CSE 222

Data Structures and Algorithms

7th Assignment

Report

Author

Mert Emir ŞEKER

200104004085

Date

23.05.2024

Table Of Contents

1. Makefile:

C: > Users > merts > OneDrive > Masaüstü > src > makefile

```
JC = javac
JFLAGS = -g
TARGET = Main
SOURCES = Main.java GUIVisualization.java AVLTree.java Stock.java StockDataManager.java RandomCommandGenerator.java

default: $(TARGET)

$(TARGET): $(SOURCES)
    $(JC) $(JFLAGS) $^
    java -Xint $(TARGET)

doc:
    javadoc -d javadoc $(SOURCES)

clean:
    rm -f *.class
    rm -rf javadoc

.PHONY: clean default javadoc

random: RandomCommandGenerator.java
    $(JC) $(JFLAGS) RandomCommandGenerator.java
    java RandomCommandGenerator
```

2. Makefile Commands:

JC = javac: Uses javac as the Java compiler.

JFLAGS = -g: Adds debugging information during compilation.

TARGET = Main: Sets the main target class to Main.

SOURCES: Lists the source files to be compiled.

default: $(TARGET): The default target compiles and runs the Main class.

$(TARGET): $(SOURCES): Compiles and runs the Main class from the source files.

doc: Generates documentation using javadoc.

clean: Deletes compiled class files and the javadoc directory.

.PHONY: Declares clean, default, and javadoc as phony targets.

random: Compiles and runs the RandomCommandGenerator class.

3. How to run the code?

To run the program, you can write **make** in terminal and you can write **java Main <input_file>**. If you want to create Javadoc you can write **make doc**. If you want to generate random command file you can write in terminal **make random**

4. Code Explanation:

4.1. Stock.java: The Stock class represents a stock with attributes such as symbol, price, volume, and market capitalization. It includes methods to get and set these attributes: getSymbol and setSymbol for the stock symbol, getPrice and setPrice for the stock price, getVolume and setVolume for the stock volume, and getMarketCap and setMarketCap for the stock market capitalization. Additionally, it provides a toString method to return a string representation of the stock. The constructor initializes these attributes when a new Stock object is created.

4.2. AVLTree.java: The AVLTree class represents a balanced binary search tree (AVL tree) for managing Stock objects.

4.2.1. Node class: The nested Node class represents a node in the AVL tree, containing a Stock object, pointers to left and right children, and the height of the node.

```
private class Node
{
    Stock stock;
    Node left, right;
    int height;

    Node(Stock stock)
    {
        this.stock = stock;
        this.height = 1;
    }
}

private Node root;
```

4.2.2. Insert: The insert method adds a new Stock object to the AVL tree by calling the private insert method, which recursively inserts the stock and balances the tree as necessary. The private insert method handles the recursive insertion of a Stock object starting from the given node, updating the node's height, and balancing the tree if required.

```
public void insert(Stock stock)
{
    root = insert(root, stock);
}

/**
 * Inserts a new stock into the AVL tree starting from the given node.
 *
 * @param node  The current node in the AVL tree.
 * @param stock The stock to be inserted.
 * @return The updated node after insertion.
 */
private Node insert(Node node, Stock stock)
{
    if (node == null)
    {
        return new Node(stock);
    }

    int cmp = stock.getSymbol().compareTo(node.stock.getSymbol());
    if (cmp < 0)
    {
        node.left = insert(node.left, stock);
    }
    else if (cmp > 0)
    {
        node.right = insert(node.right, stock);
    }
    else
    {
        // If stock with same symbol exists, update its attributes
        node.stock.setPrice(stock.getPrice());
        node.stock.setVolume(stock.getVolume());
        node.stock.setMarketCap(stock.getMarketCap());
        return node;
    }

    // Update height and balance the node
    node.height = 1 + Math.max(height(node.left), height(node.right));
    return balance(node);
}
```

4.2.3. Delete: The delete method removes a Stock object from the AVL tree by calling the private delete method, which recursively deletes the stock and balances the tree as necessary. The private delete method handles the recursive deletion of a Stock object starting from the given node, updating the node's height, and balancing the tree if required, including handling nodes with one or no children and finding the in-order successor for nodes with two children.

```
public void delete(String symbol)
{
    root = delete (root, symbol);
}

/**
 * Deletes a stock from the AVL tree starting from the given node.
 *
 * @param node   The current node in the AVL tree.
 * @param symbol The symbol of the stock to be deleted.
 * @return The updated node after deletion.
 */
private Node delete(Node node, String symbol)
{
    if (node == null)
    {
        return null;
    }

    int cmp = symbol.compareTo(node.stock.getSymbol());
    if (cmp < 0)
    {
        node.left = delete (node.left, symbol);
    }
    else if (cmp > 0)
    {
        node.right = delete (node.right, symbol);
    }
    else
    {
        // Node with only one child or no child
        if (node.left == null || node.right == null)
        {
            node = (node.left != null) ? node.left : node.right;
        }
        else
        {
            // Node with two children: Get the inorder successor (smallest in the right subtree)
            Node temp = min_value_node(node.right);
            node.stock = temp.stock;
            node.right = delete (node.right, temp.stock.getSymbol());
        }
    }

    // If the tree had only one node then return
    if (node == null)
    {
        return null;
    }

    // Update height and balance the node
    node.height = 1 + Math.max(height(node.left), height(node.right));
    return balance(node);
}
```

