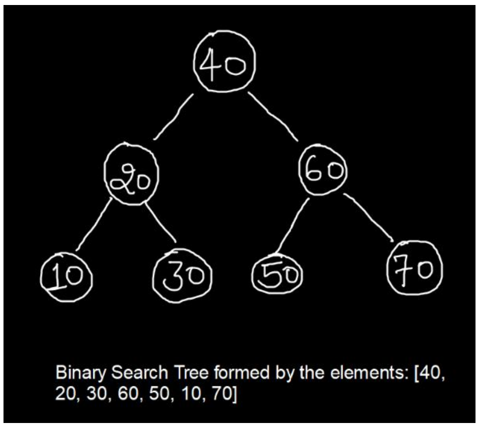
AVL Trees

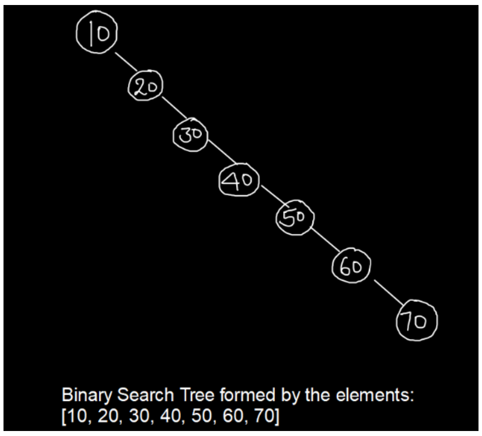
Drawbacks of BST

In a BST, all the operations such as search, minimum, maximum, insertion, deletion are performed with time complexity directly proportional to the height of BST i.e. (O(h)) and height of a BST depends on the order in which elements are fed to the algorithm. Let us comprehend the concept with help of an example.

Let the order in which elements are fed to the insertion algorithm be: [40, 20, 30, 60, 50, 10, 70]. The height of the resultant tree is of the order of O(log n).


40
20
60
10
30
50
70
Binary Search Tree formed by the elements: [40, 20, 30, 60, 50, 10, 70]

Let us take another order in which elements are fed to the algorithm: [10, 20, 30, 40, 50, 60, 70]. The height of the resultant tree is of the order O(n).


10
20
30
40
50
60
70
Binary Search Tree formed by the elements: [10, 20, 30, 40, 50, 60, 70]

This makes BST having complexities of various operations similar to that of a binary tree. If we want to leverage the advantages of BST, we have to ensure that height of BST is O(log n).

We know that in a balanced binary tree, the height of binary tree is proportional to log(n), which is the minimum possible height, so all the operations can be performed with a time complexity of O(log n). So, to improve this time complexity, need for balanced binary search trees arise. AVL trees are one example of balanced binary search trees.

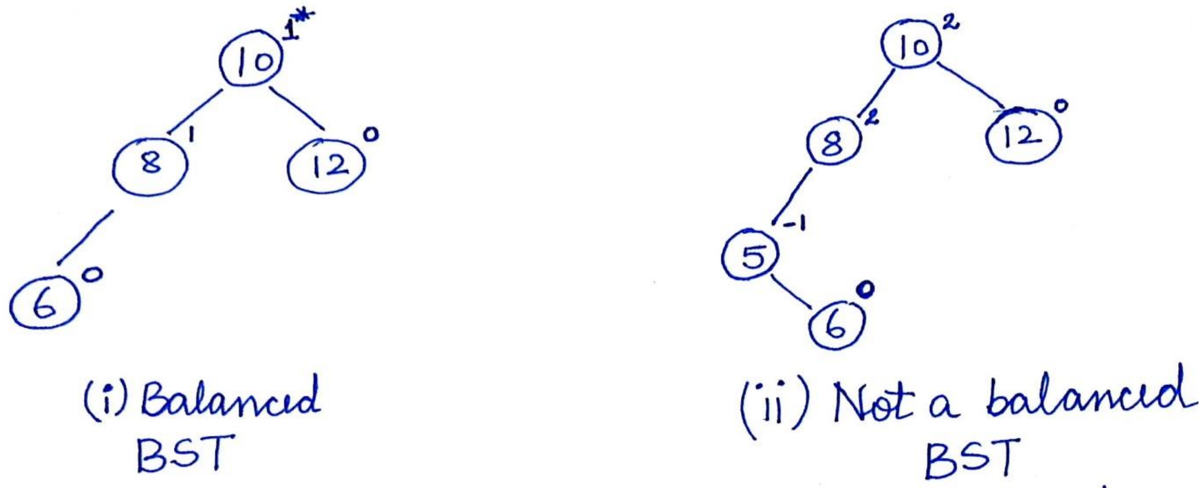
AVL Trees

AVL tree is named after their inventors, Adelson-Velskii and Landis. AVL tree checks the height of the left and the right sub-trees and assures that the difference is not more than 1. Height of an AVL tree is balanced using balance factor.

