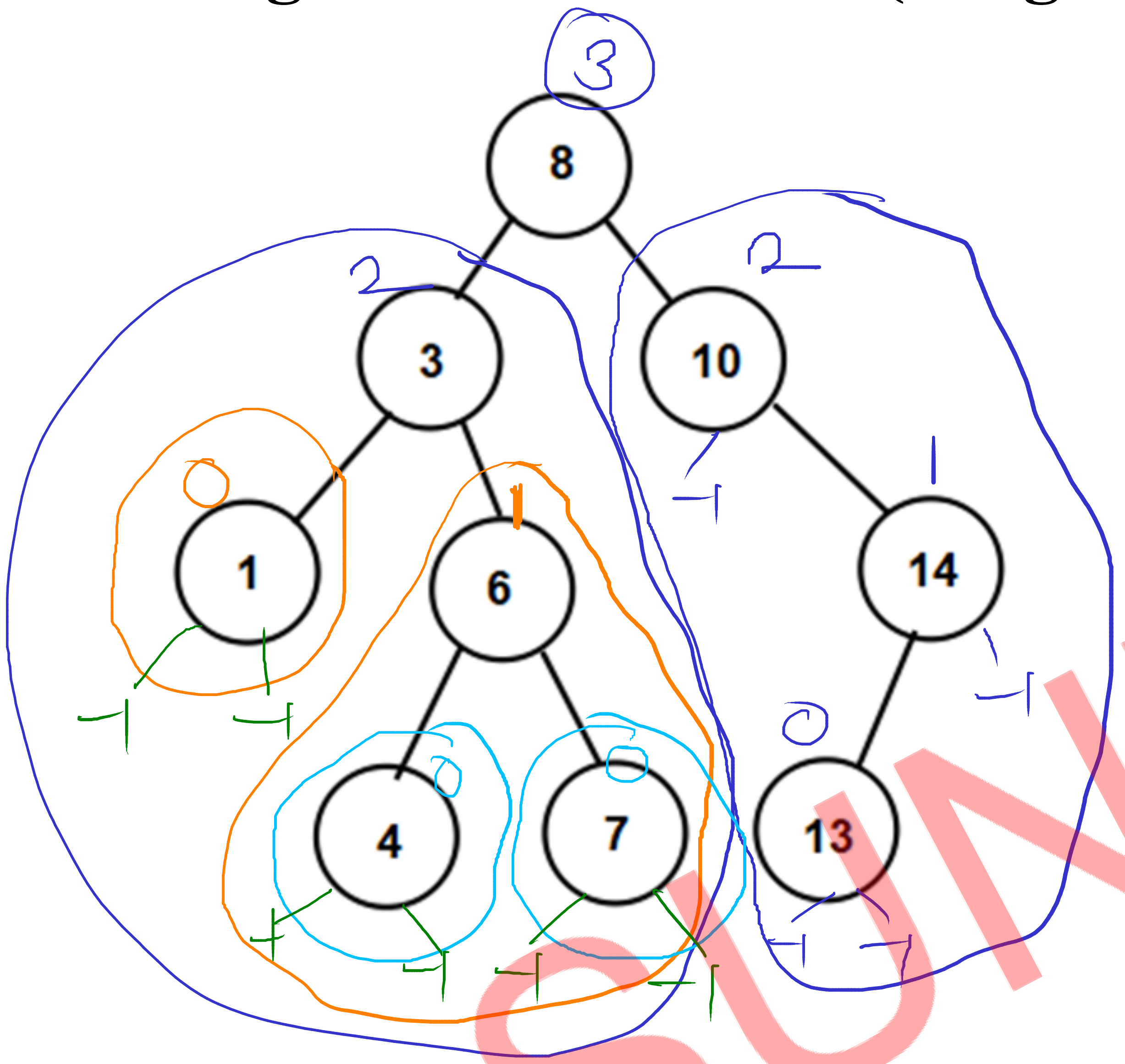
//1. push root on queue
//2. pop one node from queue
//3. visit(print) node
//4. if left exist, push it on queue
//5. if right exist, push it on queue
//6. while queue is not empty
//repeat ste 2 to 5

Traversal :