4.2.4. min_value_node: The min_value_node method finds and returns the node with the minimum value in the given subtree by traversing to the leftmost node.

```
private Node min_value_node(Node node)
{
    Node current = node;
    while (current.left != null)
    {
        current = current.left;
    }
    return current;
}
```

4.2.5. Search: The search method looks for a Stock object in the AVL tree by calling the private search method, returning the Stock if found or null if not found. The private search method handles the recursive search for a Stock object starting from the given node, comparing the symbol to navigate left or right in the tree.

```
public Stock search(String symbol)
{
    Node result = search(root, symbol);
    return (result != null) ? result.stock : null;
}

/**
 * Searches for a stock in the AVL tree starting from the given node.
 *
 * @param node   The current node in the AVL tree.
 * @param symbol The symbol of the stock to search for.
 * @return The node containing the stock if found, otherwise null.
 */
private Node search(Node node, String symbol)
{
    if (node == null || node.stock.getSymbol().equals(symbol))
    {
        return node;
    }

    int cmp = symbol.compareTo(node.stock.getSymbol());
    if (cmp < 0)
    {
        return search(node.left, symbol);
    }
    else
    {
        return search(node.right, symbol);
    }
}
```

4.2.6. **Balance:** The balance method checks the balance factor of a node and performs rotations (left, right, or double rotations) to ensure the AVL tree remains balanced after insertions or deletions.

```
private Node balance(Node node)
{
    int balanceFactor = getBalanceFactor(node);

    // Left Left Case
    if (balanceFactor > 1 && getBalanceFactor(node.left) >= 0)
    {
        return rightRotate(node);
    }

    // Left Right Case
    if (balanceFactor > 1 && getBalanceFactor(node.left) < 0)
    {
        node.left = leftRotate(node.left);
        return rightRotate(node);
    }

    // Right Right Case
    if (balanceFactor < -1 && getBalanceFactor(node.right) <= 0)
    {
        return leftRotate(node);
    }

    // Right Left Case
    if (balanceFactor < -1 && getBalanceFactor(node.right) > 0)
    {
        node.right = rightRotate(node.right);
        return leftRotate(node);
    }

    return node;
}
```

4.2.7. **rightRotate:** The rightRotate method performs a right rotation on the subtree rooted at the given node, adjusting the heights and returning the new root of the rotated subtree.

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

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

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

    return x;
}
```

4.2.8. leftRotate: The leftRotate method performs a left rotation on the subtree rooted at the given node, adjusting the heights and returning 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 = Math.max(height(x.left), height(x.right)) + 1;
    y.height = Math.max(height(y.left), height(y.right)) + 1;

    return y;
}
```

4.2.9. height: The height method returns the height of the given node, or 0 if the node is null.

```
private int height(Node node)
{
    return (node == null) ? 0 : node.height;
}
```

4.2.10. getBalanceFactor: The getBalanceFactor method calculates and returns the balance factor of the given node by subtracting the height of the right child from the height of the left child.

```
private int getBalanceFactor(Node node)
{
    return (node == null) ? 0 : height(node.left) - height(node.right);
}
```

4.2.11. inOrderTraversal: The inOrderTraversal method prints the in-order traversal of the AVL tree by calling the private inOrderTraversal method starting from the root. The private inOrderTraversal method recursively traverses the tree in in-order (left, root, right) and prints each Stock object.

```
public void inOrderTraversal()
{
    inOrderTraversal(root);
}

/**
 * Prints the in-order traversal of the AVL tree starting from the given node.
 *
 * @param node The root of the subtree to traverse.
 */
private void inOrderTraversal(Node node)
{
    if (node != null)
    {
        inOrderTraversal(node.left);
        System.out.println(node.stock);
        inOrderTraversal(node.right);
    }
}
```

4.2.12. preOrderTraversal: The preOrderTraversal method prints the pre-order traversal of the AVL tree by calling the private preOrderTraversal method starting from the root. The private preOrderTraversal method recursively traverses the tree in pre-order (root, left, right) and prints each Stock object.

```
public void preOrderTraversal()
{
    preOrderTraversal(root);
}

/**
 * Prints the pre-order traversal of the AVL tree starting from the given node.
 *
 * @param node The root of the subtree to traverse.
 */
private void preOrderTraversal(Node node)
{
    if (node != null)
    {
        System.out.println(node.stock);
        preOrderTraversal(node.left);
        preOrderTraversal(node.right);
    }
}
```

4.2.13. postOrderTraversal: The postOrderTraversal method prints the post-order traversal of the AVL tree by calling the private postOrderTraversal method starting from the root. The private postOrderTraversal method recursively traverses the tree in post-order (left, right, root) and prints each Stock object.

```
public void postOrderTraversal()
{
    postOrderTraversal(root);
}

/**
 * Prints the post-order traversal of the AVL tree starting from the given node.
 *
 * @param node The root of the subtree to traverse.
 */
private void postOrderTraversal(Node node)
{
    if (node != null)
    {
        postOrderTraversal(node.left);
        postOrderTraversal(node.right);
        System.out.println(node.stock);
    }
}
```

