CSE222 HW7 – Balanced Tree-based Stock Data Management Report

Arife Yurtseven

210104004294

1.Introduction

In this assignment, I have defined a Java application that uses AVL trees to manage and analyze stock data. The application processes commands read from a file to perform stock addition, deletion, updating, and searching operations. Additionally, in another part of my assignment, I determine the average time it takes to perform 100 operations (add, remove, search, and update) on randomly generated 1000 operations. Furthermore, I analyze its performance by determining the average time it takes to perform 100 operations (add, remove, search, and update) on operations ranging from 5000 to 9000... and visualize it graphically.

2. Fundamentals of AVL Trees

AVL trees are self-balancing binary search trees where the height difference between the left and right subtrees of each node is at most 1. This feature makes AVL trees balanced and efficient. Basic operations include insertion, deletion, update, and search, all of which have a time complexity of O(log n).

3.Class Hierarchy and Design

There are three main classes in the assignment:

- AVLTree.java
- Stock.java
- StockDataManager.java
- InputGenerator

For graphical visualization, there is:

- GUIVisualization.java

Additionally, there is the main section:

Main.java

Stock.java

This Stock class is used to store and access data related to stocks (symbol, price, volume, market Cap).

Getter method(getSymbol, getPrice, getVolume, getMarketCap):

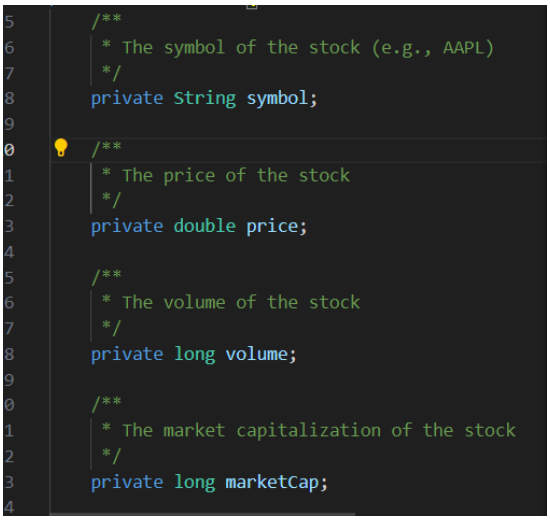
- Returns the value of the relevant member variable.

Setter methods (setSymbol, setPrice, setVolume, setMarketCap):

- Updates the value of the relevant member variable with the new value given as its parameter.

The toString method provides a readable representation of the Stock object

- return "Stock{" + "symbol='" + symbol + '\'' + ", price=" + price + ", volume=" + volume + ", marketCap=" + marketCap + '}';:



StockDataManager.java

This class performs operations such as adding, updating, removing, searching stocks, and displaying the structure of the AVL tree.

The StockDataManager class manages stock information using AVL tree.

private AVLTree avlTree: AVL tree used to store and manage stocks.

public void addOrUpdateStock(String symbol, double price, long volume, long marketCap):

Creates a new Stock object. Adds or updates this stock to the AVL tree.

public void removeStock(String symbol):

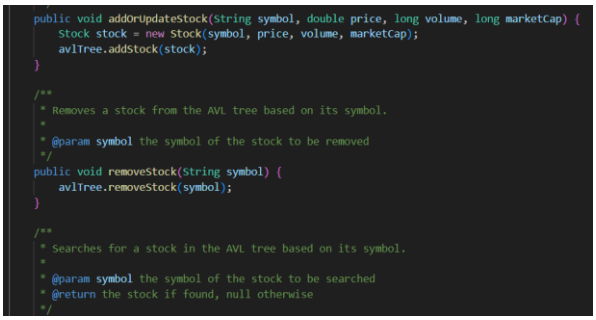
Removes the stock with the specified symbol from the AVL tree.

public Stock searchStock(String symbol):

Searches the AVL tree for the stock with the specified symbol. Returns the stock found, null if not found.

public void updateStock(String oldSymbol, String newSymbol, double price, long volume, long marketCap):