8, 3, 10, 1, 6, 14, 4, 7, 13

BST - Height

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



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

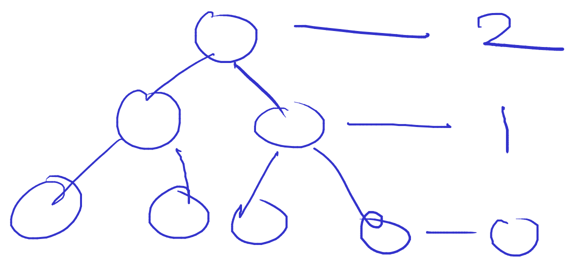
BST - Time complexity of operations

No. of elements $= n$

height of BST $= h$

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

h	n
-1	0
0	1
1	3
2	7
3	15
⋮	⋮



Add : $O(h) \,/\, O(\log n)$

search : $O(h) \,/\, O(\log n)$

Delete : $O(h) \,/\, O(\log n)$

Traversal : $O(n)$

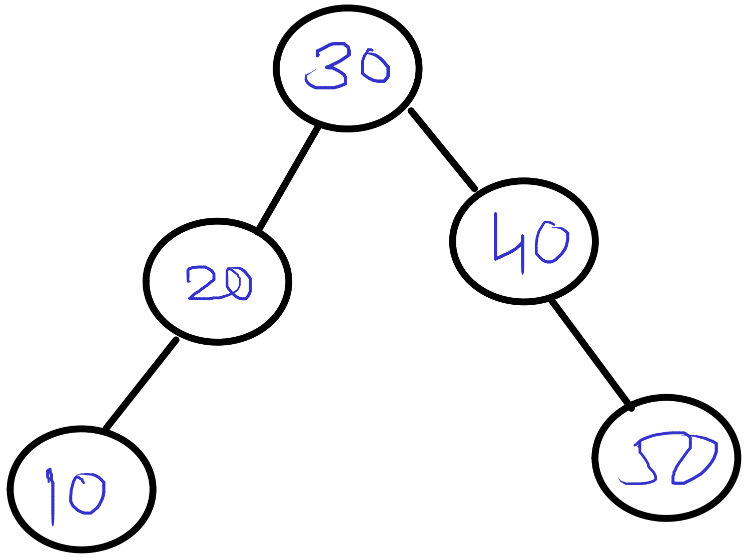
$$2^{h+1} = n+1$$

$$2^{h} = n$$

$$h = \frac{\log n}{\log 2}$$

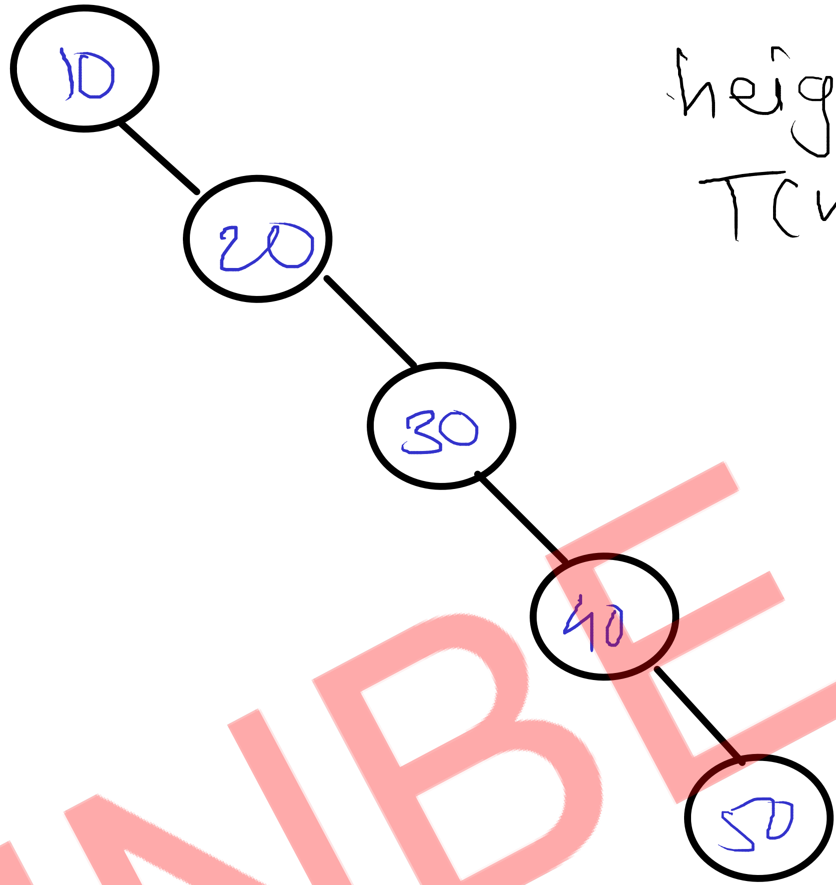
Skewed BST

Keys : 30, 40, 20, 50, 10



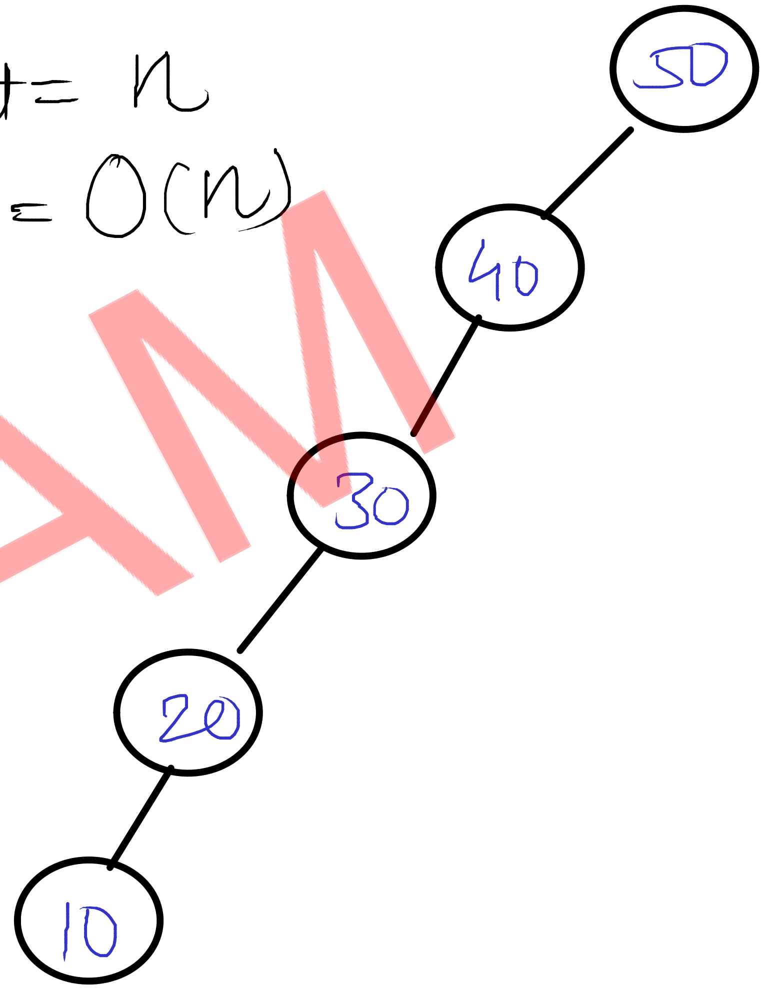
height = log n
$T(n) = O(\log n)$

Keys : 10, 20, 30, 40, 50



height = n
$T(n) = O(n)$

Key : 50, 40, 30, 20, 10



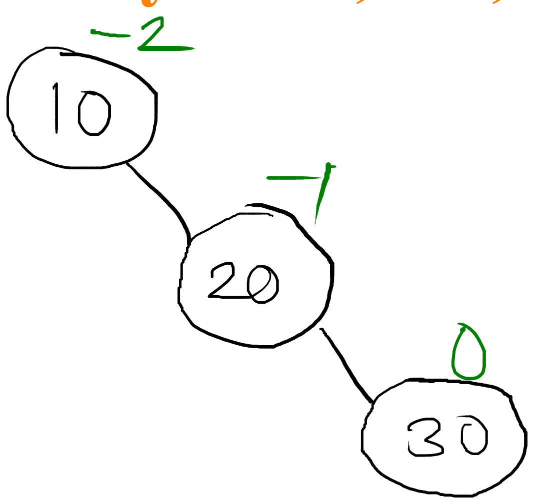
- **if tree is growing only in one direction, that tree is skewed BST**
- **if tree is growing only in left direction, that tree is left skewed BST**
- **if tree is growing only in right direction, that tree is right skewed BST**

Balanced BST

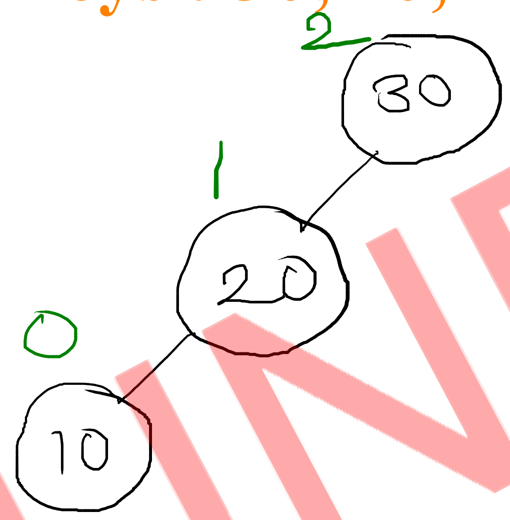
Balance Factor = **height(left sub tree)** - **height(right sub tree)**