4.3. StockDataManager.java: The StockDataManager class manages stock data using an AVL tree.

4.3.1. addOrUpdateStock: The addOrUpdateStock method either adds a new stock or updates an existing stock in the AVL tree based on the given symbol, price, volume, and market capitalization.

```
public void addOrUpdateStock(String symbol, double price, long volume, long marketCap) {
    Stock existingStock = avlTree.search(symbol);
    if (existingStock != null) {
        existingStock.setPrice(price);
        existingStock.setVolume(volume);
        existingStock.setMarketCap(marketCap);
    } else {
        Stock newStock = new Stock(symbol, price, volume, marketCap);
        avlTree.insert(newStock);
    }
}
```

4.3.2. removeStock: The removeStock method removes a stock from the AVL tree based on the given symbol.

```
public void removeStock(String symbol) {
    avlTree.delete(symbol);
}
```

4.3.3. searchStock: The searchStock method searches for a stock in the AVL tree by its symbol and returns the Stock object if found, or null if not found.

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

4.3.4. updateStock: The updateStock method updates the details of an existing stock, including changing its symbol, price, volume, and market capitalization. It first deletes the old stock, updates the stock details, and reinserts it into the AVL tree.

```
public void updateStock(String oldSymbol, String newSymbol, double newPrice, long newVolume, long newMarketCap) {
    Stock stock = avlTree.search(oldSymbol);
    if (stock != null) {
        avlTree.delete(oldSymbol); // Remove old stock
        stock.setSymbol(newSymbol); // Update symbol
        stock.setPrice(newPrice);
        stock.setVolume(newVolume);
        stock.setMarketCap(newMarketCap);
        avlTree.insert(stock); // Insert updated stock with new symbol
    }
}
```

4.3.5. printInOrder: The printInOrder method prints the in-order traversal of the AVL tree, showing the stocks in sorted order based on their symbols.

```
public void printInOrder() {
    avlTree.inOrderTraversal();
}
```

4.3.6. printPreOrder: The printPreOrder method prints the pre-order traversal of the AVL tree, showing the stocks in root-left-right order.

```
public void printPreOrder() {
    avlTree.preOrderTraversal();
}
```

4.3.7. printPostOrder: The printPostOrder method prints the post-order traversal of the AVL tree, showing the stocks in left-right-root order.

```
public void printPostOrder() {
    avlTree.postOrderTraversal();
}
```

4.3.8. performPerformanceAnalysis: The performPerformanceAnalysis method conducts a performance analysis by measuring the average time taken for adding, searching, and removing a specified number of stocks (size) in the StockDataManager. It prints the average time for each operation and displays the in-order, pre-order, and post-order traversals of the AVL tree.

```
private static void performPerformanceAnalysis(StockDataManager manager, int size) {
    long startTime, endTime;

    // Measure time for ADD operation
    startTime = System.nanoTime();
    for (int i = 0; i < size; i++) {
        manager.addOrUpdateStock("SYM" + i, Math.random() * 100, (long) (Math.random() * 1000000), (long) (Math.random() * 1000000000));
    }
    endTime = System.nanoTime();
    System.out.println("Average ADD time: " + (endTime - startTime) / size + " ns");

    // Measure time for SEARCH operation
    startTime = System.nanoTime();
    for (int i = 0; i < size; i++) {
        manager.searchStock("SYM" + i);
    }
    endTime = System.nanoTime();
    System.out.println("Average SEARCH time: " + (endTime - startTime) / size + " ns");

    // Measure time for REMOVE operation
    startTime = System.nanoTime();
    for (int i = 0; i < size; i++) {
        manager.removeStock("SYM" + i);
    }
    endTime = System.nanoTime();
    System.out.println("Average REMOVE time: " + (endTime - startTime) / size + " ns");

    // Print traversals
    System.out.println(x:"In-order traversal:");
    manager.printInOrder();
    System.out.println(x:"Pre-order traversal:");
    manager.printPreOrder();
    System.out.println(x:"Post-order traversal:");
    manager.printPostOrder();
}
```

4.4. GUIVisualization.java: VThe GUIVisualization class creates a graphical user interface to visualize performance data as a graph using JFrame.

4.4.1. GUIVisualization: The constructor initializes the frame with the specified plot type ("line" or "scatter"), x-axis data points, y-axis data points, and graph title, setting up the frame's properties like size, title, and default close operation.

4.4.2. paint: The paint method overrides the default paint method to draw the graph on the JFrame using the Graphics object, calling the drawGraph method to render the graph.

4.4.3. drawGraph: The drawGraph method handles the actual drawing of the graph. It sets up the drawing area, background, and grid lines, draws axis labels, and plots the data points either as a line or scatter plot depending on the plot type. It includes detailed steps for drawing y-axis and x-axis labels, grid lines, and data points.

4.4.4. getMaxYValue: The getMaxYValue method calculates and returns the maximum value from the list of y-axis data points by iterating through the list and finding the highest value.

