AVL Tree

AVL(Adelson -Velskii-Landis) Tree

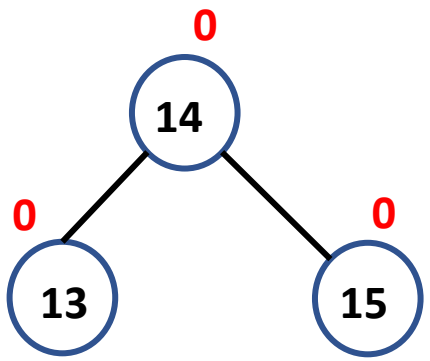
Definition

- Every AVL tree is a Binary Search tree
- AVL trees are height balanced trees i.e., the internal nodes in AVL trees are such that the height of left subtree and height of the right subtree differ by at most 1.

Balance factor of a node = (height of its left subtree) – (height of its right subtree)

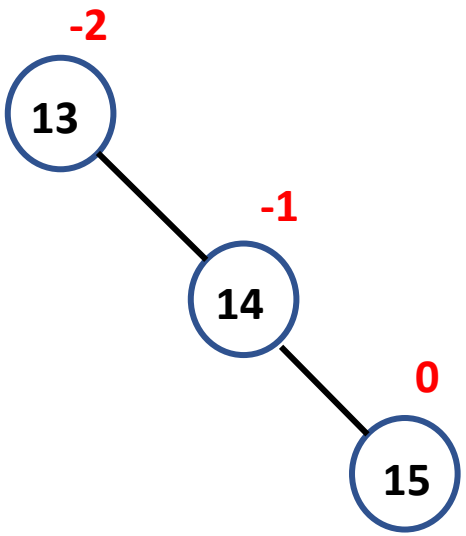
Balance factor of a node 14 = ht. of Left Sub tree – ht. of right sub tree