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

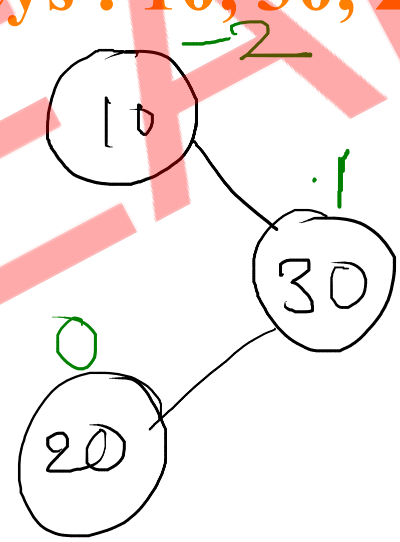
Keys : 10, 20 ,30



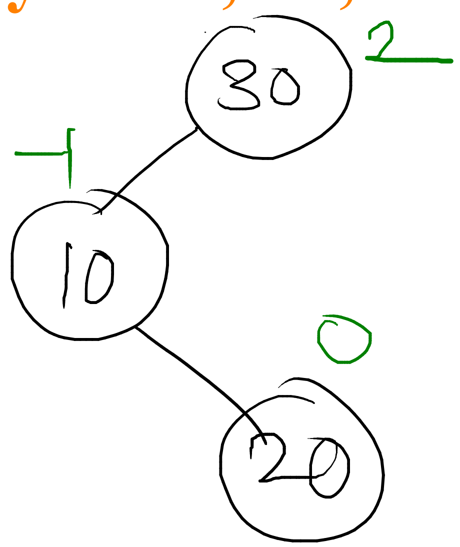
Keys : 30, 20, 10



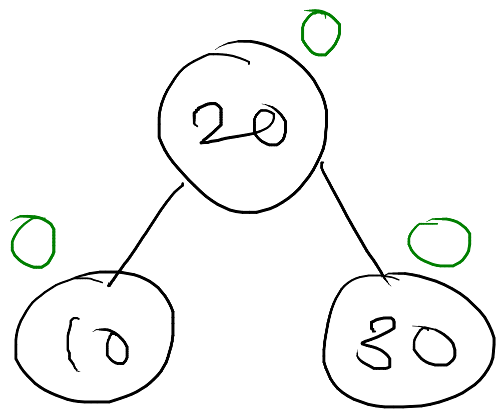
Keys : 10, 30, 20



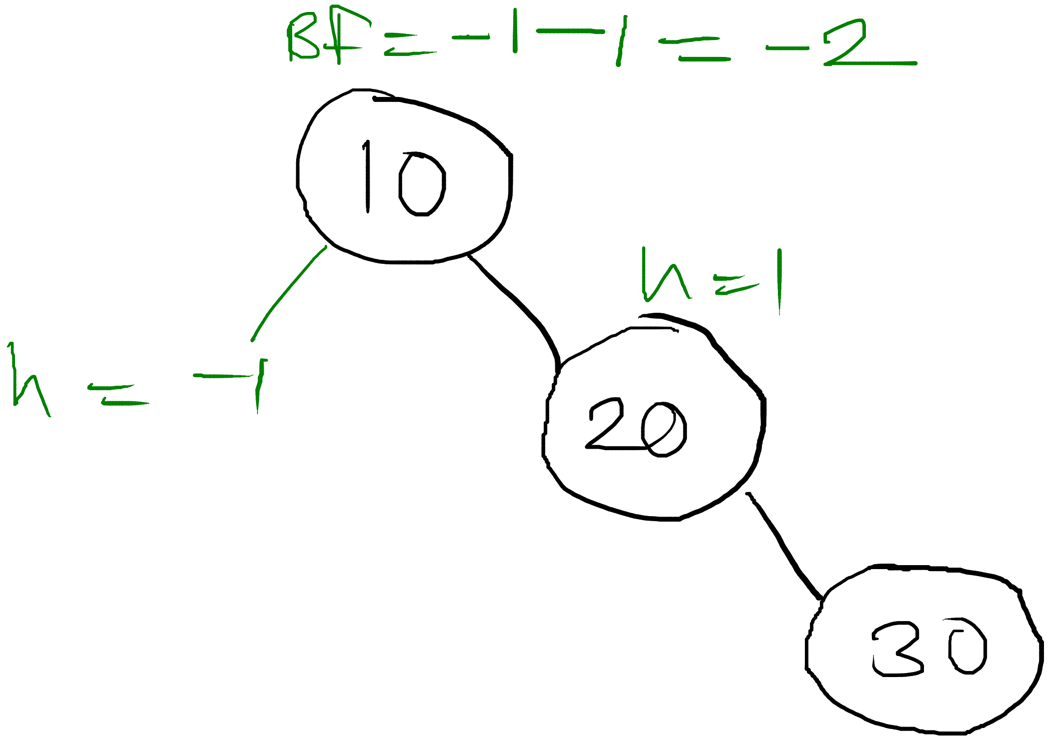
Keys : 30, 10, 20

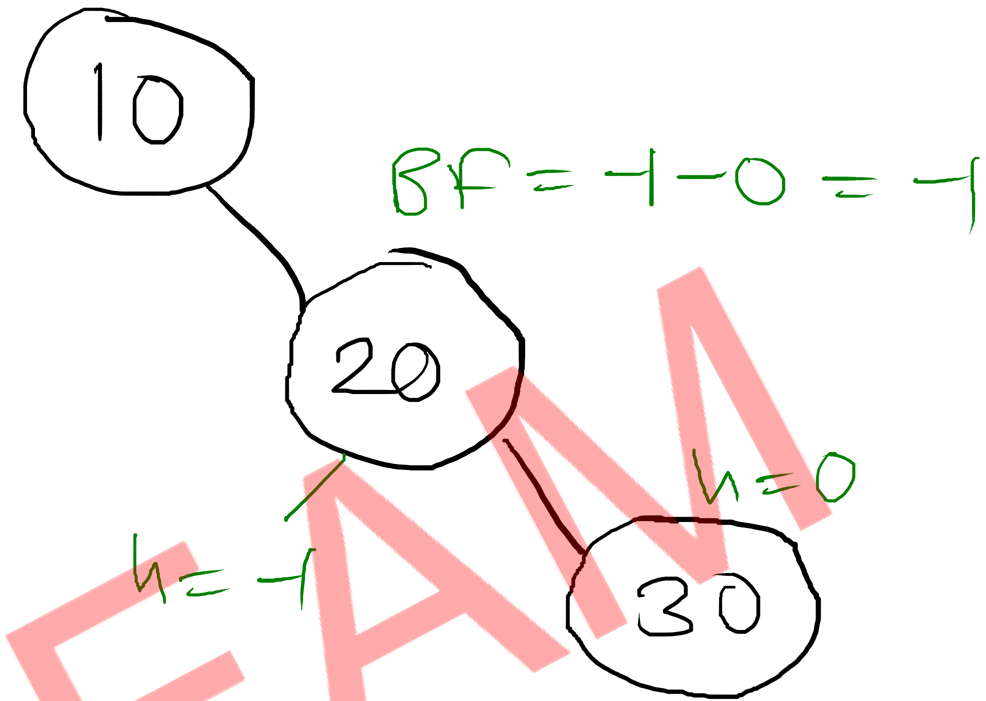


Keys : 20, 10, 30
Keys : 20, 30, 10



Balanced BST

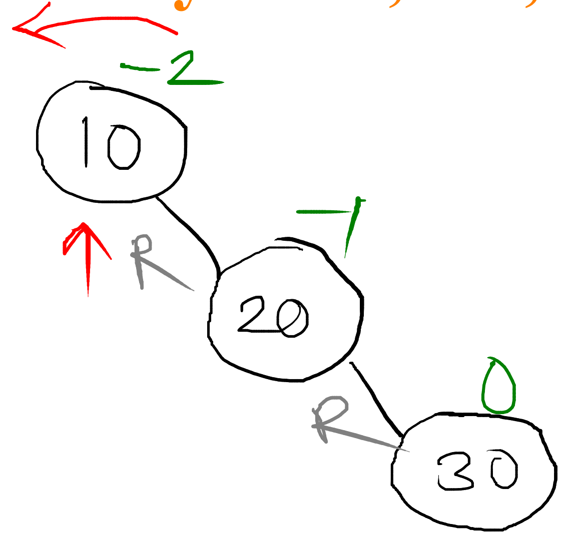

BF = -1-1 = -2
10
h = -1
h=1
20
30


10
BF = -1-0 = -1
20
h=0
h=-1
30

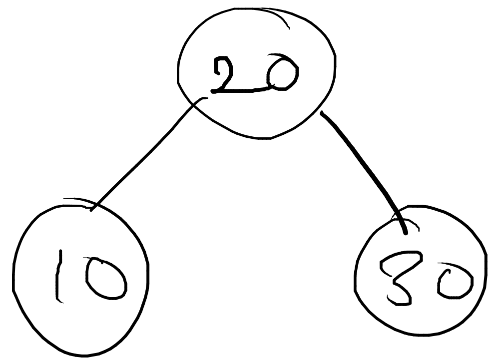
Rotations

RR Imbalance

Keys : 10, 20 ,30

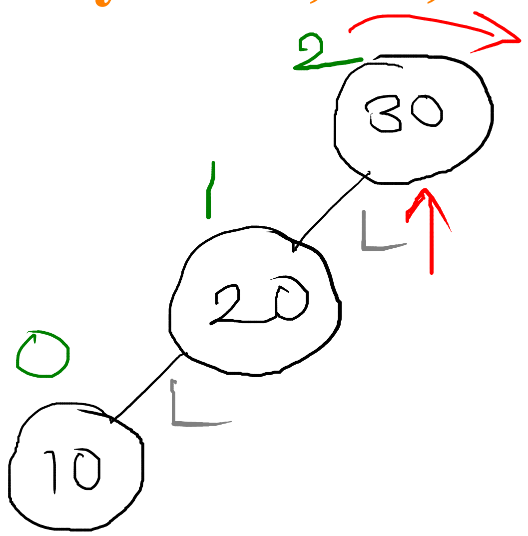


Left Rotation

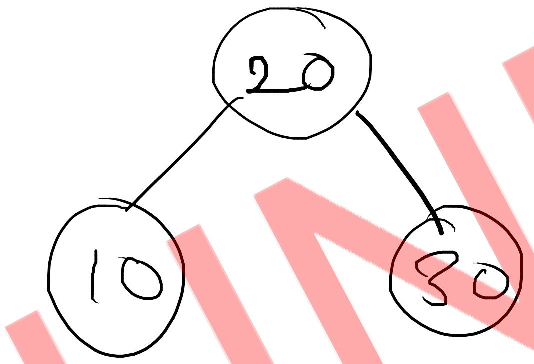


LL Imbalance