Balance factor = height of left subtree - height of right subtree

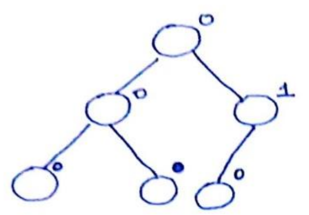
We calculate this balance factor for each node. For a height balanced node, BF=HL - HR= {-1,0,1}. |BF|=|HL - HR|<=1. Similarly, for a height imbalanced node, |BF|=|HL - HR| > 1.

For a tree to be height balanced, each node in the tree must be balanced. If any node is height imbalanced, then the whole tree becomes height imbalanced.

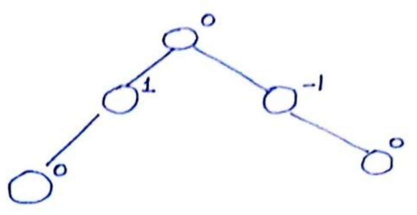


* Balance factors are written in superscript of each node.

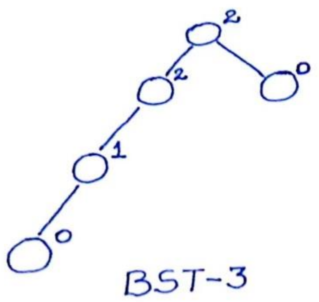
To make a node balanced, we perform rotations. Before delving deep into rotations, let us first understand balance factor through examples.



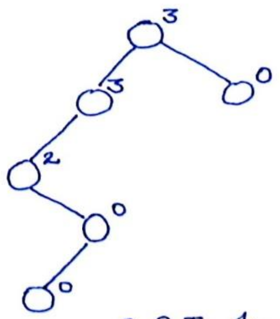
BST-1
This is balanced BST



BST-2
This is balanced BST



BST-3
This is unbalanced BST



BST-4
This is unbalanced BST

AVL rotations

Let us move to rotations now.

As we all know, whenever a new element is inserted in BST, it gets added as leaf node. It may happen that after we have inserted an element, then some nodes may become imbalanced. Due to which, the whole tree becomes imbalanced.

To make a node balanced, we must make rotations. Tree rotation is an operation that changes the structure, without interfering in the order of elements of trees and makes it balanced. There are four rotations possible.

To make the tree balanced we have 4 kind of rotations:

1. Left -left rotation
2. Left -right rotation
3. Right-right rotation
4. Right-left rotation

These rotations are done while inserting a new node in the AVL tree for rebalancing the tree.

