DATA STRUCTURES – FALL 2023

LAB 10

Learning Outcomes

In this lab, you will learn to implement the AVL Tree.

National University of Computer & Emerging Sciences (NUCES) Islamabad,
Department of Cyber Security

AVL Tree Datastructure

AVL tree is a height-balanced binary search tree. That means, an AVL tree is also a binary search tree but it is a balanced tree. A binary tree is said to be balanced if, the difference between the heights of left and right subtrees of every node in the tree is either -1, 0 or +1. In other words, a binary tree is said to be balanced if the height of left and right children of every node differ by either -1, 0 or +1. In an AVL tree, every node maintains an extra information known as **balance factor**. The AVL tree was introduced in the year 1962 by G.M. Adelson-Velsky and E.M. Landis.

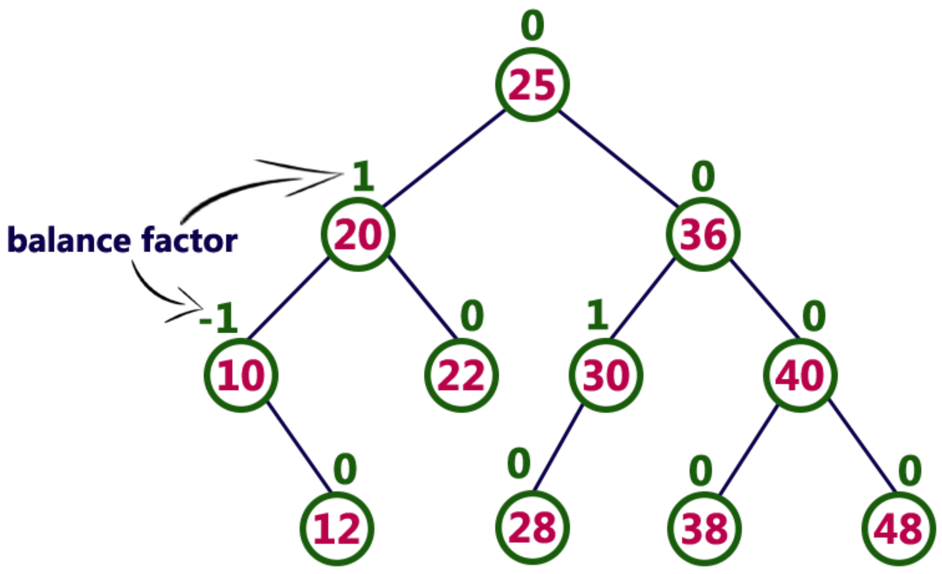
An AVL tree is defined as follows...

An AVL tree is a balanced binary search tree. In an AVL tree, balance factor of every node is either -1, 0 or +1.

Balance factor of a node is the difference between the heights of the left and right subtrees of that node. The balance factor of a node is calculated either **height of left subtree - height of right subtree** (OR) **height of right subtree - height of left subtree**. In the following explanation, we calculate as follows...

Balance factor = heightOfLeftSubtree - heightOfRightSubtree

Example of AVL Tree



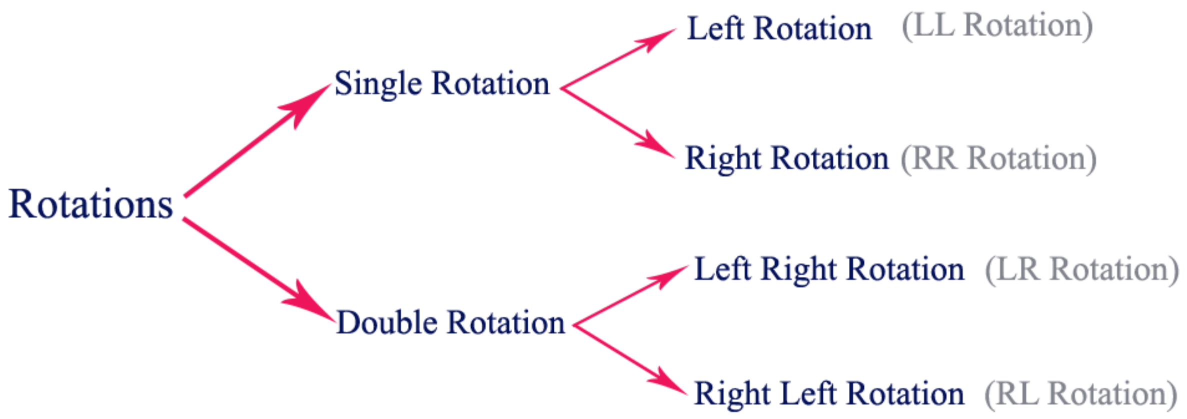
AVL Tree Rotations

In AVL tree, after performing operations like insertion and deletion we need to check the **balance factor** of every node in the tree. If every node satisfies the balance factor condition, then we conclude the operation otherwise we must make it balanced. Whenever the tree becomes imbalanced due to any operation, we use **rotation** operations to make the tree balanced.