Keys : 30, 20, 10

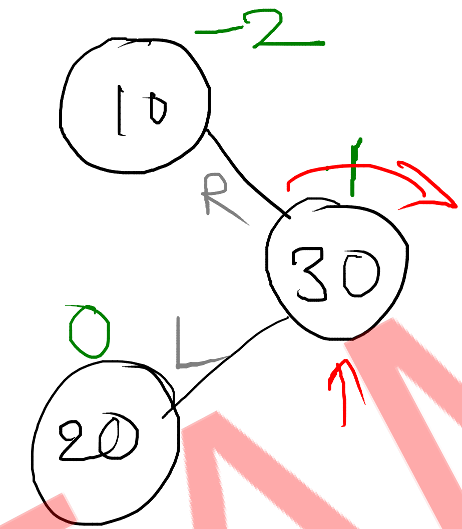


Right Rotation

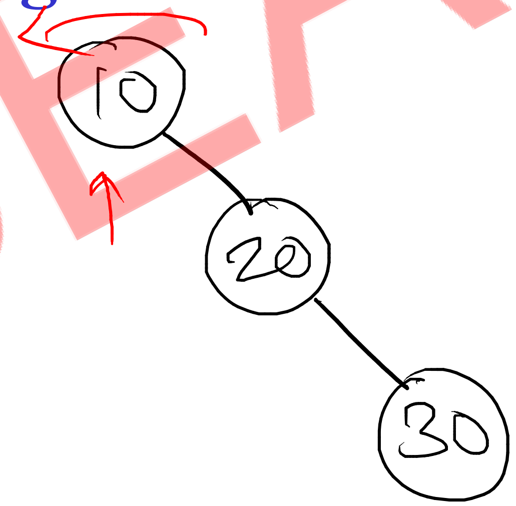


RL Imbalance

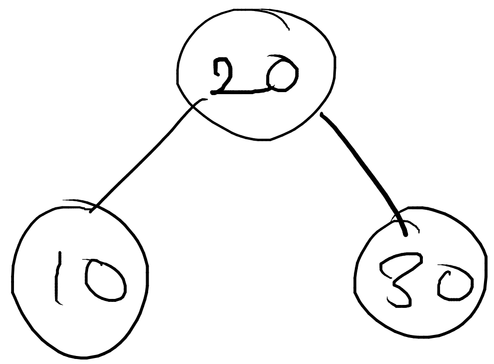
Keys : 10, 30, 20



Right Rotation

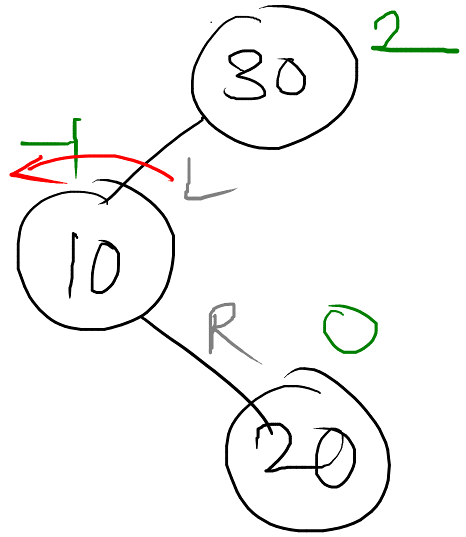


Left Rotation

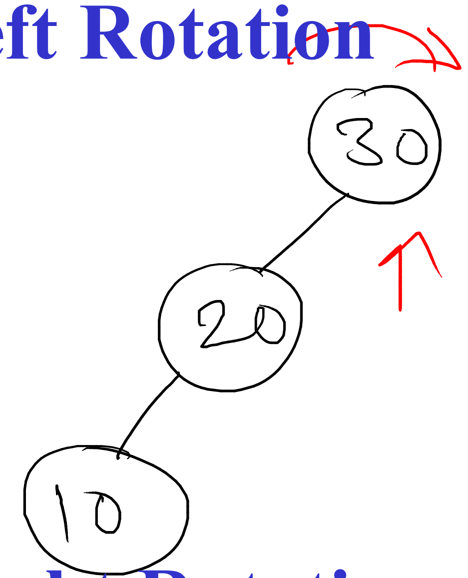


LR Imbalance

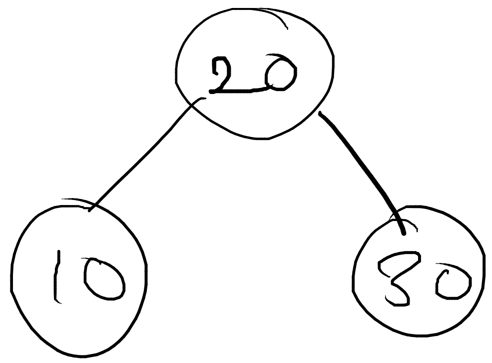
Keys : 30, 10, 20



Left Rotation



Right Rotation

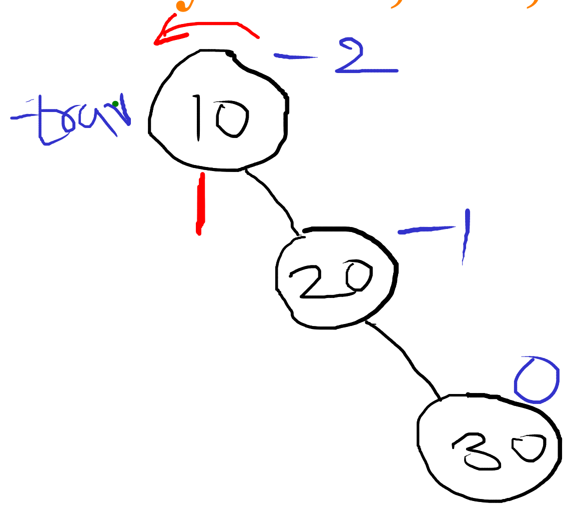


Single Rotation

Double Rotation

RR Imbalance

Keys : 10, 20 ,30

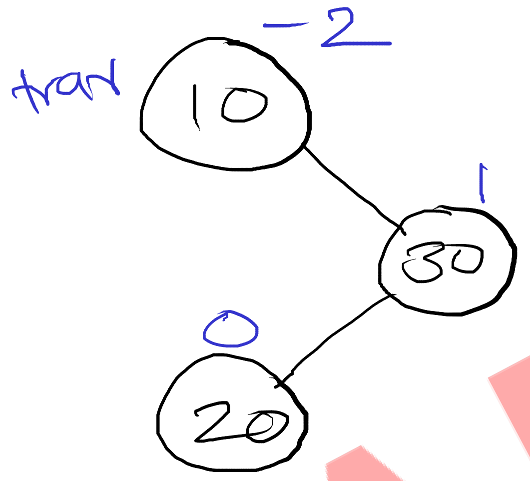


$bf < -1$

value > trav.right.data

RL Imbalance

Keys : 10, 30, 20

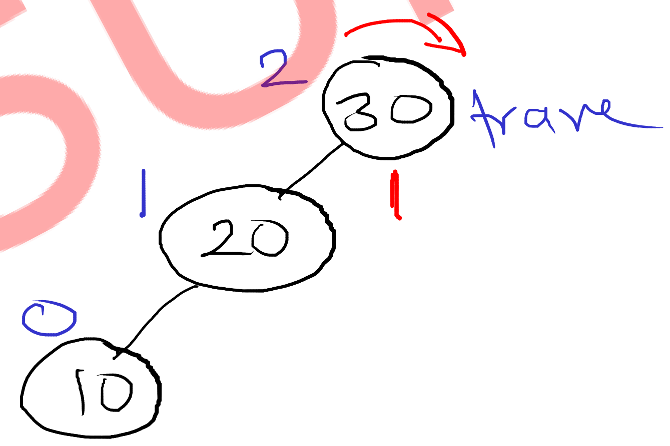


$bf < -1$