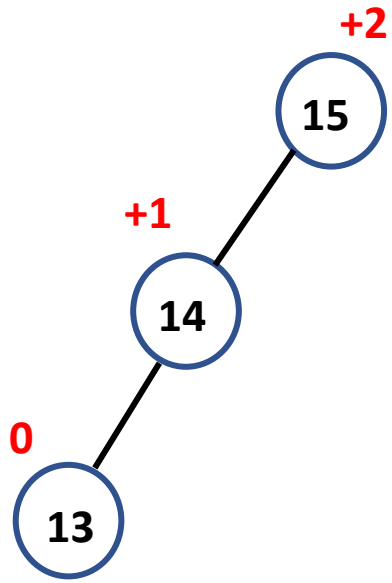
1 – 1 = 0



AVL tree

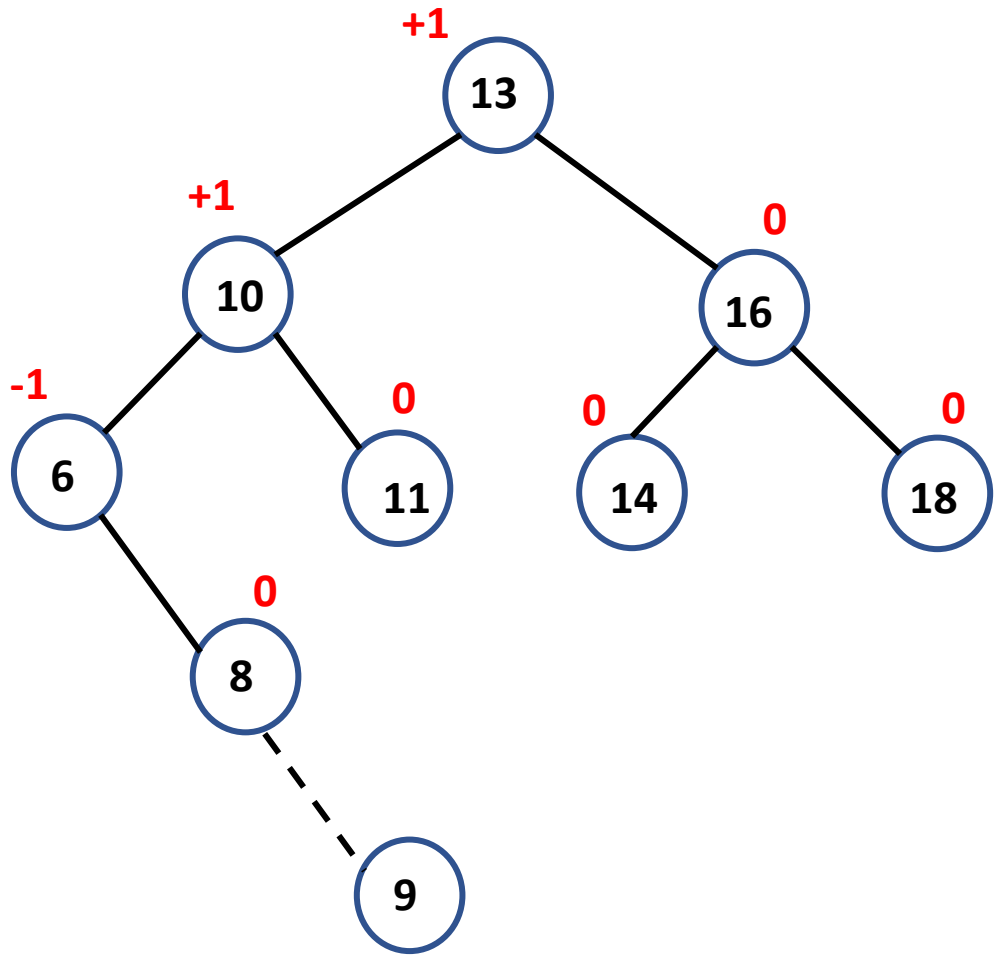
height of the leaf node is 1

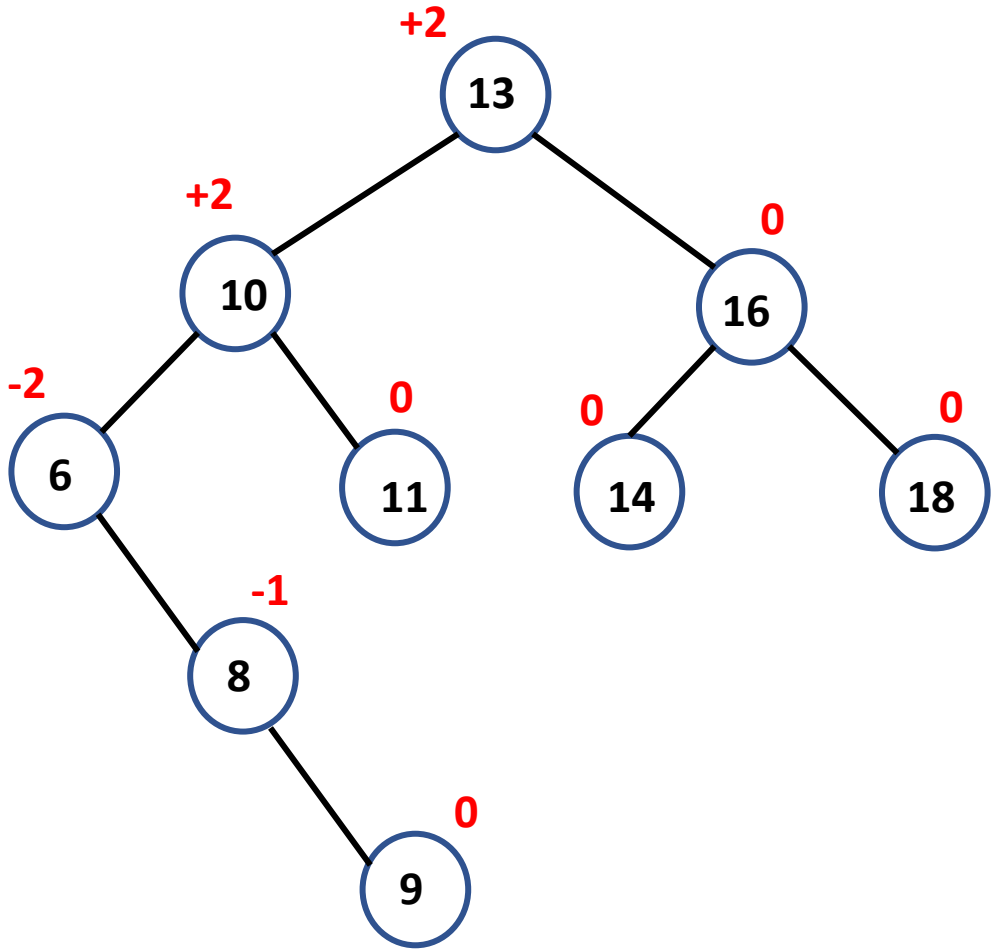




not AVL tree

Example for AVL and non- AVL trees



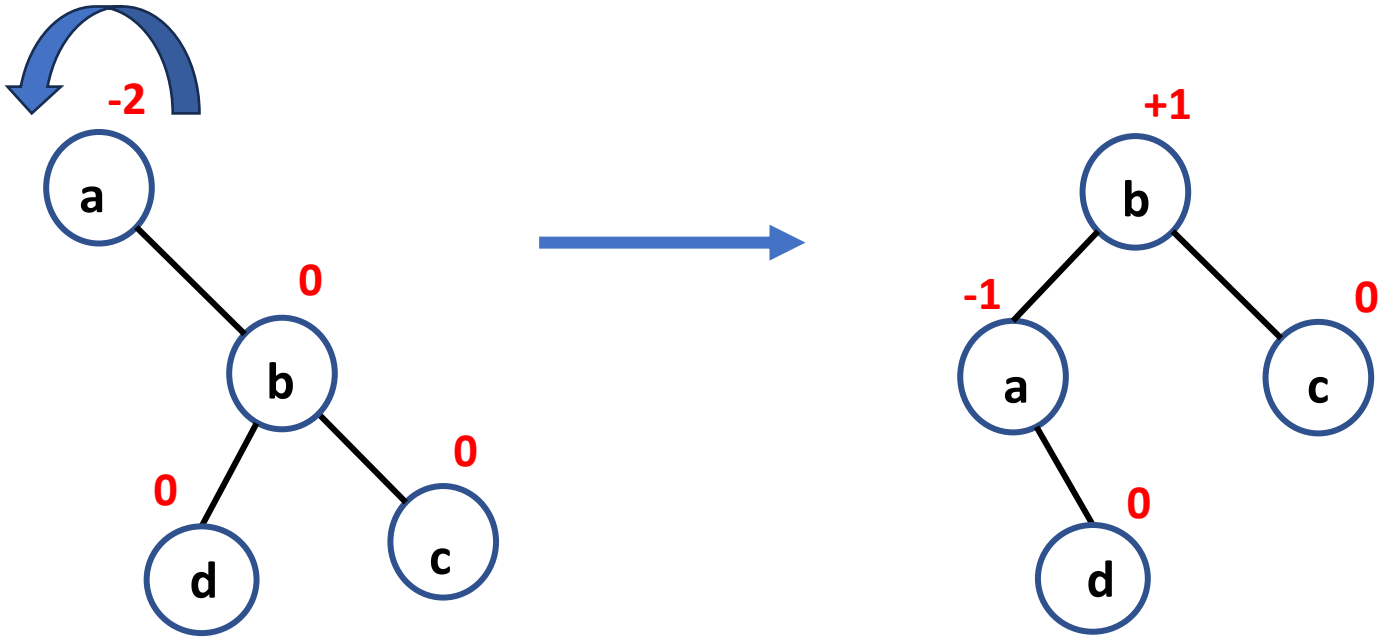


Use Tree rotations to make the tree balanced

AVL – Tree Rotations

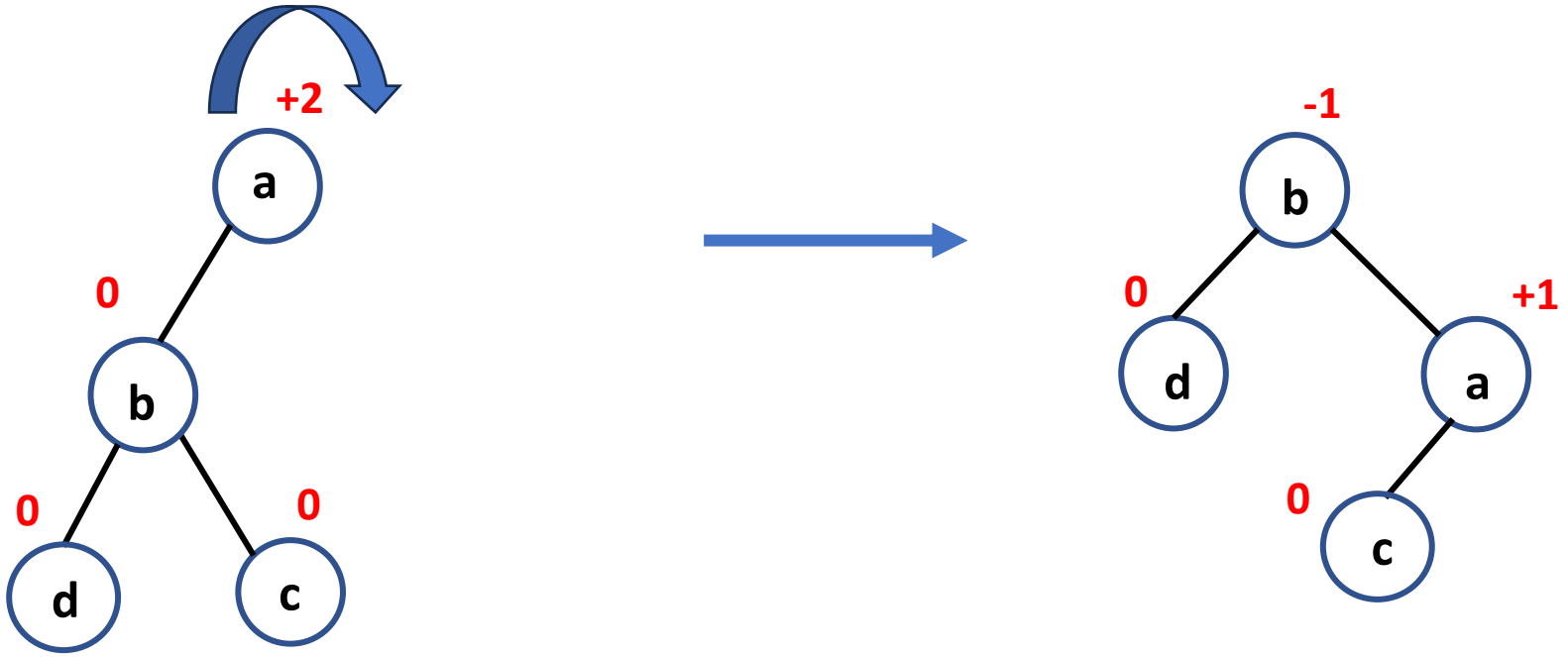
- Rotation can be 4 types
  - Left Rotation(LL)
  - Right Rotation (RR)
  - Left-Right Rotation (LR) or Double Left
  - Right-Left Rotation (RL) or Double right

Left Rotation (LL)



- Node b becomes the new root
- Node ‘a’ takes ownership of b’s left child as its right child(if exists)
- Node ‘b’ takes ownership and Node ‘a’ as its left child.

