

**Ayesha Salam (SP21-BCS-011)**

**Syeda Humna Ashfaq(SP21-BCS-108)**

**Data Structures and Algorithms**

**Assignment 2**

**AVL tree(Height Balanced Tree)**

**Height Balanced Tree:**

A height-balanced binary tree is defined as a binary tree in which the height of the left and the right subtree of any node differ by not more than 1. AVL tree, red-black tree are examples of height-balanced trees.

**BST vs HBT:**

Binary search trees can sometimes result in right skewed or left skewed cases. In these cases, if there are n nodes then we have to make search n times to perform every operation which increases time complexity. The height of BST is n whereas the height of HBT log(n). Decreased height result less time complexity while searching, sorting and other operations.

**AVL TREE**

AVL tree is a self-balancing binary search tree in which each node maintains extra information called a balance factor whose value is either -1, 0 or +1.

AVL tree got its name after its inventor Georgy Adelson-Velsky and Landis.

**Balance Factor**

Balance factor of a node in an AVL tree is the difference between the height of the left subtree and that of the right subtree of that node.

**Balance Factor = (Height of Left Subtree - Height of Right Subtree)**

The self-balancing property of an AVL tree is maintained by the balance factor. The value of balance factor should always be -1, 0 or +1.

An example of a balanced AVL tree is:

Chart

Description automatically generated

After applying rotations check the balance factor again if it’s not in range apply rotations again.

**Operations on an AVL tree**

Various operations that can be performed on an AVL tree are:

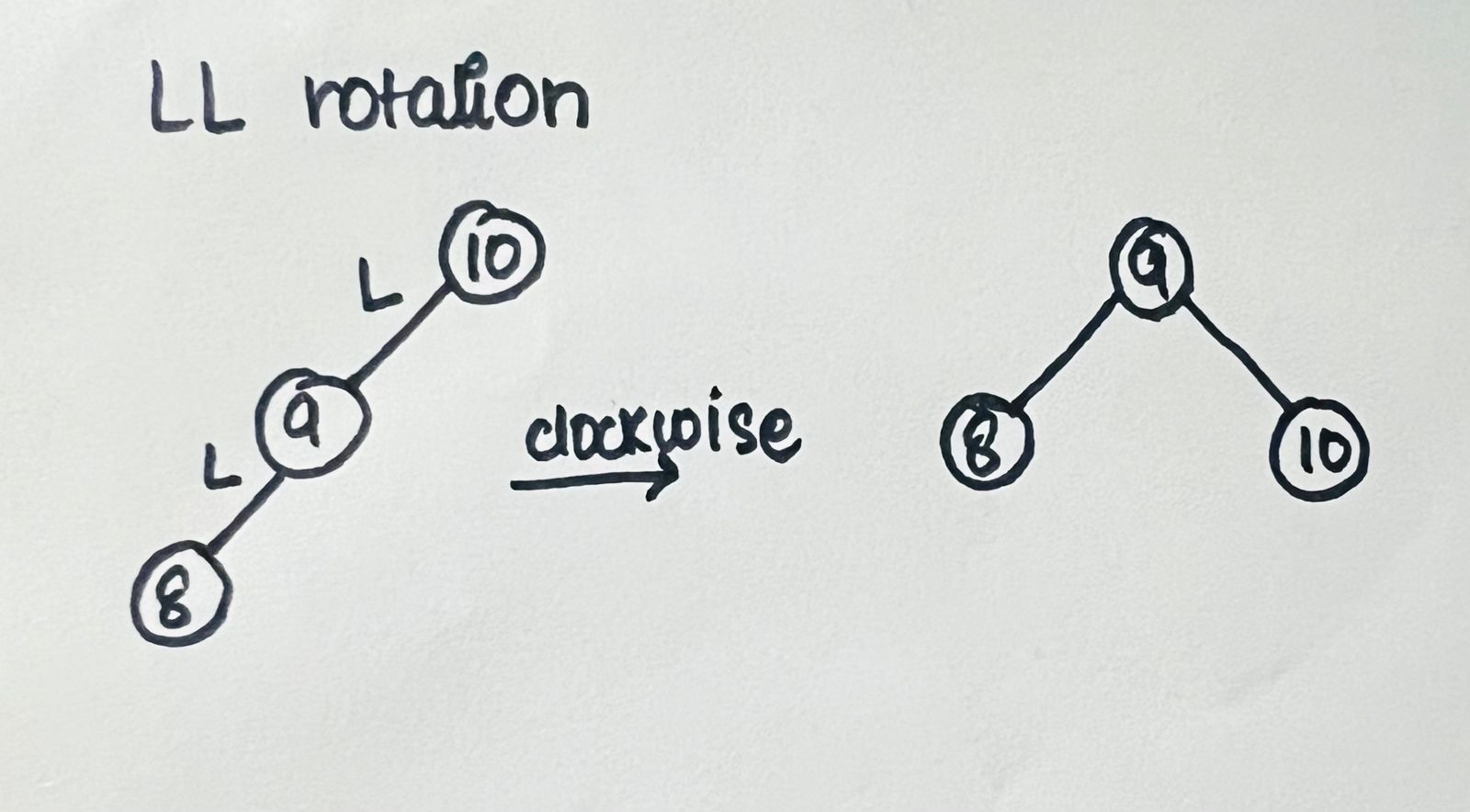
**Rotating the subtrees in an AVL Tree**

