National University of Computer & Emerging Sciences, Karachi

Computer Science Department

Fall 2022, Lab Manual – 10

Course Code: CL-2001	**Course : Data Structures - Lab**
Instructor(s) :	**Eman Shahid**

LAB - 10

AVL & its Rotations, basic utility functions

AVL TREE:

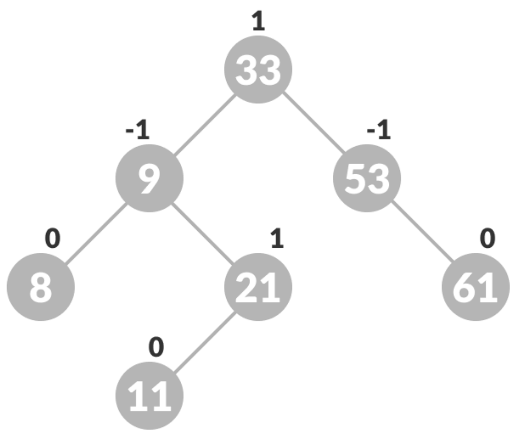
AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes. The valid BF range lies in { -1, 0, +1. In an AVL tree, every node maintains extra information known as the balance factor.

```
  class Node {
  int key, height;
          Node *left, *right;

        Node(int d) {
        key = d;
                 height = 1;
            }
     } ;
```

Balance Factor:

A balanced binary tree, also referred to as a height-balanced binary tree, is defined as a binary tree in which the height of the left and right subtree of any node differ by not more than 1. difference between the left and the right subtree for any node is not more than one the left subtree is balanced the right subtree is balanced
To calculate the balance factor for the AVL tree, we first need to calculate the height of each subtree.



```
int height(Node N) {     if (N == null)
            return 0;
      return N->height;
      }

int getBalance(Node N) {       if (N ==
null)
            return 0;
     return height(N->left) - height(N->right);
      }
```

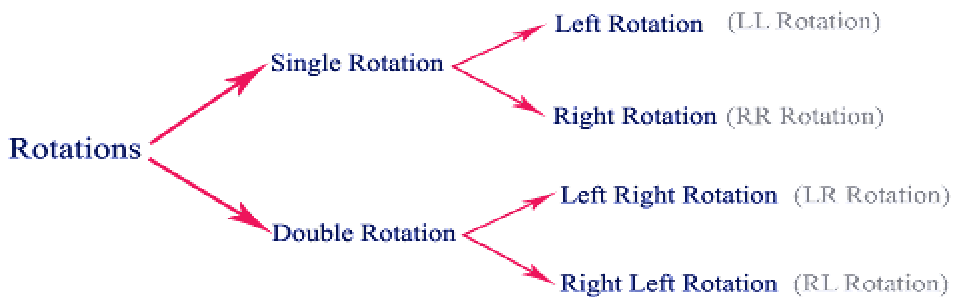
Consider the Following Binary Search Tree now: `Node`

```
insert(Node node, int key) {
          int lstH , rstH;
          /* 1. Perform the normal BST insertion */
          if (node == null)
               return (new Node(key));

       if (key < node->key)
       node->left = insert(node->left, key);
       else if (key > node.key)
       node->right = insert(node->right, key);
          else // Duplicate keys not allowed
               return node;
// set the height of every node
          lstH = height(node->left);
       rstH = height(node->right);
  node->height = 1 + max(lstH  , rstH);
}
Main(){
insert(tree->root, 10);
insert(tree->root, 20);
  insert(tree->root, 30);
  insert(tree->root, 40);
  insert(tree->root, 50);
  insert(tree->root, 45);
}
```

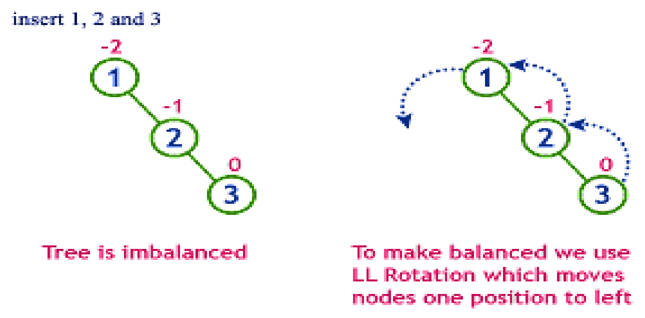
The AVL Tree Rotations :