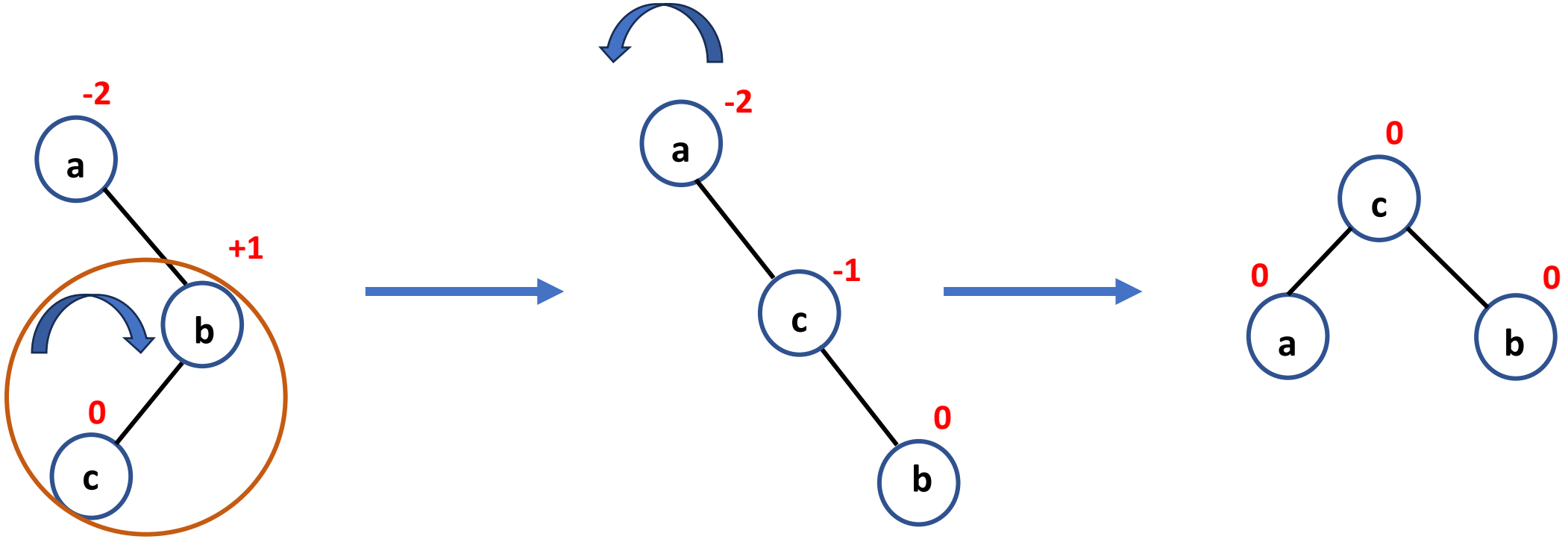
Right Rotation (RR)



- Node b becomes the new root
- Node ‘a’ takes ownership of b’s right child as its left child(if exists)
- Node ‘b’ takes ownership and Node ‘a’ as its right child.

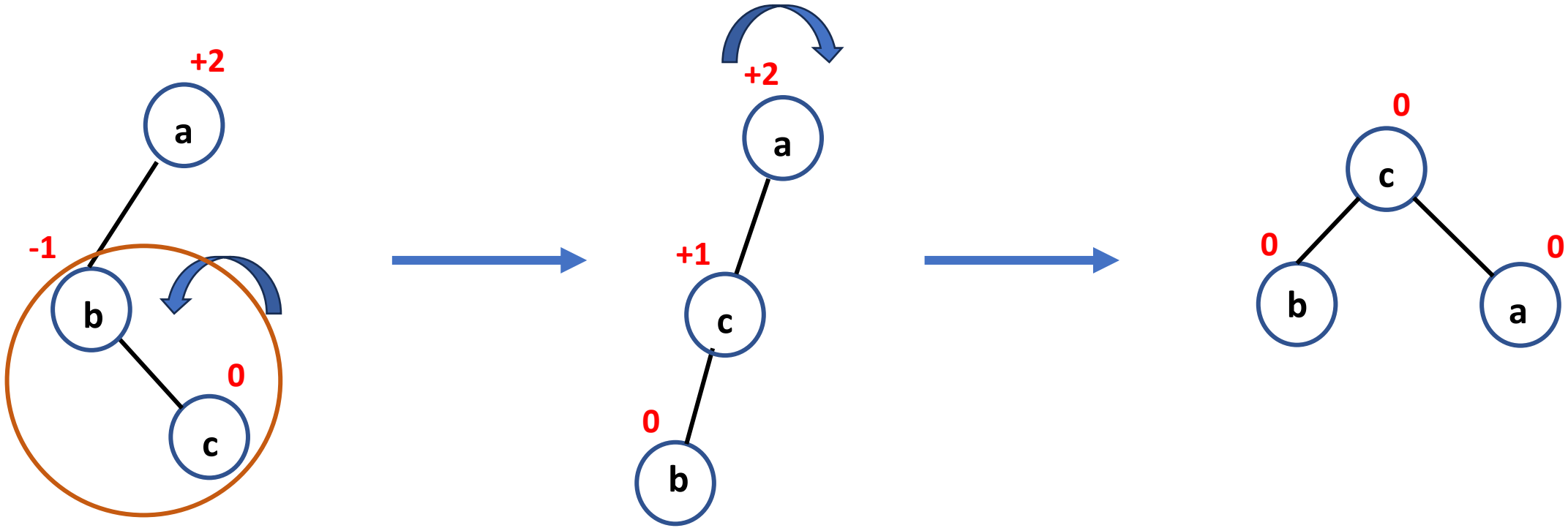
Left – Right Rotation (LR) or Double left

- Perform right rotation on right subtree, and
- Perform left rotation.



Right – Left Rotation (RL) or “Double right”

- Perform Left rotation on left subtree of the root.
- Perform right rotation.



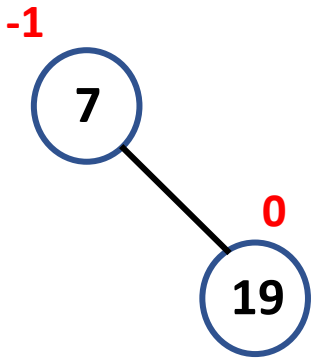
Rules to Identify the kind of rotation

If tree is right heavy
{
    if **tree's right subtree** is **left heavy**
        perform double left rotation (LR)
    else
        perform single left rotation (LL)
}
else if tree is left heavy
{
    if **tree's left subtree** is **right heavy**
        perform double right rotation (RL)
    else
        perform single right rotation (RR)
}

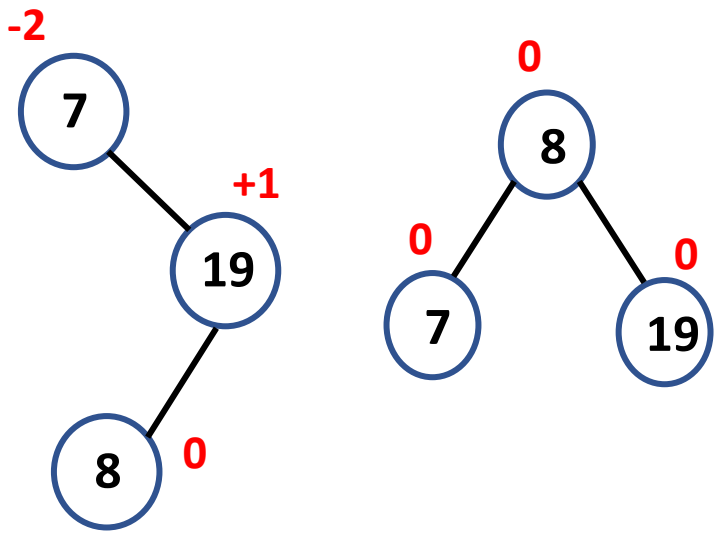
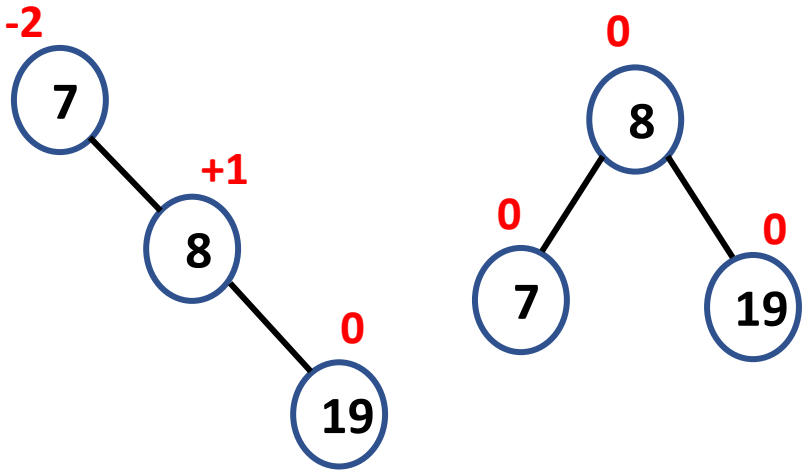
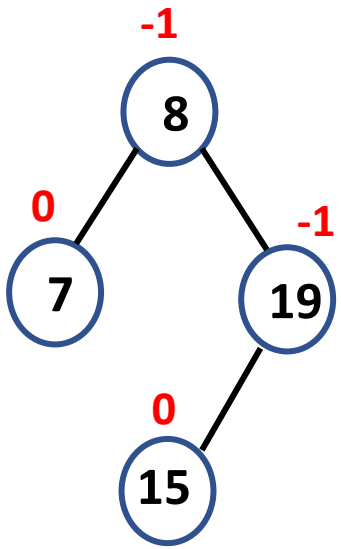
Creation of AVL tree

Create an AVL tree for the elements 7, 19, 8, 15, 23, 59, 45, 4, 2, 10

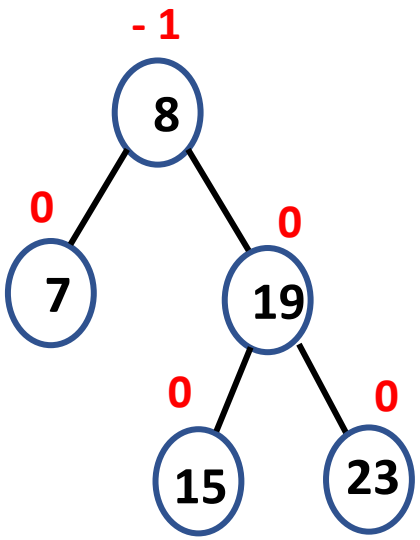
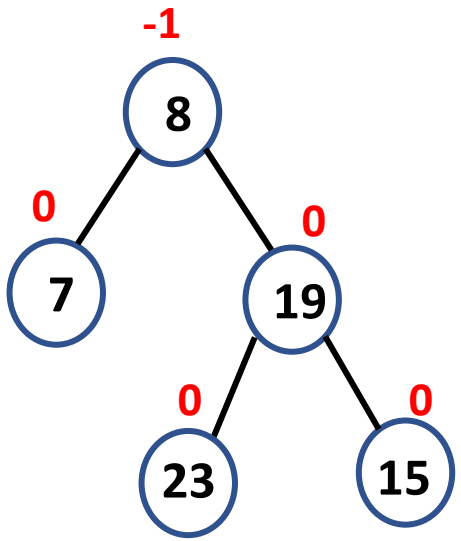
Insertion element	**AVL tree**	**Balanced or not**
Insert 7		Balanced

Insert 19		Balanced

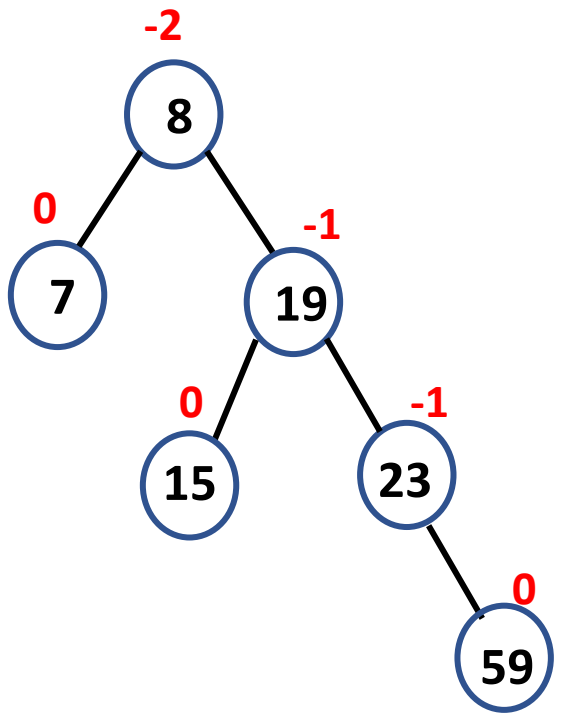
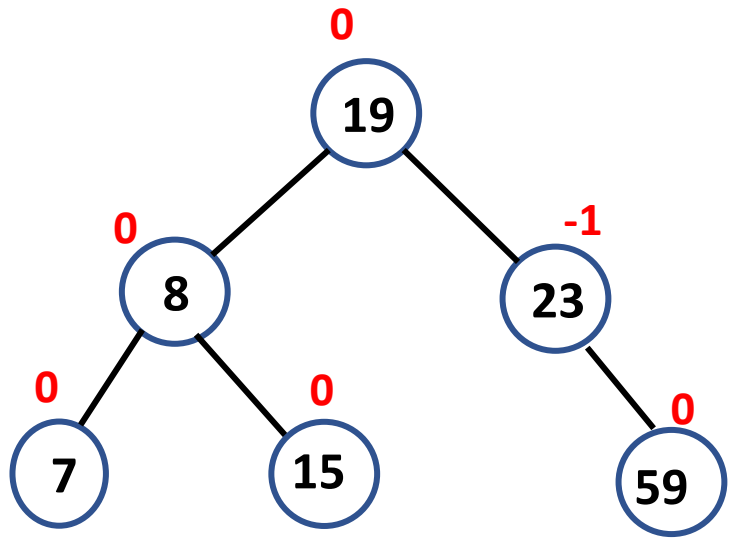
7, 19, 8, 15, 23, 59, 45, 4, 2, 10

