CSE 222 HW7 REPORT

Name : Yahya Kemal

Surname: Kuyumcu

ID : 220104004106

1.1)Program Structure

Classes:

AVLTree

GUIVisualization

randomInputGenerator

- RandomStringGenerator
- Stock
- StockDataManager

AVLTree:

The AVLTree class is a self-balancing binary search tree, where each node contains a Stock object. The key feature of an AVL tree is its ability to maintain its balance after insertions or deletions, ensuring that the tree remains optimally balanced at all times. This balance is maintained by tracking the height of each node and rebalancing the tree through rotations when the height difference between the left and right subtrees of any node exceeds 1. This self-balancing functionality guarantees that operations such as search, insertion, and removal of nodes are performed in O(logn) time complexity, making the AVL tree highly efficient for operations that involve frequent retrievals.

GUIVisualization:

The GUIVisualization class is responsible for visualizing the performance metrics of various operations on the AVL tree. It takes in data from the main class, which includes evaluation times for different operations, and generates graphical representations of these metrics. This class is crucial for providing a visual understanding of the performance characteristics of the AVL tree. It allows for easy comparison of operation

times and can help identify potential bottlenecks or areas for optimization in the AVL tree implementation. The visualization is achieved through the use of line graphs, where the x-axis represents the data points and the y-axis represents the time taken for each operation.

Stock:

The Stock class serves as a representation of a stock in the stock market. It encapsulates the key attributes of a stock, including its symbol, price, volume, and market capitalization. The symbol is a unique identifier for the stock, typically a string of characters that represents the company's ticker symbol. The price is a long value that represents the current trading price of the stock. The volume is a long value that represents the number of shares of the stock that have been traded during a given period. The marketcap is a long value that represents the total market value of all of a company's outstanding shares, calculated as the stock's current price times the total number of outstanding shares.

StockDataManager:

The StockDataManager class is responsible for managing the stock data within the application. It utilizes an AVLTree data structure to store and manage the stock data efficiently. The AVLTree, being a self-balancing binary search tree, ensures that operations such as insertion, deletion, and search are performed in O(logn) time complexity. This makes the StockDataManager class highly efficient for managing large volumes of stock data. The StockDataManager class provides methods for adding, updating, and removing stocks from the AVLTree, making it the primary interface for interacting with the stock data in the application.

RandomInputGenerator:

The RandomInputGenerator class is responsible for generating an input file that contains a series of command lines. These command lines correspond to operations such as ADD, REMOVE, SEARCH, or UPDATE that are to be performed on the AVL tree. The number of nodes and operations are specified by the user, and the class generates a text file named randominput.txt based on these inputs. Each line in the text file represents a single operation, and the entire file serves as the input for the main program. The RandomInputGenerator class is invoked before the main program, and the output it generates is used to create and manipulate the AVL tree in the main program. This class plays a crucial role in automating the process of testing the AVL tree with a variety of operations and data sizes.

RandomStringGenerator:

The RandomStringGenerator class is a utility class designed to generate random strings of a specified length. These strings are primarily used as stock symbols when performing add operations in the context of the stock management system. The class uses a combination of letters and numbers to generate unique identifiers for each stock. This is particularly useful in testing scenarios where a large number of unique stock symbols need to be generated on the fly. The class include methods to specify the length and character set of the generated strings, providing flexibility in the type of strings that can be generated.

Input Format:

My input format is generated randomly by the `RandomInputGenerator` class, which creates random command lines. When the program runs, it prompts the user for the number of nodes in the tree, and the number of add, remove, search, and update operations to be performed. Based on this information, a `randominput.txt` file is generated.

AVL TREE BALANCING :