value < trav.right.data

LL Imbalance

Keys : 30, 20, 10

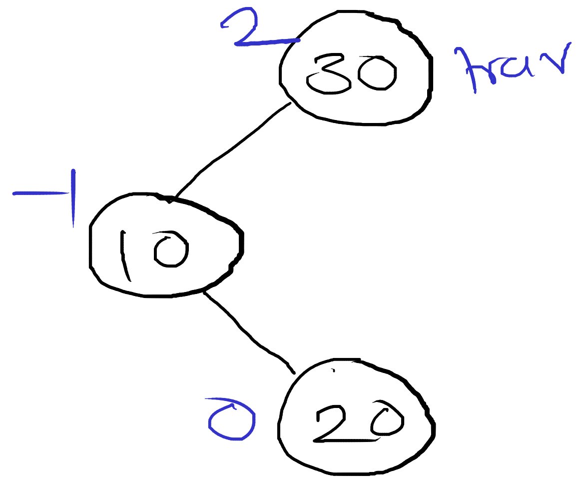


$bf > 1$

value < trav.left.data

LR Imbalance

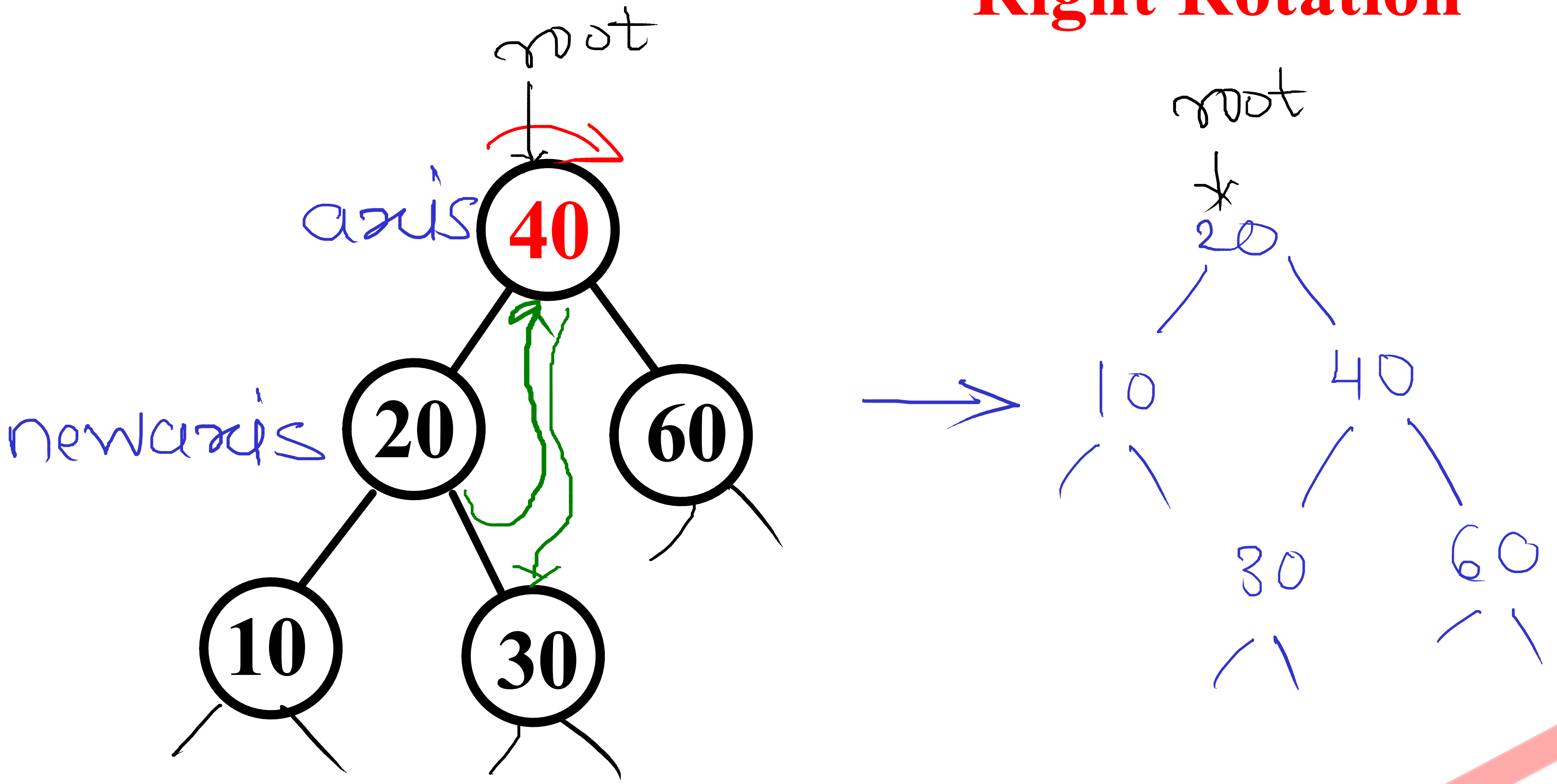
Keys : 30, 10, 20



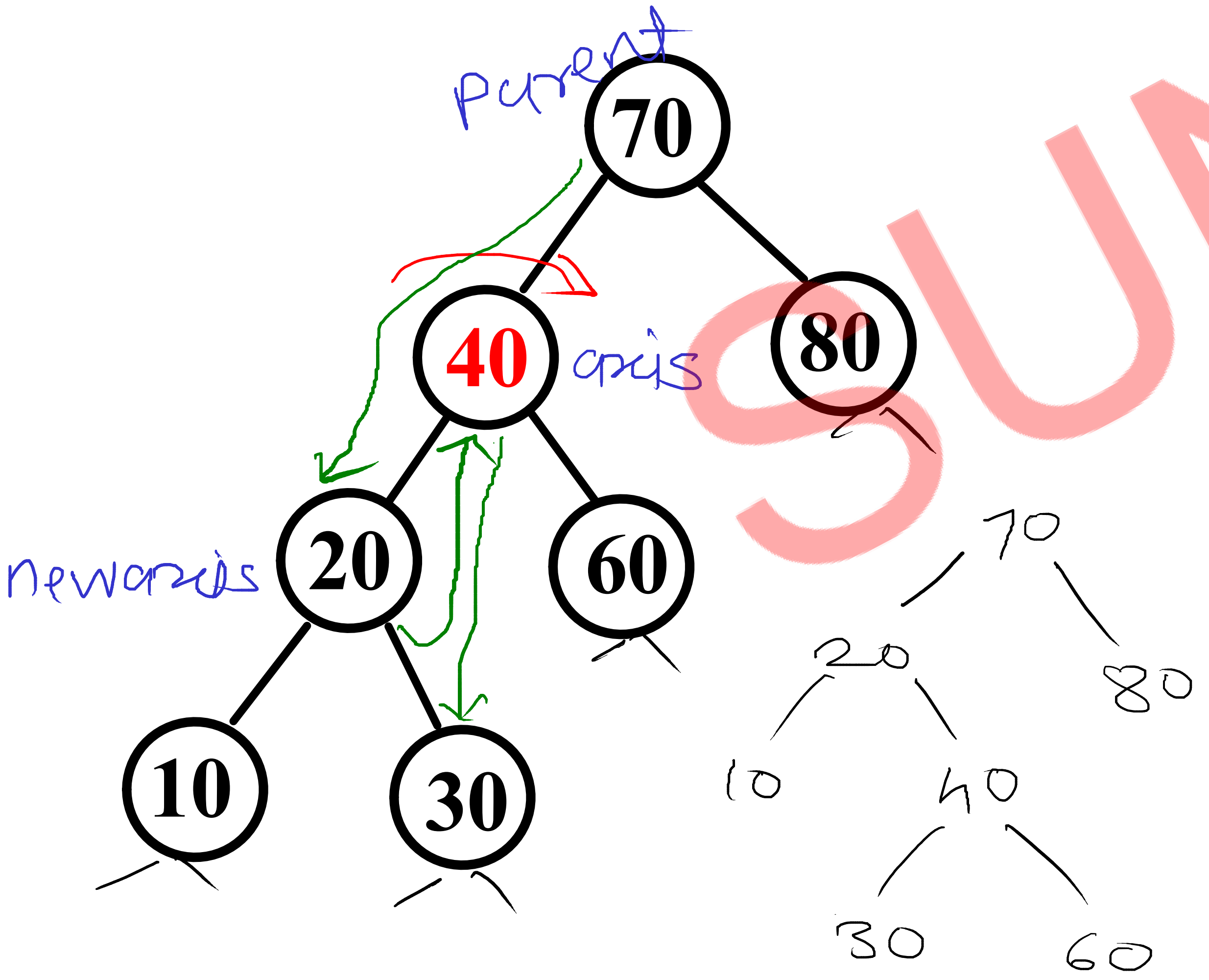
$bf > 1$

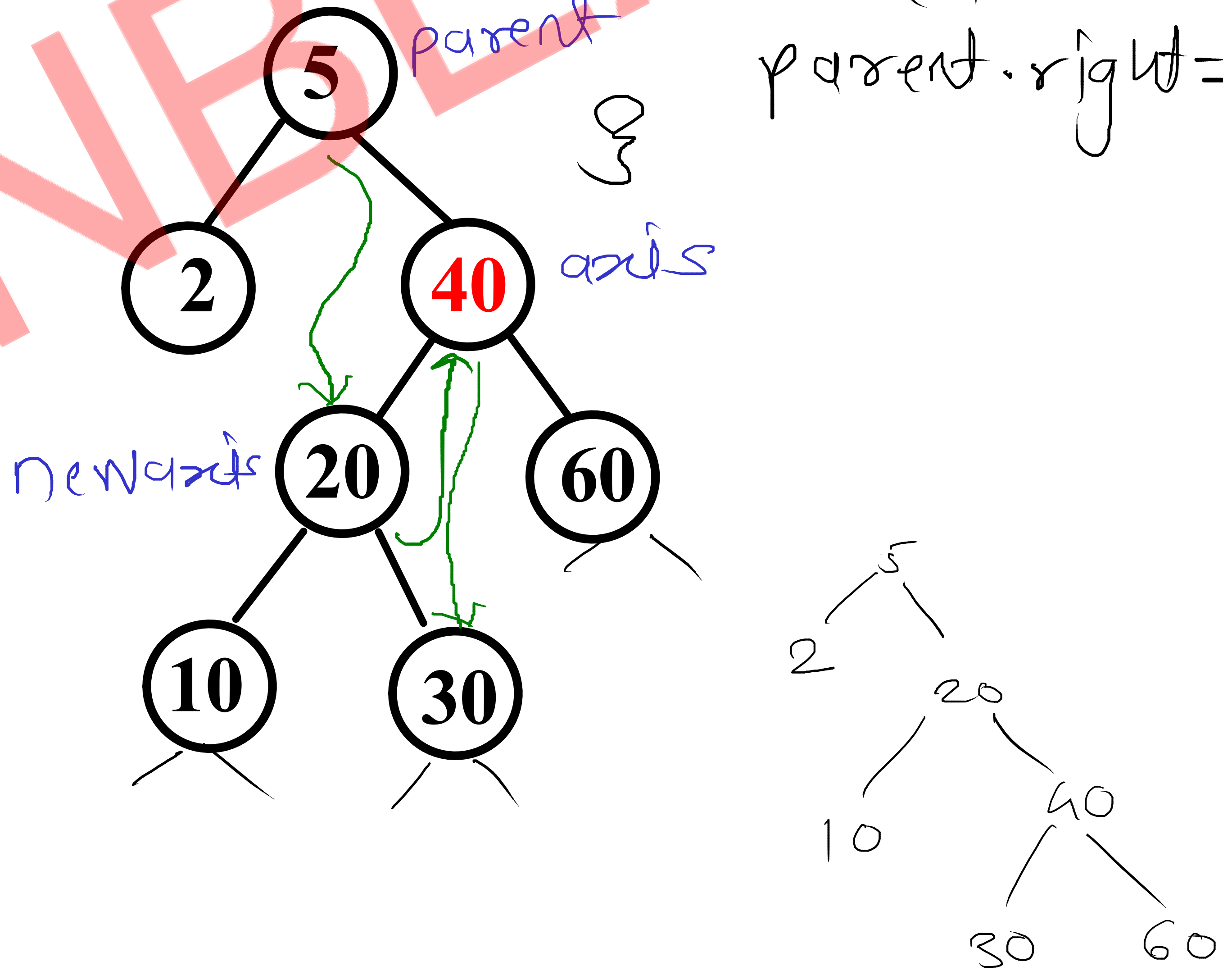
value > trav.left.data

Right Rotation

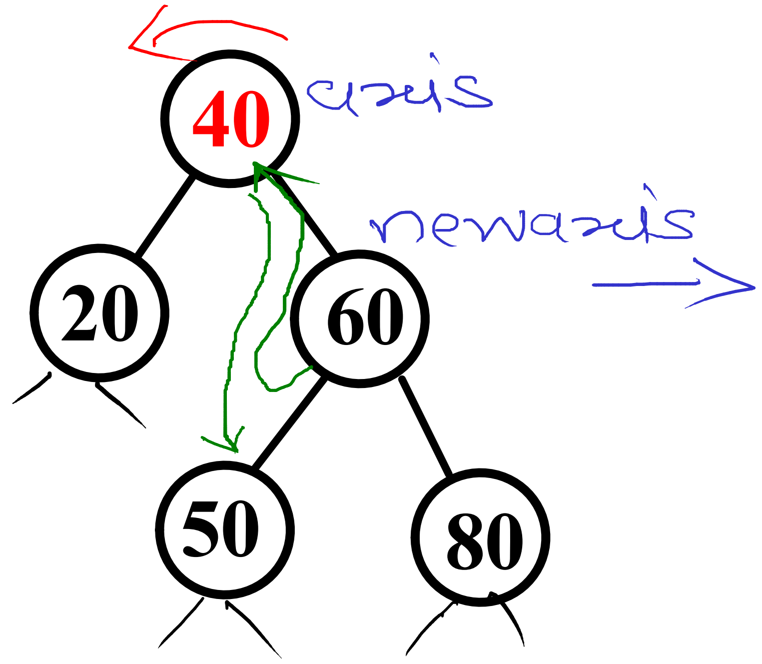


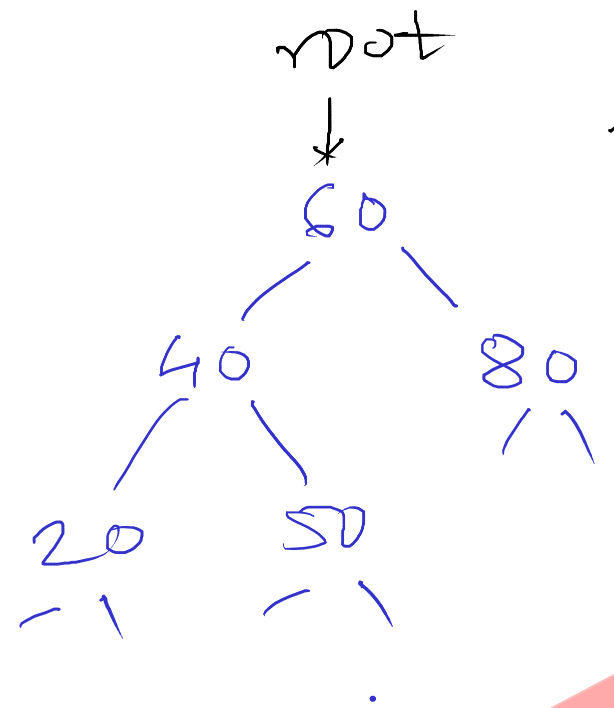

```
right_rotation(axis, parent){
    newaxis = axis.left
    axis.left = newaxis.right
    newaxis.right = axis
    if(axis == root)
        root = newaxis;
    else if(axis == parent.left)
        parent.left = newaxis
    else if(axis == parent.right)
        parent.right = newaxis
}
```



