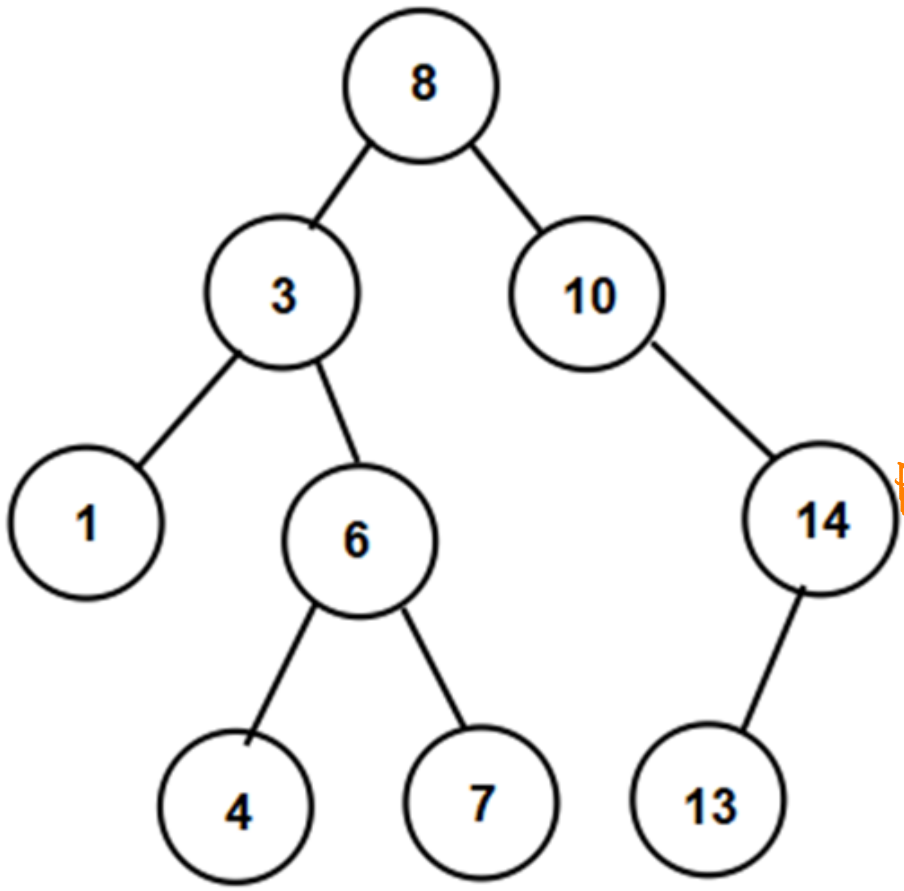
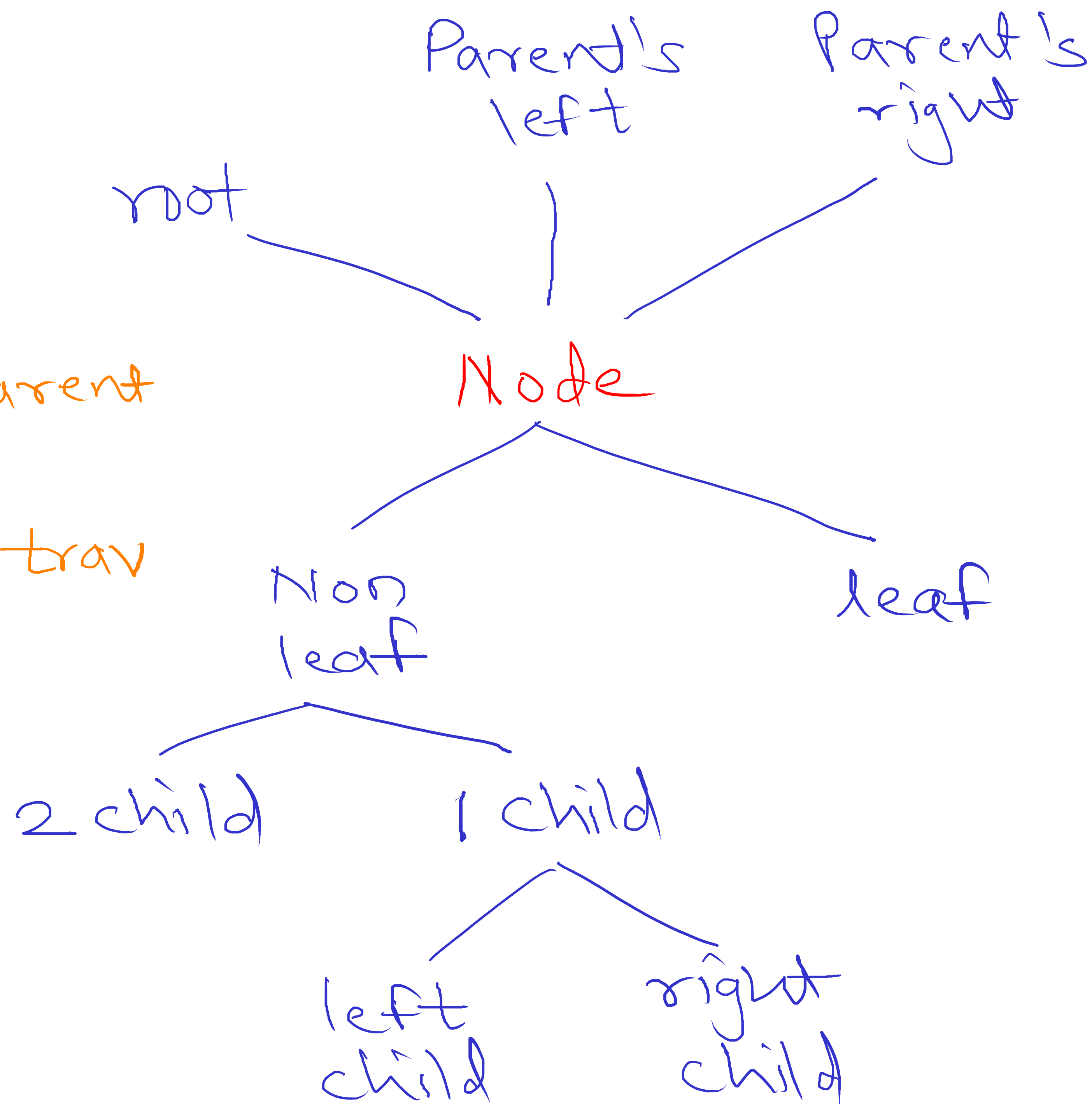
BST - Delete Node