Rotation operations are used to make the tree balanced.

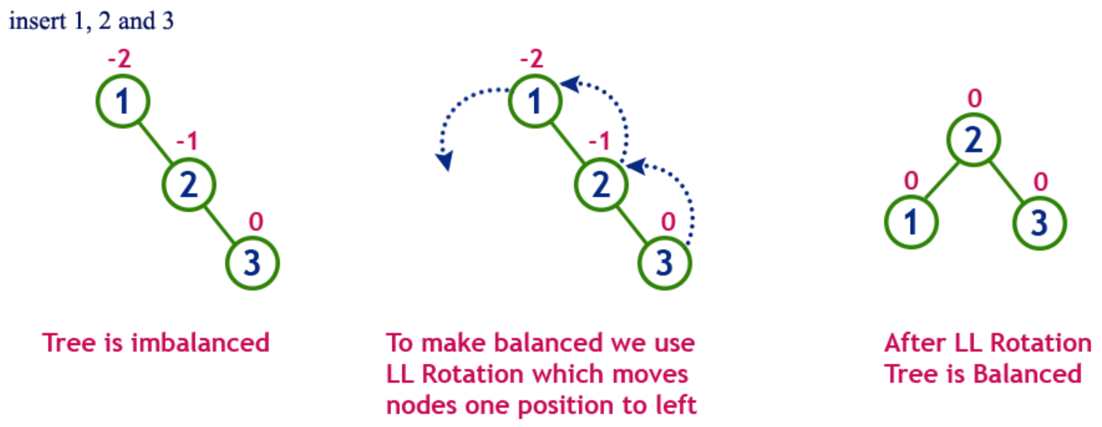
Rotation is the process of moving nodes either to left or to right to make the tree balanced.

There are **four** rotations and they are classified into **two** types.


Rotations
Single Rotation
Double Rotation
Left Rotation (LL Rotation)
Right Rotation (RR Rotation)
Left Right Rotation (LR Rotation)
Right Left Rotation (RL Rotation)

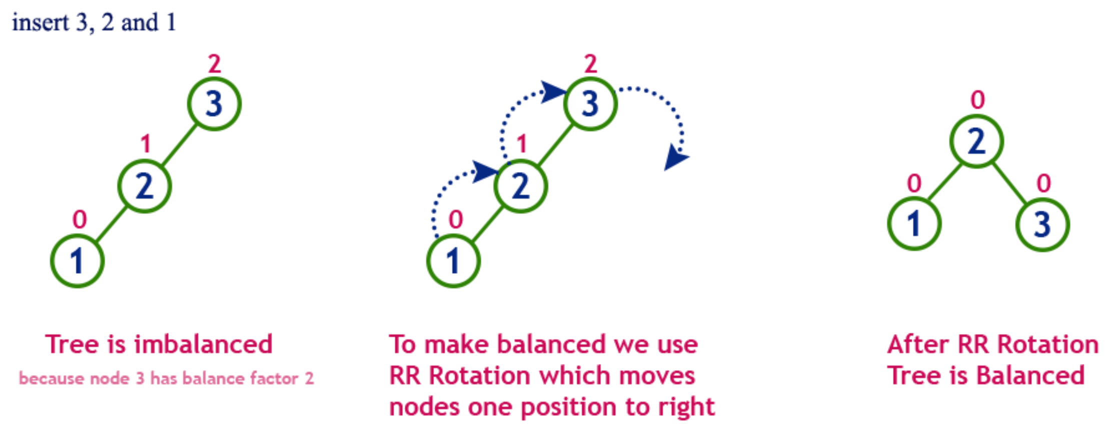
Single Left Rotation (LL Rotation)

In LL Rotation, every node moves one position to left from the current position. To understand LL Rotation, let us consider the following insertion operation in AVL Tree...


insert 1, 2 and 3
-2
1
-1
2
0
3
Tree is imbalanced
-2
1
-1
2
0
3
To make balanced we use LL Rotation which moves nodes one position to left
0
2
0
1
0
3
After LL Rotation Tree is Balanced