4.5. RandomCommandGenerator: The RandomCommandGenerator class is designed to create random stock commands for testing the StockDataManager. It generates commands of types ADD, SEARCH, REMOVE, and UPDATE, and writes them to a file named "operations.txt". This class helps in automating the creation of a diverse set of stock commands, which can be used to test the functionality and performance of the StockDataManager.

```java
public class RandomCommandGenerator {

    private static final int NUM_COMMANDS = 1000;
    private static final int NUM_ADDS = 1000;
    private static final String FILE_NAME = "operations.txt";

    /**
     * The entry point of the command generator application. Generates random ADD, SEARCH, REMOVE, and UPDATE commands and writes them to a file.
     *
     * @param args The command-line arguments.
     */
    Run | Debug
    public static void main(String[] args) {
        Random random = new Random();
        StringBuilder sb = new StringBuilder();

        // First generate 1000 ADD commands
        for (int i = 0; i < NUM_ADDS; i++) {
            String symbol = "SYM" + i;
            double price = 10 + (1000 - 10) * random.nextDouble();
            long volume = 1 + (1000000 - 1) * Math.abs(random.nextLong());
            long marketCap = 1 + (1000000000 - 1) * Math.abs(random.nextLong());
            sb.append(String.format(format:"ADD %s %.2f %d %d%n", symbol, price, volume, marketCap));
        }

        // Generate remaining commands
        for (int i = 0; i < NUM_COMMANDS; i++) {
            int commandType = random.nextInt(bound:3);
            int index = random.nextInt(NUM_ADDS);

            switch (commandType) {
                case 0: // SEARCH
                    sb.append(String.format(format:"SEARCH SYM%d%n", index));
                    break;
                case 1: // REMOVE
                    sb.append(String.format(format:"REMOVE SYM%d%n", index));
                    break;
                case 2: // UPDATE
                    String newSymbol = "SYM" + index;
                    double newPrice = 10 + (1000 - 10) * random.nextDouble();
                    long newVolume = 1 + (1000000 - 1) * Math.abs(random.nextLong());
                    long newMarketCap = 1 + (1000000000 - 1) * Math.abs(random.nextLong());
                    sb.append(String.format(format:"UPDATE SYM%d %s %.2f %d %d%n", index, newSymbol, newPrice, newVolume, newMarketCap));
                    break;
            }
        }

        // Write commands to file
        try (BufferedWriter writer = new BufferedWriter(new FileWriter(FILE_NAME))) {
            writer.write(sb.toString());
        } catch (IOException e) {
            e.printStackTrace();
        }

        System.out.println("Random commands have been generated and written to " + FILE_NAME);
    }
}
```

4.6. Main.java: The Main class serves as the entry point for running the stock management application. It reads commands from an input file, processes them using the StockDataManager, and performs performance analysis on AVL tree operations such as ADD, SEARCH, REMOVE, and UPDATE.

4.6.1. Main: The main method is the entry point of the application. It expects a single command-line argument specifying the path to the input file containing stock management commands. The method reads the commands from the file, processes them using the StockDataManager, and then performs performance analysis for various operations. If the correct number of arguments is not provided, it prints a usage message and exits.

```
public static void main(String[] args) {
    if (args.length != 1) {
        System.out.println(x:"Usage: java Main <input_file>");
        return;
    }

    String inputFile = args[0];
    StockDataManager manager = new StockDataManager();

    try (BufferedReader br = new BufferedReader(new FileReader(inputFile))) {
        String line;
        while ((line = br.readLine()) != null) {
            processCommand(line, manager);
        }
    } catch (IOException e) {
        e.printStackTrace();
    }

    // Perform a simple performance analysis
    int size = 1000;
    AddPerformanceAnalysis(manager, size);
    SearchPerformanceAnalysis(manager, size);
    RemovePerformanceAnalysis(manager, size);
    UpdatePerformanceAnalysis(manager, size);
}
```

4.6.2. processCommand: The processCommand method takes a single command line and executes the corresponding operation on the StockDataManager. It splits the command line into tokens and determines the command type (ADD, REMOVE, SEARCH, UPDATE). Based on the command type, it either adds/updates a stock, removes a stock, searches for a stock, or updates an existing stock with new details. For unknown commands, it prints an error message.

```
private static void processCommand(String line, StockDataManager manager) {
    String[] tokens = line.split(regex:" ");
    String command = tokens[0];

    switch (command) {
        case "ADD":
            manager.addOrUpdateStock(tokens[1], Double.parseDouble(tokens[2]), Long.parseLong(tokens[3]), Long.parseLong(tokens[4]));
            break;
        case "REMOVE":
            manager.removeStock(tokens[1]);
            break;
        case "SEARCH":
            Stock stock = manager.searchStock(tokens[1]);
            if (stock != null) {
                System.out.println(stock);
            } else {
                System.out.println("Stock not found: " + tokens[1]);
            }
            break;
        case "UPDATE":
            manager.updateStock(tokens[1], tokens[2], Double.parseDouble(tokens[3]), Long.parseLong(tokens[4]), Long.parseLong(tokens[5]));
            break;
        default:
            System.out.println("Unknown command: " + command);
            break;
    }
}
```

