**Assignment 3 – Implementation of AVL Tree Operations**

**Data Structures and Algorithms**

**Submission Date: 20:59, Sunday, December 26, 2021**

**Mapped to CLO-2**

Dear Students,

The objective of this assignment is to implement a simple application using AVL Trees as the data structure. You may choose any application like maintaining students record, dictionary words or any other list; you may also decide to store some fields from the dataset you have chosen for your term project. The new operations to be implemented in additions to the ones you coded for BSTs are the following:

* Compute balance factor of a node x
* Compute height of a node x
* Left rotation operation
* Right rotation operation
* Left-Right rotation operation
* Right-Left rotation operation.

To implement the afore mentioned operations, you shall have to modify the following:

* **TreeNode class**

Declare a variable height so that every node stores its current height in the said variable. We know that a new node always gets inserted as a leaf. Thus, the height should be initialized to 0 in the constructor. Note that maintaining the height of a node shall reduce the complexity of insert and delete operations; after insertion or deletion operations, only the height of the affected branches shall have to be computed and updated.

* **Insertion Operation**

You shall need a recursive version of the insert function. I have shared it below for your convenience.

* + Inside the insert function, you may call the balanceFactor(Node\* current) function. If a node with a balance factor +2 or -2 gets detected, you should check which of the four rotation operations is required and then call the concerned function to rebalance the tree.
  + If no rotation is applied around a node e.g. x, then the insert function should return the address of the same node x which will be stored in the left or right child field of its parent node. In case, a rotation operation is applied then the insert function should return the address of **node y** so that the left or right childe field of the parent of x gets updated.

Node\* insert(Node\* root, int key)

{

    // if the root is null, create a new node and return it

    if (root == nullptr) {

        return new Node(key);

    }

     // if the given key is less than the root node, recur for the left subtree

    if (key < root->data) {

        root->left = insert(root->left, key);

    }

    // if the given key is more than the root node, recur for the right subtree

    else {

        root->right = insert(root->right, key);

    }

    return root;

}

* **Deletion Operation**

Once a node has been deleted, then update height of the nodes in the affected branches, and the balance factor as well. If any node with a balance factor +2 or -2 gets detected, then check which of the four rotation operations are required. Call the concerned rotation operation. You may modify the delete operation by implementing its recursive version, if necessary. You may find a solution without doing so. Why? Because you already have the address of the deleted node in the ploc pointer variable!

**Note:** Once a node has been deleted, and the tree has been rebalanced, you shall need to update the height of those nodes in the updated tree which are on the path from the root to the deleted node.

|  |
| --- |
| **Submission Guidelines:**   * Upload your source code along with the screenshots of the called functions in a word file. * Briefly mention the application you have implemented. * Moreover, also upload your source code in a separate .cpp file. |