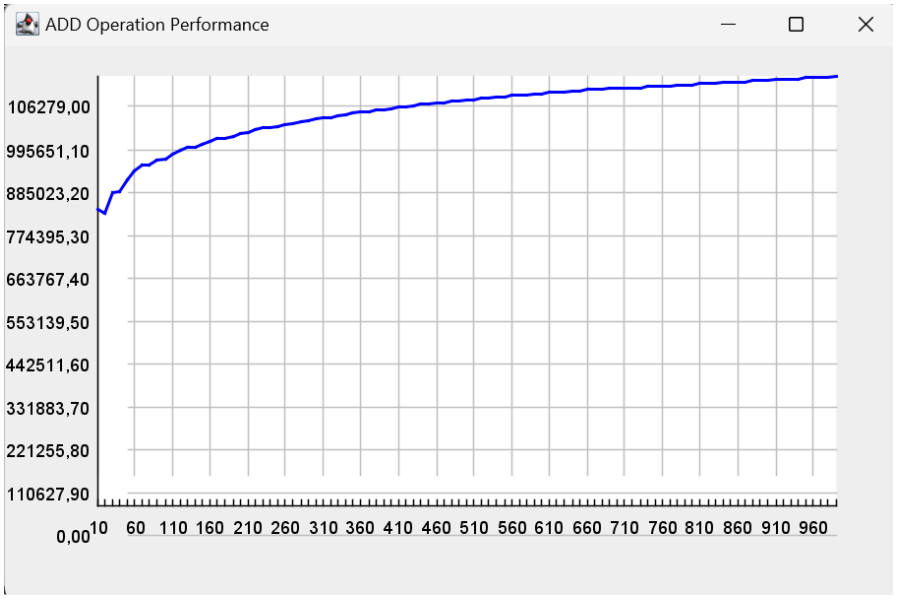
An AVL tree is a type of self-balancing binary search tree that ensures the tree remains balanced by maintaining a balance factor of -1, 0, or 1 for each node. This balance factor is the difference in heights between the left and right subtrees of a node. When an operation such as insertion or deletion makes the tree unbalanced (balance factor exceeds 1 or -1), rotations are performed to restore balance. There are four types of rotations: right rotation (to fix left-heavy imbalance), left rotation (to fix right-heavy imbalance), left-right rotation (a double rotation to fix left-right imbalance), and right-left rotation (a double rotation to fix right-left imbalance). These rotations ensure that the tree remains balanced, thereby maintaining efficient O(log n) performance for search, insertion, and deletion operations.

COLLECTING INPUTS FOR VISUALIZATION:

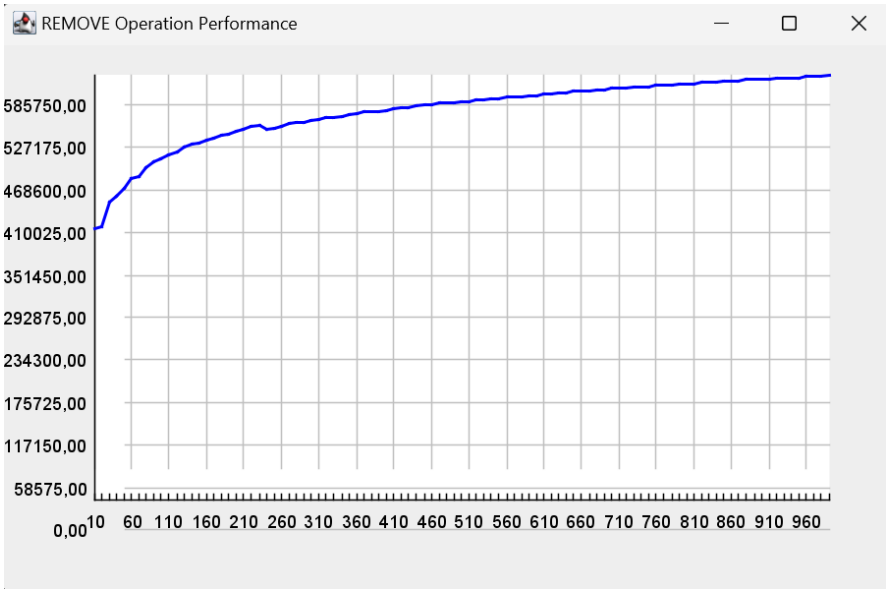
For each operation, I start by creating a new empty tree. Into this tree, I insert nodes based on pre-determined size values, which range from 10 to 1000, incrementing by 10. For each operation, I record the time values for a number of unique operations equal to the current size value. I then divide these time values by 10 and add the resulting values to a list that holds the time data for that specific operation. I repeat this process for three different operations, creating three separate lists for each size value. Finally, I send these lists to an instance of the `GUIVisualization` class

GRAPH OUTPUTS:

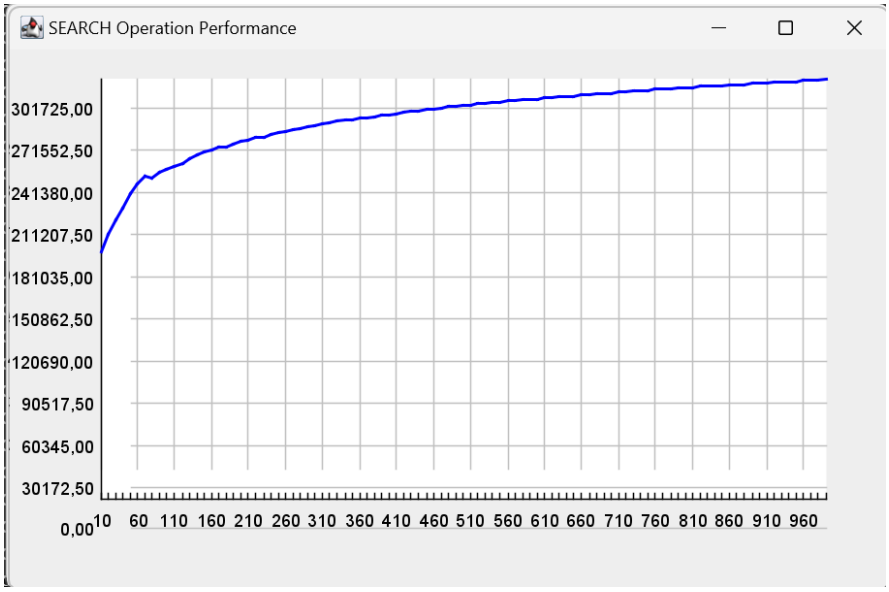
ADD GRAPH:



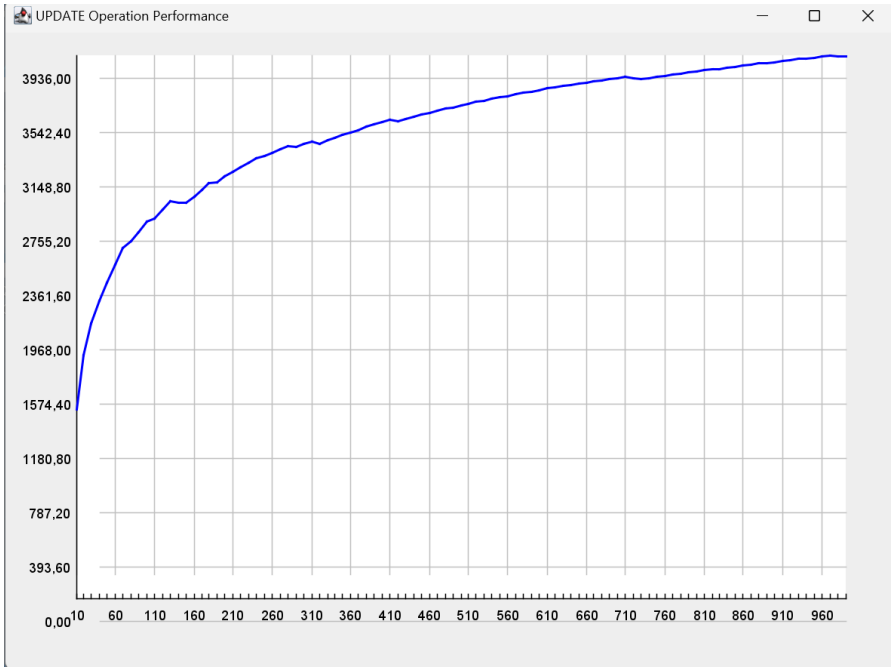
REMOVE GRAPH:



SEARCH OPERATION:



UPDATE OPERATION:



Problems I have faced and how i fixed them:

While working on the assignment, the biggest problem I encountered was generating the graphs with a log(n) scale. Although my AVLTree implementation was correct, the algorithms I initially used to collect the operation time values resulted in unexpected graphs. Initially, I was performing all operations within the same loop on a populated tree and recording specific time values. However, these operations were affecting each other's times, leading to distorted graphs.

To address this, the first change I made was to separate each operation. This resulted in more meaningful graphs, but they were still not as expected. I then created new trees from scratch for each operation and filled them according to a specified size value before performing the operations and recording the time values. This produced more acceptable graphs, but some data points were still significantly off.

Finally, I decided to perform each operation multiple times and take the average of the times. After placing these average values into lists specific to each operation, the resulting graphs resembled the expected log(n) shape more closely. However, there were still some outlier values distorting the graphs. To eliminate these outliers, I implemented a method to calculate the average difference for each list and used 1.1

times this average difference as a threshold to normalize the outlier data. This approach led to the graphs presented in the report.