4.6.3. AddPerformanceAnalysis: The AddPerformanceAnalysis method performs performance analysis for the ADD operation on the StockDataManager. It measures the time taken to add a specified number of stocks (given by size) and records the total time at intervals of 100 operations. It then calculates the average time per operation and prints it. Finally, it visualizes the performance data using a scatter plot with the help of the GUIVisualization class.

```
private static void AddPerformanceAnalysis(StockDataManager manager, int size) {
    ArrayList<Integer> xAxis = new ArrayList<>();
    ArrayList<Long> yAxis = new ArrayList<>();
    long totalTime = 0;

    for (int i = 0; i < size; i++) {
        long operationStart = System.nanoTime();
        manager.addOrUpdateStock("SYM" + i, Math.random() * 100, (long) (Math.random() * 1000000), (long) (Math.random() * 1000000000));
        long operationTime = System.nanoTime() - operationStart;
        totalTime += operationTime;
        if ((i + 1) % 100 == 0) {
            xAxis.add(i + 1);
            yAxis.add(totalTime);
        }
    }

    long averageTime = totalTime / size;
    System.out.println("Average ADD time: " + averageTime + " ns");

    SwingUtilities.invokeLater(() -> {
        GUIVisualization frame = new GUIVisualization(plotType:"scatter", xAxis, yAxis, graphTitle:"Add Operation Performance");
        frame.setVisible(b:true);
    });
}
```

4.6.4. SearchPerformanceAnalysis:

The SearchPerformanceAnalysis method performs performance analysis for the SEARCH operation on the StockDataManager. It first adds a specified number of stocks to the manager and then measures the time taken to search for each stock. It records the total time at intervals of 100 operations, calculates the average time per operation, and prints it. The performance data is visualized using a scatter plot created by the GUIVisualization class.

```
private static void SearchPerformanceAnalysis(StockDataManager manager, int size) {
    ArrayList<Integer> xAxis = new ArrayList<>();
    ArrayList<Long> yAxis = new ArrayList<>();
    long totalTime = 0;

    for (int i = 0; i < size; i++) {
        manager.addOrUpdateStock("SYM" + i, Math.random() * 100, (long) (Math.random() * 1000000), (long) (Math.random() * 1000000000));
    }

    for (int i = 0; i < size; i++) {
        long operationStart = System.nanoTime();
        manager.searchStock("SYM" + i);
        long operationTime = System.nanoTime() - operationStart;
        totalTime += operationTime;

        if ((i + 1) % 100 == 0) {
            xAxis.add(i + 1);
            yAxis.add(totalTime);
        }
    }

    long averageTime = totalTime / size;
    System.out.println("Average SEARCH time: " + averageTime + " ns");

    SwingUtilities.invokeLater(() -> {
        GUIVisualization frame = new GUIVisualization(plotType:"scatter", xAxis, yAxis, graphTitle:"Search Operation Performance");
        frame.setVisible(b:true);
    });
}
```

4.6.5. RemovePerformanceAnalysis:

The RemovePerformanceAnalysis method performs performance analysis for the REMOVE operation on the StockDataManager. Similar to the SEARCH analysis, it first adds a specified number of stocks to the manager. Then, it measures the time taken to remove each stock, records the total time at intervals of 100 operations, calculates the average time per operation, and prints it. The performance data is visualized using a scatter plot with the GUIVisualization class.

```
private static void RemovePerformanceAnalysis(StockDataManager manager, int size) {
    ArrayList<Integer> xAxis = new ArrayList<>();
    ArrayList<Long> yAxis = new ArrayList<>();
    long totalTime = 0;

    for (int i = 0; i < size; i++) {
        manager.addOrUpdateStock("SYM" + i, Math.random() * 100, (long) (Math.random() * 1000000), (long) (Math.random() * 1000000000));
    }

    for (int i = 0; i < size; i++) {
        long operationStart = System.nanoTime();
        manager.removeStock("SYM" + i);
        long operationTime = System.nanoTime() - operationStart;
        totalTime += operationTime;
        if ((i + 1) % 100 == 0) {
            xAxis.add(i + 1);
            yAxis.add(totalTime);
        }
    }

    long averageTime = totalTime / size;
    System.out.println("Average REMOVE time: " + averageTime + " ns");

    SwingUtilities.invokeLater(() -> {
        GUIVisualization frame = new GUIVisualization(plotType:"scatter", xAxis, yAxis, graphTitle:"Remove Operation Performance");
        frame.setVisible(b:true);
    });
}
```