Insert 8		Unbalanced. Perform rotation(LR) 
Insert 15		Balanced

7, 19, 8, 15, 23, 59, 45, 4, 2, 10

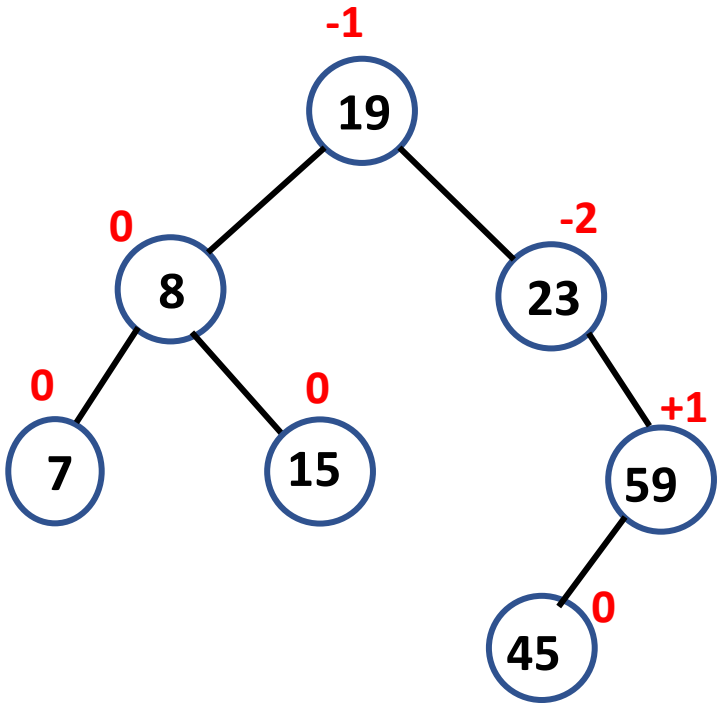
Insert 23		balanced 

7, 19, 8, 15, 23, 59, 45, 4, 2, 10

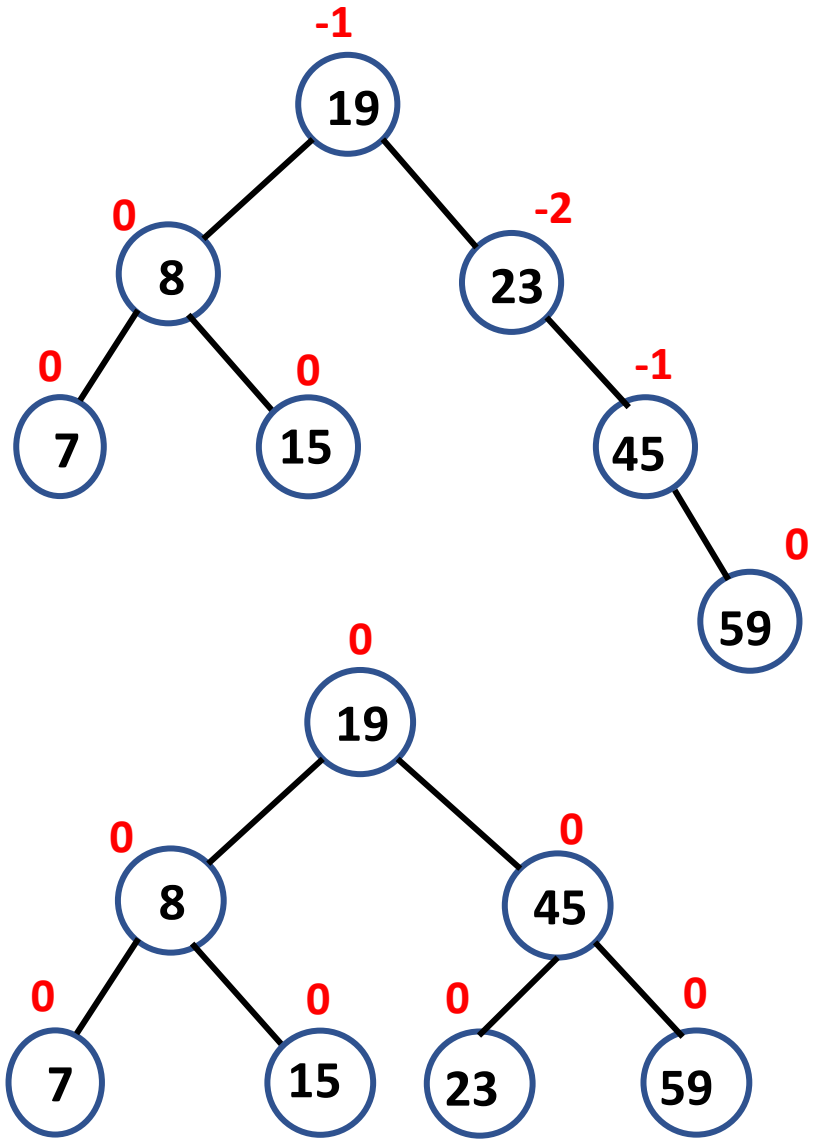
Insert 59		Unbalanced. Perform left rotation 

