# CSE222 Homework 7: Report
# Emre Kibar – 210104004093

1. **AVL Tree Implementation**
   Key Methods and Their Functionalities

   1. Node Class
      The Node class represents the individual nodes in the AVL tree. Each node stores a Stock object, its height, and references to its left and right children. The height of a new node is initialized to 1.
   2. Insertion

      - `insert(Stock stock)`: Public method to insert a new stock into the AVL tree. Calls the recursive helper method `insert(Node node, Stock stock)`.
      - `insert(Node node, Stock stock)`: Recursively finds the correct position for the new stock based on its symbol. Updates the height of the current node after insertion. Balances the subtree rooted at the current node if necessary. Balancing involves four cases: left-left, right-right, left-right, and right-left.

   3. Deletion

      - `delete(String symbol)`: Public method to delete a stock with the specified symbol from the AVL tree. Calls the recursive helper method `delete(Node node, String symbol)`.
      - `delete(Node node, String symbol)`: Recursively finds the node to be deleted based on the symbol. Handles cases where the node to be deleted has one or zero children. Finds the successor if the node to be deleted has two children. Updates the height of the current node after deletion. Balances the subtree rooted at the current node if necessary.

   4. Searching

      - `search(String symbol)`: Public method to search for a stock with the specified symbol. Calls the private helper method `search(Node node, String symbol)`.
      - `search(Node node, String symbol)`: Recursively searches for the stock in the subtree which is rooted at the given node.

   5. Balancing the Tree

      - `getBalance(Node node)`: Computes the balance factor of the given node by subtracting the height of the right subtree from the height of the left subtree.
      - `height(Node node)`: Computes the height of the given node.
      - `rightRotate(Node node)`: Performs a right rotation on the given node to correct a left-heavy imbalance.

- leftRotate(Node node): Performs a left rotation on the given node to correct a right-heavy imbalance.
- Balancing Cases:
    - Left-Left Case
    - Right-Right Case
    - Left-Right Case
    - Right-Left Case

6. Tree Traversal Methods
    - preOrder(Node node): Performs a pre-order traversal starting from the given node.
    - inOrder(Node node): Performs an in-order traversal starting from the given node.
    - postOrder(Node node): Performs a post-order traversal starting from the given node.

Balancing the Tree
Balancing an AVL tree ensures that the height difference between the left and right subtrees of any node is at most one. This balance is achieved through rotations:

- Right Rotation
- Left Rotation
- Double Rotations

**2. Input File Format and Processing the Commands Input File Format**

The input file contains a list of randomized commands, each specifying an operation on stock data. Each line in the input file represents a single command to be executed. The possible operations are ADD, REMOVE, SEARCH, and UPDATE. Each command follows a specific format, as described below:

1. ADD Operation
    - Format: ADD <symbol> <price> <volume> <marketCap>
    - Example: ADD ABC 123.45 1000000 500000000
2. REMOVE Operation
    - Format: REMOVE <symbol>
    - Example: REMOVE ABC
3. SEARCH Operation
    - Format: SEARCH <symbol>
    - Example: SEARCH ABC

4. UPDATE Operation

- Format: UPDATE <symbol> <price> <volume> <marketCap>
- Example: UPDATE ABC 543.21 2000000 800000000

Command Processing

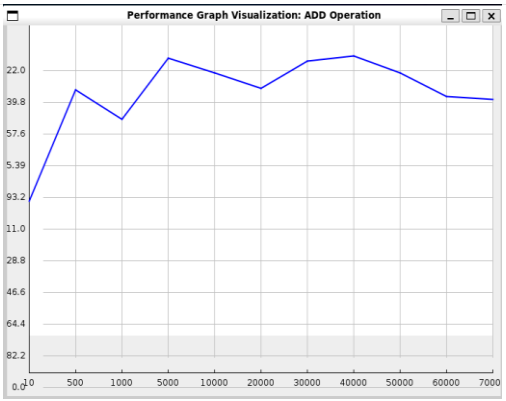
5. Reading the Input File

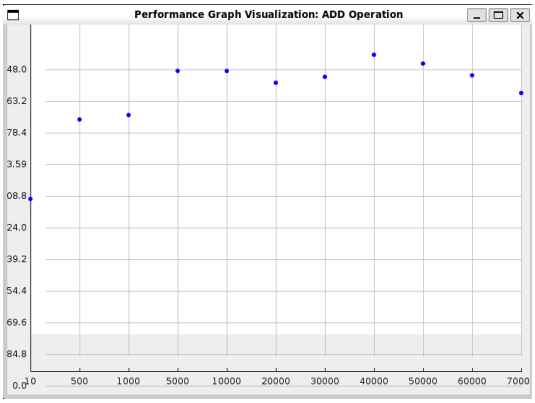
- The method `readFileAndInitializeTree(manager)` reads the input file line by line.
- Each line is passed to the `processCommand(line, manager)` method for further processing.

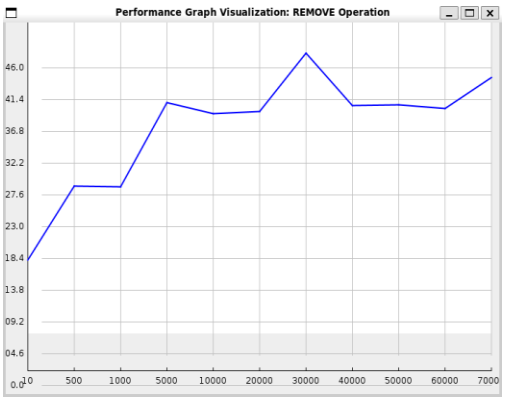
6. Processing Each Command

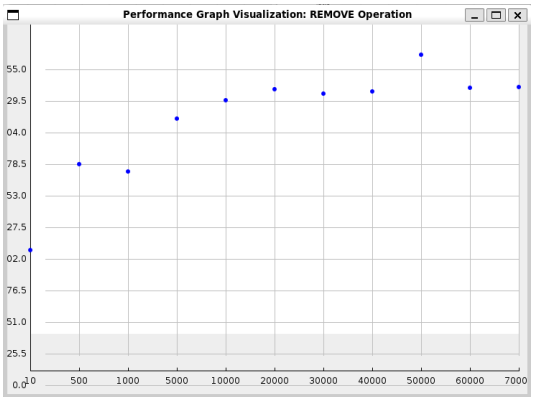
- The `processCommand(line, manager)` method splits the line into tokens to determine the command and its parameters.
- Based on the command (ADD, REMOVE, SEARCH, UPDATE), it calls the appropriate method on the StockDataManager instance (manager).

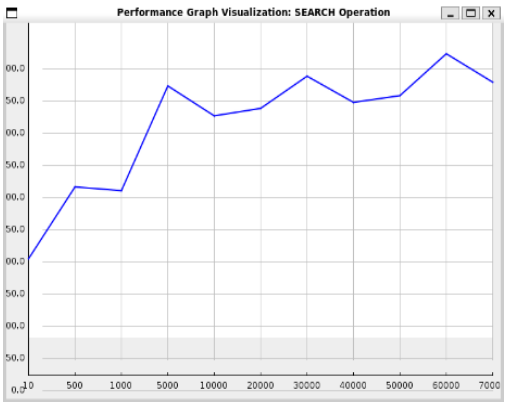
## 3. Performance Analysis

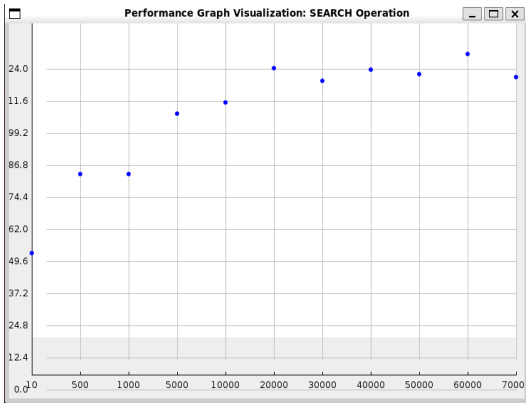


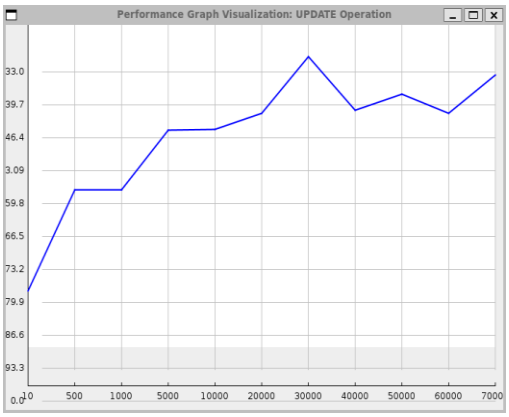



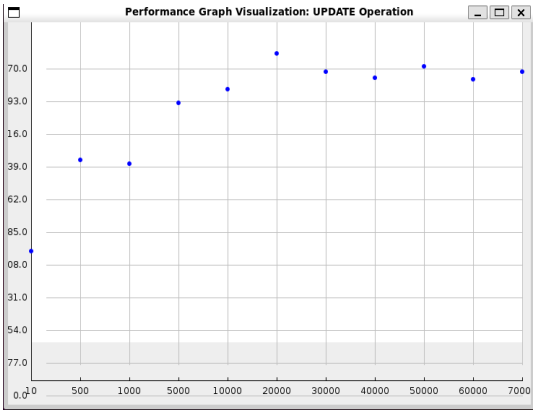






The performance of all operations must have logarithmic graphs theoretically because we are using an AVL Tree to hold the stocks which balances itself automatically after each operation. That's why, the operations have O(logn) complexity. In the graphs on the upside, shows the graphs of operations and they are showing logarithmic graph characteristics as expected. So, we can validate the balance system of AVL Tree.

## 4. Challenges on Homework

- When I am creating a random command for the inputFile with a large initial node size, the size of the "inputFile.txt" file gets larger and stops generating more commands. To prevent that I increment the rate of ADD comment while generating random comments.
- To make the compilation faster, for the search operation if the searched symbol is found, I am not printing the searched stock to terminal because it makes the compilation slower.
- At first, when I am trying to get the graph of operations, the program creates irrelevant graphs that don't match with the expected graphs. Probably, it is caused by the Java optimization during compilation. To eliminate the Java optimization, I am using "-Xİnt" command and with this way I get expected graphs.