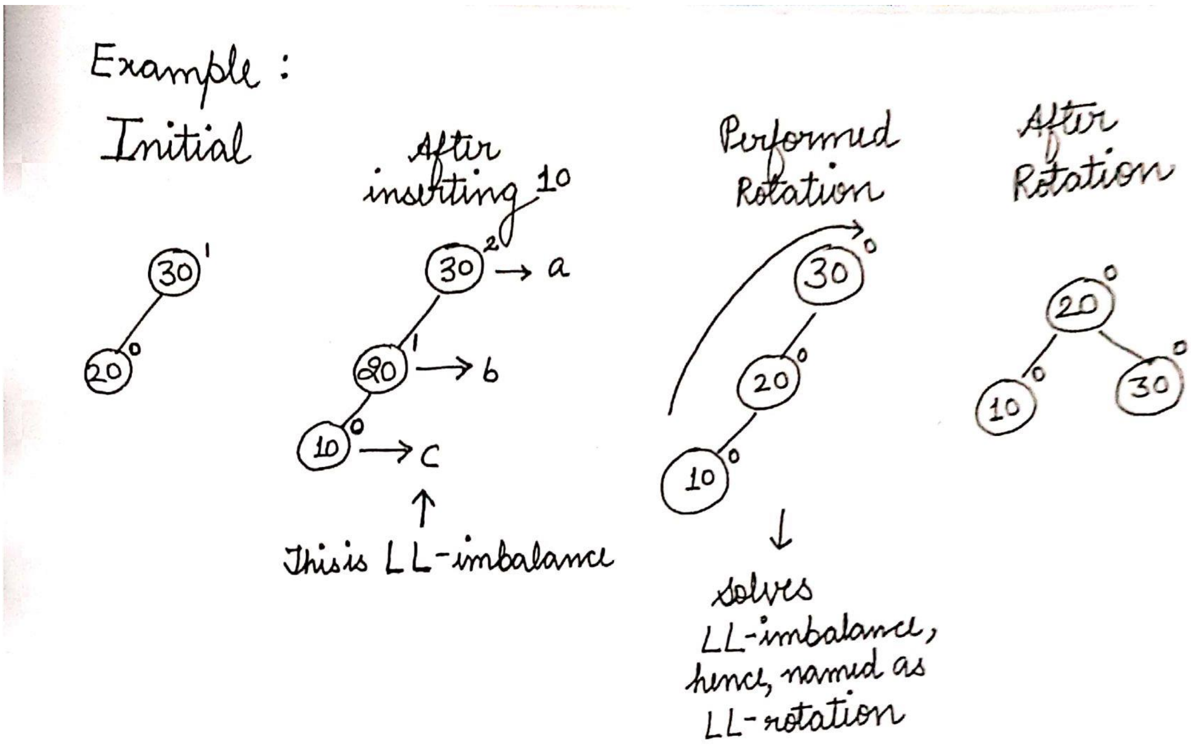
Steps to perform for insertion:

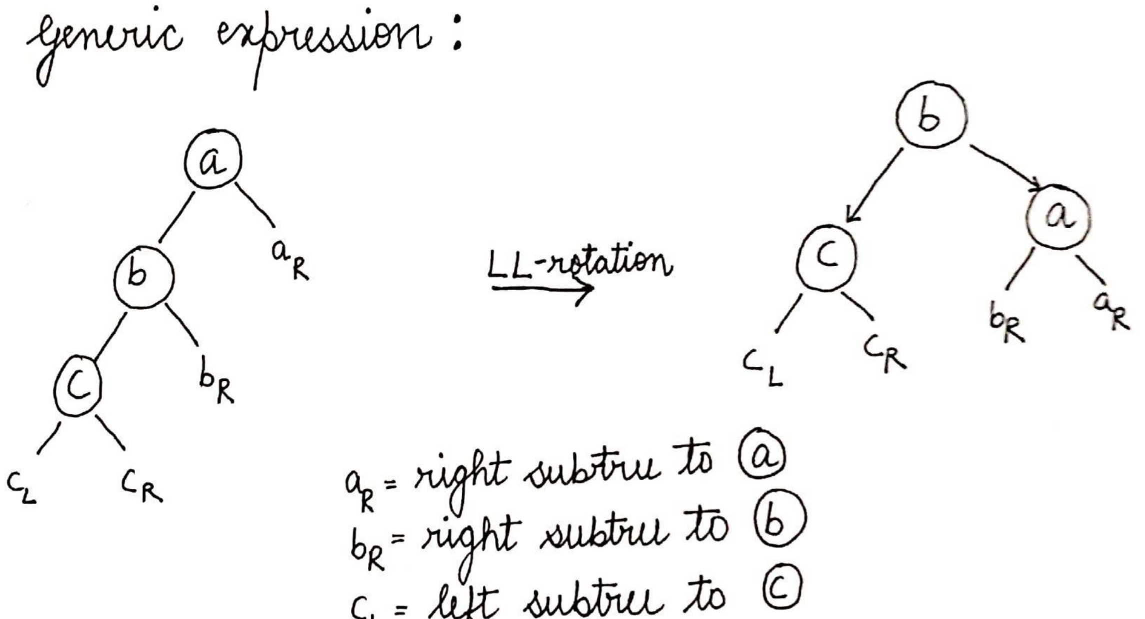
1. Add a new node in an AVL tree similar to that of BST. The new node added will always be a leaf node. By adding this new node, it is possible that we violate the balancing of tree of property.