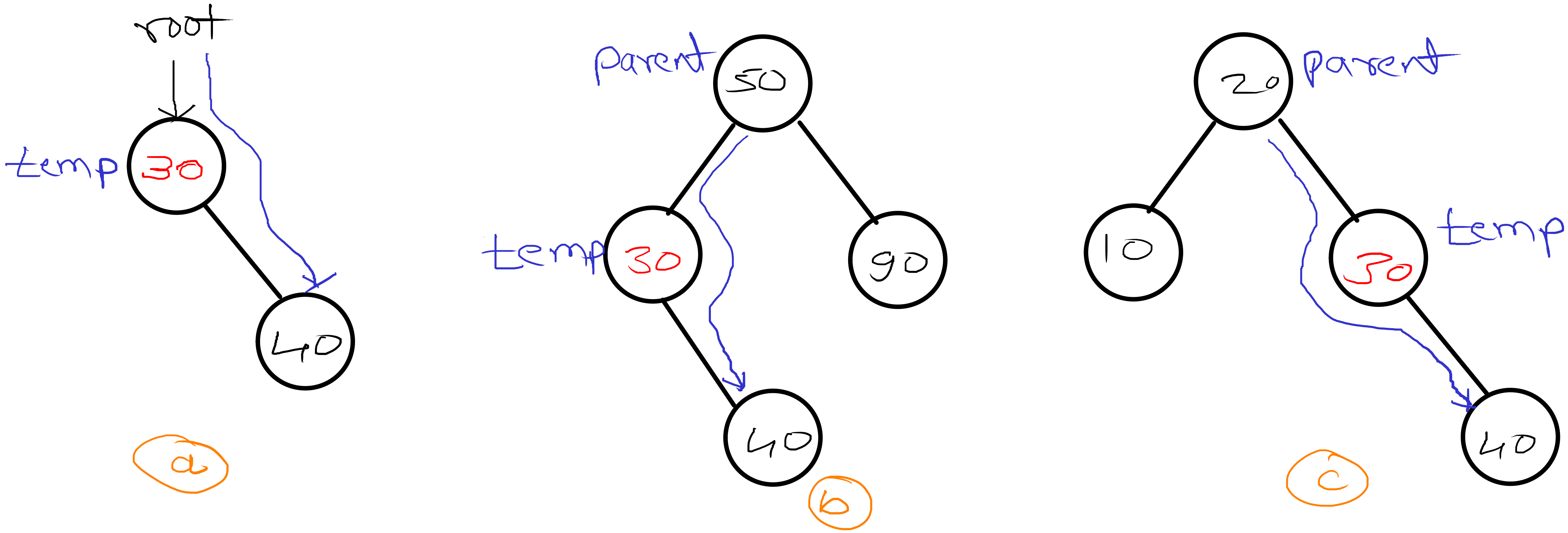
parent

trav

Key = 4

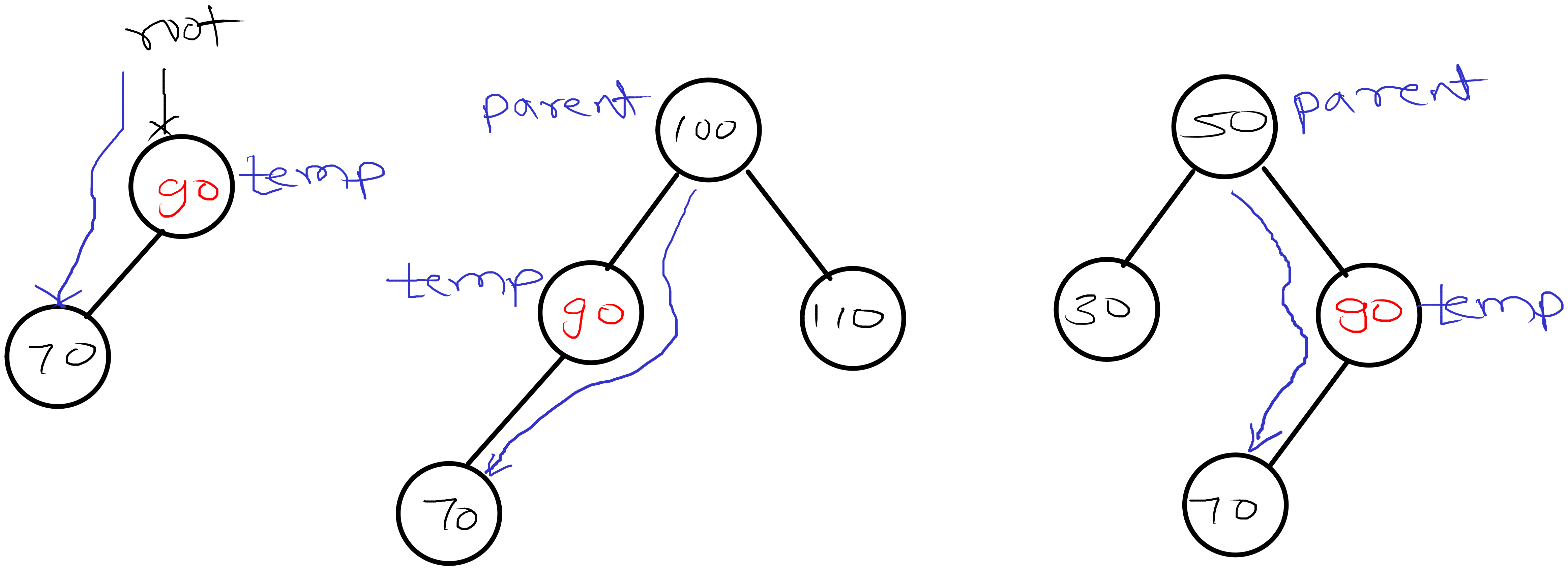


BST - Delete node which has single child (right child)



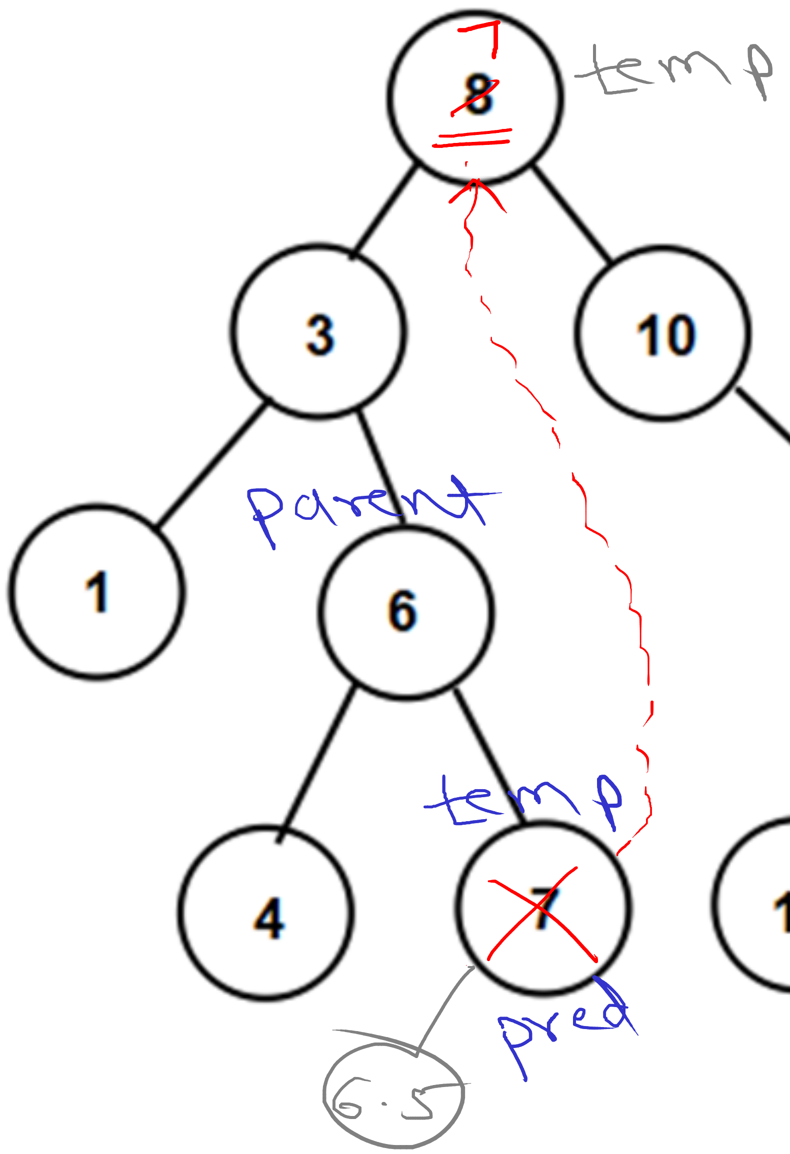

```
if(temp.left == null){
        if(temp == root)                // root node
a               root = temp.right;
        else if(temp == parent.left)            // parent's left child
b               parent.left = temp.right;
        else  // if(temp == parent.right)    // parent's right child
c               parent.right = temp.right;
}
```

BST - Delete node which has single child (left child)




```
if(temp.right == null){
        if(temp == root)                        // root node
                root = temp.left;
        else if(temp == parent.left)            // parent's left child
                parent.left = temp.left;
        else  // if(temp == parent.right)       // parent's right child
                parent.right = temp.left;
}
```

BST - Delete node which has two childs




```
if(temp.left != null && temp.right != null){
        //1. find predecessor with its parent
        Node pred = temp.left;
        parent = temp;
        while(pred.right != null){
                parent = pred;
                pred = pred.right;
        }
        //2. replace data of temp by data of pred
        temp.data = pred.data;
        //3. mark pred for deletion
        temp = pred;
}
```