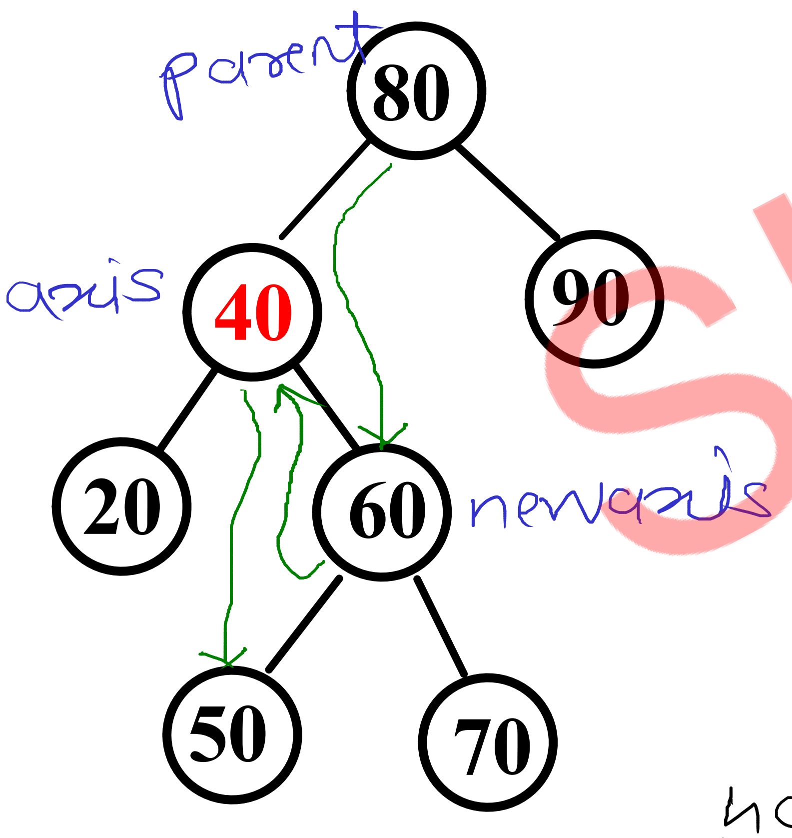


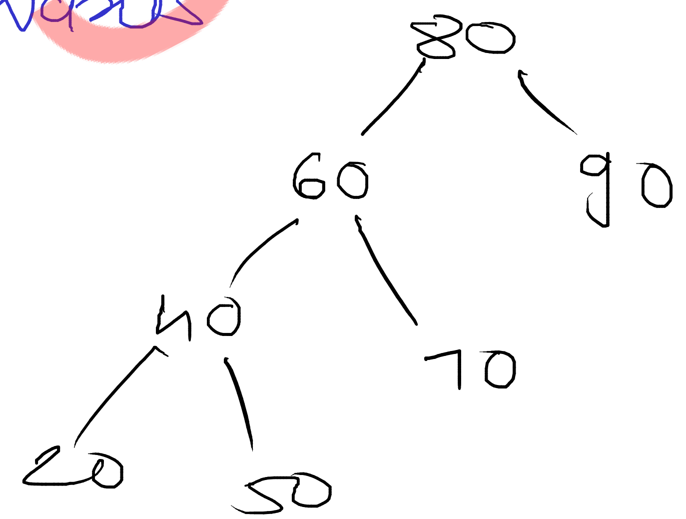
Left Rotation

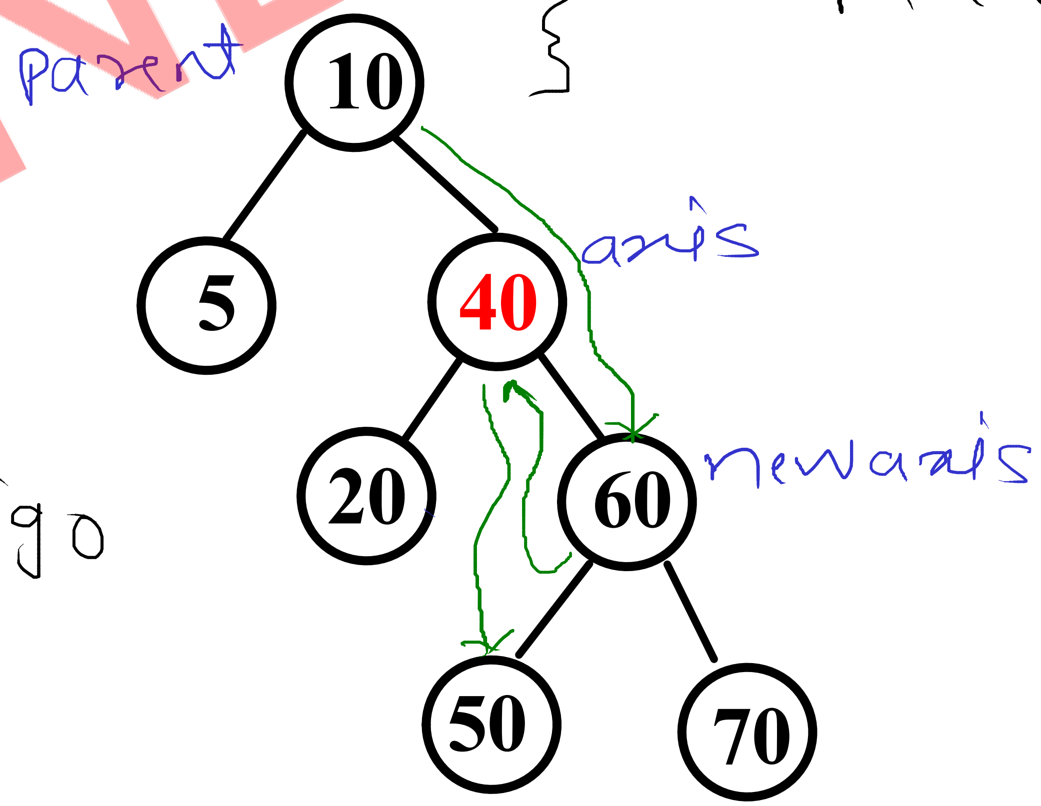


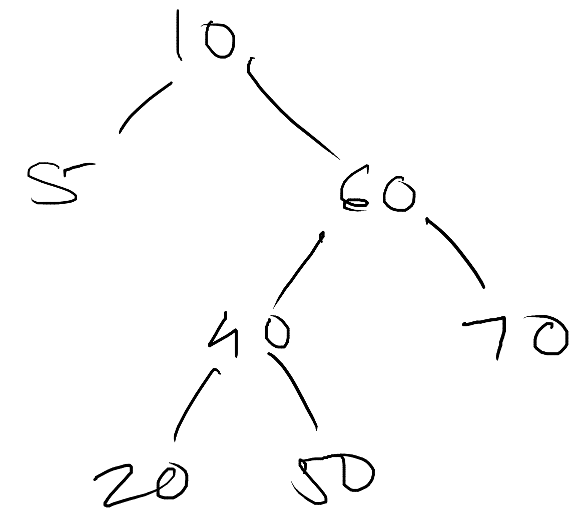



```
left_rotation(axis, parent){
    newaxis = axis.right
    axis.right = newaxis.left
    newaxis.left = axis
    if (axis == root)
        root = newaxis;
    else if(axis==parent.left)
        parent.left=newaxis;
    else if(axis == parent.right)
        parent.right = newaxis;
}
```