2. Now, start traversing up the tree and find the 1st unbalanced node. Let say ***a*** be the 1st unbalanced node, ***b*** be the child of ***a*** and ***c*** be the grandchild of ***a*** that comes on the path of new node.
3. If the tree becomes unbalanced due to insertion of new node, rebalance the tree by performing the appropriate rotation from the above four, on the subtree with root ***a***.

1. **Left-left rotation:**

   b is left child of a and c is left child of b. Let us take an example:

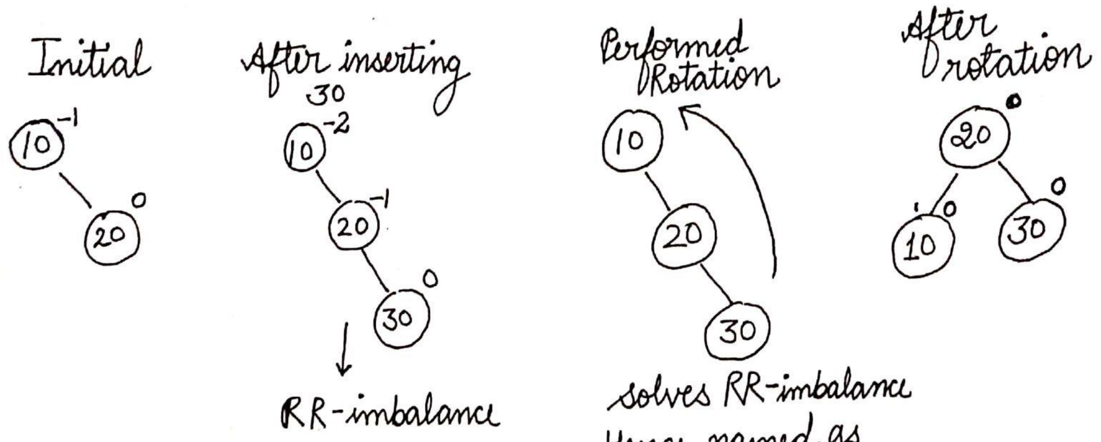




a_R = right subtree to $\textcircled{a}$
b_R = right subtree to $\textcircled{b}$
c_L = left subtree to $\textcircled{c}$
c_R = right subtree to $\textcircled{c}$

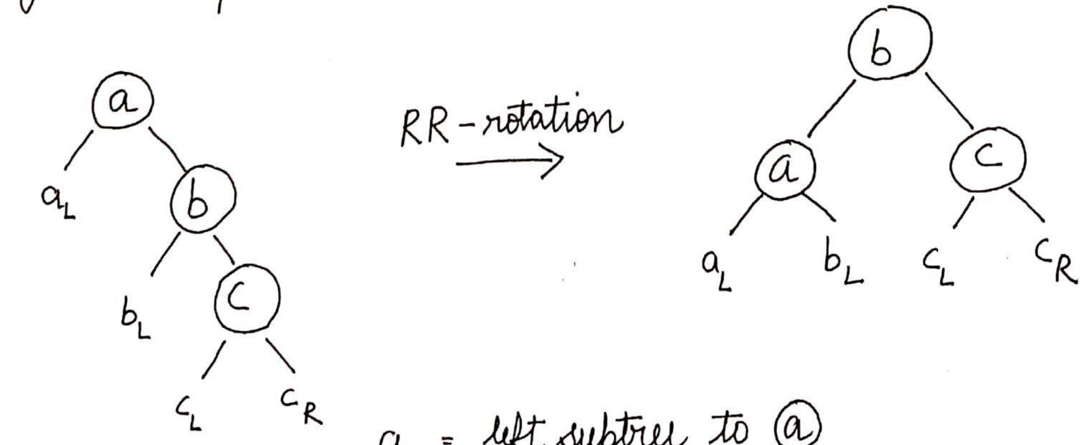
2. **Right – right rotation:**
   b is the right child of ***a*** and ***c*** is the right child of b.