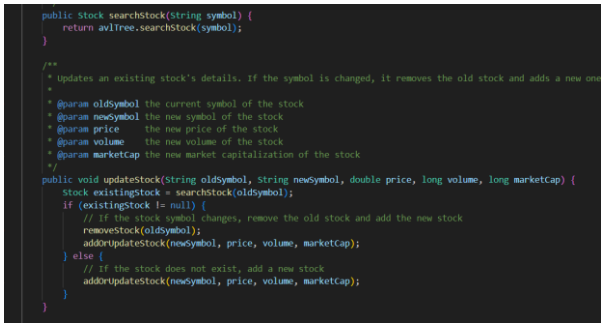
Updates the details of an existing stock. If the stock is found, it removes the stock with the old symbol. It adds the stock with the new symbol and updated values. If the stock is not found, it adds a new stock.



```
public void addOrUpdateStock(String symbol, double price, long volume, long marketCap) {
    Stock stock = new Stock(symbol, price, volume, marketCap);
    avlTree.addStock(stock);
}

/**
 * Removes a stock from the AVL tree based on its symbol.
 *
 * @param symbol the symbol of the stock to be removed
 */
public void removeStock(String symbol) {
    avlTree.removeStock(symbol);
}

/**
 * Searches for a stock in the AVL tree based on its symbol.
 *
 * @param symbol the symbol of the stock to be searched
 * @return the stock if found, null otherwise
 */
```



```
public Stock searchStock(String symbol) {
    return avlTree.searchStock(symbol);
}

/**
 * Updates an existing stock's details. If the symbol is changed, it removes the old stock and adds a new one
 *
 * @param oldSymbol the current symbol of the stock
 * @param newSymbol the new symbol of the stock
 * @param price     the new price of the stock
 * @param volume    the new volume of the stock
 * @param marketCap the new market capitalization of the stock
 */
public void updateStock(String oldSymbol, String newSymbol, double price, long volume, long marketCap) {
    Stock existingStock = searchStock(oldSymbol);
    if (existingStock != null) {
        // If the stock symbol changes, remove the old stock and add the new stock
        removeStock(oldSymbol);
        addOrUpdateStock(newSymbol, price, volume, marketCap);
    } else {
        // If the stock does not exist, add a new stock
        addOrUpdateStock(newSymbol, price, volume, marketCap);
    }
}
```

public void displayInOrder():

Displays the AVL tree in-order. Prints the AVL tree in order.

public void displayPreOrder():

Displays the AVL tree root-first (pre-order). Prints the AVL tree root-first.

public void displayPostOrder():

Displays the AVL tree in post-order. Prints the AVL tree in post-order.

public void printTree():

Visually prints the structure of the AVL tree.

```
     */
    public void displayInOrder() {
        avlTree.printInOrder();
    }

    /**
     * Displays the AVL tree in a pre-order traversal.
     */
    public void displayPreOrder() {
        avlTree.printPreOrder();
    }

    /**
     * Displays the AVL tree in a post-order traversal.
     */
    public void displayPostOrder() {
        avlTree.printPostOrder();
    }

    /**
     * Prints the structure of the AVL tree.
     */
    public void printTree() {
        avlTree.printTree();
    }
```

AVLTree.java

In this class, I manage stock information using the AVL tree data structure. The main functions of this class are as follows:

Stock Addition: New stocks can be added to the AVL tree using the addStock method. During the addition process, the tree is balanced, and the height of each node is updated.

Stock Removal: A specific stock can be removed from the tree based on its symbol using the removeStock method. The tree is automatically balanced after the removal operation.

Stock Search: Stocks can be searched based on their symbol using the searchStock method. If the stock is found, the corresponding stock object is returned.

Updating Stock Information: The updateStock method is used to update the symbol of an existing stock. If the symbol changes, the old stock is removed, and the new stock is added.

Viewing the Tree: The displayInOrder, displayPreOrder, and displayPostOrder methods traverse the AVL tree in in-order, pre-order, and post-order respectively, and print it to the console.

Viewing the Tree Structure: The printTree method prints the structure of the tree to the console in a visual representation showing the relationships between nodes.

private int height(Node N): Returns the height of the specified node. If the node is empty, it returns 0.

private Node rightRotate(Node y): Rotates the specified subtree to the right and returns the new root node.

private Node leftRotate(Node x): Rotates the specified subtree to the left and returns the new root node.

```
    private Node rightRotate(Node y) {
        Node x = y.left;
        Node T2 = x.right;

        x.right = y;
        y.left = T2;

        y.height = max(height(y.left), height(y.right)) + 1;
        x.height = max(height(x.left), height(x.right)) + 1;

        return x;
    }

    /**
     * Performs a left rotation on the specified subtree.
     *
     * @param x the root of the subtree to be rotated
     * @return the new root of the rotated subtree
     */
    private Node leftRotate(Node x) {
        Node y = x.right;
        Node T2 = y.left;

        y.left = x;
        x.right = T2;

        x.height = max(height(x.left), height(x.right)) + 1;
        y.height = max(height(y.left), height(y.right)) + 1;

        return y;
```

Balancing is performed by rotations to preserve the structure of the AVL tree. The AVL tree ensures that the height of each node is balanced; That is, the height difference of the right and left subtrees of any node must be at most 1.

Height Update: The height of each node is updated according to the height of its subtrees. Height information is important to check the stability of the AVL tree.

Balance Factor Calculation: The balance factor of each node is calculated as the difference between the height of the right subtree and the height of the left subtree.

Balance Control: The balance factor determines the balance state of any node. If the balance factor is -1, 0 or 1, the tree is in balance. However, if the balance factor is less than -1 or greater than 1, the tree is unstable and requires balancing.

Rotations: Appropriate rotations are performed depending on the balance factor and the position of the node. There are two basic types of turns: right turns and left turns.

Right Rotation: If the height of the left subtree is 2 or more than the height of the right subtree, right rotation is performed. This allows nodes to be shifted to the right.

Left Rotation: If the height of the right subtree is 2 or more than the height of the left subtree, left rotation is performed. This allows nodes to be shifted to the left.

Height Update After Rotation: After the rotation, the heights of the relevant nodes are recalculated and updated.

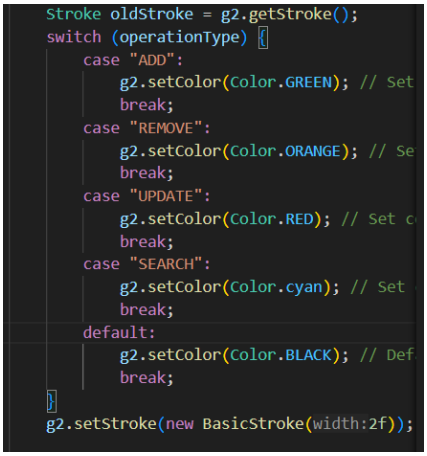
These steps outline the balancing operations performed to maintain the balance of the AVL tree. In this way, the AVL tree always remains in a balanced structure and optimum performance is achieved in search, insertion and deletion operations.

```
122     int balance = getBalance(node);
123
124     if (balance > 1 && stock.getSymbol().compareTo(node.left.stock.getSymbol()) < 0)
125         return rightRotate(node);
126
127     if (balance < -1 && stock.getSymbol().compareTo(node.right.stock.getSymbol()) > 0)
128         return leftRotate(node);
129
130     if (balance > 1 && stock.getSymbol().compareTo(node.left.stock.getSymbol()) > 0) {
131         node.left = leftRotate(node.left);
132         return rightRotate(node);
133     }
134
135     if (balance < -1 && stock.getSymbol().compareTo(node.right.stock.getSymbol()) < 0) {
136         node.right = rightRotate(node.right);
137         return leftRotate(node);
138     }
```

GUIVisualization.java

The GUIVisualization class provides a tool for visualizing performance chart data. You can plot data points on a graph or create a graph as a line if you prefer. As data points are added or updated, the graph is dynamically updated and displayed in different colors according to various types of operations (add, remove, update, search). Data points are plotted on the graph according to locations specified on the x and y axes. The chart is automatically scaled and plotted according to the size of the window and the values of the data points.