Inorder traversal : 1 3 4 6 $\boxed{7}$ $\boxed{8}$ $\boxed{10}$ 13 14

inorder predecessor

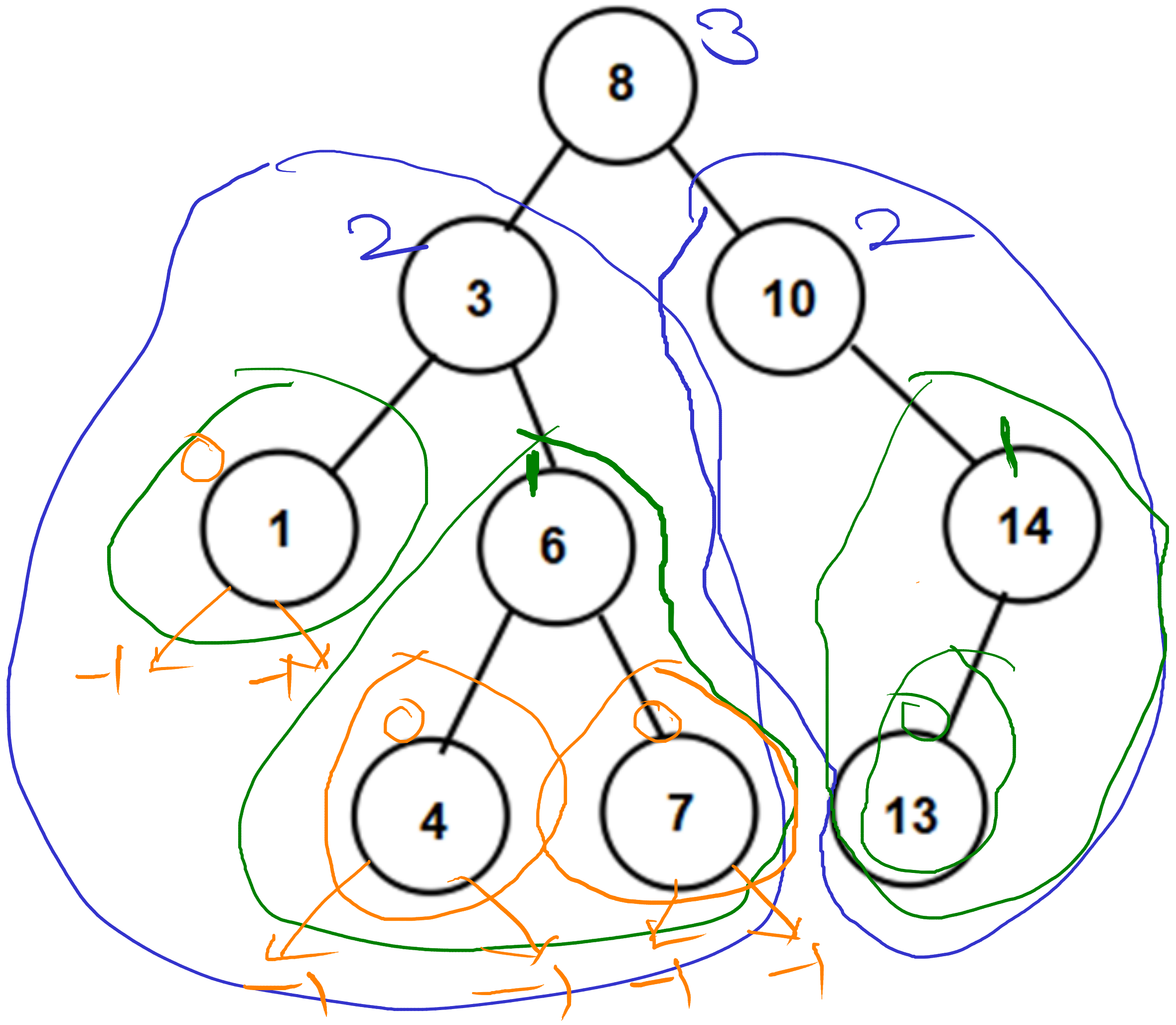
left extreme right

inorder successor

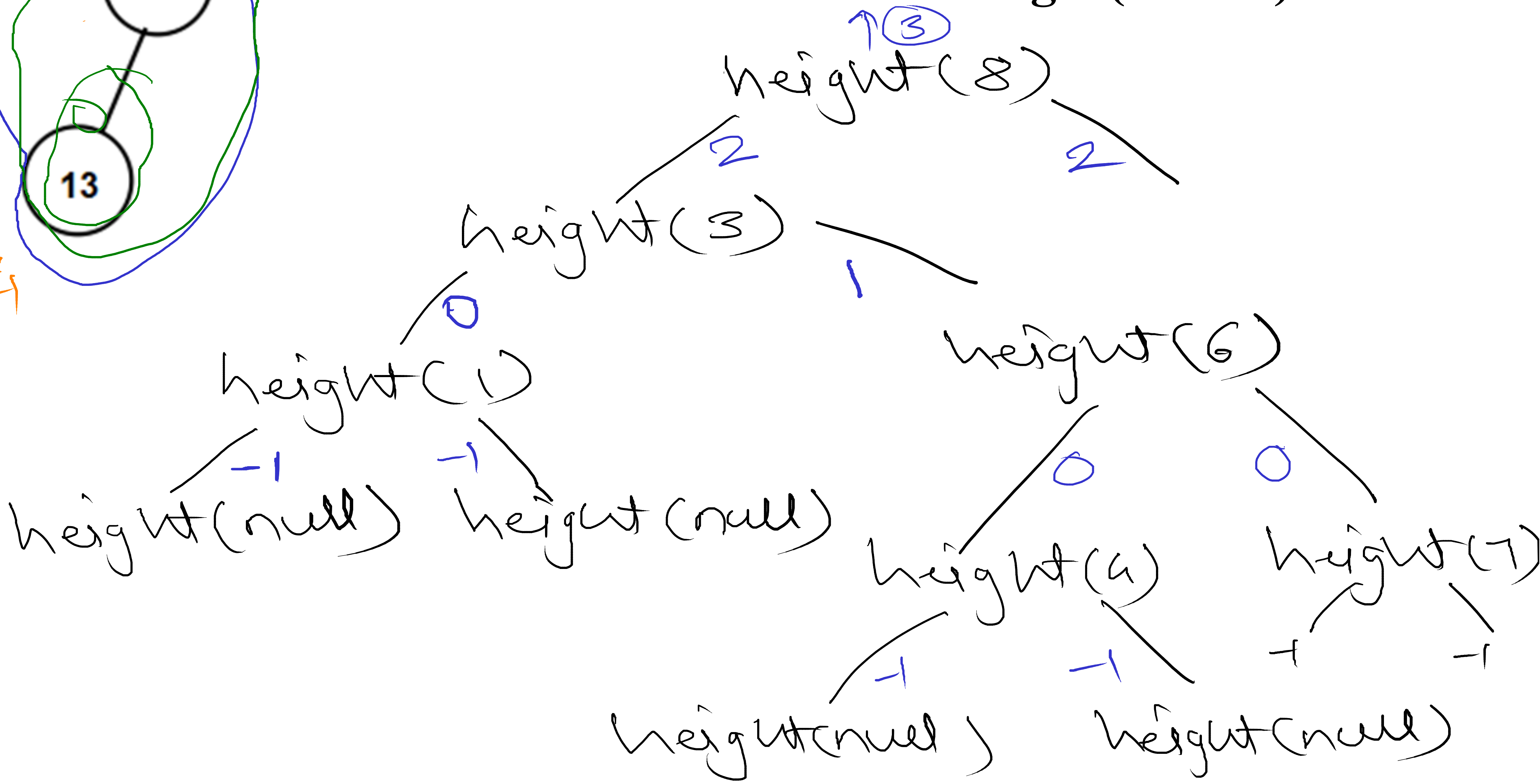
right extreme left

BST - Height

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

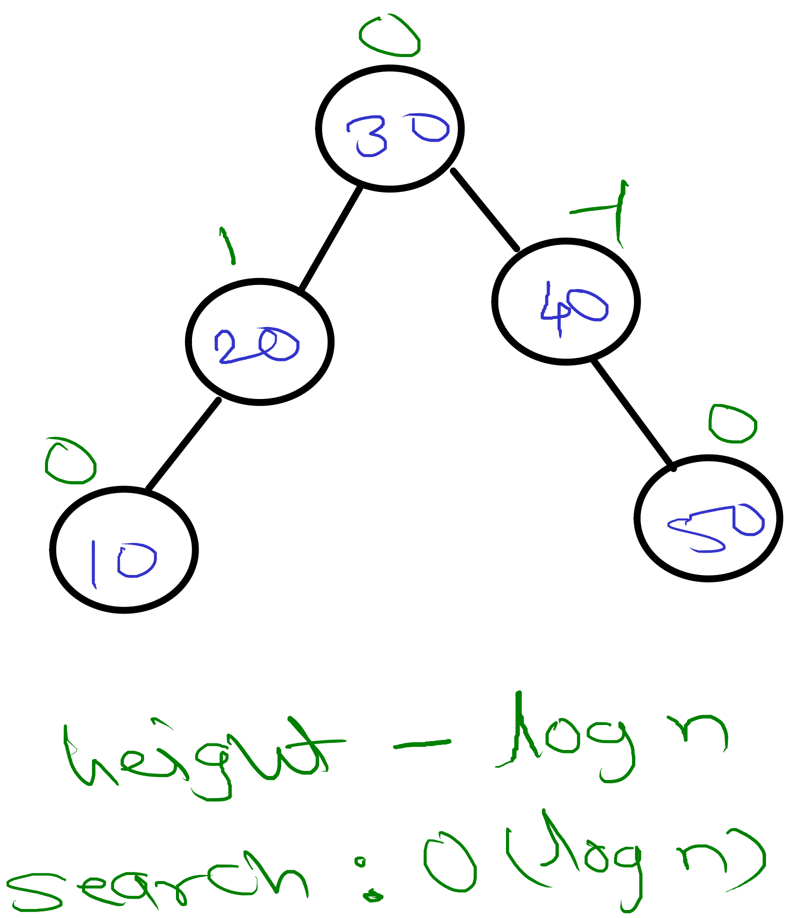


//0. if left or right sub tree is absent
//then return -1
//1. find height of left subtree
//2. find height of right subtree
//3. find max height
//4. add one into max height(return)