Example :



solves RR-imbalance
Hence, named as
RR-rotation

Generic Expression:

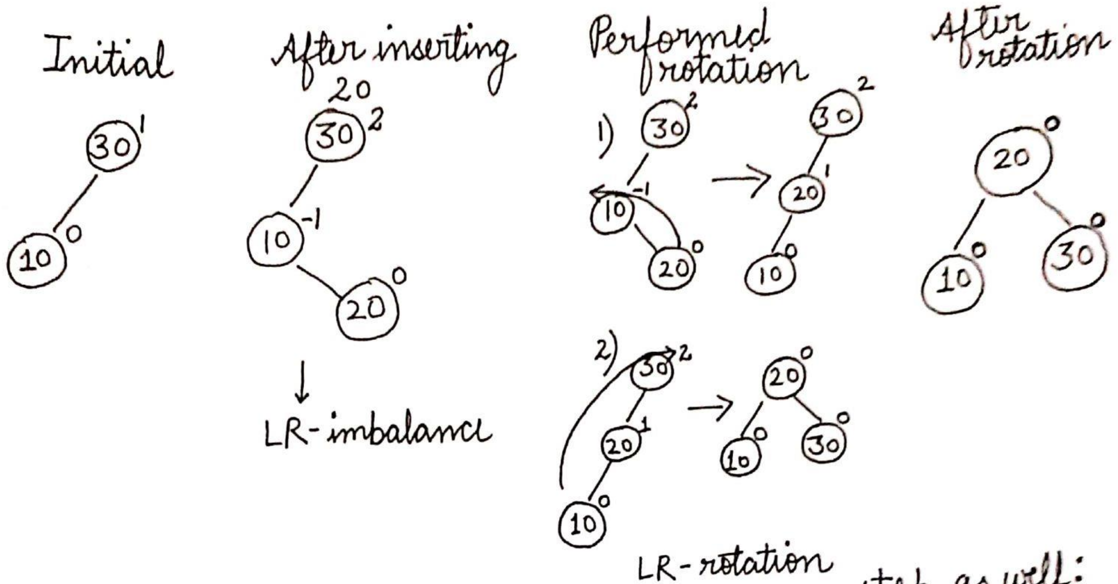


a_L = left subtree to ⓐ
b_L = left subtree to ⓑ
c_L = left subtree to ⓒ
c_R = right subtree to ⓒ

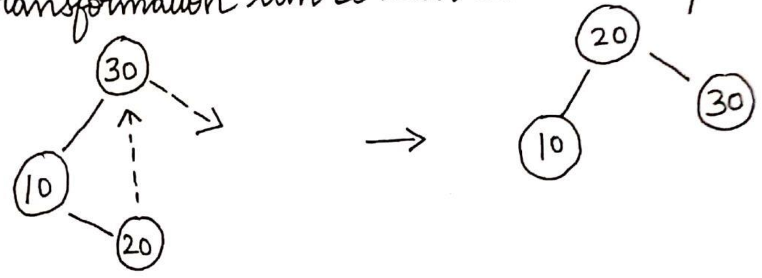
3. **Left -right rotation:**

   b is left child of ***a and c*** is right child of *b*.

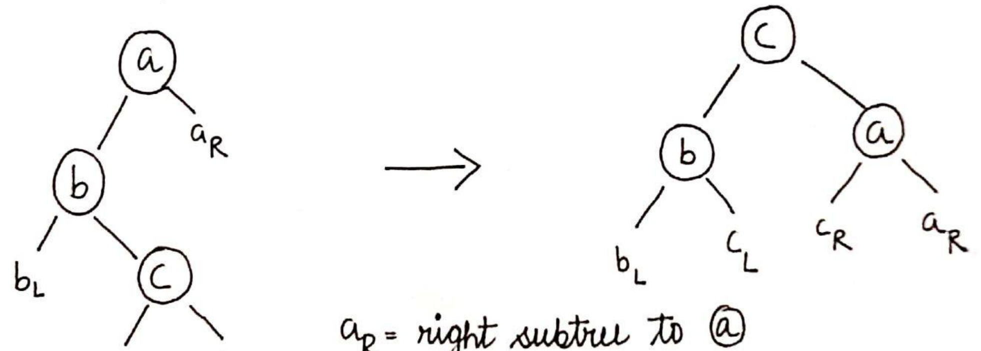
Example:



This transformation can be seen as one step as well:



Generic expression:



a_R = right subtree to ⓐ

b_L = left subtree to ⓑ

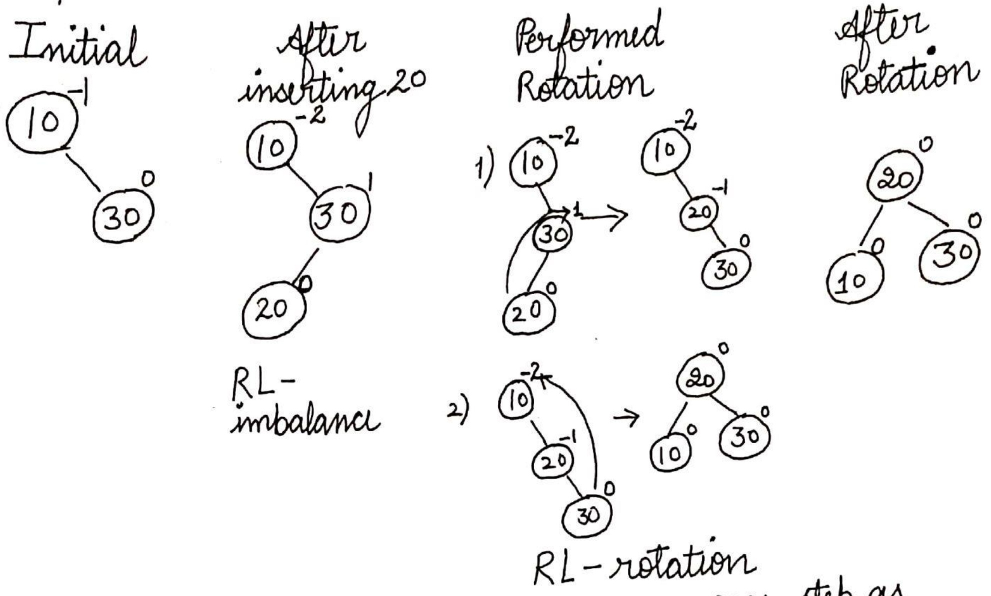
c_L = left subtree to ⓒ

c_R = right subtree to ⓒ

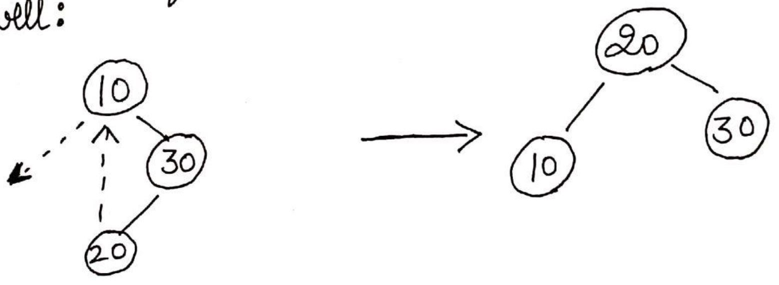
4. **Right-left rotation:**

b is right child of ***a and c*** is left child of *b*.

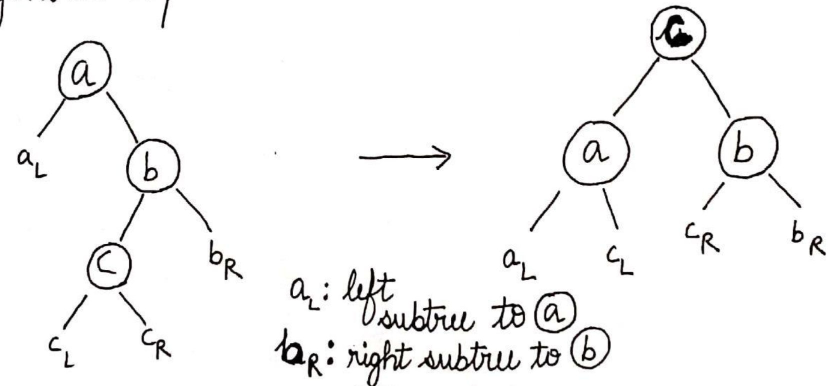
Example:



This transformation can be seen as one step as well:



Generic expression:



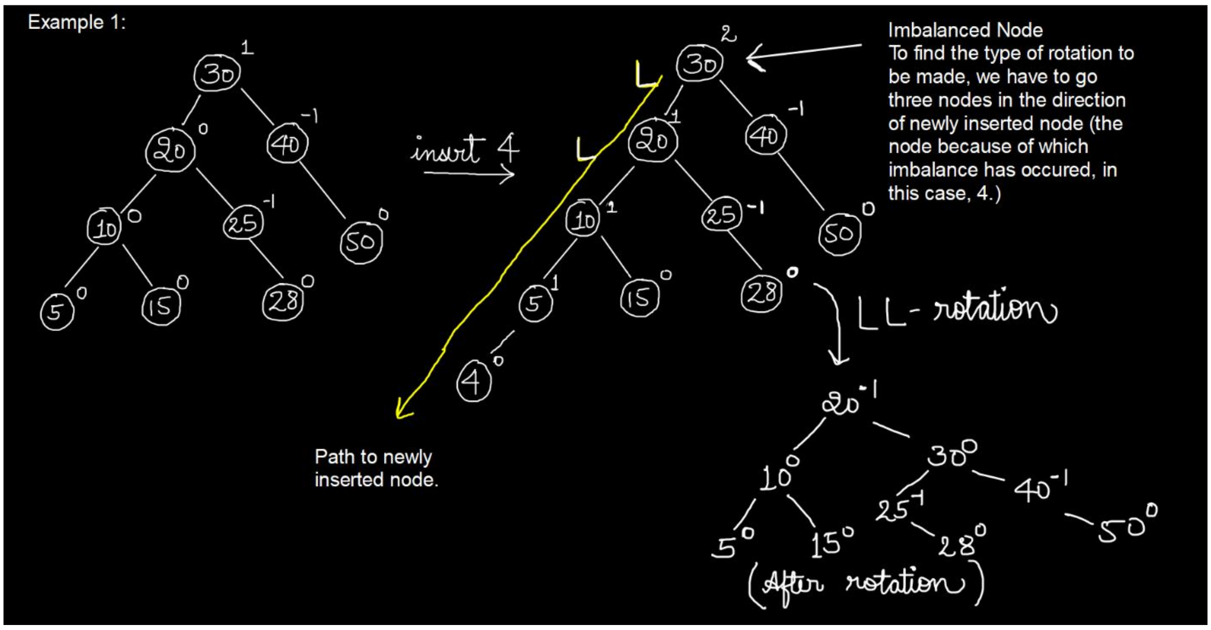
a_L: left subtree to a

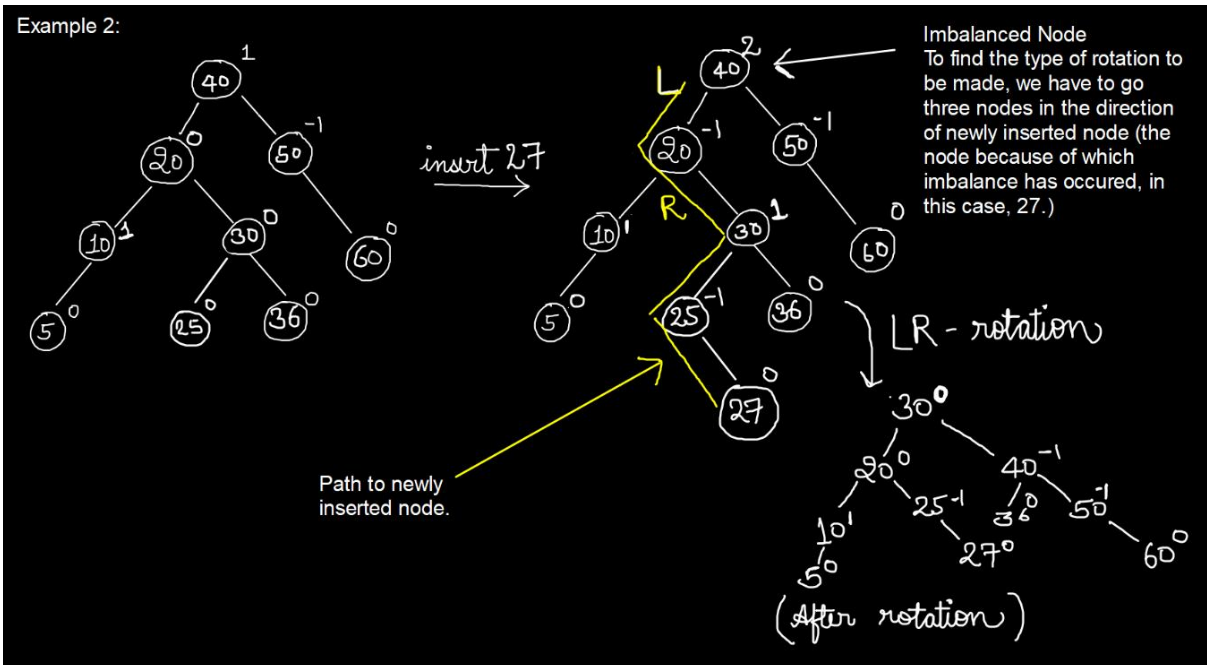
b_R: right subtree to b

c_L: left subtree to c

c_R: right subtree to c

Let us solve a few examples:


Example 1:
insert 4
L
L
Imbalanced Node
To find the type of rotation to be made, we have to go three nodes in the direction of newly inserted node (the node because of which imbalance has occured, in this case, 4.)
LL- rotation
Path to newly inserted node.
(After rotation)


Example 2:
insert 27
L
R
Imbalanced Node
To find the type of rotation to be made, we have to go three nodes in the direction of newly inserted node (the node because of which imbalance has occured, in this case, 27.)
LR - rotation
Path to newly inserted node.
(After rotation)

Implementation of AVL Tree:

In the implementation of AVL tree, we will insert each node recursively. The code of recursion travels up itself so in a bottom up manner, so we don't need to maintain a pointer for its ancestors.

The steps are:

1. Insert a new node similar to a BST.
2. Now the present node needs to be the ancestor of newly inserted node and update the height of present node.
3. Get the balanced factor of the present node, to check whether it is balanced or not.
4. If the balanced factor > 1, then the tree is unbalanced and we need to perform either left-left or left-right rotations.
5. If the balance factor < -1, then again, the tree is unbalanced and we need to perform either right- right or right- left rotations.

Code:

```
#include<iostream>
#include<cmath>
using namespace std;

class avlnode
{
    public:
    int key;
    avlnode *left;
    avlnode *right;
    int height;
};

// To get the height of the tree
int height(avlnode *N)
{
    if (N == NULL)
        return 0;
    return N->height;
}
// creating a newnode
avlnode* newNode(int key)
{
    avlnode* node = new avlnode();
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    node->height = 1; // as it is a leaf node
    return(node);
}
// right function to rotate subtree
avlnode *rightRotate(avlnode *b)
{
    avlnode *c = b->left;
    avlnode *T2 = c->right;
    c->right = b;
    b->left = T2;

    // Updating heights
    b->height = max(height(b->left),
                    height(b->right)) + 1;
    c->height = max(height(c->left),
                    height(c->right)) + 1;

    return c;
}

//left rotate subtree
avlnode *leftRotate(avlnode *c)
{
    avlnode *b = c->right;
    avlnode *T2 = b->left;

    b->left = c;
    c->right = T2;

    // Updating heights
    c->height = max(height(c->left),
                    height(c->right)) + 1;
    b->height = max(height(b->left),
                    height(b->right)) + 1;
    return b;
}

// Balanced factor
int getBalance(avlnode *N)
{
    if (N == NULL)
        return 0;
    return height(N->left) - height(N->right);
}
  //Insertion
avlnode* insert(avlnode* node, int key)
{
    // Insert a new node similar to a BST.

    if (node == NULL)
        return(newNode(key));

    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
    else
        return node;

    //Update ancestor height
    node->height = 1 + max(height(node->left),
                        height(node->right));

    // Balanced factor
    int balance = getBalance(node);
    // Left-Left rotation
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);

    // Right Right rotation
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);

    // Left Right rotation
    if (balance > 1 && key > node->left->key)
    {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    // Right Left rotation
    if (balance < -1 && key < node->right->key)
    {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

     return node;
}

//to print the preorder sequence of the constructed avl tree
void preOrder(avlnode *root)
{
    if(root != NULL)
    {
        cout << root->key << " ";
        preOrder(root->left);
        preOrder(root->right);
    }
}

int main()
{
    avlnode *root = NULL;
    /* Constructing tree given in
    the above figure */
    root = insert(root, 15);
    root = insert(root, 25);
    root = insert(root, 35);
    root = insert(root, 45);
    root = insert(root, 55);
    root = insert(root, 30);

    /* The constructed AVL Tree would be
             35
            / \
          25 45
          / \ \
         15 30 55
    */
    cout << "The Preorder traversal of AVL tree is:"<<endl;
    preOrder(root);
}
```