Lab 11 AVL Trees Implementation

Learning Outcomes:

After successfully completing this lab the students will be able to:

1. Understand the properties of AVL Trees and their balancing features.
2. Develop C programs for implementing AVL Trees and their balancing features.
3. Use AVL Trees to load/store data to/from a file on the hard disk.

Pre-Lab Reading Task:

AVL Trees:

In an AVL tree, the heights of the two child subtrees of any node differ by at most one; if at any time they differ by more than one, re-balancing is done to restore this property. Lookup, insertion, and deletion all take ***O(log n)*** time in both the average and worst cases, where ***n*** is the number of nodes in the tree prior to the operation. Insertions and deletions may require the tree to be rebalanced by one or more tree rotations.

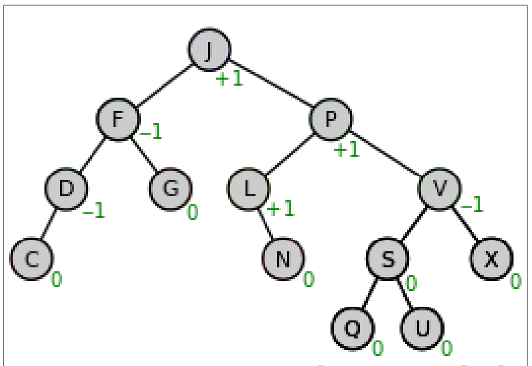


Figure 1: AVL Tree. Nodes are marked with their Balance Factors

Why AVL Trees?

Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take ***O(h)*** time where ***h*** is the height of the BST. The cost of these operations may become ***O(n)*** for a skewed Binary tree. If we make sure that height of the tree remains ***O(log n)*** after every insertion and deletion, then we can guarantee an upper bound of ***O(log n)*** for all these operations. The height of an AVL tree is always ***O(log n)*** where ***n*** is the number of nodes in the tree

Balance Factor:

The balance factor of any node of an AVL tree is in the integer range [-1,+1]. If after any modification in the tree, the balance factor becomes less than −1 or greater than +1, the subtree rooted at this node is unbalanced, and a rotation is needed.

balanceFactor = height(left subtree) - height(right subtree)

Insertion

To make sure that the given tree remains AVL after every insertion, we must augment the standard BST insert operation to perform some re-balancing. Following are two basic operations that can be performed to re-balance a BST without violating the BST property (keys(left) < key(root) < keys(right)).

1. Left Rotation
2. Right Rotation

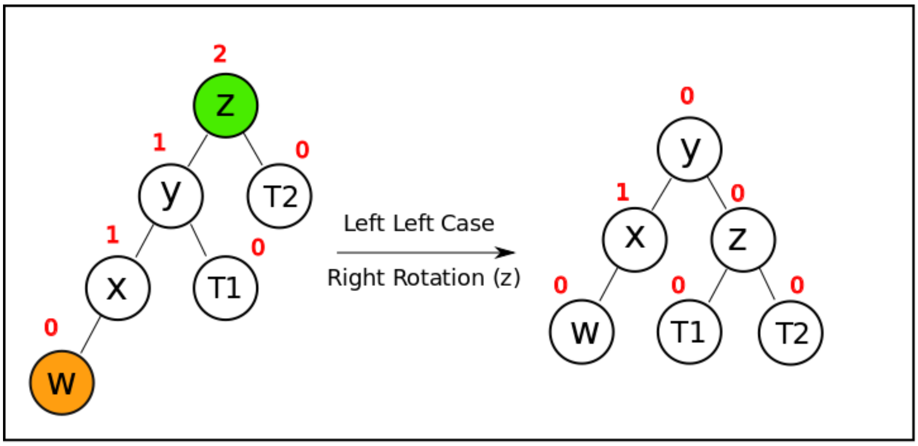
Steps to follow for insertion

Let the newly inserted node be ***w***

1. Perform standard BST insert for ***w***.
2. Starting from ***w***, travel up and find the first unbalanced node. Let ***z*** be the first unbalanced node, ***y*** be the child of ***z*** that comes on the path from ***w*** to ***z*** and ***x*** be the grandchild of ***z*** that comes on the path from ***w*** to ***z***.
3. Re-balance the tree by performing appropriate rotations on the subtree rooted with ***z***. There can be 4 possible cases that needs to be handled as ***x***, ***y*** and ***z*** can be arranged in 4 ways. Following are the possible 4 arrangements:
   - y is left child of z and x is left child of y (Left Left Case)
   - y is left child of z and x is right child of y (Left Right Case)
   - y is right child of z and x is right child of y (Right Right Case)
   - y is right child of z and x is left child of y (Right Left Case)

Following are the operations to be performed in above mentioned 4 cases. In all of the cases, we only need to re-balance the subtree rooted with z and the complete tree becomes balanced as the height of subtree (After appropriate rotations) rooted with z becomes same as it was before insertion.

Left Left Case: (We will need to perform a right rotation)



For more information read **Chapter 10.4** from the book: ***"Data Structures using C"*** by Reema Thareja.

In-Lab Tasks:

You are provided with skeleton code that builds a Binary Search Tree by adding 10 nodes to it. Functions for node insertion and printing the tree (in-order traversal only) are already implemented. Your task is to **modify the *insert*** function to incorporate AVL insertion. You will find Programming Example on Page 324 of the above mentioned book useful.

Post-Lab Tasks:

Complete the following functions for the BST:

1. Save the Tree data to a file (In-Order, Pre-Order and Post-Order)
2. Load tree from a file containing numbers.