AVL TREE IMPLEMENTATION

About:

An AVL Tree is a self-balancing binary search tree where lookup, insertion, deletion all take O(lg *n*) time in even the worst case. In an AVL Tree, the difference of the heights of the two subtrees is at most one. This allows for quicker insertion, deletion, and lookups in the average case.

How we did it:

We started by declaring nodes for the AVL tree. We did so creating a struct with members data, height, and pointers to the left and right.

After that we dove right in declaring the AVLTree class with all the private and public class members and methods.

Members:

There are several members of the AVL Class which we reuse to create the AVL Tree. In our implementation we have 6 private members and 7 public members. The private members keep the encapsulation.

We started off by creating a max method that returns the maximum element of two integers. We did this because we needed to use this functionality a lot in the implementation. The height method returns the total height of the AVL Tree. We passed the AVL Tree pointer and recursively got the height of the tree.

The right rotate and left rotate methods do exactly as the names of the methods explain. They rotate the tree to the orientation that is specified. We do so by assigning right node of the node to the left of the right node of the root. We put right of the left node from the root to a temp value. After that, we assign the right of the root's left node to the left of the root. We assign the right of the root's left to the temp value we got. Finally, we update the height of the tree and return the respective root. Keep in mind, that the process is altered for right rotate, since we use right instead of left.

The check balance returns 0 if root is NULL, if not, it returns the difference between the heights of the two subtrees.

The add method inserts a new incoming element to the tree. It is broken down into 4 cases. Left-Left case, Right-Right case, Left-Right case, and Right-left case where the directions determine where in the tree we are navigating. We use the right rotate and left rotate methods to achieve this. After adding the element, we return the original root.

FindMin returns the minimum element of the tree by traversing to the leftmost node of the tree and returning the value. The delete_node() function deletes a node from the AVL Tree. The implementation is long and complex; we use the right rotate and left rotate functions for the

cases just like the insert functions. We achieved this implementation by breaking down into cases such as, if a node is a left, we can simply delete it, If the node's left child is NULL, we can connect the root to it's right child, and other cases.

We also implemented the inorder and preorder traversal of the AVL tree. This was pretty simple using a simple base case of root equals NULL and recursively calling the function. We also have a destroy function that destroys the whole AVL Tree which is a recursive function. The final method which is the search function searches for an element in the AVL tree. It returns a boolean value. It returns true if the element is present in the tree and returns false otherwise.