In my chart, I draw 4 separate graphs showing the ADD operation with green dots, the REMOVE operation with orange dots, the UPDATE operation with red dots, and the SEARCH operation with black dots.



InputGenerator.java

SYMBOLS: This array I specified the stock symbols to be used to generate random commands

main() Method: Commands are created in the random_input.txt file in the range of 0 to 100.

generateRandomCommand() Method: This method generates random stock management commands. It randomly selects a command type and symbol, then creates a command based on that information. Command types can be "ADD", "REMOVE", "SEARCH" and "UPDATE". The "ADD" command adds stock, the "REMOVE" command removes stock, the "SEARCH" command searches for stock and the "UPDATE" command updates stock. It generates random parameters according to each command type and combines them according to the command format.

Main.java

main() Metodu: This method reads and processes stock commands from an input file received from the command line and visualizes performance. First, it checks the required arguments and then reads the input file. The processCommand() method is called to process each row. Next, a set of StockDataManager instances are created and populated with stocks of a certain size. Finally, performance analysis is performed using the performAnalysis() method and the results are visualized within the GUIVisualization framework.

processCommand() Metodu: This method processes a command line and performs the relevant inventory transaction. Depending on the command type, the corresponding methods in the StockDataManager class are called: addOrUpdateStock(), removeStock(), searchStock() and updateStock().

performPerformanceAnalysis() Metodu: This method performs performance analysis for a specific StockDataManager and transaction type. Performs a specified number of stock operations: adding stock, removing stock, updating stock, or searching .

stock.**performAnalysis() Metodu:** This method takes a set of occupied stock managers and a transaction type, then analyzes the performance and visualizes it in a chart. For a certain transaction type, 100 transactions are performed, these transactions are performed 50 times (to give more accurate results) and the average duration of these transactions is calculated. The results are added to the GUIVisualization framework as data points representing the average processing time for a given tree size.

```
private static void performAnalysis(GUIVisualization frame, StockDataManager[] managers, String operation) {
    for (int i = 0; i < TREE_SIZES.length; i++) {
        long totalTime = 0;

        for (int j = 0; j < TEST_ITERATIONS; j++) {
            long startTime = System.nanoTime();
            performPerformanceAnalysis(managers[i], OPERATIONS_COUNT, operation);
            long endTime = System.nanoTime();
            long averageTime = (endTime - startTime) / OPERATIONS_COUNT;
            totalTime += averageTime;
        }

        frame.addDataPointX(TREE_SIZES[i]);
        frame.addDataPointY(totalTime / TEST_ITERATIONS);
    }
}
```

displayTree() Metodu: This method displays the tree structure and trees (pre-order and post-order, respectively) of a given StockDataManager instance.

```
private static void displayTree(StockDataManager manager) {
    manager.printTree();
    System.out.println(x:"Display in-order:");
    manager.displayInOrder();
    System.out.println(x:"Display pre-order:");
    manager.displayPreOrder();
    System.out.println(x:"Display post-order:");
    manager.displayPostOrder();
}
```

Challenges and solutions I encountered:

Frankly, I had a little difficulty in understanding the homework. To be honest, I could only understand later that we were trying to find the time required to perform operations such as adding or subtracting data to an already full file and placing it in the graph, but I was not very clear on this issue. I had a hard time finding the right points on the graph and getting a logarithmic graph. Because it did not give correct data in Windows or on my own computer. I got much more accurate results when I tried it from Ubuntu. Also, while writing AVL Tree, I had a hard time updating because that part was not working, the tree was being re-formed, it was not updating correctly, so I used resources on the internet.

Since I did not read the PDF in detail, there were situations where I had to change the code at the last minute. For example, I was always performing the operations in my already existing inout txt file, but I realized later that I should have created a random file instead of that and printed the operations there in the terminal. Finally, in order to get accurate results regarding the graphics, I managed to get better results by adding 100 commands to a tree with 1000 commands 50 times and taking the average of this.

OUTPUTS

Random_input.txt

```
ADD AA 26.65 148319 296358945
UPDATE TK TKU 35.96 9002 22328
SEARCH FB
UPDATE AA AAU 96.35 561 78374
UPDATE AA AAU 7.58 2622 76730
UPDATE BAR BARU 29.08 9041 84189
REMOVE AA
REMOVE TK
REMOVE GS
SEARCH FB
ADD MS 2.83 681546 921535214
ADD FB 69.82 190465 759525465
REMOVE TESLA
ADD MS 67.99 421093 863770077
REMOVE TK
UPDATE FB FBU 37.18 3102 52645
REMOVE YBB
REMOVE MS
UPDATE MS MSU 13.31 4563 11343
REMOVE BAR
REMOVE AA
ADD MS 63.21 454707 802523646
ADD YBB 28.06 551055 895663800
SEARCH YBB
UPDATE MS MSU 84.90 2977 46721
```

```
SEARCH YBB
UPDATE MS MSU 84.90 2977 46721
UPDATE TK TKU 43.96 9513 22117
SEARCH MS
REMOVE BAR
REMOVE BAR
ADD TK 84.06 488957 959551514
SEARCH FB
REMOVE MS
SEARCH YBB
ADD FB 83.53 110561 612692653
SEARCH MS
SEARCH GS
UPDATE TESLA TESLAU 43.32 8088 18241

```

Terminal

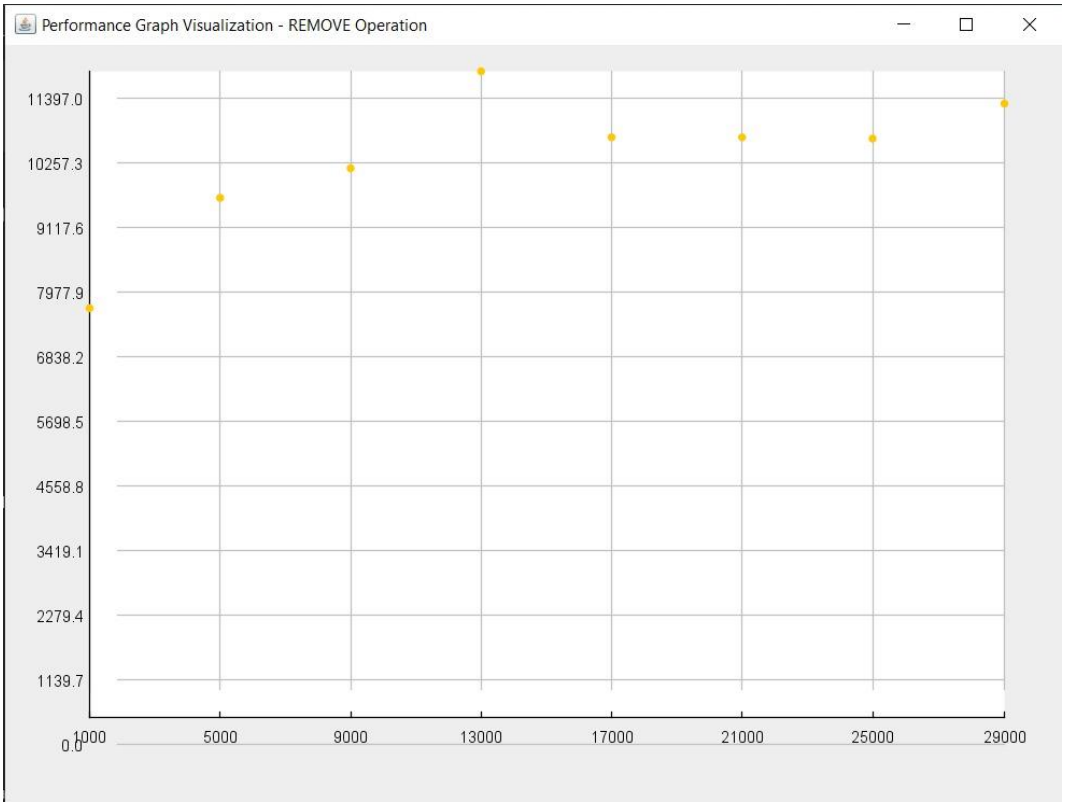
```
java InputGenerator random_input.txt
Random commands generated: 37
java -Xint Main random_input.txt
Added: AA
Updated: TK to TKU
Searching for: FB
Stock not found: FB
Updated: AA to AAU
Updated: AA to AAU
Updated: BAR to BARU
Removed: AA
Removed: TK
Removed: GS
Searching for: FB
Stock not found: FB
Added: MS
Added: FB
Removed: TESLA
Added: MS
Removed: TK
Updated: FB to FBU
Removed: YBB
Removed: MS
Updated: MS to MSU
Removed: BAR
Removed: AA
Added: MS
Added: YBB
Searching for: YBB
Stock{symbol='YBB', price=28.06, volume=551055, marketCap=895663800}
Updated: MS to MSU
Updated: TK to TKU
Searching for: MS
Stock not found: MS
Removed: BAR
Removed: BAR
Added: TK
Searching for: FB
```

```
Searching for: FB
Stock not found: FB
Removed: MS
Searching for: YBB
Stock{symbol='YBB', price=28.06, volume=551055, marketCap=895663800}
Added: FB
Searching for: MS
Stock not found: MS
Searching for: GS
Stock not found: GS
Updated: TESLA to TESLAU
R---- FBU (4)
   L---- BARU (2)
   |  L---- AAU (1)
   |  R---- FB (1)
   R---- TKU (3)
      L---- TESLAU (2)
      |  L---- MSU (1)
      |  R---- TK (1)
      R---- YBB (1)
```

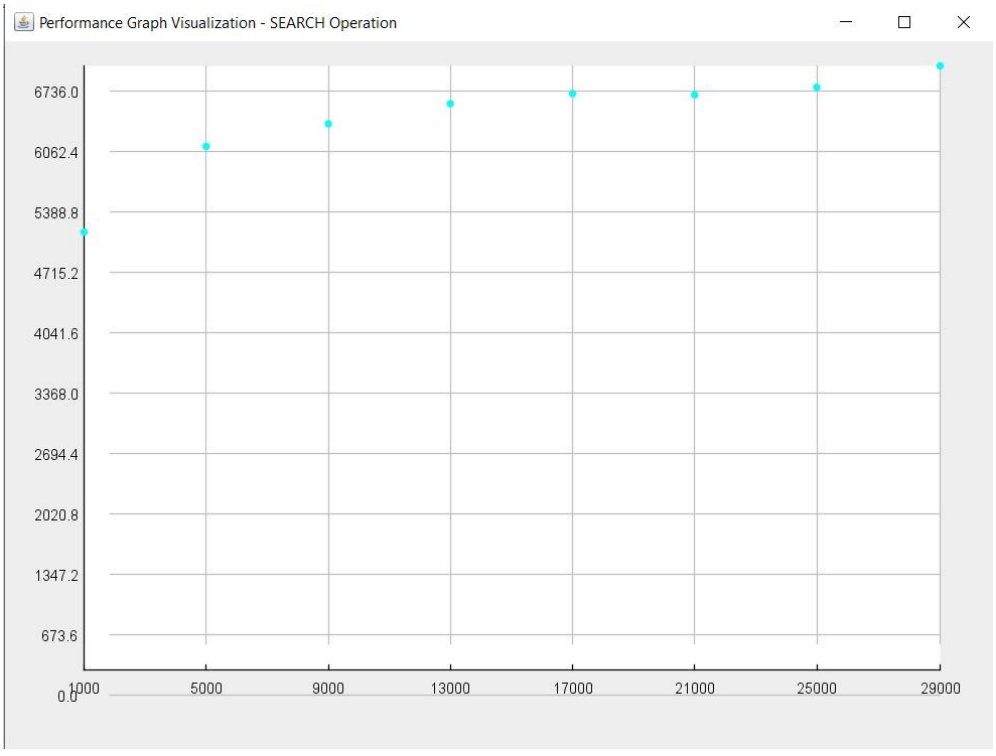
,

```
Display in-order:
Stock{symbol='AAU', price=7.58, volume=2622, marketCap=76730}
Stock{symbol='BARU', price=29.08, volume=9041, marketCap=84189}
Stock{symbol='FB', price=83.53, volume=110561, marketCap=612692653}
Stock{symbol='FBU', price=37.18, volume=3102, marketCap=52645}
Stock{symbol='MSU', price=84.9, volume=2977, marketCap=46721}
Stock{symbol='TESLAU', price=43.32, volume=8088, marketCap=18241}
Stock{symbol='TK', price=84.06, volume=488957, marketCap=959551514}
Stock{symbol='TKU', price=43.96, volume=9513, marketCap=22117}
Stock{symbol='YBB', price=28.06, volume=551055, marketCap=895663800}
Display pre-order:
Stock{symbol='FBU', price=37.18, volume=3102, marketCap=52645}
Stock{symbol='BARU', price=29.08, volume=9041, marketCap=84189}
Stock{symbol='AAU', price=7.58, volume=2622, marketCap=76730}
Stock{symbol='FB', price=83.53, volume=110561, marketCap=612692653}
Stock{symbol='TKU', price=43.96, volume=9513, marketCap=22117}
Stock{symbol='TESLAU', price=43.32, volume=8088, marketCap=18241}
Stock{symbol='MSU', price=84.9, volume=2977, marketCap=46721}
Stock{symbol='TK', price=84.06, volume=488957, marketCap=959551514}
Stock{symbol='YBB', price=28.06, volume=551055, marketCap=895663800}
Display post-order:
Stock{symbol='AAU', price=7.58, volume=2622, marketCap=76730}
Stock{symbol='FB', price=83.53, volume=110561, marketCap=612692653}
Stock{symbol='BARU', price=29.08, volume=9041, marketCap=84189}
Stock{symbol='MSU', price=84.9, volume=2977, marketCap=46721}
Stock{symbol='TK', price=84.06, volume=488957, marketCap=959551514}
Stock{symbol='TESLAU', price=43.32, volume=8088, marketCap=18241}
Stock{symbol='YBB', price=28.06, volume=551055, marketCap=895663800}
Stock{symbol='TKU', price=43.96, volume=9513, marketCap=22117}
Stock{symbol='FBU', price=37.18, volume=3102, marketCap=52645}
```

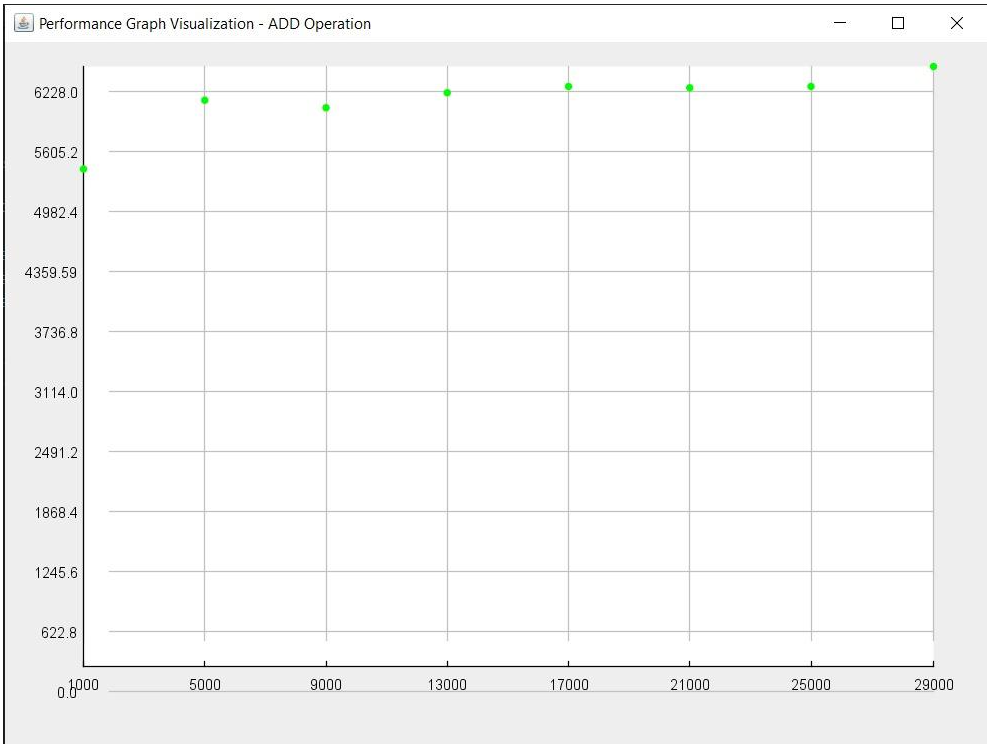
REMOVE



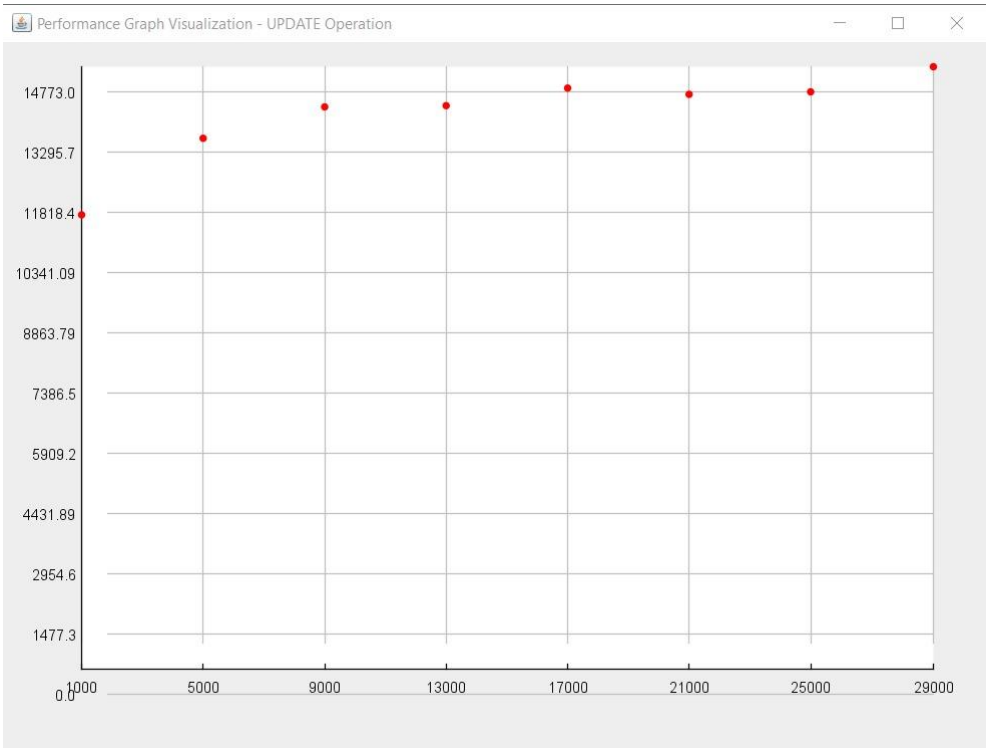
SEARCH



ADD


Performance Graph Visualization - ADD Operation
6228.0
5605.2
4982.4
4359.59
3736.8
3114.0
2491.2
1868.4
1245.6
622.8
0.0
1000
5000
9000
13000
17000
21000
25000
29000

UPDATE


Performance Graph Visualization - UPDATE Operation
14773.0
13295.7
11818.4
10341.09
8863.79
7386.5
5909.2
4431.89
2954.6
1477.3
0.0
1000
5000
9000
13000
17000
21000
25000
29000