In rotation operation, the positions of the nodes of a subtree are interchanged.

There are two types of rotations:

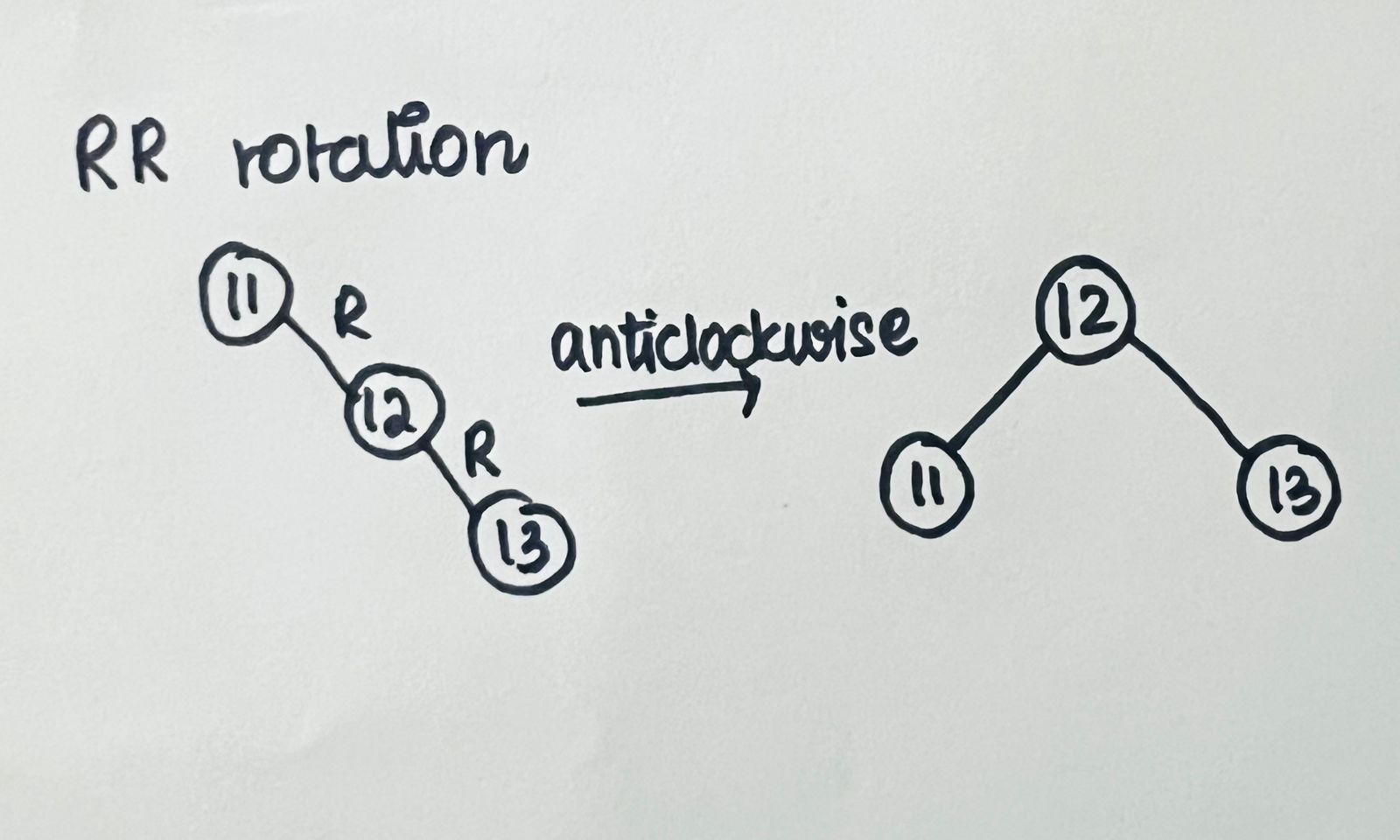
**Left Rotate**

In left-rotation, we perform a single rotation only and the node is moved in clockwise direction so the balance factor comes in range.(right rotation)