In the 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. 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)

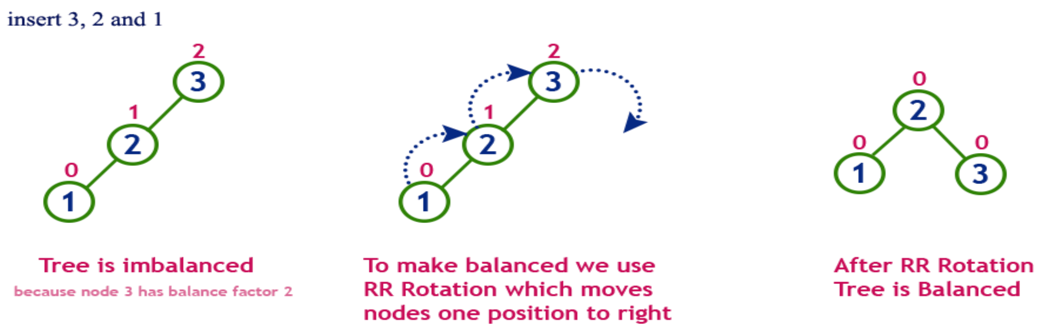
Single Left Rotation (LL Rotation):

In LL Rotation, every node moves from one position to the 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
To make balanced we use LL Rotation which moves nodes one position to left
After LL Rotation Tree is Balanced

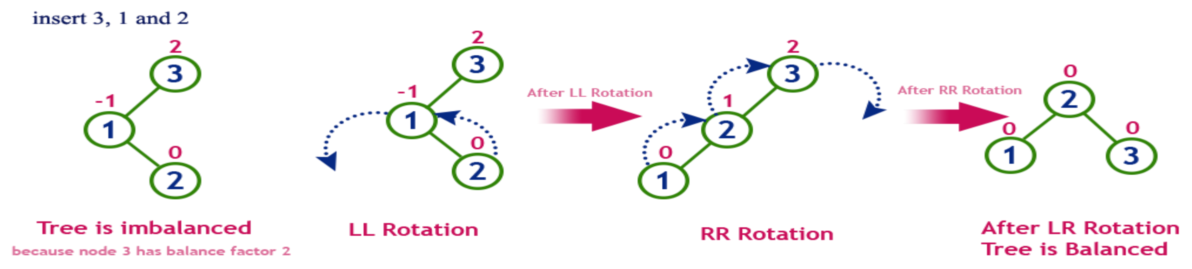
Single Right Rotation (RR Rotation)

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


insert 3, 2 and 1
Tree is imbalanced
because node 3 has balance factor 2
To make balanced we use RR Rotation which moves nodes one position to right
After RR Rotation Tree is Balanced

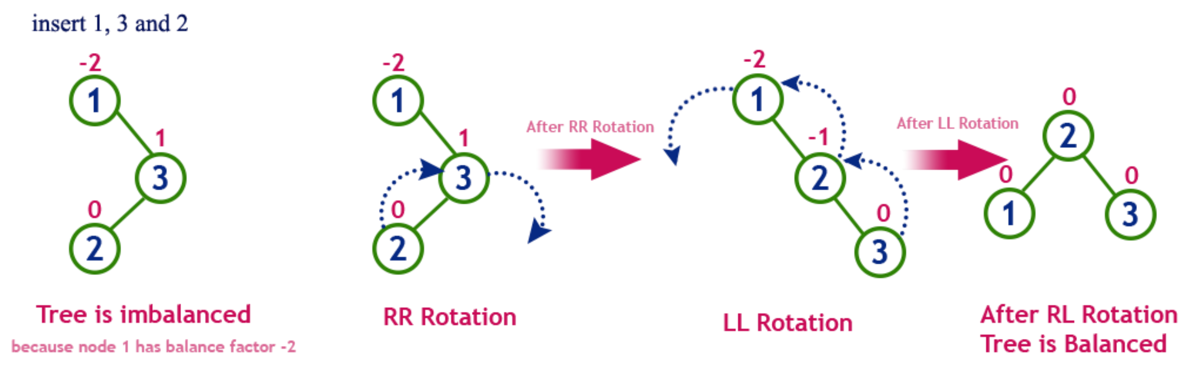
Left Right Rotation (LR Rotation)