7, 19, 8, 15, 23, 59, 45, 4, 2, 10

Insert 45

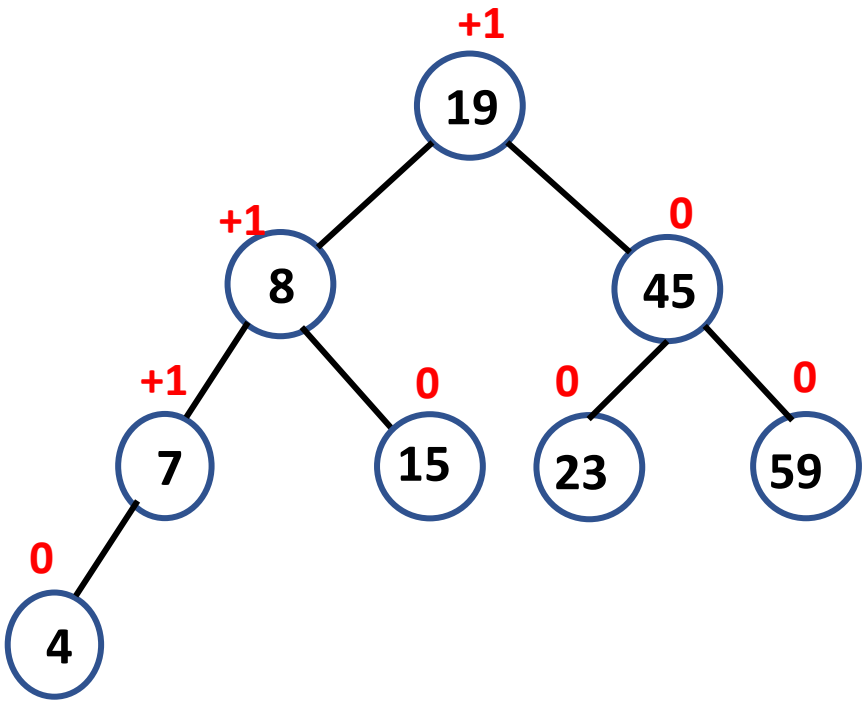


Unbalanced. Perform LR rotation



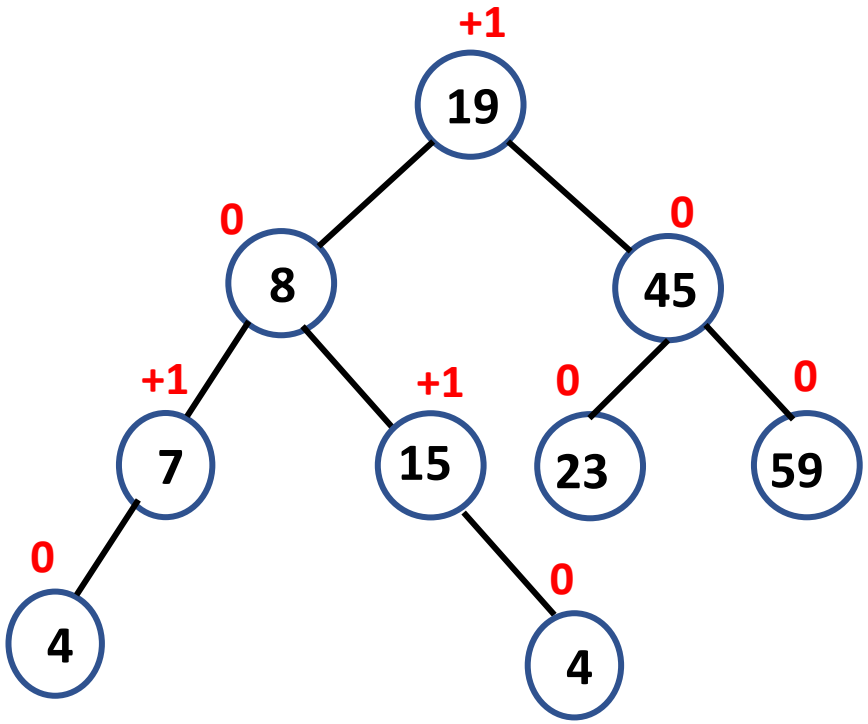
7, 19, 8, 15, 23, 59, 45, 4, 2, 10

Insert 4



balanced

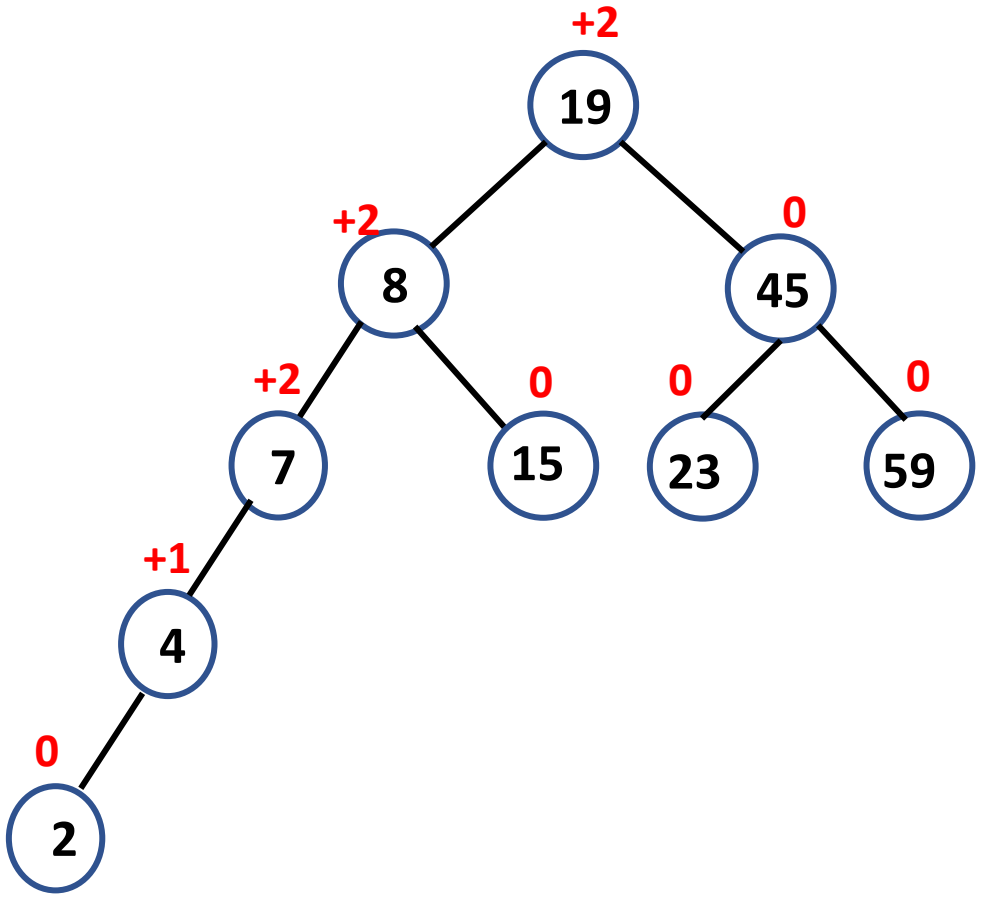
Insert 18



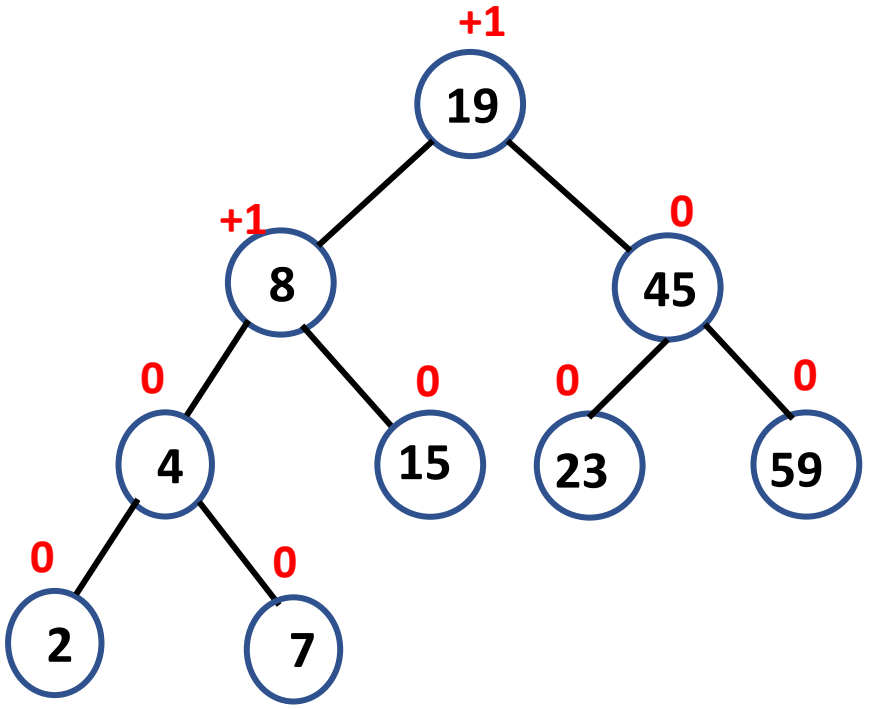
balanced

7, 19, 8, 15, 23, 59, 45, 4, 2, 10

Insert 2

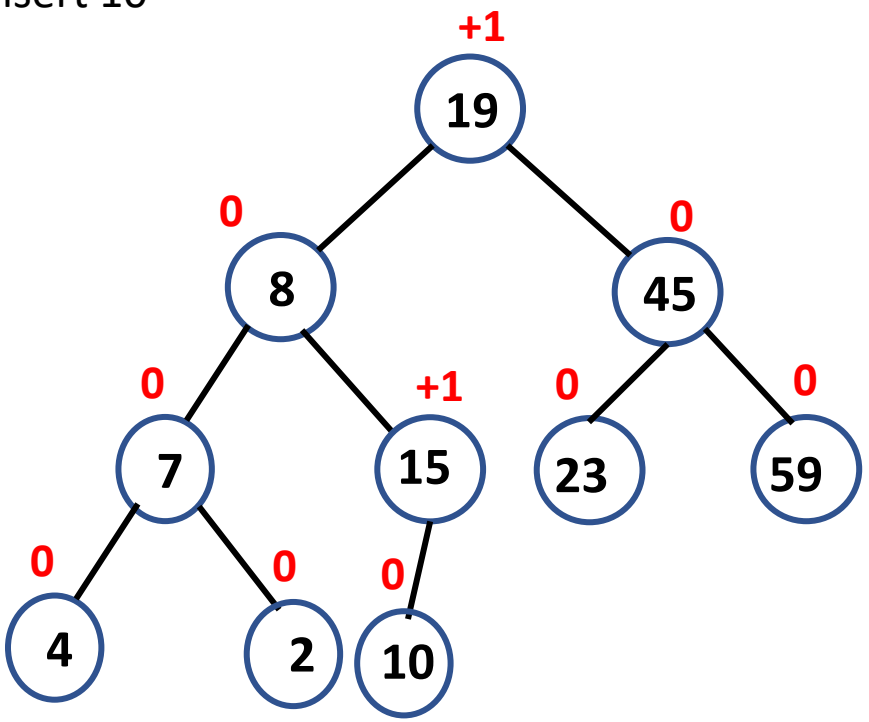


Unbalanced



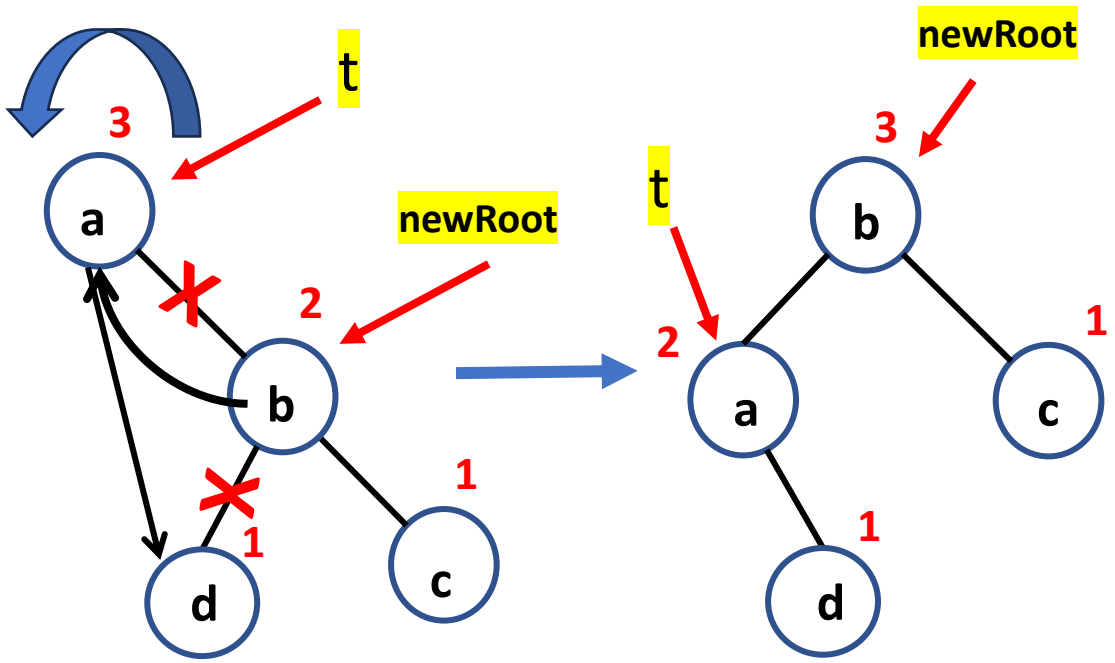
7, 19, 8, 15, 23, 59, 45, 4, 2, 10

Insert 10



balanced

Left Rotation (LL)



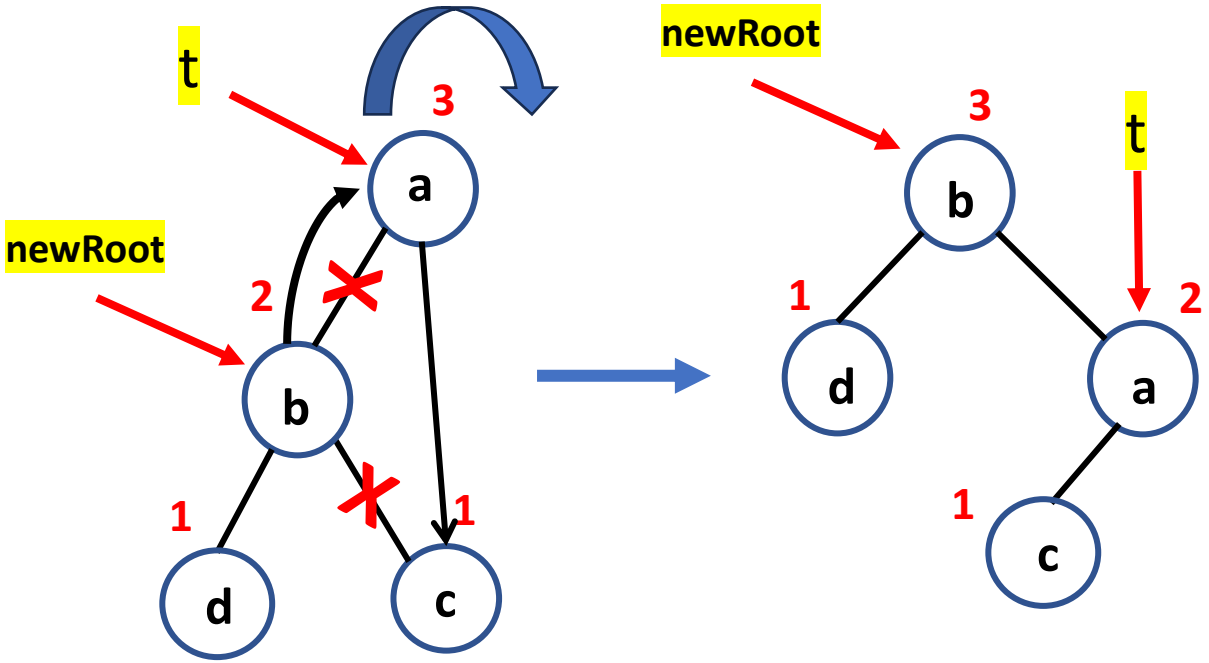
- Node b becomes the new root
- Node ‘a’ takes ownership of b’s left child as its right child(if exists)
- Node ‘b’ takes ownership and Node ‘a’ as its left child.

```
node* rotate_left(node* t){
    node* newRoot = t->right;
    t->right = t->right->left;
    newRoot->left=t;
    updateHeight(t);
    updateHeight(newRoot);
    return newRoot;
}
```

```
void updateHeight(node* t){
    int tl=getHeight(t->left);
    int th=getHeight(t->right);
    t->height = max(tl,th)+1;
}
```

```
int max(int x,int y){
    return x>y?x:y;
}
```

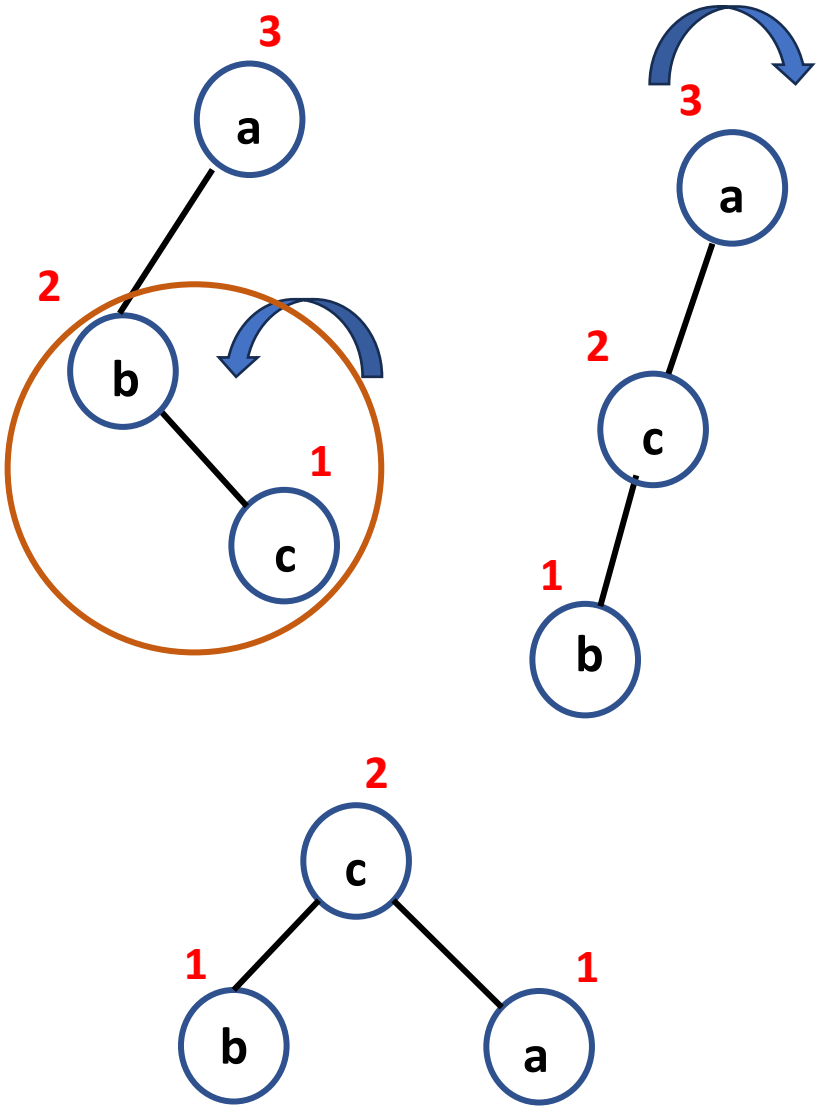
Right Rotation (RR)




```
node* rotate_right(node* t){
    node* newRoot = t->left;
    t->left = t->left->right;
    newRoot->right=t;
    updateHeight(t);
    updateHeight(newRoot);
    return newRoot;
}
```

- Node b becomes the new root
- Node ‘a’ takes ownership of b’s right child as its left child(if exists)
- Node ‘b’ takes ownership and Node ‘a’ as its right child.

Right -Left Rotation (RL) or Double right




```
node* rebalance_left(node* t){
    if(getHeight(t->right) - getHeight(t->left) == -2)
    {
       if(getHeight(t->left->right)> getHeight(t->left->left))
       {
            // double right
            t->left=rotate_left(t->left);
            t=rotate_right(t);
        }
        else
            t=rotate_right(t); // single right
    }
    else
        updateHeight(t); //no rotation is required
    return t;
}
```

Time complexity Analysis

```
node* rebalance_left(node* t){
    if(getHeight(t->right) - getHeight(t->left) == -2)
    {
       if(getHeight(t->left->right)> getHeight(t->left->left))
       {
            // double right
            t->left=rotate_left(t->left);
            t=rotate_right(t);
        }
        else
            t=rotate_right(t); // single right
    }
    else
        updateHeight(t); //no rotation is required
    return t;
}
```

O(1)

O(1)

O(1)

O(1)

O(1)

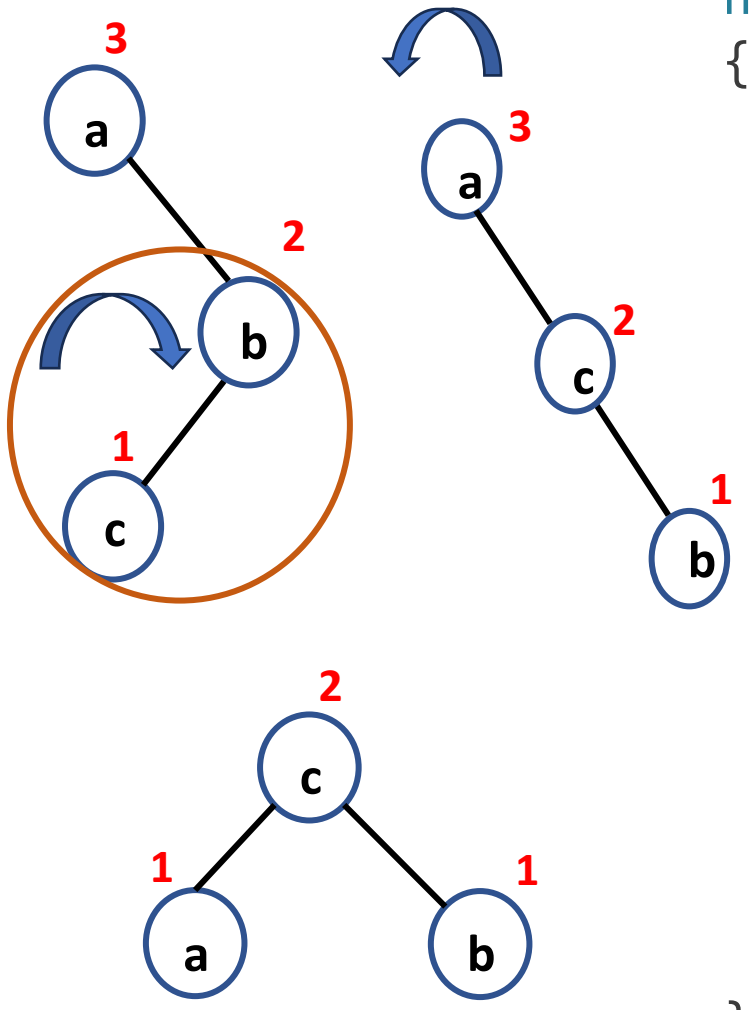
O(1)

O(1)

Time complexity = O(1) - Constant

max(O(1), max(O(1)+O(1),O(1)) =O(1)

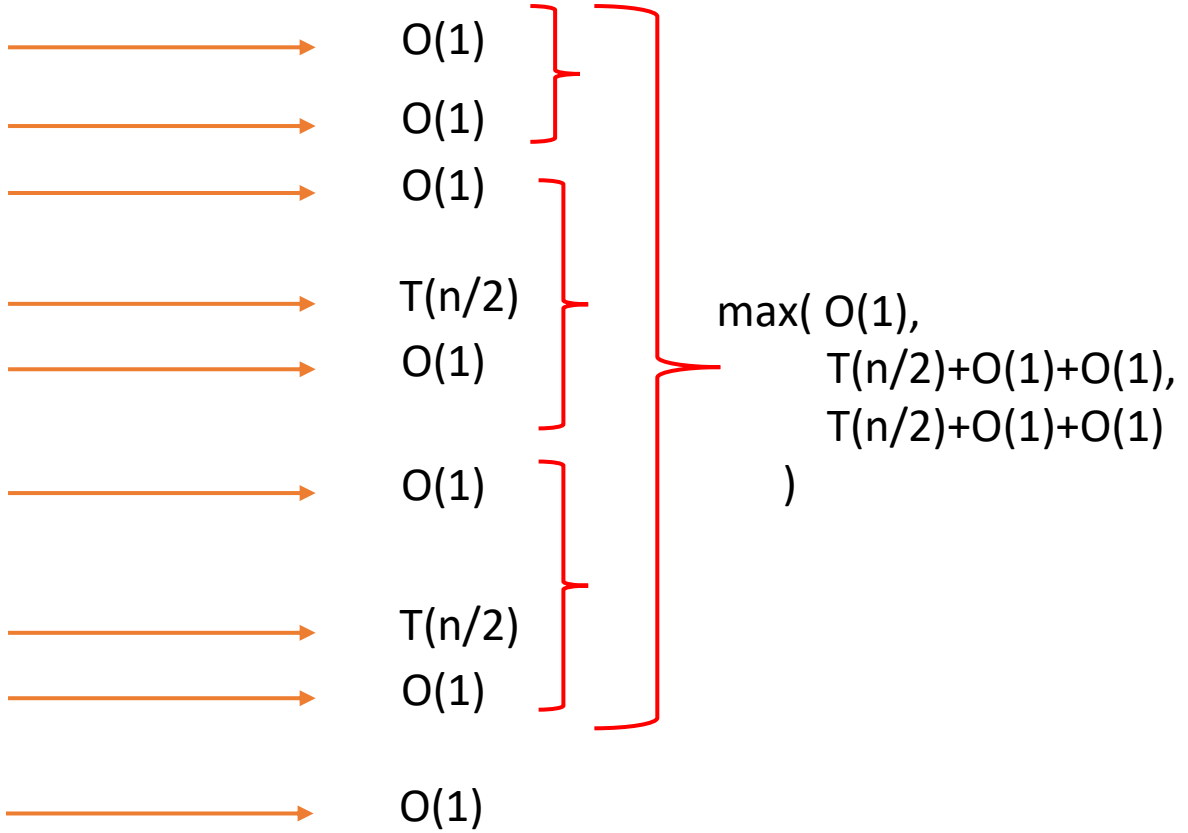
Left – Right Rotation (LR) or Double left




```
node* rebalance_right(node* t)
{
    if( Height(t->right) - Height(t->left) == 2){
        if(Height(t->right->left) > Height(t->right->right))
        {
            //Double left
            t->right=rotate_right(t->right);
            t=rotate_left(t);
        }
        else
            //Single left
            t=rotate_left(t);
    }
    else
        updateHeight(t); //no rotation is required
    return t;
}
```

AVL-Insert

```
node* avl_insert(node* t,int ele){
    if(t==NULL)
        return new node(ele);
    else if(ele<t->data)
    {
        t->left=avl_insert(t->left,ele);
        t=rebalance_left(t);
    }
    else if(ele>t->data)
    {
        t->right=avl_insert(t->right,ele);
        t=rebalance_right(t);
    }
    return t;
}
```



Time Complexity : Θ(log n)

max(O(1),T(n/2)+c, T(n/2)+c)) + O(1)

Analysis of Algorithm

- Summation method
- Recurrence relation
  - Substitution method
  - Recursion tree
  - Master’s theorem (Used for only divide-and-conquer strategy)

Summation Method

Analysis of Bubble sort

```
for(int i=1; i <= n-1; i++)
{
        for(int j=0; j < n-i; j++)
        {
            if(a[j+1] > a[j])
                swap(a[j+1],a[j])
        }
}
```

For pass 1 – inner loop executes n-1 times
For pass 2 – inner loop executes n-2 times
.
.
.
For pass n-1 – inner loop executes 1 time

$= n-1 + n-2 + n-3 + \ldots\ldots + 1$

$= ((n-1) * n)/ 2$ [formula : sum of n natural no.]

$= n^2/2 - n/2$

$<= c * n^2$

$= O(n^2)$

Recurrence Relation

$$T(n) = T(n-1) + c \quad \text{if } n \geq 1$$
$$= c \quad \text{if } n = 0$$

$$T(n) = T(n-1) + T(n-2) + c \quad \text{if } n > 1$$
$$= c \quad \text{if } n = 1$$
$$= c \quad \text{if } n = 0$$

```
int fact(int n)
{
      if ( n == 0 )
         return 1;
      else
         return n * fact(n-1);
}
```

```
int fib(int n)
{
      if ( n==0 || n==1)
         return n;
      else
         return fib(n-1)+fib(n-2);
}
```

AVL- Delete

```
node* avl_delete(node* t,int ele)
{
    if(t==NULL)
        return t;
    else if(ele<t->data)
    {
        t->left=avl_delete(t->left,ele);
        t=rebalance_left(t);
    }
    else if(ele>t->data)
    {
        t->right=avl_delete(t->right,ele);
        t=rebalance_right(t);
    }
    else
    {
        //node to be deleted is leaf node
        if(t->left==NULL && t->right==NULL)
            t=NULL;
        // node to be deleted has one child
        else if(t->left==NULL || t->right==NULL )
        {
            if(t->left==NULL)
                *t=*(t->right);
            else
                *t=*(t->left);
        }
        // node to be deleted has both left and right
        else
        {
            node* successor=find_minimum(t->right);
            t->data=successor->data;
            t->right=avl_delete(t->right,successor->da
            t=rebalance_right(t);
        }
    }
    return t;
}
```

AVL- Search

```
int avl_search(node* t,int key){
    if(t==NULL)
        return -1;
    else if(key == t->data)
        return 1;
    else if(key < t->data)
        return avl_search(t->left,key);
    else
        return avl_search(t->right,key);
}
```

Advantages and Disadvantages of AVL Trees

Advantages:

1.Search is O(log n) since AVL trees are always balanced.
2.Insertion and deletions are also O(log n)
3.The height balancing adds no more than a constant factor to the speed of insertion.

Dis Advantages:

- Difficult to program & debug; more space for balance factor.
- Asymptotically faster but rebalancing costs time.
- Most STL implementations of the ordered associative containers (sets, multisets, maps , and Multimaps) use red-black trees instead of AVL trees. Unlike AVL trees, red-black trees require only one restricting for removal or insertion.
- Most large searches are done in database systems on disk and use other structures (e.g. B-trees).

Applications

In memory sorts of sets and dictionaries

Applications

- In memory sorts of sets and dictionaries.
- Database applications in which insertions and deletions are fewer but there are frequent lookups for data required.