CSE 222 – Data Structures and Algorithms HW7 Report

AVLTree Implementations

INSERTION

- AVLTree insertion is not very different than normal BST insertion. First we check if the node is null or not if it is it means that the tree is empty or we have reached a leaf node, and a new node is created with the Stock object and returned. This is the best case.
- If the node is not null, the method compares the symbol of the Stock object with the symbol of the Stock object in the current node. If the symbol of the Stock object to be inserted is less than the symbol of the Stock object in the current node, the method recursively calls itself with the left child of the current node. If the symbol of the Stock object to be inserted is greater than the symbol of the Stock object in the current node, the method recursively calls itself with the right child of the current node. If the symbols are equal, it means the Stock object already exists in the tree, so the current node is returned without any changes.
- After the recursive calls, the height of the current node is updated. The height of a node in an AVL Tree is the maximum height of its two children plus one. Finally, the balance method is called on the current node to ensure that the AVL Tree maintains its balance after the insertion. The balance method checks the balance factor of the node (the difference in height between the left and right child of the node) and performs the necessary rotations if the tree is unbalanced.

DELETION

- If the node is null, it means the tree is empty or the node to be deleted is not found in the tree. In this case, it returns null.
- If the symbol is less than the symbol of the current node's stock, the method recursively calls itself to delete the node from the left subtree
- If the symbol is greater than the symbol of the current node's stock, the method recursively calls itself to delete the node from the right subtree.
- If the symbol is equal to the symbol of the current node's stock, it means the node to be deleted is found. In this case, it checks two conditions:
    - If the node has either no left child or no right child, it replaces the node with its existing child.
    - If the node has both left and right children, it finds the node with the minimum value from the right subtree, replaces the current node's stock with that node's stock, and recursively deletes that node from the right subtree.
- After deletion, if the node becomes null, it returns null.
- If the node is not null, it updates the height of the node and balances the tree to maintain the AVL tree property. The balanced node is then returned.

SEARCH

- If the node is null or the symbol of the current node's stock is equal to the given symbol, it returns the node. This means either the tree is empty, or the node has been found.
- If the given symbol is greater than the symbol of the current node's stock, the method recursively calls itself to search the node in the right subtree.

- If the given symbol is less than the symbol of the current node's stock, the method recursively calls itself to search the node in the left subtree.

<u>ADDITIONAL FUNCTIONS</u>

BALANCE

- It first calculates the balance factor of the given node. The balance factor is the difference between the heights of the left and right subtrees of the node.
- If the balance factor is greater than 1, it means the tree is left-heavy (i.e., the left subtree is taller than the right subtree). In this case, it checks the balance factor of the left child node:
    - If the balance factor of the left child is less than 0, it means the left child is right-heavy. To balance this, it performs a left rotation on the left child.
    - After that, it performs a right rotation on the given node and returns the new root node.
- If the balance factor is less than -1, it means the tree is right-heavy (i.e., the right subtree is taller than the left subtree). In this case, it checks the balance factor of the right child node:
    - If the balance factor of the right child is greater than 0, it means the right child is left-heavy. To balance this, it performs a right rotation on the right child.
    - After that, it performs a left rotation on the given node and returns the new root node.
- If the balance factor is between -1 and 1, it means the tree is already balanced. In this case, it simply returns the given node.

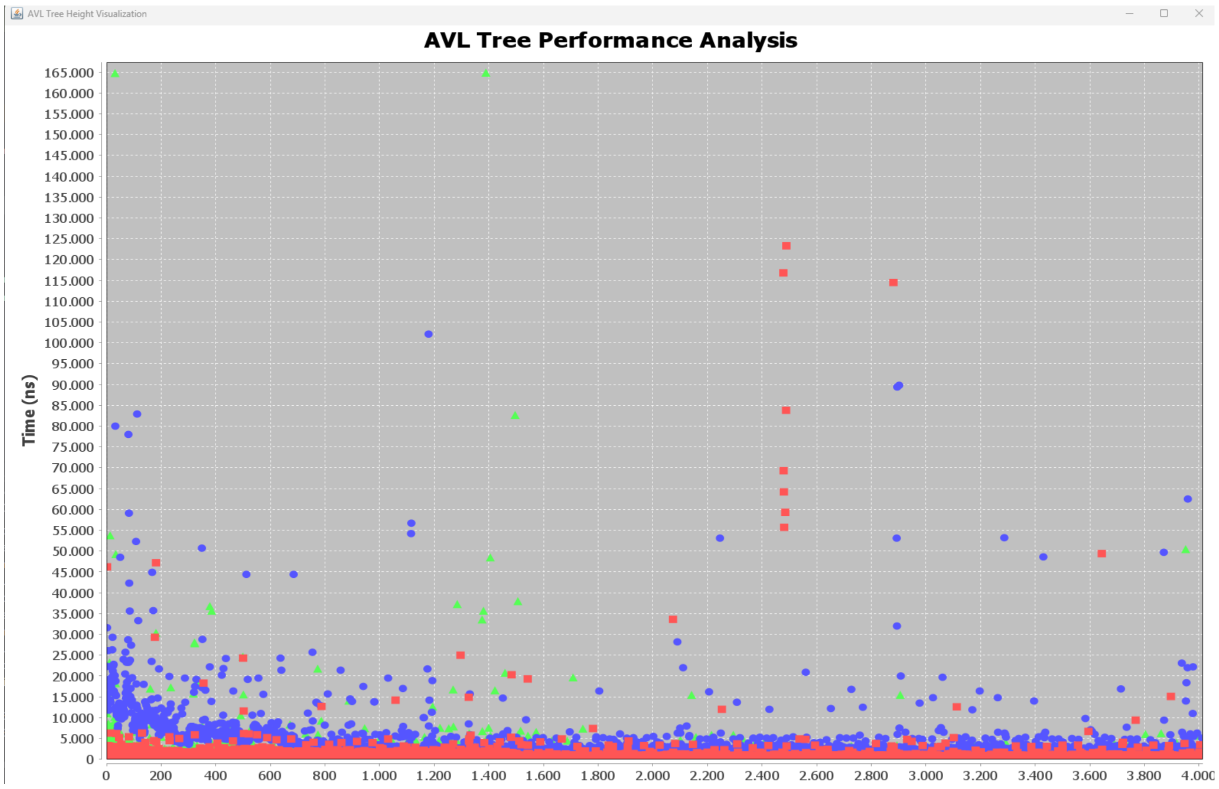
LEFT ROTATION

- It first assigns the right child of the given node (y) to a new node (x), and the left child of x to another new node (T2).
- It then makes y the left child of x, and T2 the right child of y.
- After the rotation, it updates the heights of y and x based on their respective left and right children. The height of a node is the maximum height of its left and right children plus 1.
- Finally, it returns x, which is now the root of the subtree after the rotation.

RIGHT ROTATION

- It first assigns the left child of the given node (y) to a new node (x), and the right child of x to another new node (T2).
- It then makes y the right child of x, and T2 the left child of y.
- After the rotation, it updates the heights of y and x based on their respective left and right children. The height of a node is the maximum height of its left and right children plus 1.
- Finally, it returns x, which is now the root of the subtree after the rotation.

GUI VISUALIZATION



In the latest version of my code, I didn't include this graph because I changed the graph library from the one provided in the source code to a different library, hoping to make the graph creation easier. Unfortunately, things didn't go as expected. Since the new library is external and assigment requires a makefile to run the program, I had to delete that part of the code. However, I kept a screenshot of my final graph, which I am sharing with you above. I tried logarithmic regression and various other techniques, but I couldn't achieve better results than this.

Recep Furkan Akın

210104004042