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



```
     int balance = getBalance(node);
                cout<< " " << balance << " " <<node->key);
            // If this node becomes unbalanced, then there
        // are 4 cases Left Left Case
if (balance > 1 && key < node->left->key)
               return right Rotate(node);

            // Right Right Case

         if (balance < -1 && key > node->right->key) {
                return leftRotate(node);
            }

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

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

//

```
Node leftRotate(Node x) {
        Node y = x->right;
        Node T2 = y->left;
        y->left = x;
       x->right = T2;

                // Update heights
        x->height = max(height(x-
        >left), height(x->right)) + 1;
        y->height = max(height(y-
        >left), height(y->right)) + 1;

                // Return new root
                return y;
        }
```

```
Node rightRotate(Node y) {
        Node x = y->left;
        Node T2 = x->right;
        x->right = y;
       y->left = T2;

        // Update heights
        y->height = max(height(y->left),
        height(y->right)) + 1;
        x->height = max(height(x->left),
        height(x->right)) + 1;
                // Return new root
        return x;
                                }
```

Operations on an AVL Tree

The following operations are performed on the AVL tree.

1. Insertion
2. Search
3. Deletion

Insertion:

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 the 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 the next operation

Task 01: Construct BST: {10,20,30,40,50,45} after construction, write a function to check that constructed BST are balanced or not?

Task-02: Create an AVL Tree using insertion sequence as { 55, 66, 77, 11, 33, 22, 35, 25, 44, 88,99 }, the program should print the height of tree too. **Search Operation ion AVL Tree:**

In an AVL tree, the search operation is performed with O(log n) time complexity. The search operation in the AVL tree is similar to the search operation in a Binary search tree. We use the following steps to search an element in the AVL tree.

Step 1 - Read the search element from the user.

Step 2 - Compare the search element with the value of the root node in the tree.

Step 3 - If both are matched, then display "Given node is found!!!" and terminate the function

Step 4 - If both are not matched, then check whether the search element is smaller or larger than that node value.

Step 5- If the search element is smaller, then continue the search process in the left subtree.

Step 6 - If the search element is larger, then continue the search process in the right subtree.

Step 7 - Repeat the same until we find the exact element or until the search element is compared with the leaf node.

Step 8 - If we reach the node having the value equal to the search value, then display "Element is found" and terminate the function.

Step 9 - If we reach the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function.

The implementation of the Search Operation is given below:

Task-03: Search the nodes having values in (66, 22, 44) in the tree constructed above.

Deletion Operation in AVL Trees:

The deletion operation in AVL Tree is similar to the 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 goes for the next operation otherwise perform suitable rotation to make the tree balanced.

```
if (balance > 1 &&  getBalance(root->left) >= 0)
    return rightRotate(root);

  // Left Right Case
  if (balance > 1 && getBalance(root->left) < 0)
  {
    root->left = leftRotate(root->left);
    return rightRotate(root);
  }

  // Right Right Case
  if (balance < -1 &&  getBalance(root->right) <= 0)
    return leftRotate(root);

  // Right Left Case
```

```
if (balance < -1 && getBalance(root->right) > 0)
{
    root->right = rightRotate(root->right);
    return leftRotate(root);
}
```

Task-04: Delete the nodes 35 and 99 from the tree, recalculate the bf and perform rotations to balance the tree.

Task-05: Print the resultant tree in pre, post, and in order.

Lab-11: AVL Trees and Utility Functions		
Std Name:	**Std_ID:**	
Lab11-Tasks	**Completed**	**Checked**
Task #1		
Task #2		
Task #3		
Task #4		