Project Report: Stock Management System

Implementation Overview

Main.java: Responsible for reading input data from a file, executing stock management operations, and visualizing performance metrics using a graphical interface.

GUIVisualization.java: Manages the graphical representation of performance data through graphs using Java Swing. This class provides functionalities for drawing axes, grid lines, data points, and graph lines or scatter plots.

AVLTree.java: Implements the AVL tree data structure, which is a self-balancing binary search tree. It ensures logarithmic time complexity for search, insertion, and deletion operations by maintaining the balance factor of its nodes.

Stock.java: Represents a stock entity with attributes such as symbol, price, volume, and market capitalization. This class provides methods for setting and retrieving stock attributes.

StockDataManager.java: Facilitates operations related to stock data manipulation, including addition, removal, search, update, retrieval of tree size, and tree clearing.

Design Decisions

Use of AVL Tree

Ensures that the tree remains balanced, preventing skewness and maintaining efficient search, insertion, and deletion operations with a worst-case time complexity of O(log n).

File Input

Input data is read from a text file (input.txt), which enhances the system's flexibility by allowing easy integration with external sources of stock data. This approach simplifies data input and testing processes. InputGenerator.java file generates random ADD, UPDATE, SEARCH, REMOVE commands. (each 1000 of them respectively.)

AVL Tree Balancing

Implemented AVL tree balancing algorithms, including rotations, to maintain the balance factor of nodes within the tree. This ensures that the tree remains balanced after insertions and deletions, thereby optimizing search, insertion, and deletion operations.

Performance Analysis

Utilized Java's System.nanoTime() method to accurately measure the execution time of AVL tree operations. By recording input sizes and corresponding running times, performance analysis graphs were generated to visualize the system's efficiency and scalability across different operations.

The performance analysis graphs include:

Add Operation: Graph illustrating the relationship between input size (number of stocks added) and the corresponding running time for the add operation.

Remove Operation: Graph showing the impact of input size on the running time for the remove operation.

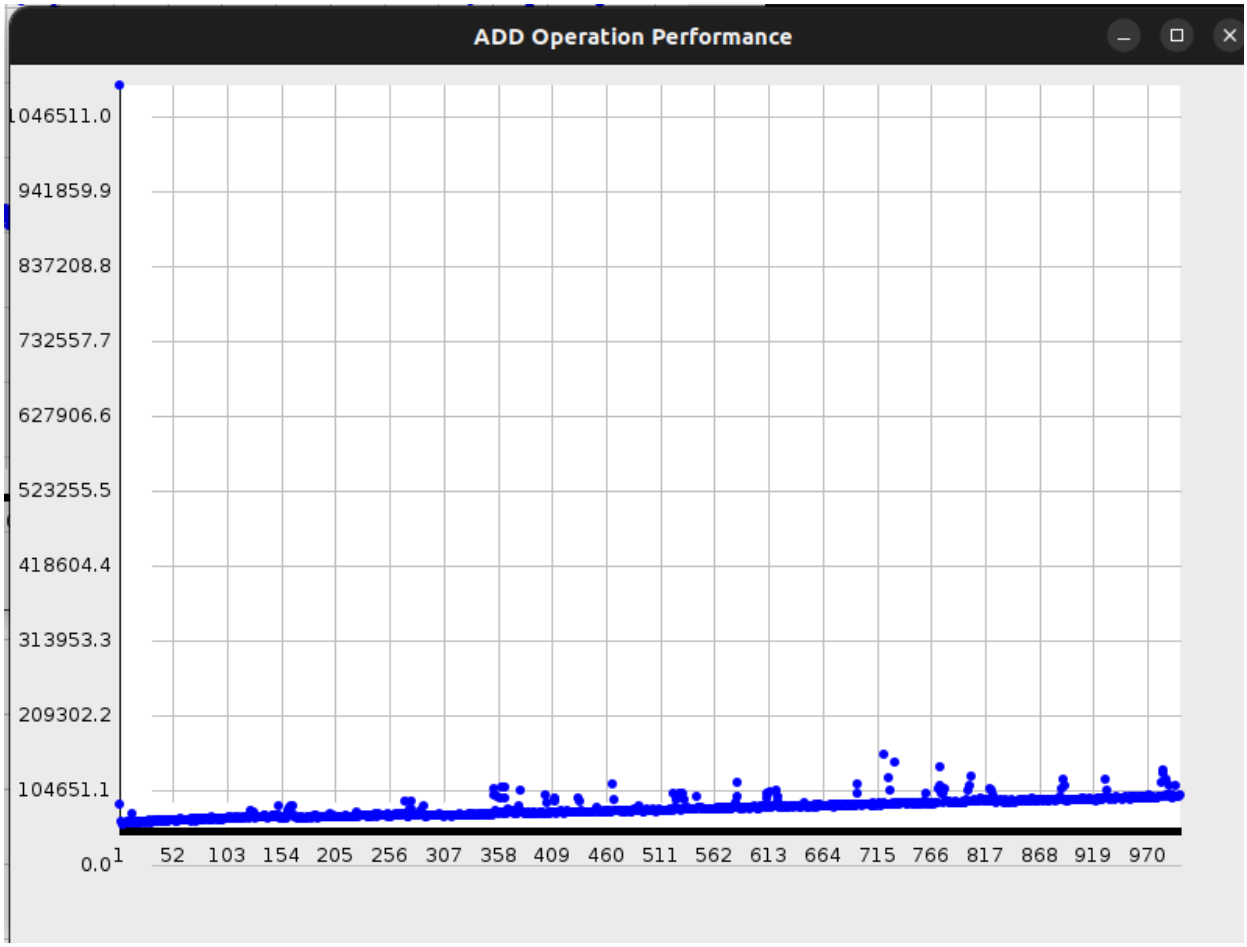
Search Operation: Performance graph depicting the efficiency of the search operation in relation to input size.