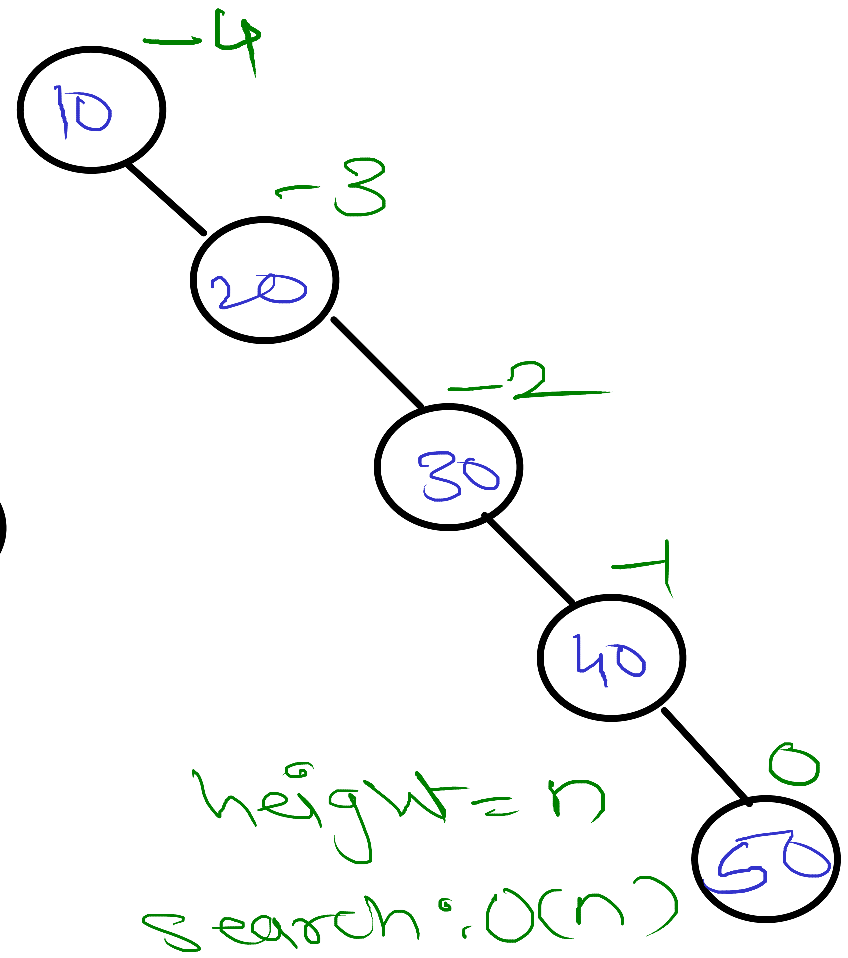


Skewed BST

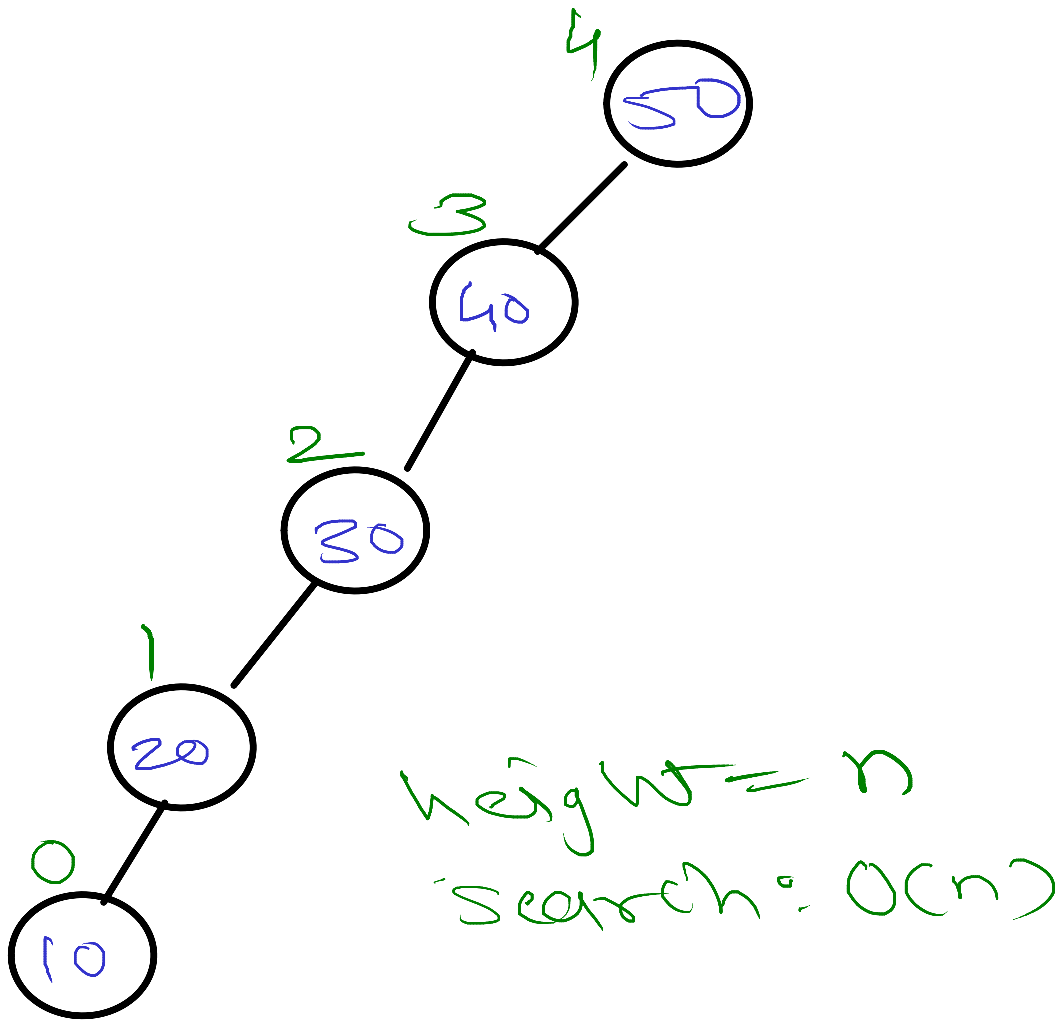
Keys : 30, 40, 20, 50, 10



Keys : 10, 20, 30, 40, 50



Key : 50, 40, 30, 20, 10



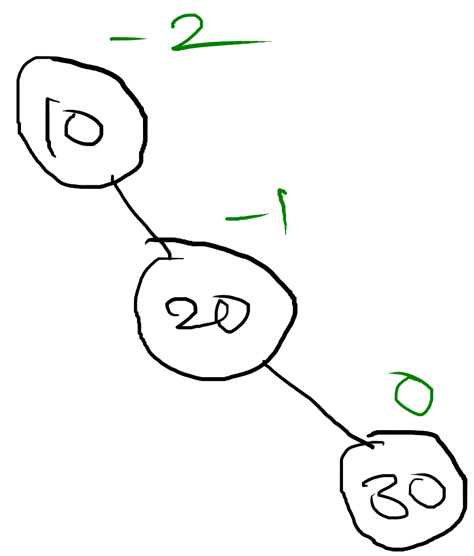
- if tree is growing in only one direction, it is called as skewed BST.
- if tree is growing in only right direction, it is called as right skewed BST.
- if tree is growing in only left direction, it is called as left skewed BST.

Balanced BST

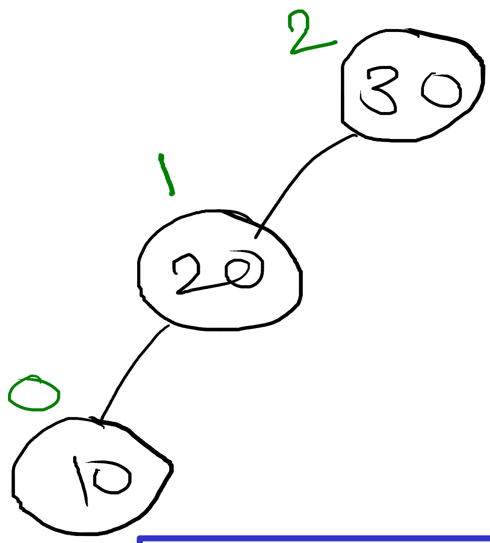
Balance Factor = height(left sub tree) - height(right sub tree)