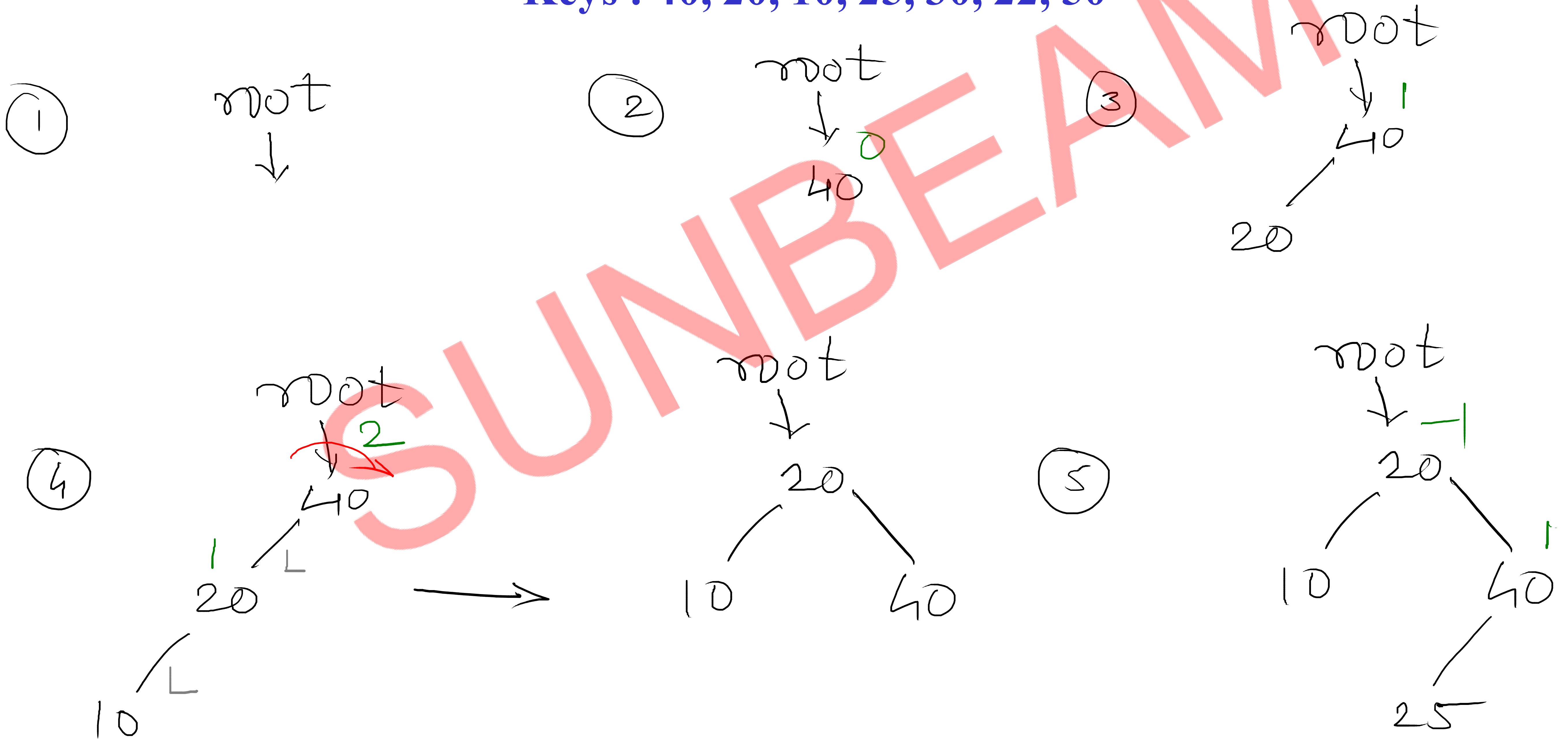




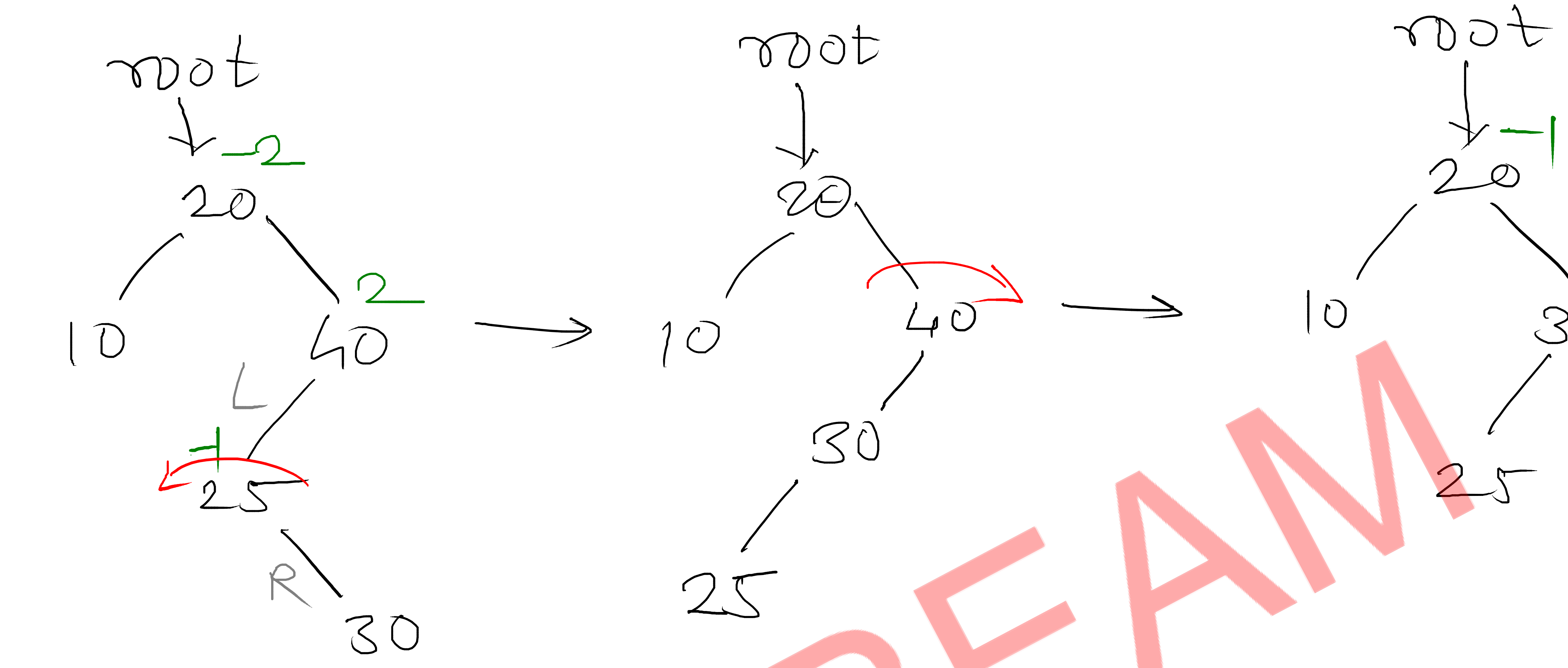
AVL Tree

- 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

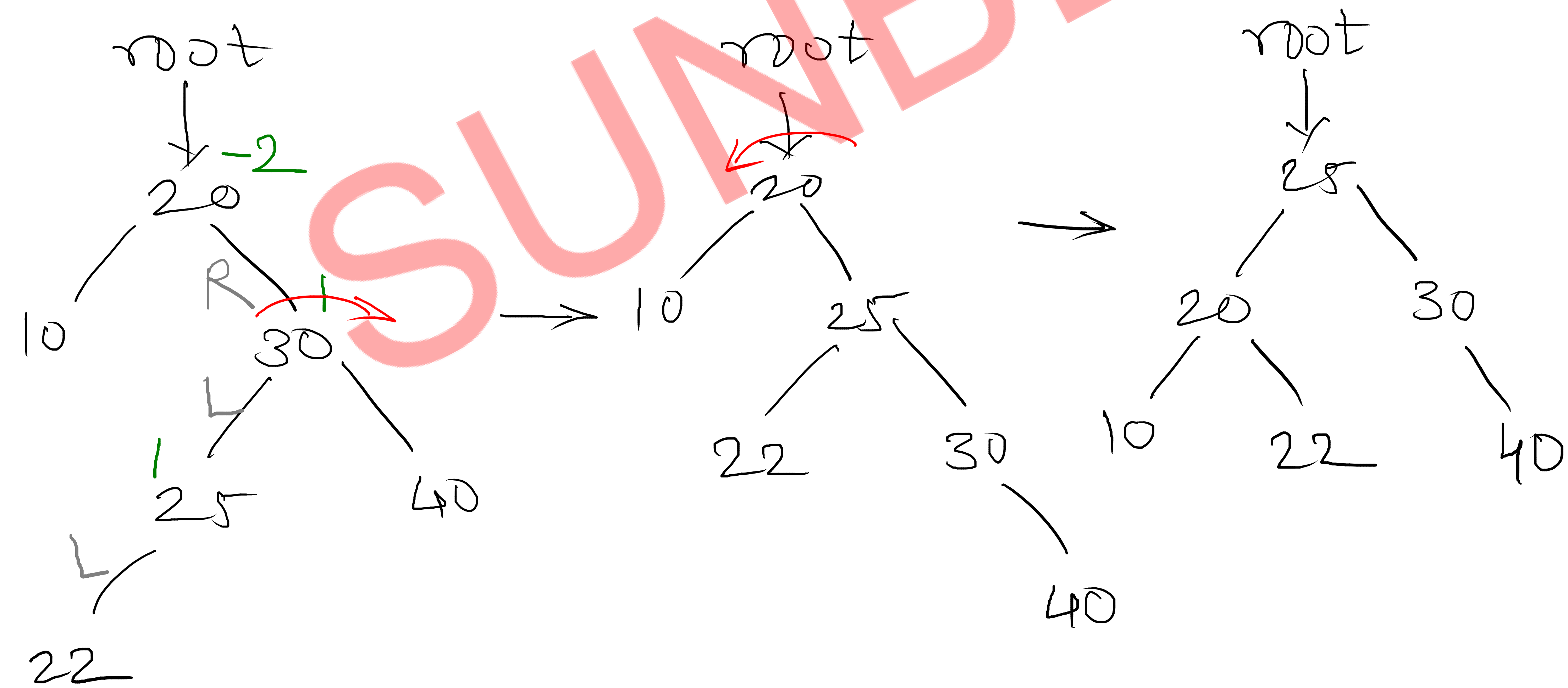
Keys : 40, 20, 10, 25, 30, 22, 50

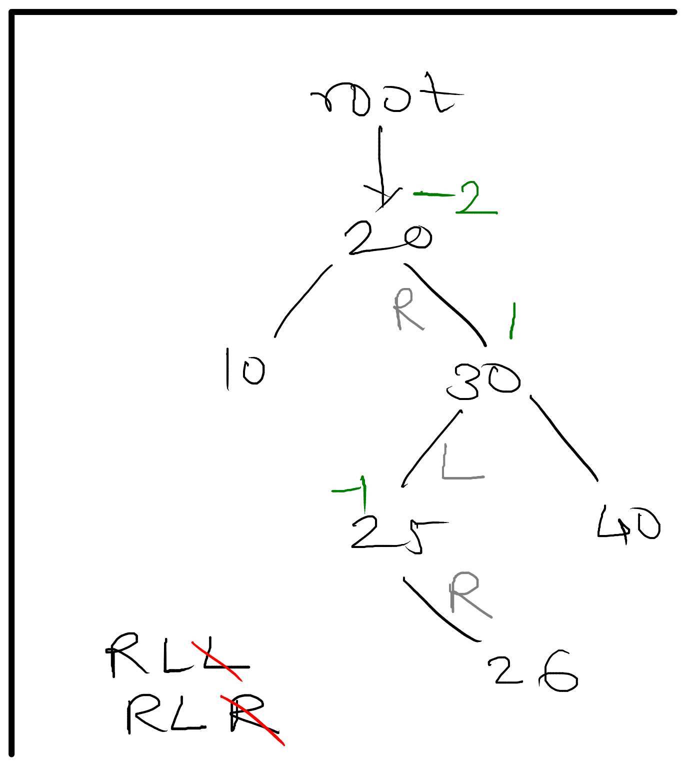


6


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

7

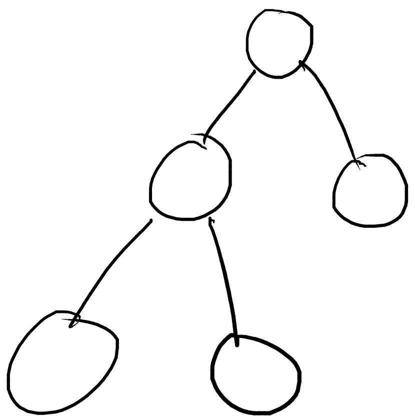

root
-2
20
10
R
1
30
L
1
25
40
L
22
root
20
10
25
22
30
40
root
25
20
30
10
22
40