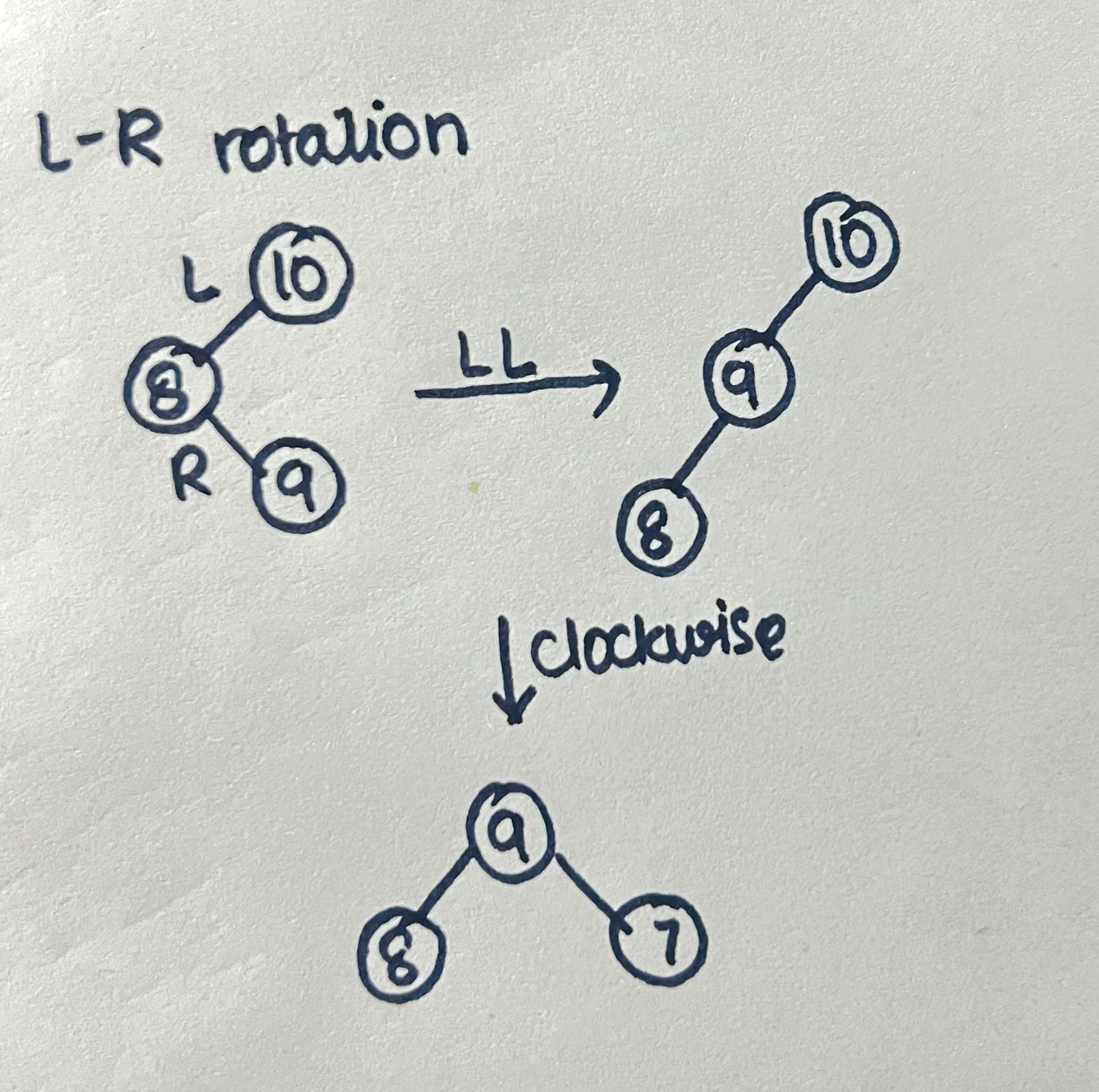
**Right Rotate**

In right rotation the rotation is performed once and the node is rotated in anticlockwise direction so the balance factor is in range.(left rotation)



**Left-Right rotation**

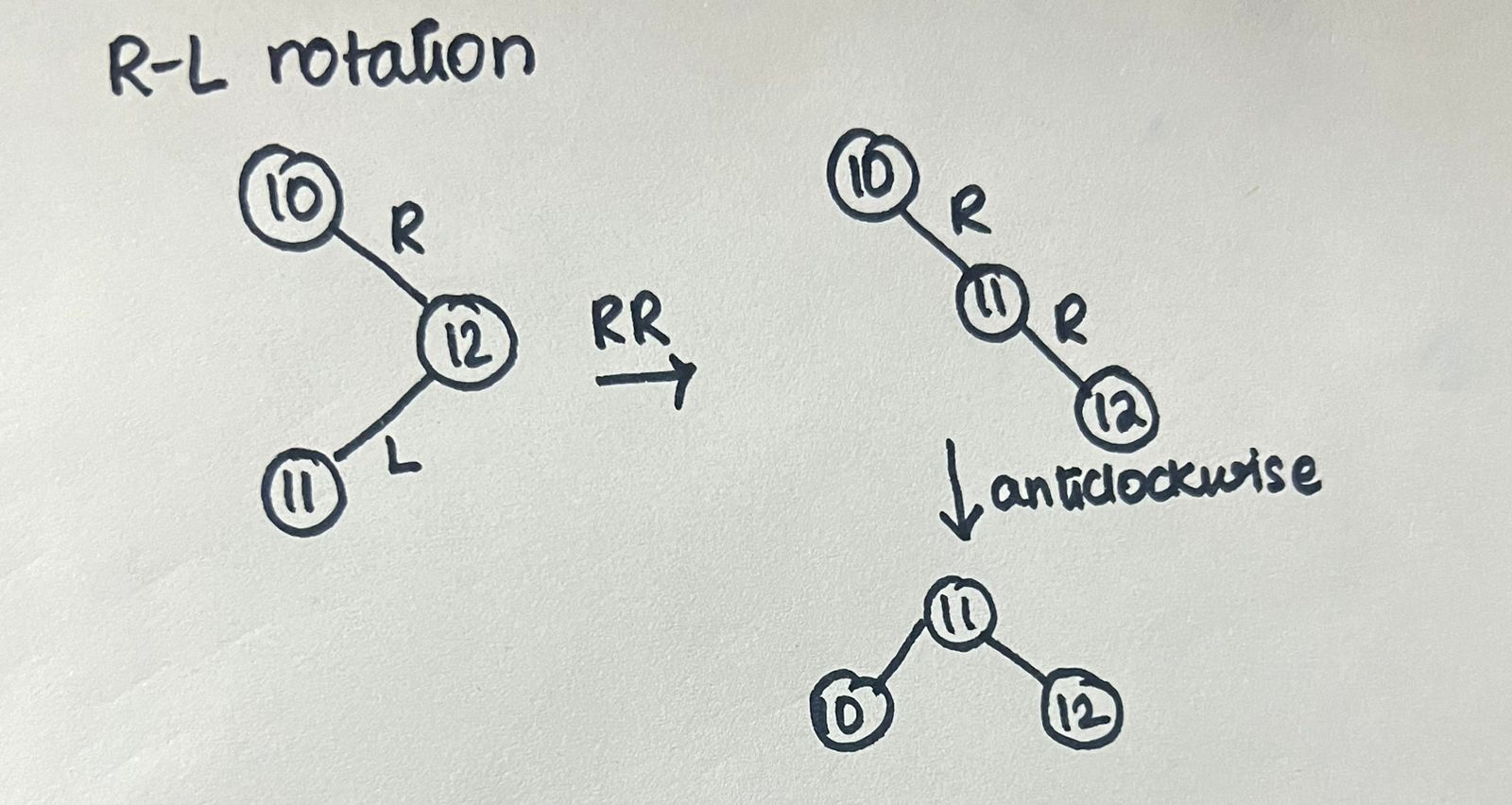
1. Do left rotation on the current to make it a case of left-left.
2. Do right rotation because now it is a left-left case.



**Right-left rotation**

1. Do right rotation on the current node to make it a right-right case.

2. Do left rotation because now it is a right-right case.



**Algorithm to insert a new Node**

1. A new node is always inserted as a leaf node which has a balance factor equal to zero.
2. While inserting a new node compare it with the root node.
3. If the value of the new node<root node than it is inserted on the left subtree of the current node.
4. If the value of new node>root node than it is inserted on the right subtree of current node.
5. After inserting every node, the balance factor of each node is checked and is it does not fall in limits than the rotations are applied till the tree becomes balanced.

**Algorithm to delete a node**

Consider x to be the node to be deleted. There are three cases for deleting a node:

1. If x is the leaf node (i.e., does not have any child), then remove x.
2. If x has one child, then substitute the contents of x with that of the child. Remove the child.
3. If x has two children, find the in-order successor, w, of x (i.e., node with a minimum value of key in the right subtree).

2) Update balance factor of the nodes.

3) Rebalance the tree if the balance factor of any of the nodes is not equal to -1, 0 or 1.

1. If BF of current node > 1,
   1. If BF of left child>= 0, do right rotation.

b. Else do left-right rotation.

1. If balance factor of current node < -1,
   * 1. If balance factor of right child <= 0, do left rotation.
     2. Else do right-left rotation.