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

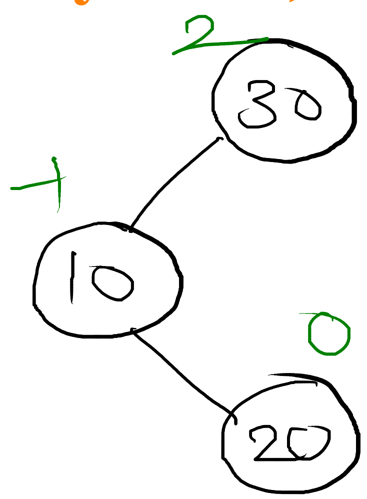
Keys : 10, 20 ,30



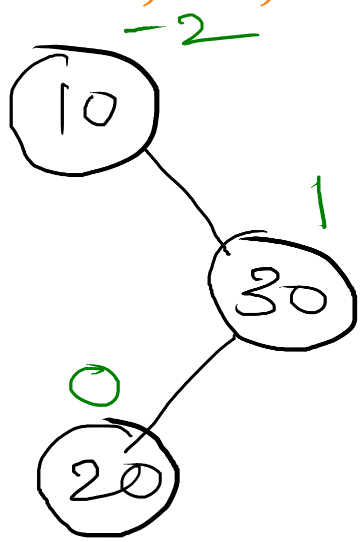
Keys : 30, 20, 10



Keys : 30, 10, 20

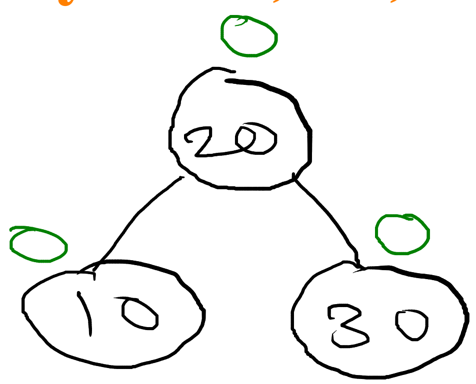


Keys : 10, 30, 20



Keys : 20, 10, 30

Keys : 20, 30, 10

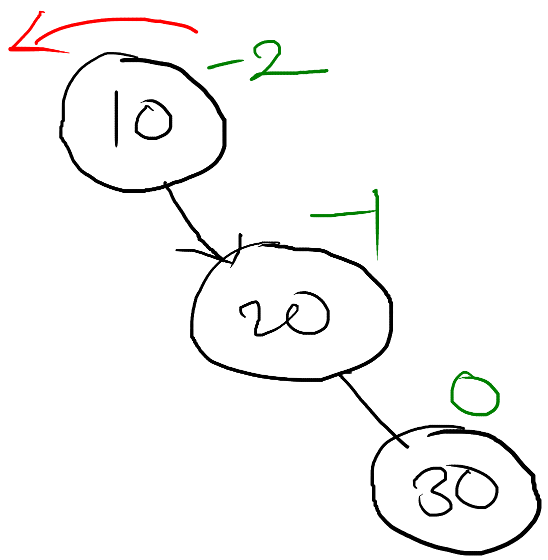


Balanced BST

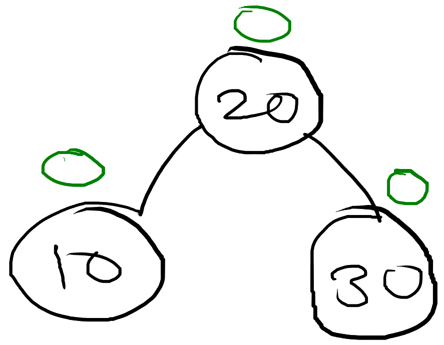
Rotations

RR Imbalance

Keys : 10, 20 ,30

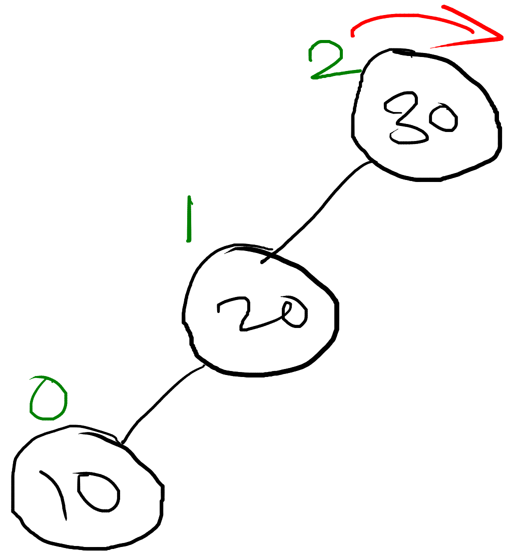


Left Rotation

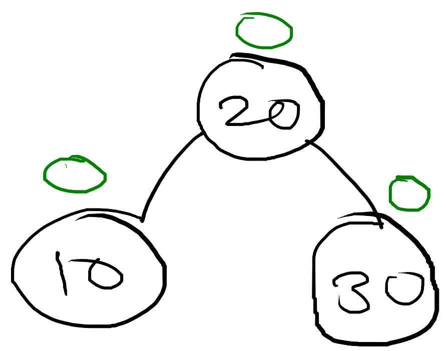


LL Imbalance

Keys : 30, 20, 10



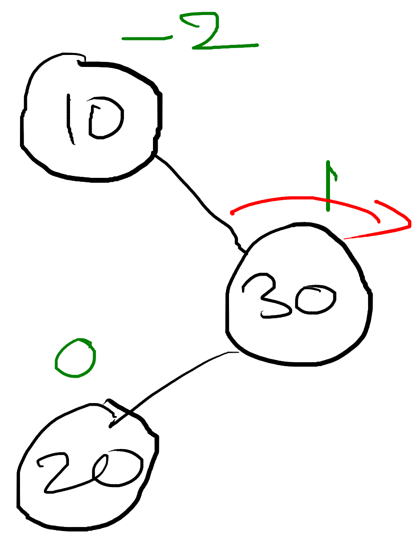
Right Rotation



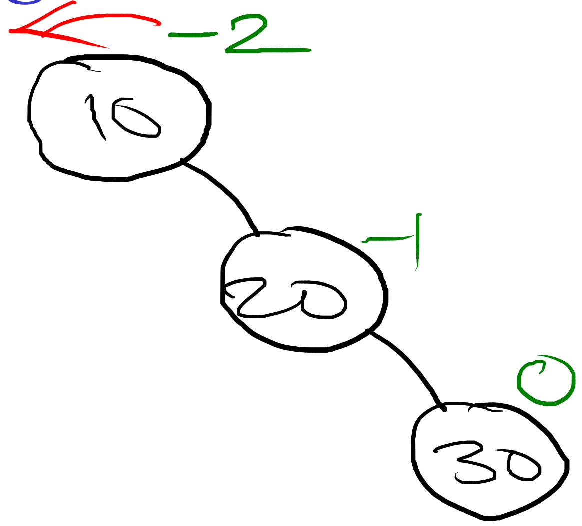
Single Roatation

RL Imbalance

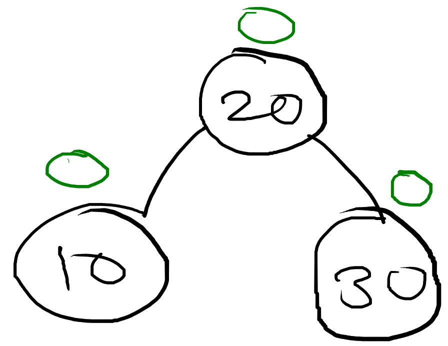
Keys : 10, 30, 20



Right Rotation

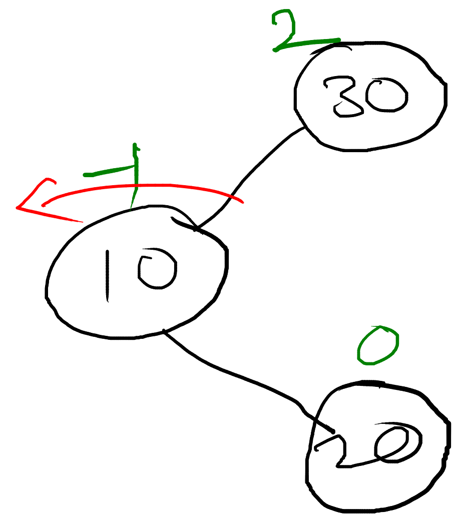


Left Rotation

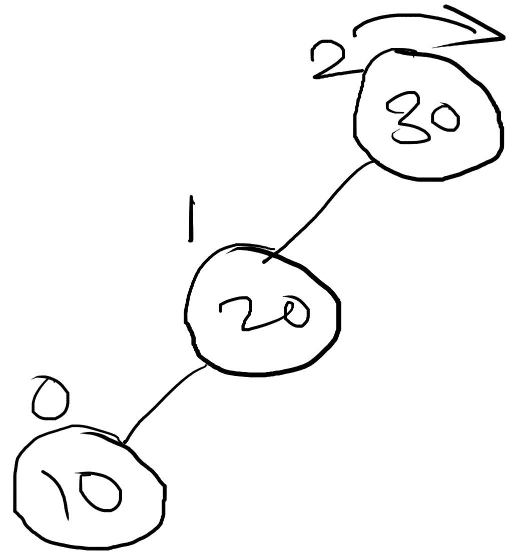


LR Imbalance

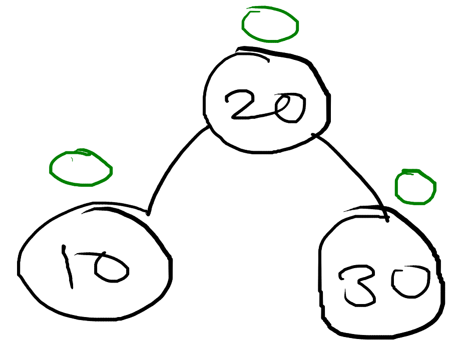
Keys : 30, 10, 20