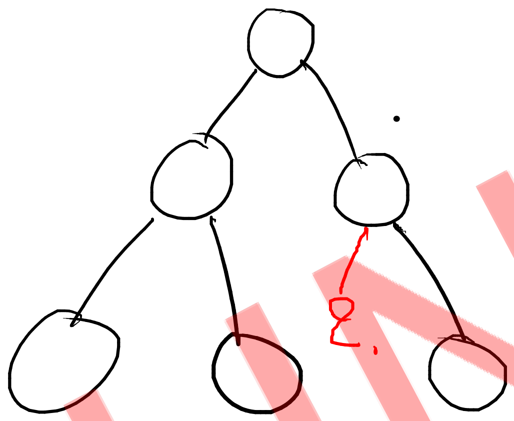

root
-2
20
10
R
1
30
L
-1
25
40
R
26
RL~~L~~
RL~~R~~

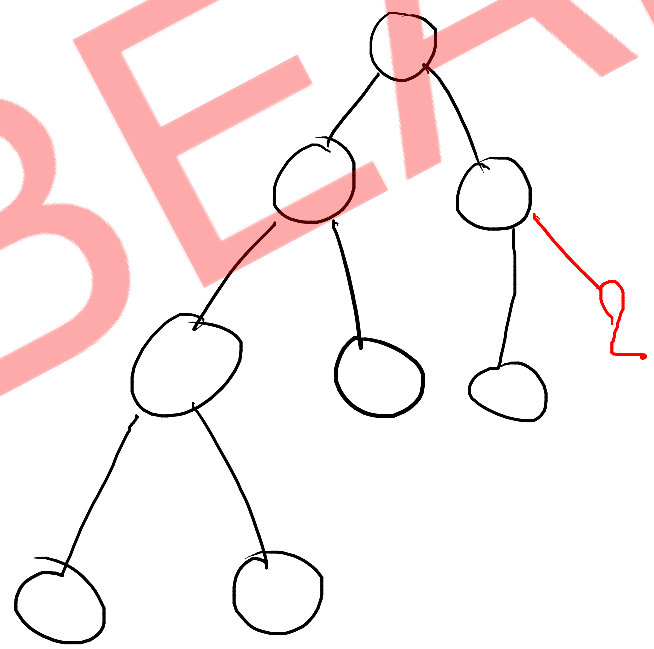
8

Almost Complete Binary Tree
(ACBT)

- this tree is filled level by level (from left to right)
- this tree should satisfy two condition
    1. all leaf nodes must be at level h or h-1
    2. nodes of last level should be filled from left to right without keeping any blank (empty)

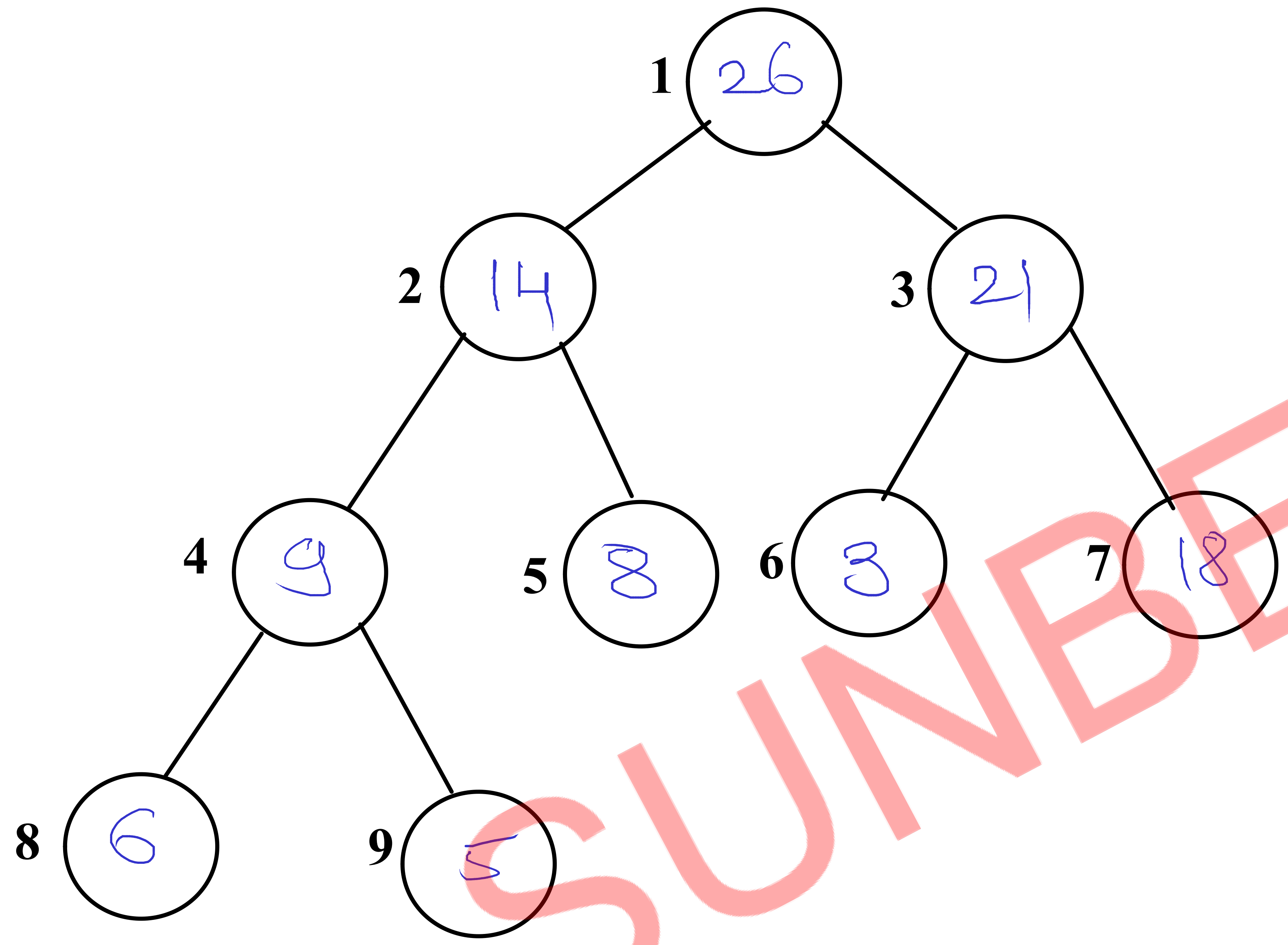






Heap - Create Heap

6 14 3 26 8 18 21 9 5

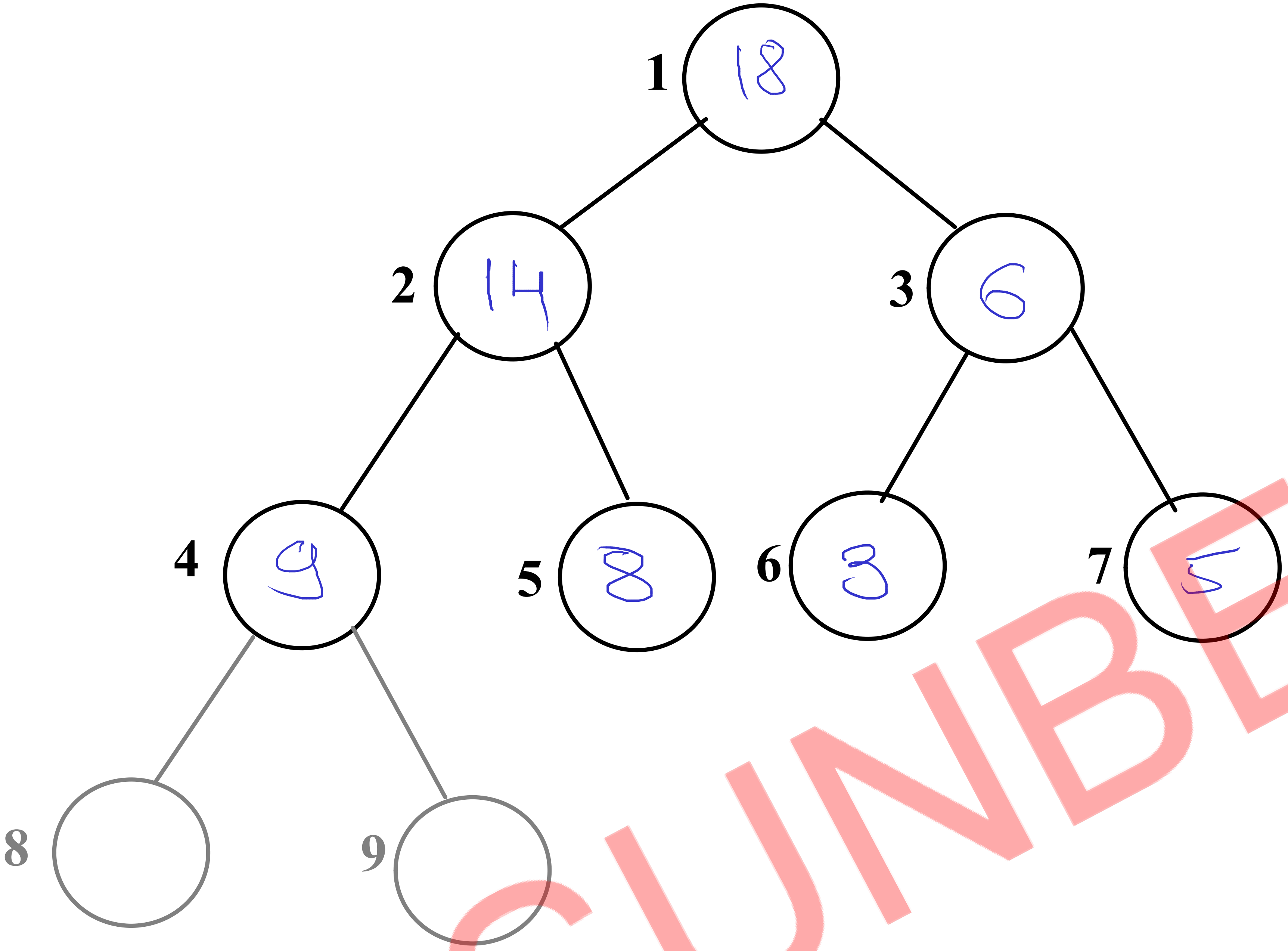


$T(n) = O(\log n)$

1) add new element at first empty location of the heap
2) adjust the position of newly added element by comparing with all its ancestors

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

Heap - Delete Heap



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

~~Max = 26~~
Max = 21

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

1) Delete root element
2) place last element at root's location
3) Adjust the position of it upto leaf nodes.