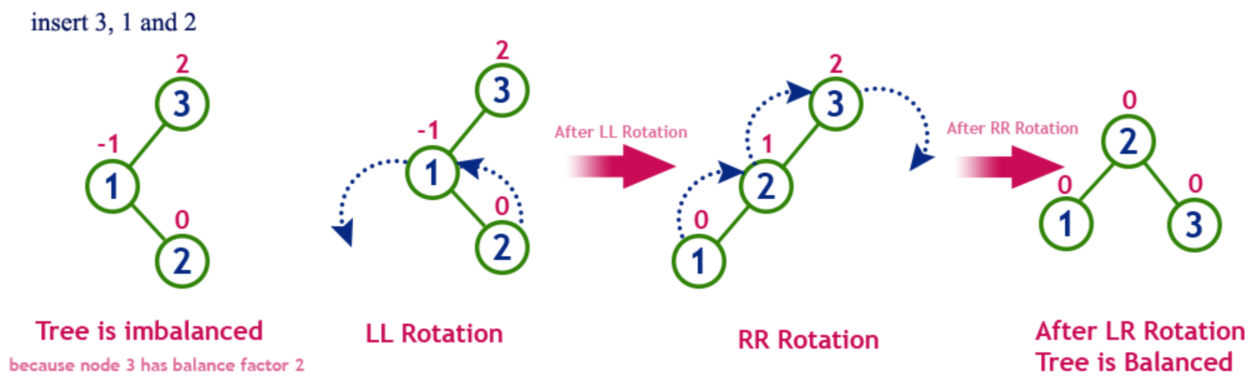
Single Right Rotation (RR Rotation)

In RR Rotation, every node moves one position to right from the current position. To understand RR Rotation, let us consider the following insertion operation in AVL Tree...



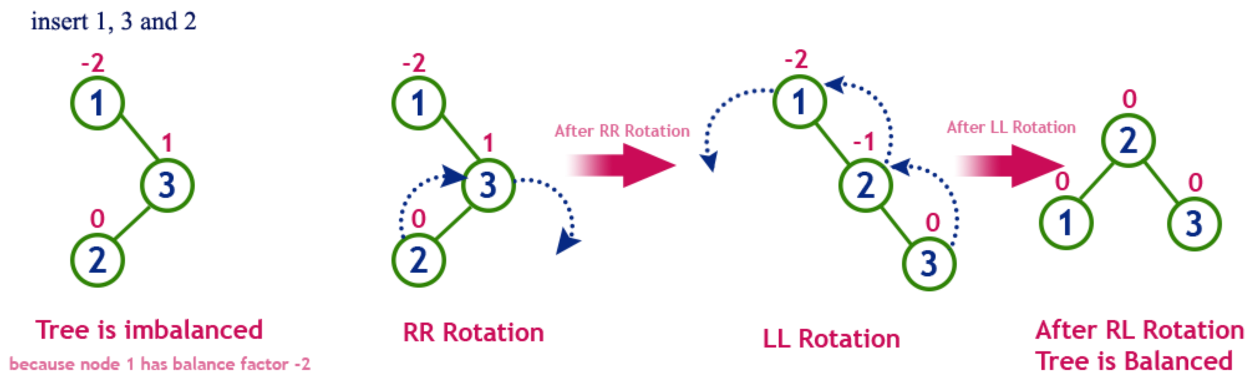
Left Right Rotation (LR Rotation)

The LR Rotation is a sequence of single left rotation followed by a single right rotation. In LR Rotation, at first, every node moves one position to the left and one position to right from the current position. To understand LR Rotation, let us consider the following insertion operation in AVL Tree...



Right Left Rotation (RL Rotation)

The RL Rotation is sequence of single right rotation followed by single left rotation. In RL Rotation, at first every node moves one position to right and one position to left from the current position. To understand RL Rotation, let us consider the following insertion operation in AVL Tree...



Operations on an AVL Tree

The following operations are performed on AVL tree...

1. **Search**
2. **Insertion**
3. **Deletion**

Insertion Operation in AVL Tree

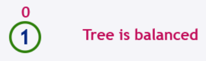
In an AVL tree, the insertion operation is performed with **O(log n)** time complexity. In AVL Tree, a new node is always inserted as a leaf node. The insertion operation is performed as follows...

- **Step 1 -** Insert the new element into the tree using Binary Search Tree insertion logic.
- **Step 2 -** After insertion, check the **Balance Factor** of every node.
- **Step 3 -** If the **Balance Factor** of every node is **0 or 1 or -1** then go for next operation.
- **Step 4 -** If the **Balance Factor** of any node is other than **0 or 1 or -1** then that tree is said to be imbalanced. In this case, perform suitable **Rotation** to make it balanced and go for next operation.

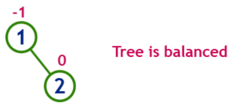
Example: Construct an AVL Tree by inserting numbers from 1 to 8.