4.6.6. UpdatePerformanceAnalysis:

The UpdatePerformanceAnalysis method performs performance analysis for the UPDATE operation on the StockDataManager. It adds a specified number of stocks to the manager and then measures the time taken to update each stock's details. The total time is recorded at intervals of 100 operations, the average time per operation is calculated and printed, and the performance data is visualized using a scatter plot with the GUIVisualization class.

```
private static void UpdatePerformanceAnalysis(StockDataManager manager, int size) {
    ArrayList<Integer> xAxis = new ArrayList<>();
    ArrayList<Long> yAxis = new ArrayList<>();
    long totalTime = 0;

    for (int i = 0; i < size; i++) {
        manager.addOrUpdateStock("SYM" + i, Math.random() * 100, (long) (Math.random() * 1000000), (long) (Math.random() * 1000000000));
    }

    for (int i = 0; i < size; i++) {
        long operationStart = System.nanoTime();
        manager.updateStock("SYM" + i, "SYM" + i, Math.random() * 100, (long) (Math.random() * 1000000), (long) (Math.random() * 1000000000));
        long operationTime = System.nanoTime() - operationStart;
        totalTime += operationTime;

        if ((i + 1) % 100 == 0) {
            xAxis.add(i + 1);
            yAxis.add(totalTime);
        }
    }

    long averageTime = totalTime / size;
    System.out.println("Average UPDATE time: " + averageTime + " ns");

    SwingUtilities.invokeLater(() -> {
        GUIVisualization frame = new GUIVisualization(plotType:"scatter", xAxis, yAxis, graphTitle:"Update Operation Performance");
        frame.setVisible(b:true);
    });
}
```

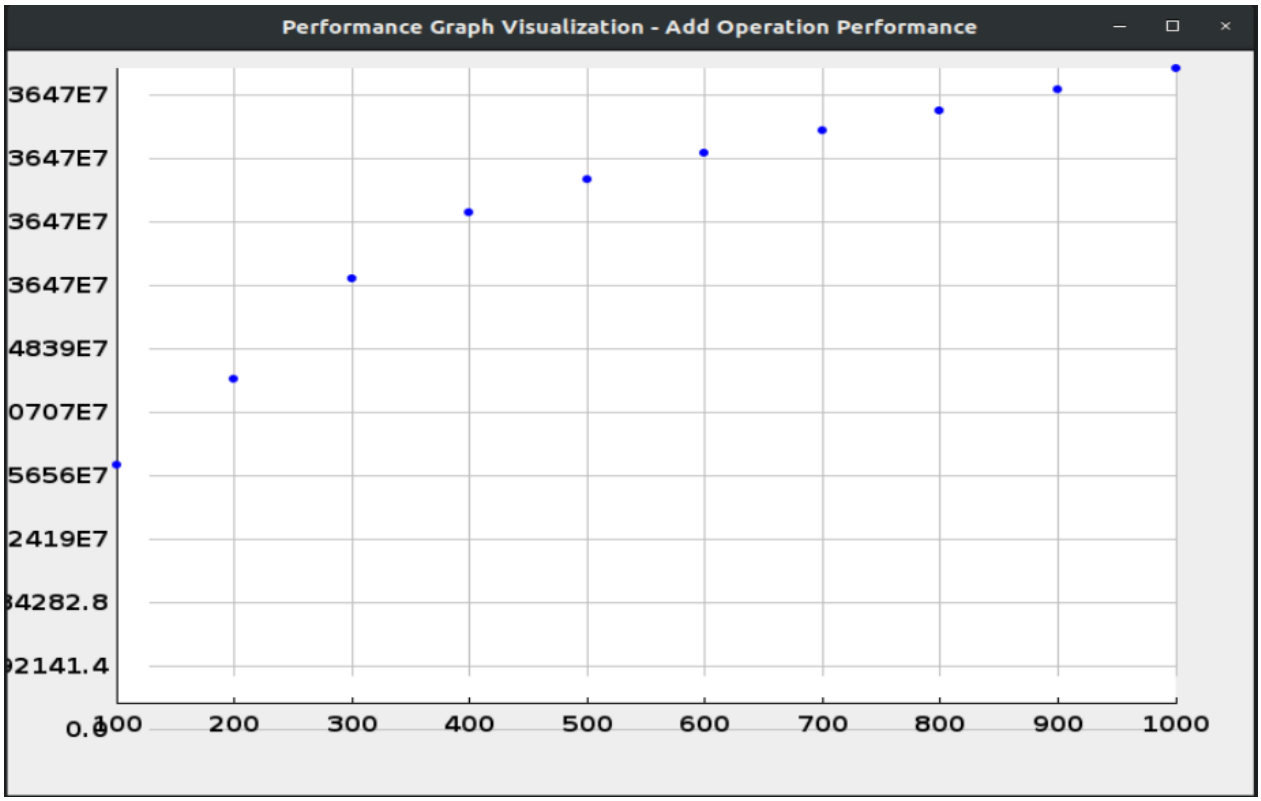
5. Example Outputs:

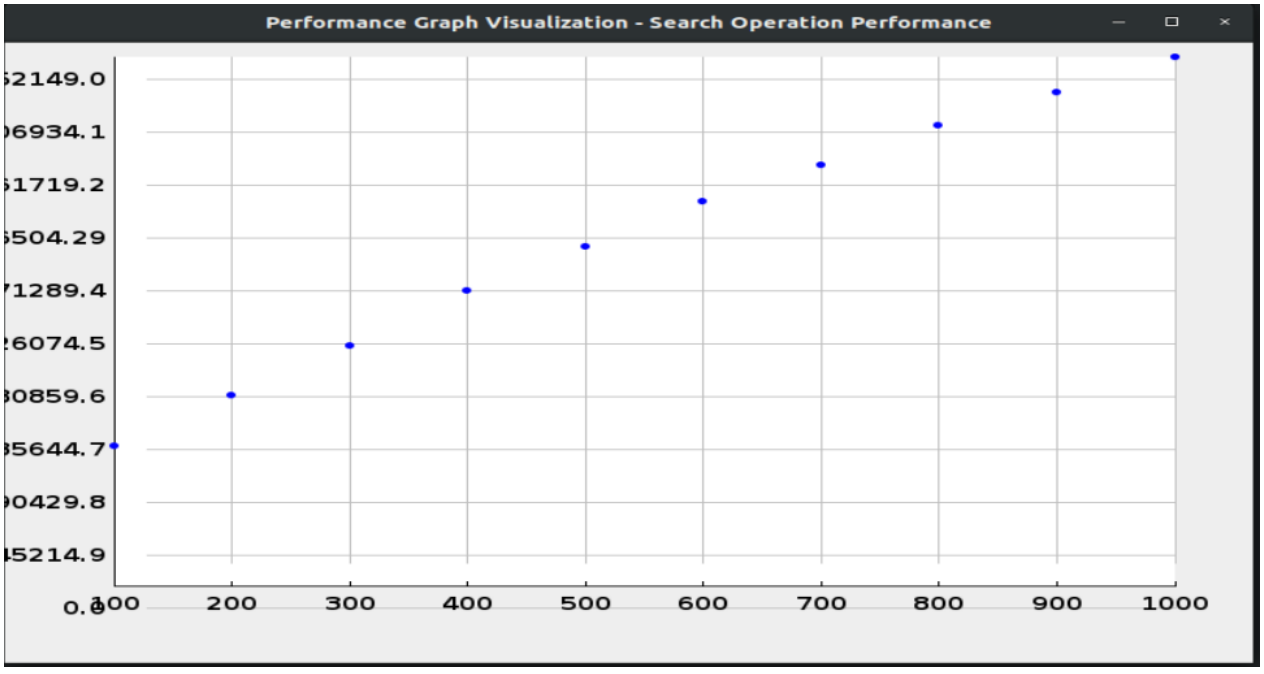
5.1. Terminal Output:

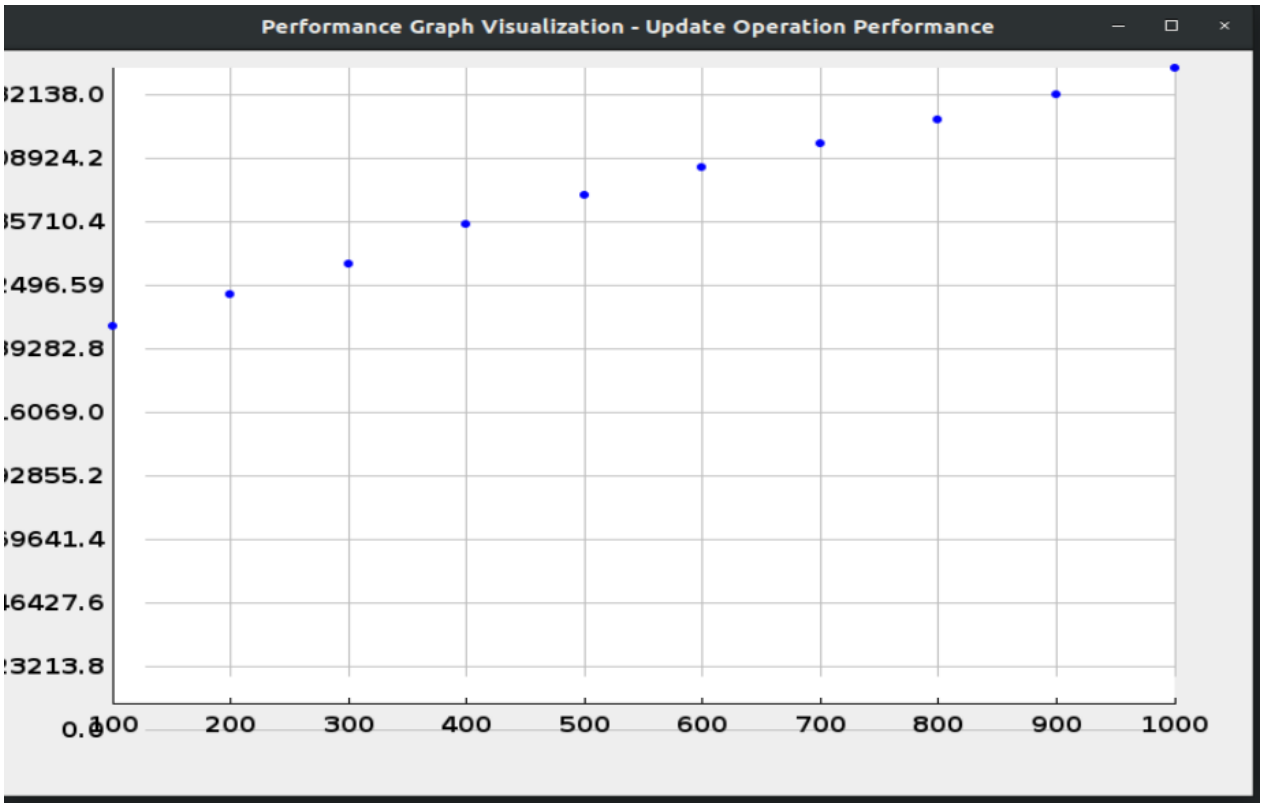
```
seker@seker-VirtualBox: ~/Desktop/src
File  Edit  View  Search  Terminal  Help
seker@seker-VirtualBox:~/Desktop/src$ make
javac -g Main.java GUIVisualization.java AVLTree.java Stock.java StockDataManager.java RandomCommandGenerator.java
java -Xint Main
Usage: java Main <input_file>
seker@seker-VirtualBox:~/Desktop/src$ java Main input.txt
Stock [symbol=XYZ, price=25.5, volume=1000000, marketCap=5000000]
Stock [symbol=DEF, price=15.75, volume=750000, marketCap=3750000]
Stock [symbol=JKL, price=22.5, volume=1500000, marketCap=7500000]
Average ADD time: 37796 ns
Average SEARCH time: 6846 ns
Average REMOVE time: 9745 ns
Average UPDATE time: 13858 ns
seker@seker-VirtualBox:~/Desktop/src$
```