Update Operation: Graph demonstrating the running time of the update operation as a function of input size.

Graph Outputs

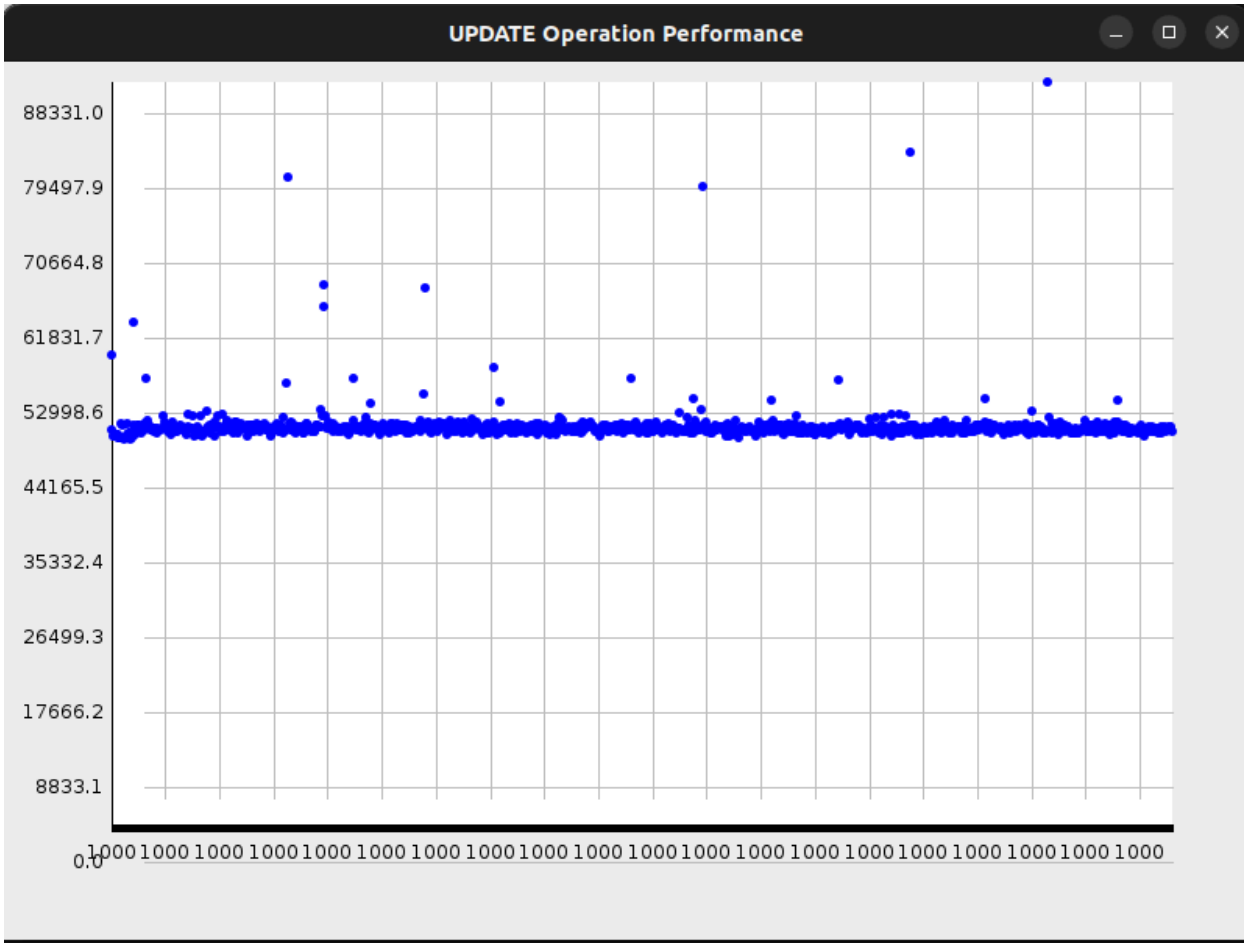
ADD

(1000 ADD operation). I have tried many things to optimize it but the first plot always becomes the biggest and ruins the graph.

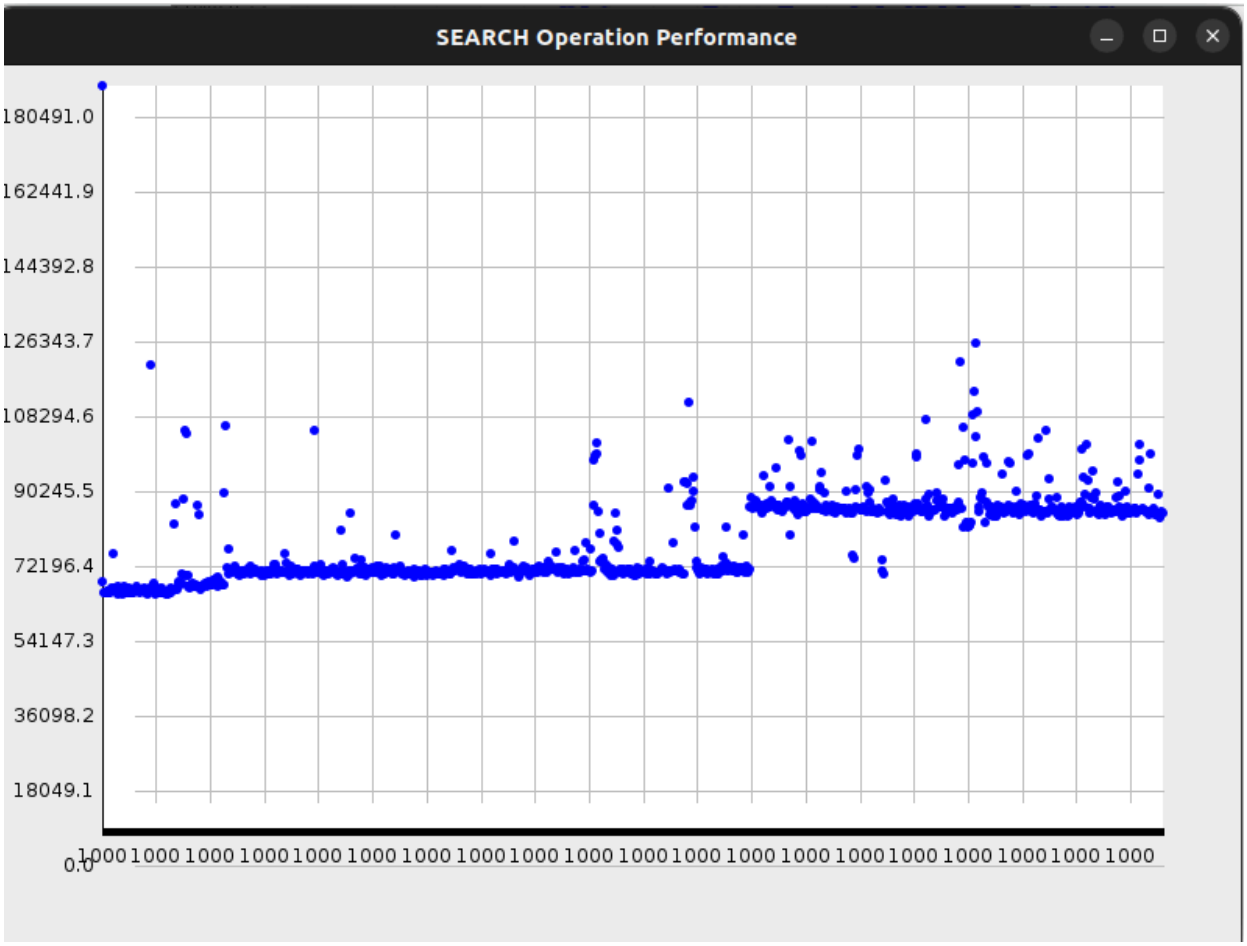


UPDATE

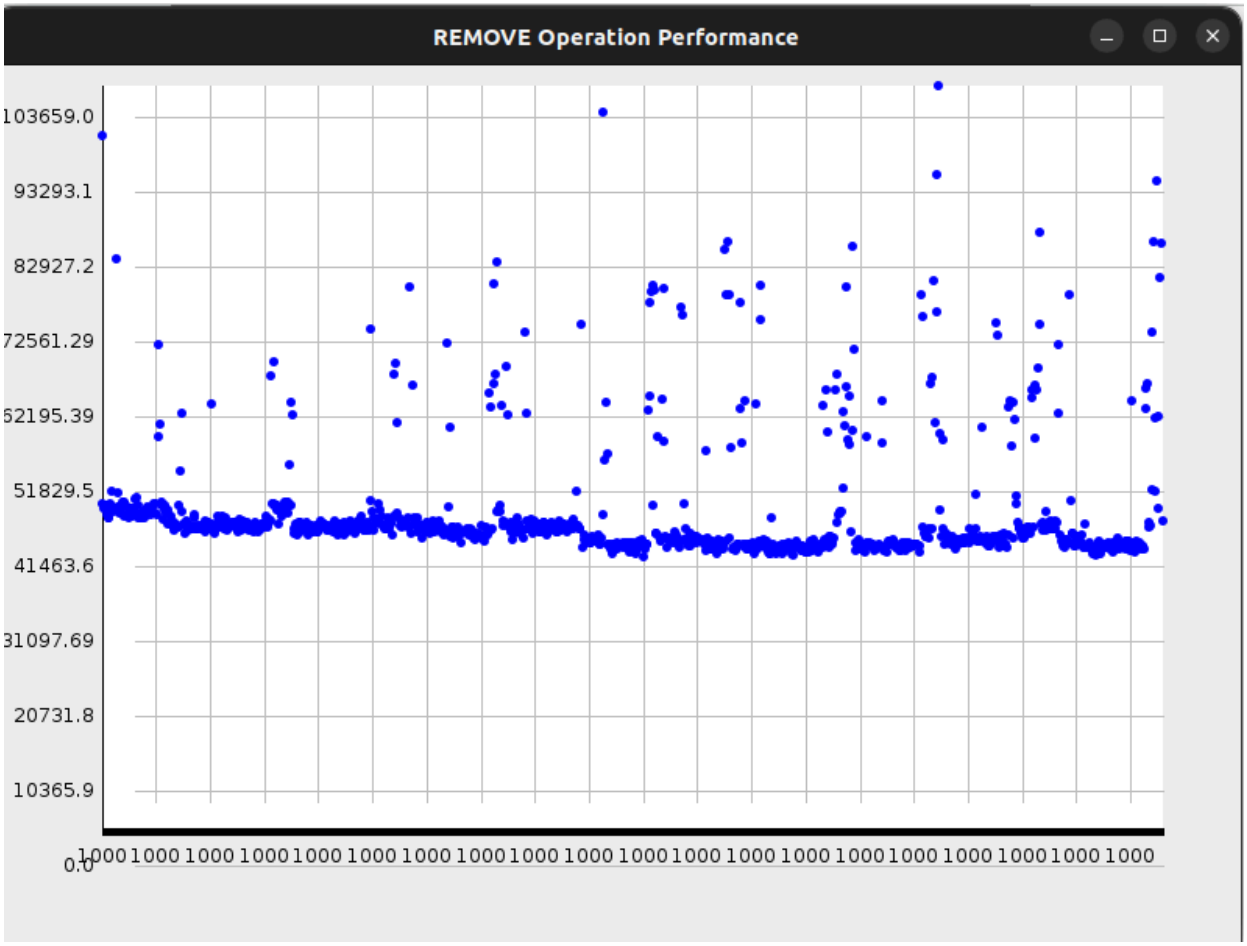
(1000 UPDATE operation after ADD operations):


UPDATE Operation Performance
88331.0
79497.9
70664.8
61831.7
52998.6
44165.5
35332.4
26499.3
17666.2
8833.1
0.0
1000 1000 1000 1000 1000 1000 1000 1000 1000 1000 1000 1000 1000 1000 1000 1000 1000 1000 1000 1000

SEARCH


SEARCH Operation Performance
180491.0
162441.9
144392.8
126343.7
108294.6
90245.5
72196.4
54147.3
36098.2
18049.1
0.0

REMOVE



Bedirhan Ömer Aksoy

200104004074