National University of Computer & Emerging Sciences (NUCES) Islamabad,
Department of Cyber Security

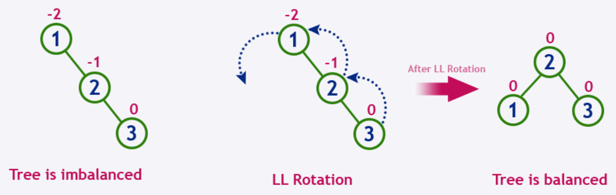
insert 1



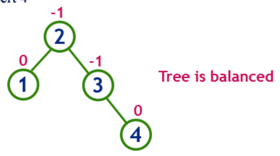
insert 2



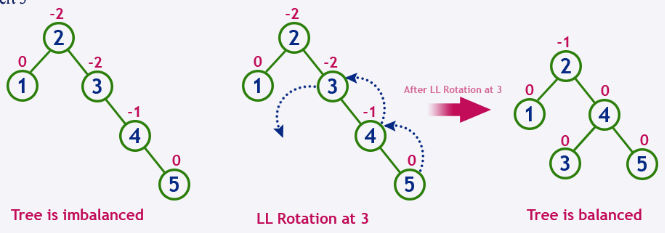
insert 3



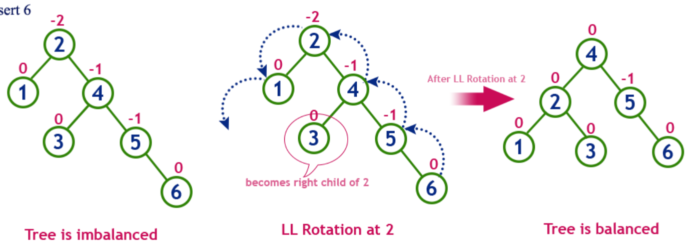
insert 4



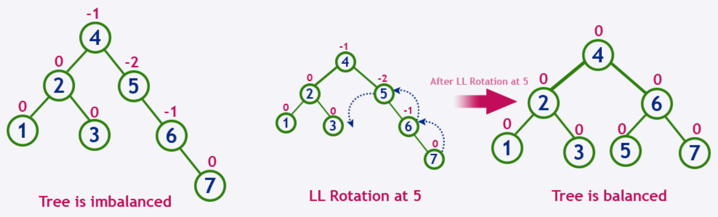
insert 5



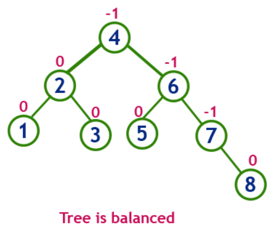
insert 6



insert 7



insert 8



Deletion Operation in AVL Tree

The deletion operation in AVL Tree is similar to deletion operation in BST. But after every deletion operation, we need to check with the Balance Factor condition. If the tree is balanced after deletion go for next operation otherwise perform suitable rotation to make the tree Balanced.

Lab Task

By completing the AVL Tree Lab, you will be able to:

1. Determine if a Binary Search Tree is critically imbalanced and distinguish between the various types of imbalances.
2. Implement functions to rotate nodes and balance a Binary Search Tree.
3. Insert and remove nodes from a Binary Search Tree while maintaining tree balance.

Step 1 - Start with the code you already have from the BST lab.

An AVL Tree is a BST but with added functionality for balancing, so much of the code you already have for BST will also be used in this lab.

Step 2 - Update your Node class.

To balance a tree, you need to be able to get the height of a node. You may find it helpful for debugging to implement a recursive get Height() function in your Node class, but keep in mind that doing so decreases the efficiency of adding and removing nodes. A more efficient algorithm may be to have a height variable with a constant-time updating function called on it at appropriate instances.

Step 3 - Add rotation and balancing functionality.

Begin by **adding rotateLeft() and rotateRight()** functions to your tree class. Then test them to ensure they work properly.

Now, add a **balance()** function to the tree class that uses the **getHeight()** function to determine if a node is critically imbalanced. If there is a critical imbalance, perform the appropriate rotations. Test your balance function to ensure it works properly. Now add the remaining two rotations **rotateLeftRight() and rotateRightLeft()**

Step 4 - Update insert() and remove() so that they rebalance the tree.

If you used recursion to write these functions originally, this will only require trivial changes. Immediately after the insertion or removal, you will recurse back up to every previous node that you visited to perform the insertion or removal. All you have to do is balance each node on the way back up through the recursion.

After updating each function, test it to make sure it works properly.

Step 5 - When you've successfully implemented and tested each part.

A website has been created to help you visualize AVL Tree algorithms. You can find it here https://www.cs.usfca.edu/~galles/visualization/AVLtree.html

Test Cases:

- Add 1
- Add 2
- Add 3
- PrintBST
- Add 4
- Add 5
- Add 6
- Add 10
- PrintBST
- Add 8
- Add 7
- Add 9
- PrintBST
- Remove 11
- Remove 10
- Remove 9
- Remove 6
- PrintBST
- Remove 8
- Remove 7
- PrintBST
- Remove 5
- Remove 4
- Remove 3
- Remove 2
- PrintBST
- Remove 1
- PrintBST