Left Rotation



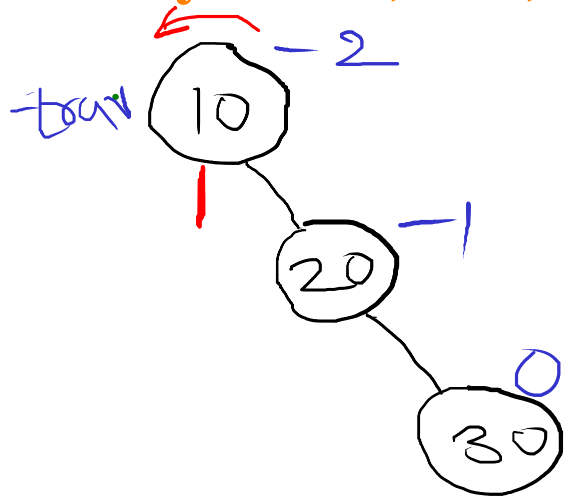
Right Rotation



Double Rotations

RR Imbalance

Keys : 10, 20 ,30

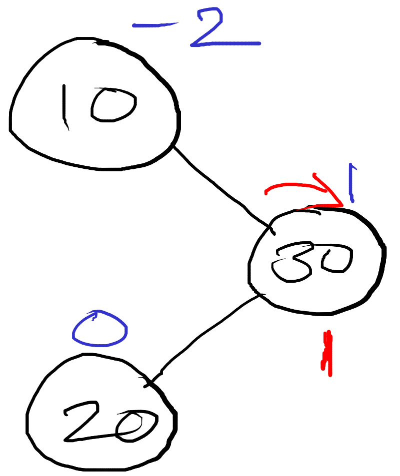


bf < -1 &&

val > trav.right.data

RL Imbalance

Keys : 10, 30, 20

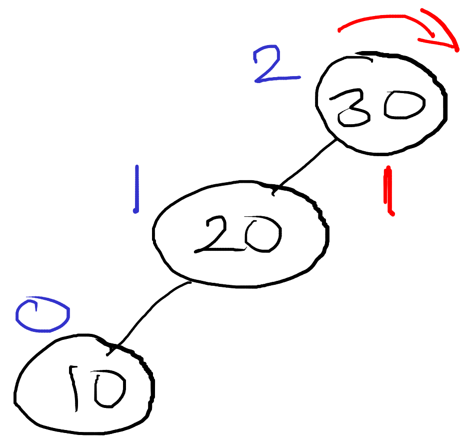


bf < -1 &&

val < trav.right.data

LL Imbalance

Keys : 30, 20, 10

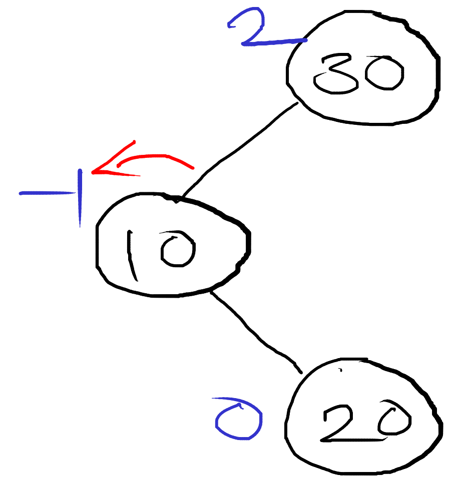


bf > 1 &&

val < trav.left.data

LR Imbalance

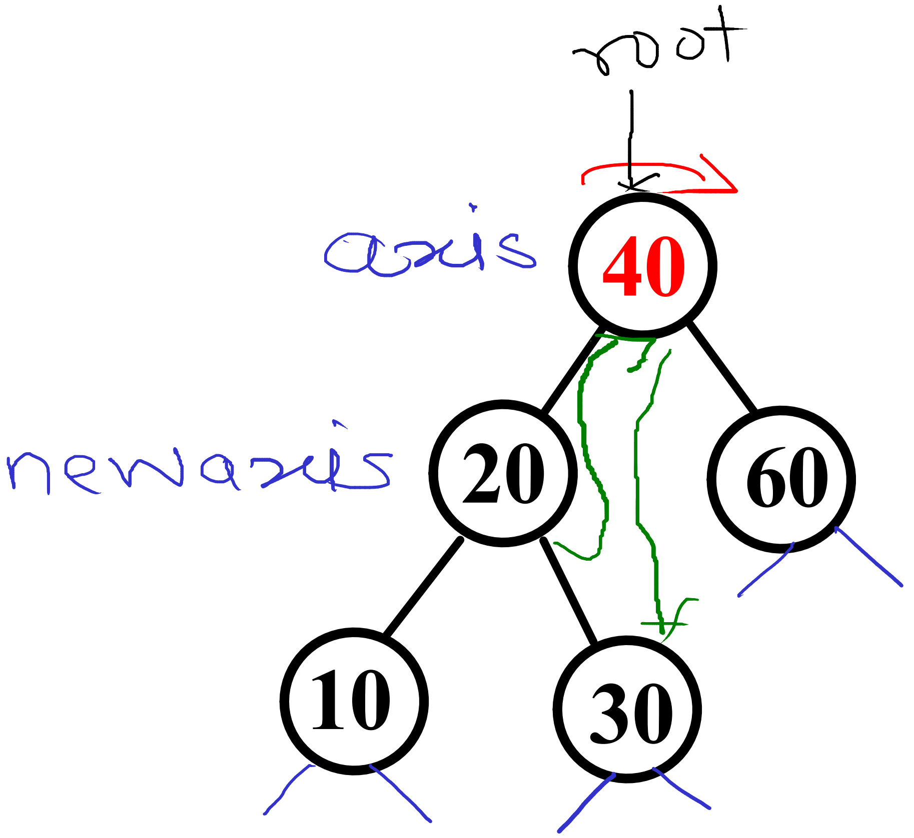
Keys : 30, 10, 20

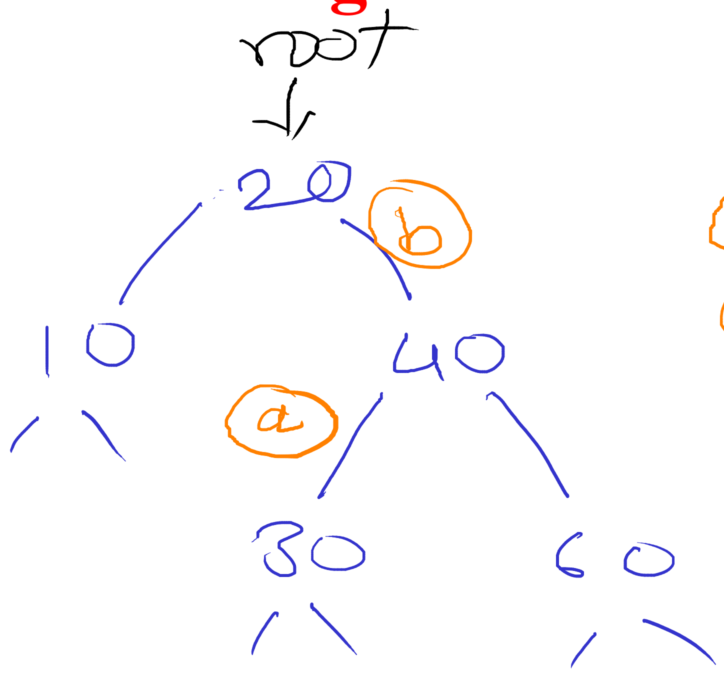


bf > 1 &&

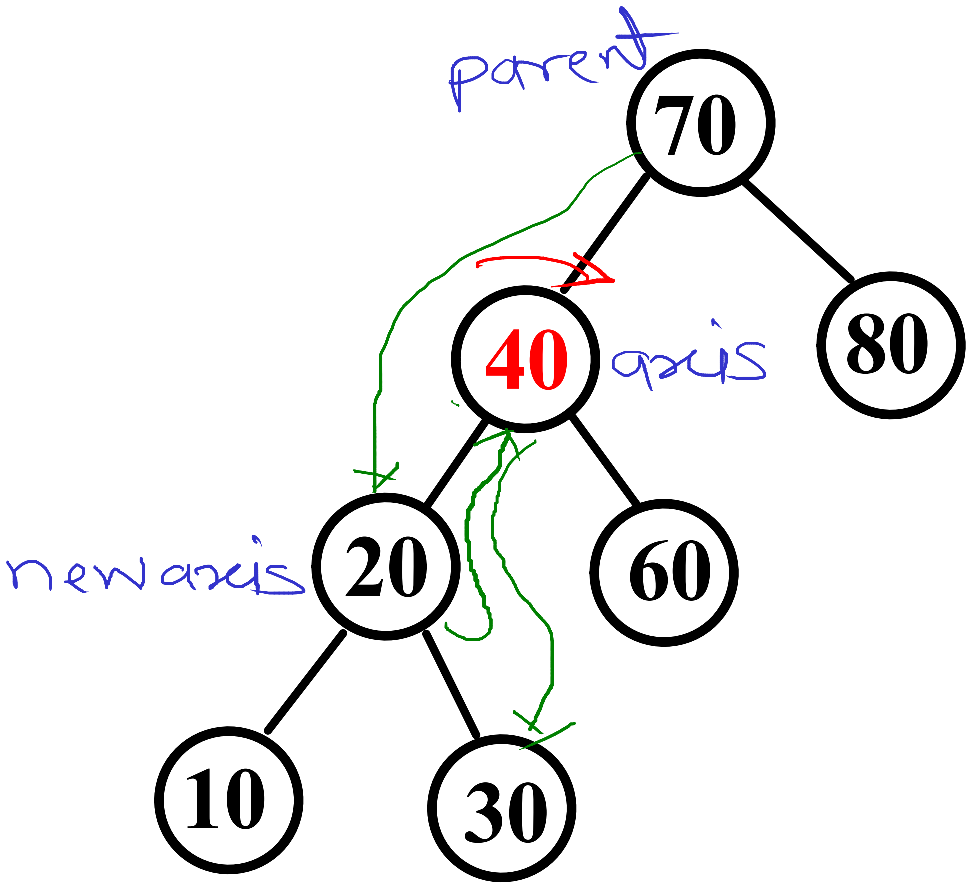
val > trav.left.data

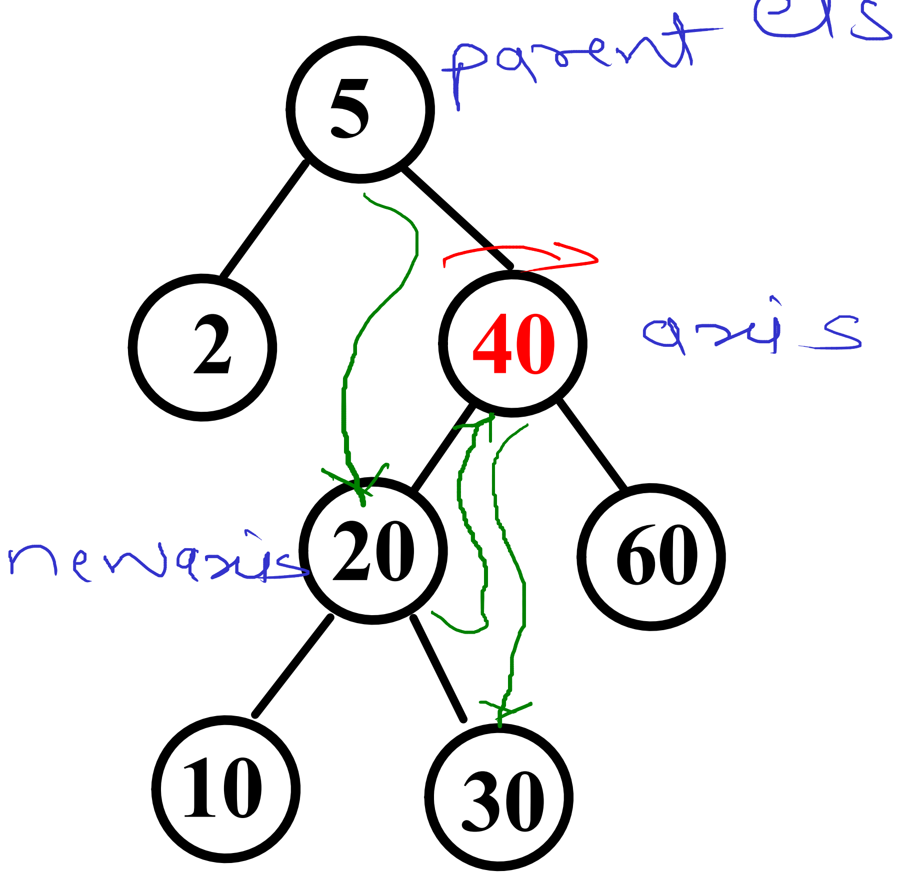
Right Rotation



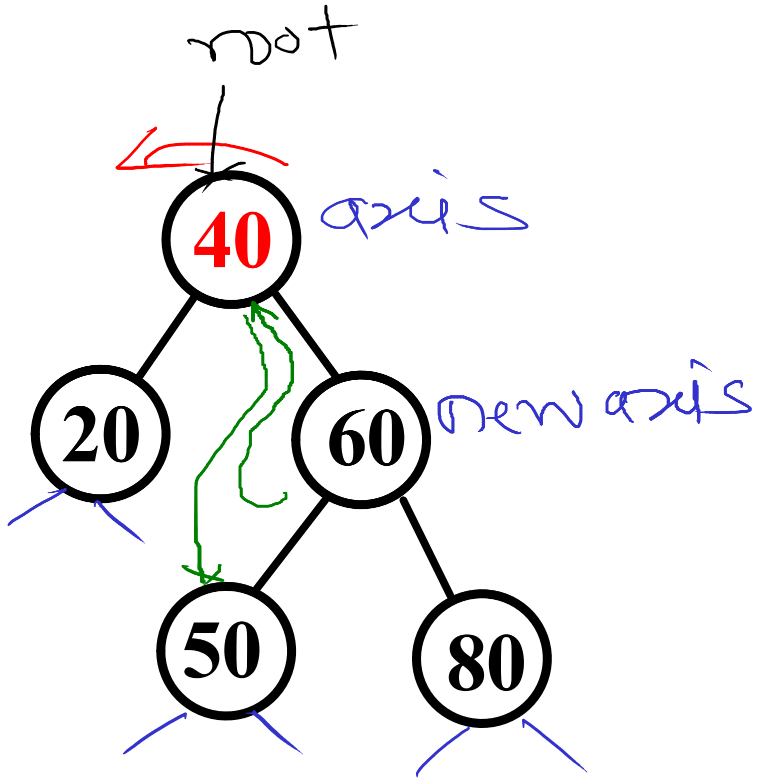


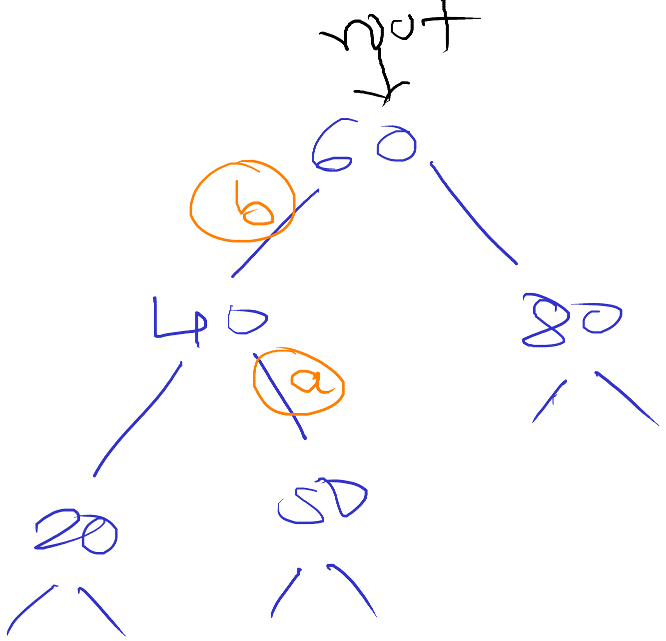

```
newaxis=axis.left;
(a) axis.left=newaxis.right;
(b) newaxis.right=axis;
if(axis==root)
    root=newaxis;
else if(axis==parent.left)
    parent.left=newaxis;
else if(axis==parent.right)
    parent.right=newaxis;
```



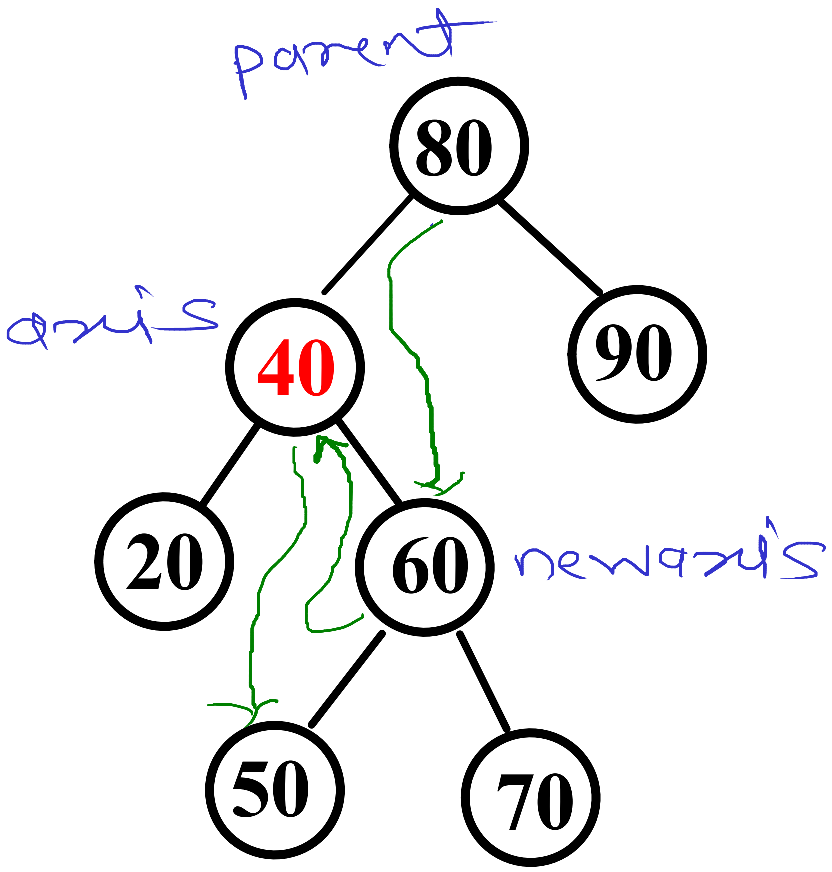


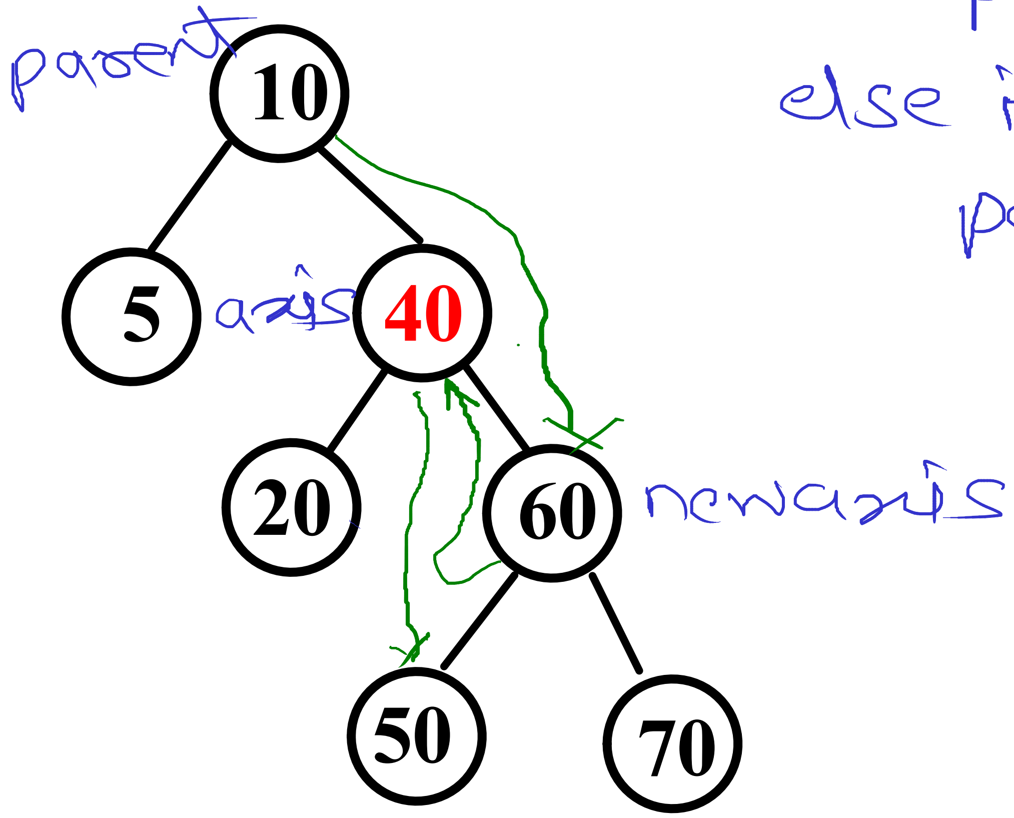
Left Rotation






```
newaxis = axis.right
(a) axis.right = newaxis.left
(b) newaxis.left = axis
if(axis == root)
    root = newaxis;
else if(axis == parent.left)
    parent.left = newaxis;
else if(axis == parent.right)
    parent.right = newaxis;
```

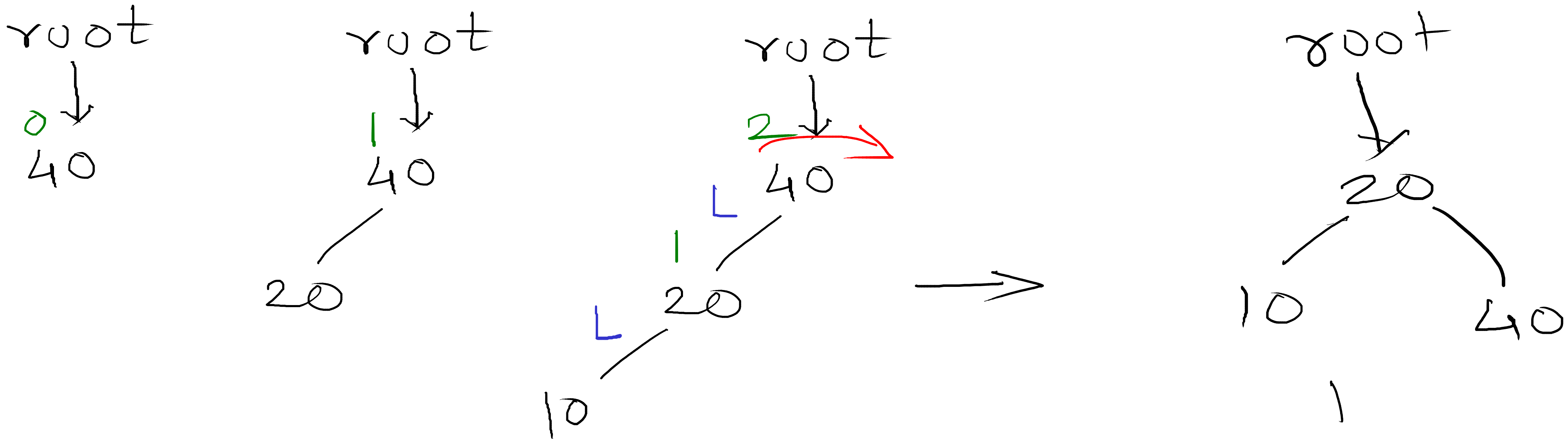


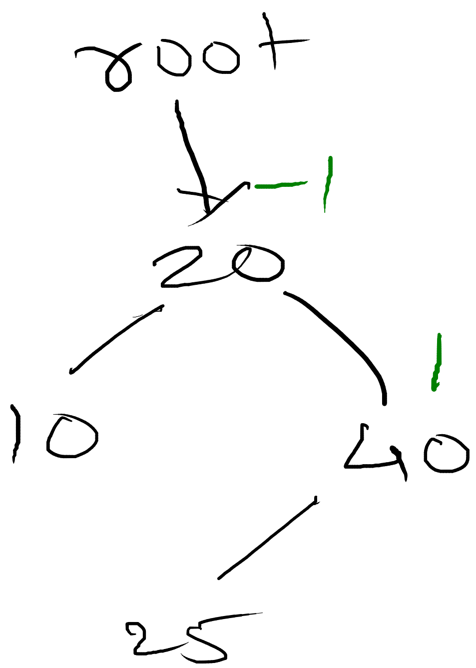


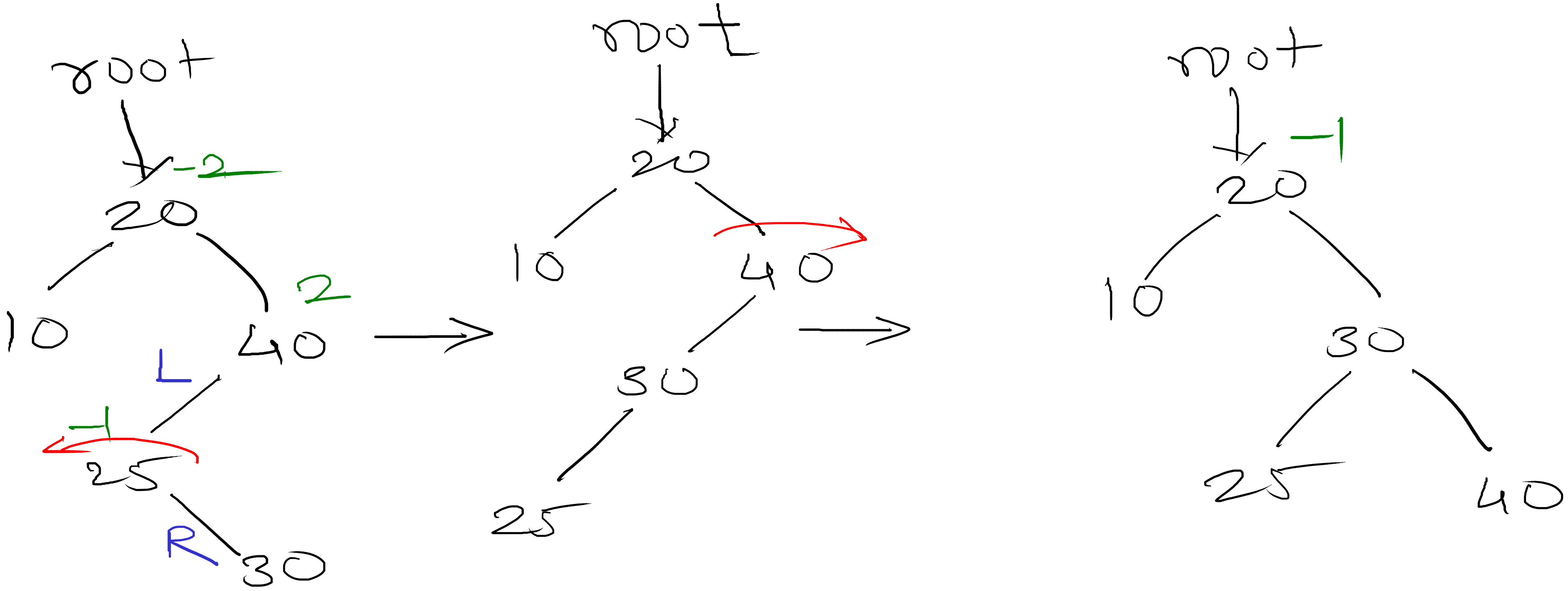
AVL Tree

- 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

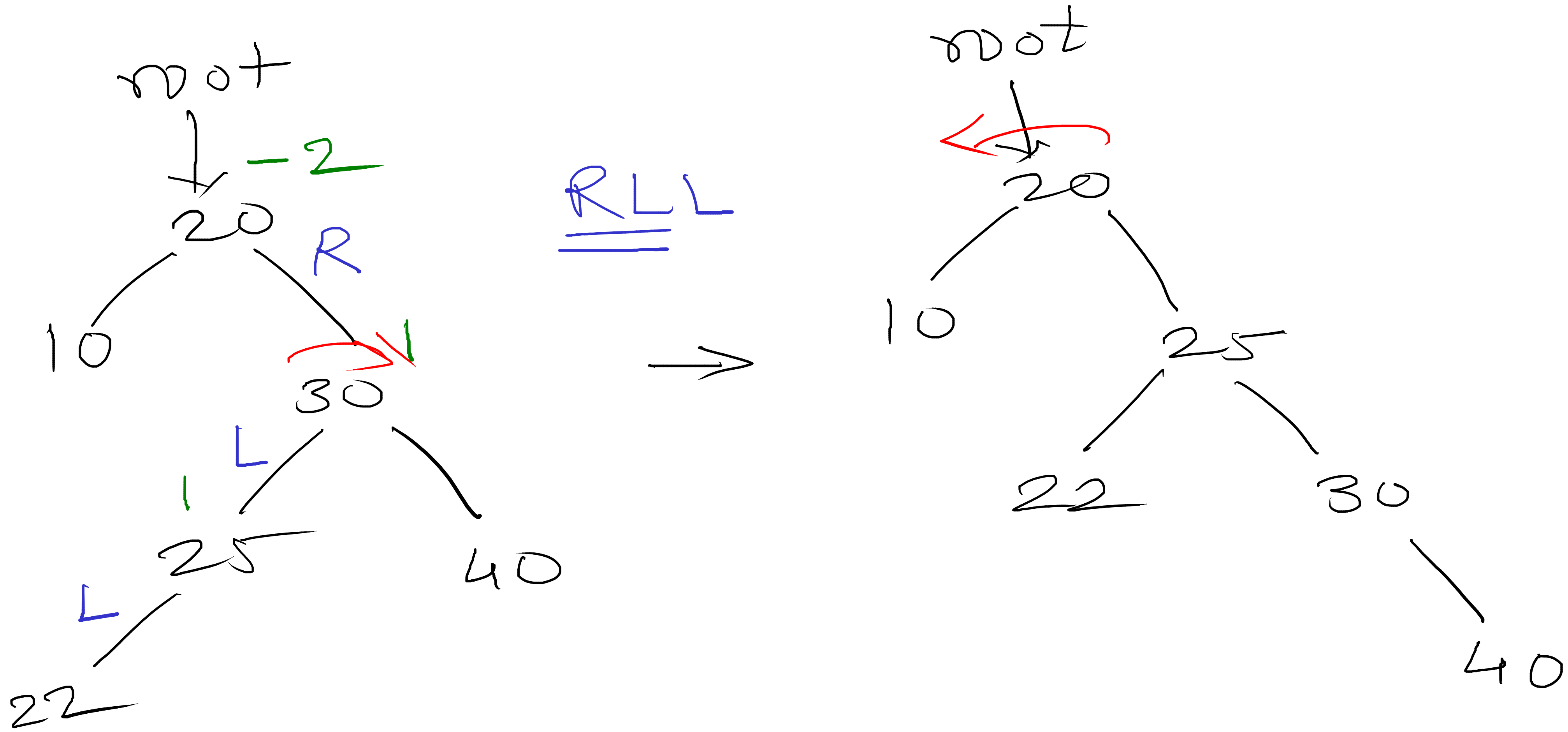
Keys : 40, 20, 10, 25, 30, 22, 50



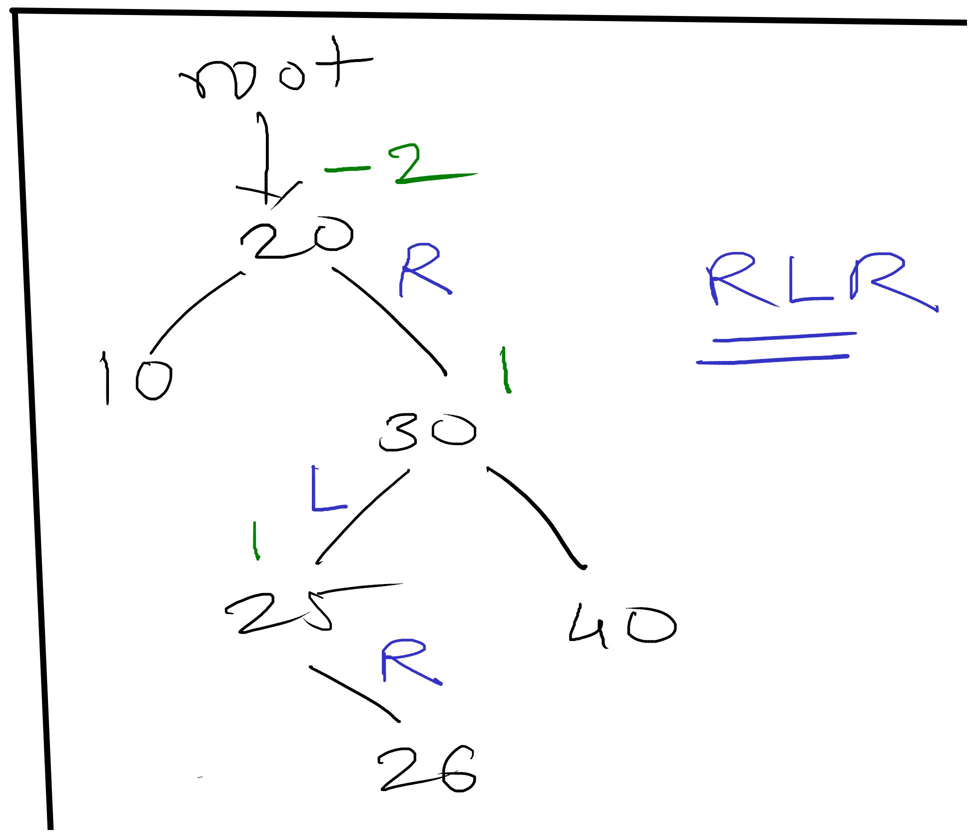




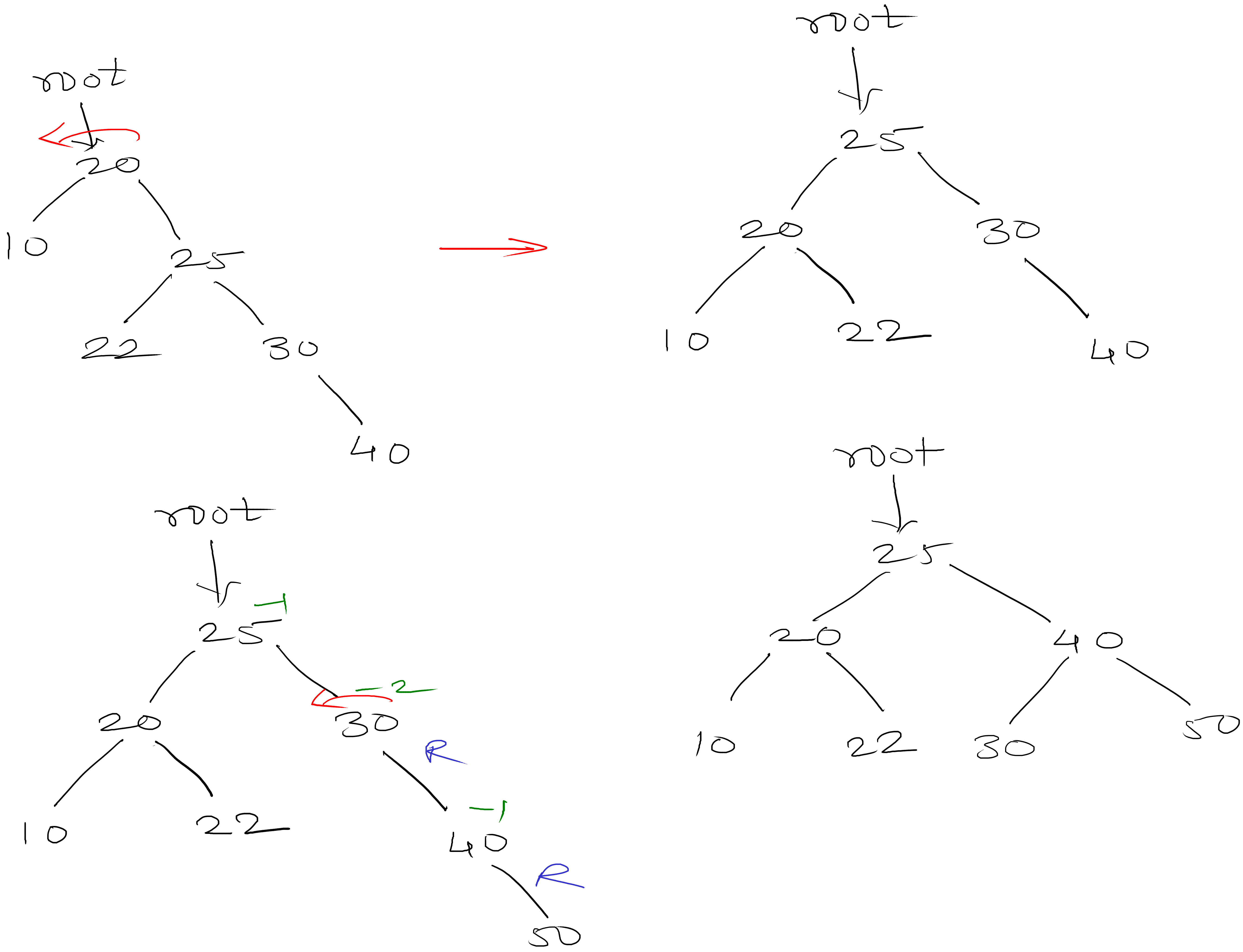
root
-2
20
10
40
2
L
-1
25
R
30
root
20
10
40
30
25
root
-1
20
10
30
25
40



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



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



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