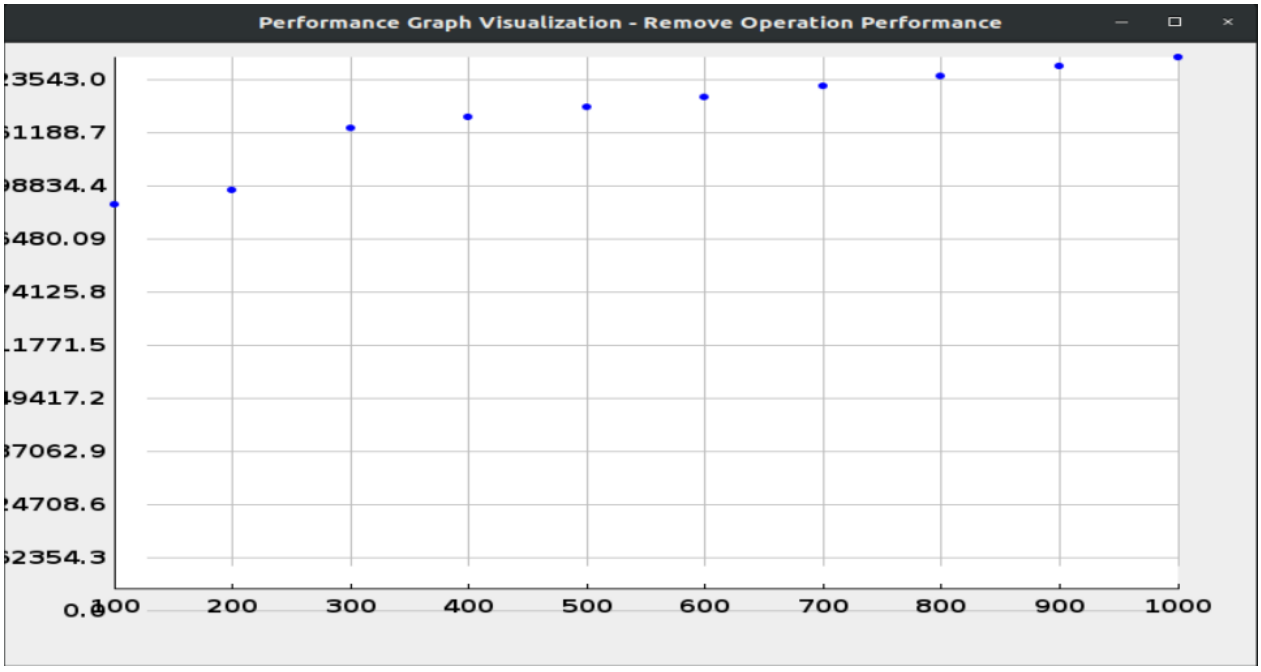
```
seker@seker-VirtualBox: ~/Desktop/src
File  Edit  View  Search  Terminal  Help
seker@seker-VirtualBox:~/Desktop/src$ make
javac -g Main.java GUIVisualization.java AVLTree.java Stock.java StockDataManager.java RandomCommandGenerator.java
java -Xint Main
Usage: java Main <input_file>
seker@seker-VirtualBox:~/Desktop/src$ java Main input.txt
Stock [symbol=AAPL, price=150.0, volume=1000000, marketCap=2500000000]
Stock [symbol=APPL, price=155.0, volume=1100000, marketCap=2600000000]
Stock not found: GOOGL
Average ADD time: 36455 ns
Average SEARCH time: 1661 ns
Average REMOVE time: 5806 ns
Average UPDATE time: 6673 ns
seker@seker-VirtualBox:~/Desktop/src$
```

5.2. Graph Output:






Performance Graph Visualization - Update Operation Performance
2138.0
8924.2
5710.4
496.59
9282.8
6069.0
2855.2
9641.4
6427.6
3213.8
0.0
100
200
300
400
500
600
700
800
900
1000


Performance Graph Visualization - Remove Operation Performance
3543.0
1188.7
8834.4
480.09
4125.8
1771.5
9417.2
7062.9
4708.6
2354.3
0.0
100
200
300
400
500
600
700
800
900
1000

5.3. All graphs are together: