AYKUT SERT HW7 REPORT

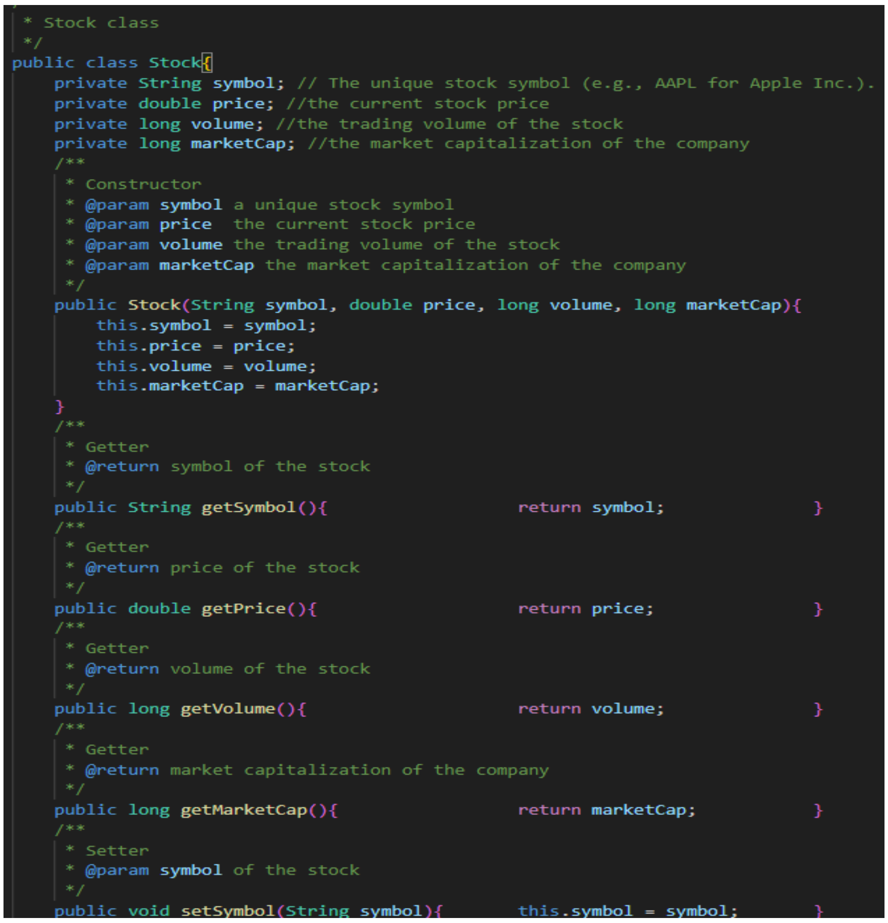
- Makefile

With make default , program runs.

With make doc , creates javadoc.

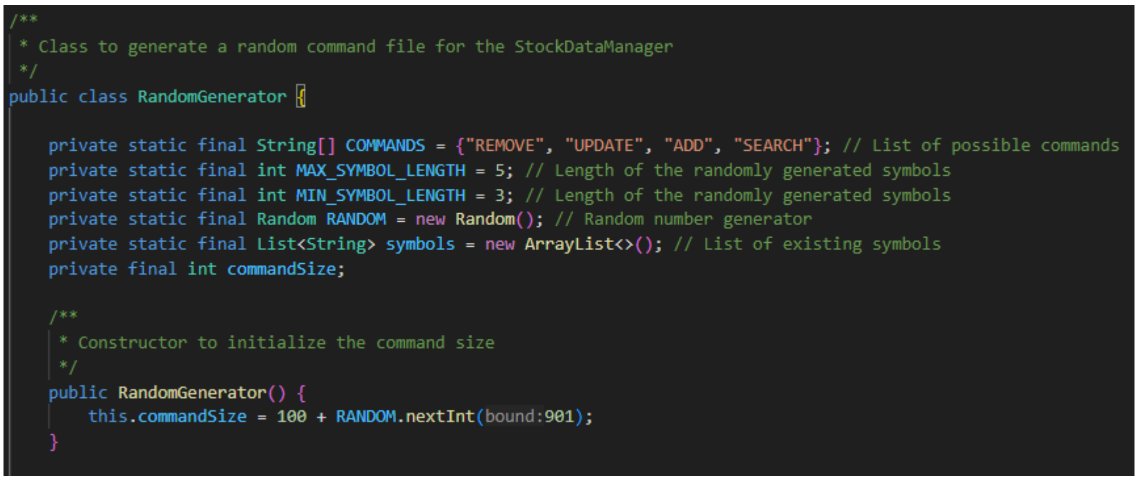
- Stock:

The data of the product was kept in the Stock class. Information about the product was obtained from Constructor. Necessary getters and setters have been implemented. While adding, a new stock object is created and inserted into the tree.



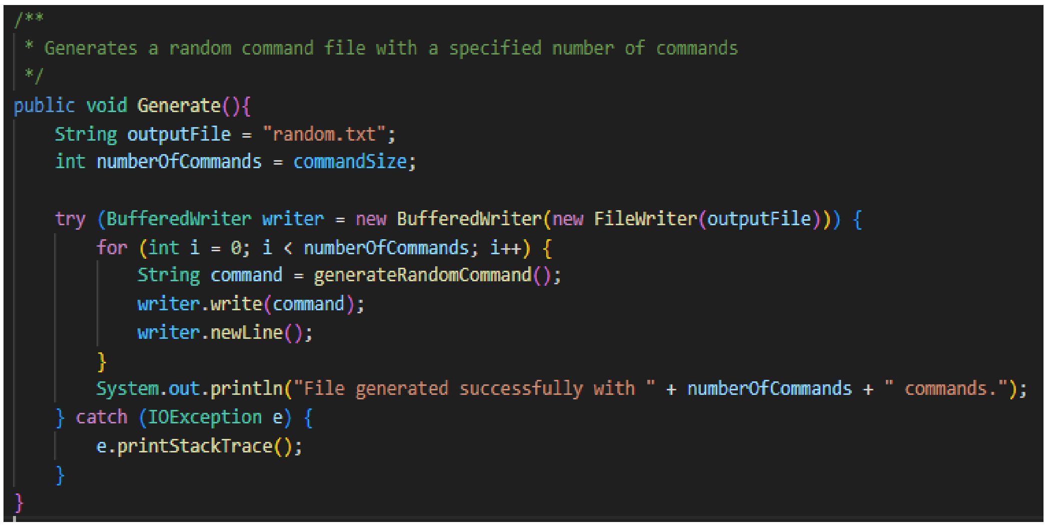
```java
 * Stock class
 */
public class Stock{
    private String symbol; // The unique stock symbol (e.g., AAPL for Apple Inc.).
    private double price; //the current stock price
    private long volume; //the trading volume of the stock
    private long marketCap; //the market capitalization of the company
    /**
     * Constructor
     * @param symbol a unique stock symbol
     * @param price  the current stock price
     * @param volume the trading volume of the stock
     * @param marketCap the market capitalization of the company
     */
    public Stock(String symbol, double price, long volume, long marketCap){
        this.symbol = symbol;
        this.price = price;
        this.volume = volume;
        this.marketCap = marketCap;
    }
    /**
     * Getter
     * @return symbol of the stock
     */
    public String getSymbol(){                  return symbol;              }
    /**
     * Getter
     * @return price of the stock
     */
    public double getPrice(){                   return price;               }
    /**
     * Getter
     * @return volume of the stock
     */
    public long getVolume(){                    return volume;              }
    /**
     * Getter
     * @return market capitalization of the company
     */
    public long getMarketCap(){                 return marketCap;           }
    /**
     * Setter
     * @param symbol of the stock
     */
    public void setSymbol(String symbol){       this.symbol = symbol;       }
```

- RandomGenerator:



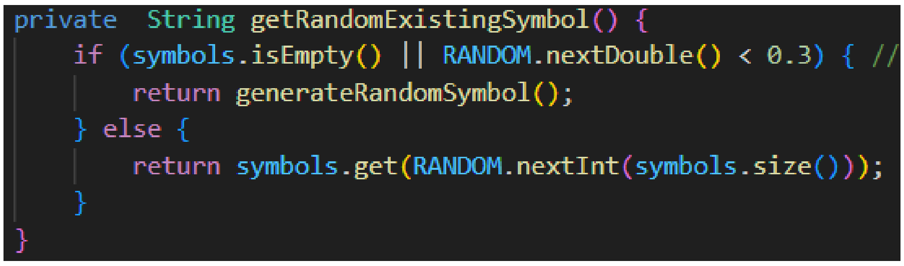
```java
/**
 * Class to generate a random command file for the StockDataManager
 */
public class RandomGenerator {

    private static final String[] COMMANDS = {"REMOVE", "UPDATE", "ADD", "SEARCH"}; // List of possible commands
    private static final int MAX_SYMBOL_LENGTH = 5; // Length of the randomly generated symbols
    private static final int MIN_SYMBOL_LENGTH = 3; // Length of the randomly generated symbols
    private static final Random RANDOM = new Random(); // Random number generator
    private static final List<String> symbols = new ArrayList<>(); // List of existing symbols
    private final int commandSize;

    /**
     * Constructor to initialize the command size
     */
    public RandomGenerator() {
        this.commandSize = 100 + RANDOM.nextInt(bound:901);
    }
```

While generating random input, our commands were first kept in an array. A length between 3-5 was chosen as the symbol. While generating random symbols, I kept a list of symbols so that search, remove and update do not always produce input that is not in the tree. It generates random data with a 30% rate or selects it randomly from this list (for remove, search and update).   The number of commands was chosen between 100 and 1000. In Main, a randomly generated file was used for the tree.



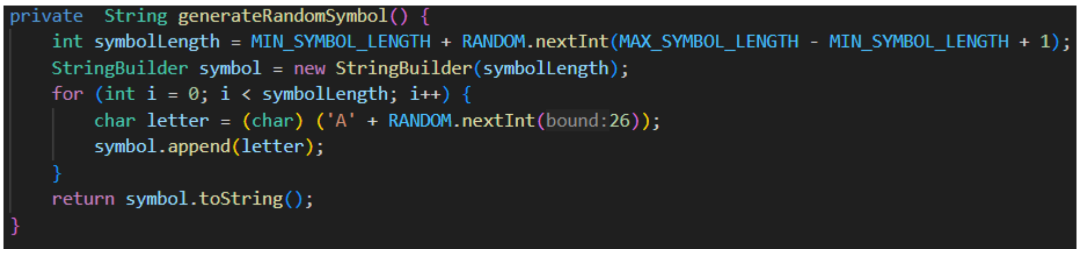
```java
/**
 * Generates a random command file with a specified number of commands
 */
public void Generate(){
    String outputFile = "random.txt";
    int numberOfCommands = commandSize;

    try (BufferedWriter writer = new BufferedWriter(new FileWriter(outputFile))) {
        for (int i = 0; i < numberOfCommands; i++) {
            String command = generateRandomCommand();
            writer.write(command);
            writer.newLine();
        }
        System.out.println("File generated successfully with " + numberOfCommands + " commands.");
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

This is the code to create a file and create as many inputs as there are commands.



```java
private  String getRandomExistingSymbol() {
    if (symbols.isEmpty() || RANDOM.nextDouble() < 0.3) { //
        return generateRandomSymbol();
    } else {
        return symbols.get(RANDOM.nextInt(symbols.size()));
    }
}
```

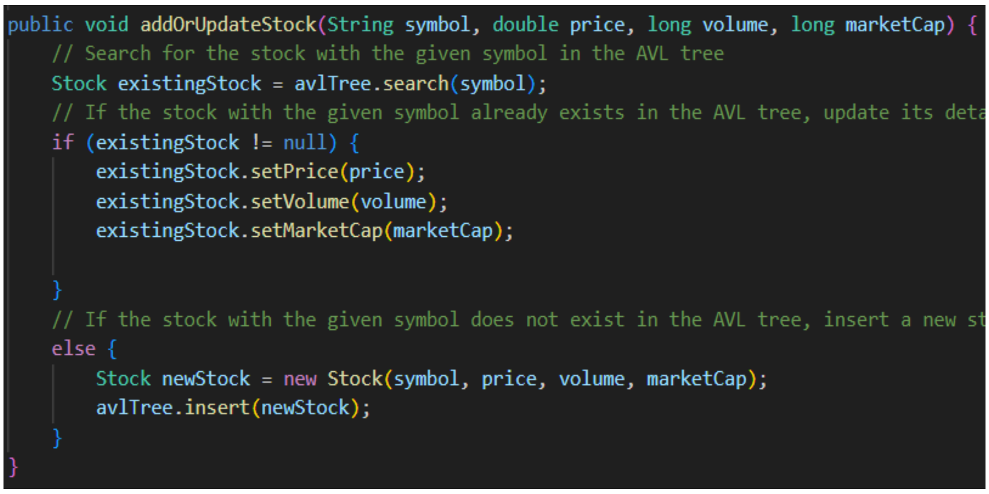
Here, if the symbol list is empty or a random symbol is generated with a rate of 30%. If the list is full, random is selected from there.



```java
private  String generateRandomSymbol() {
    int symbolLength = MIN_SYMBOL_LENGTH + RANDOM.nextInt(MAX_SYMBOL_LENGTH - MIN_SYMBOL_LENGTH + 1);
    StringBuilder symbol = new StringBuilder(symbolLength);
    for (int i = 0; i < symbolLength; i++) {
        char letter = (char) ('A' + RANDOM.nextInt(bound:26));
        symbol.append(letter);
    }
    return symbol.toString();
}
```

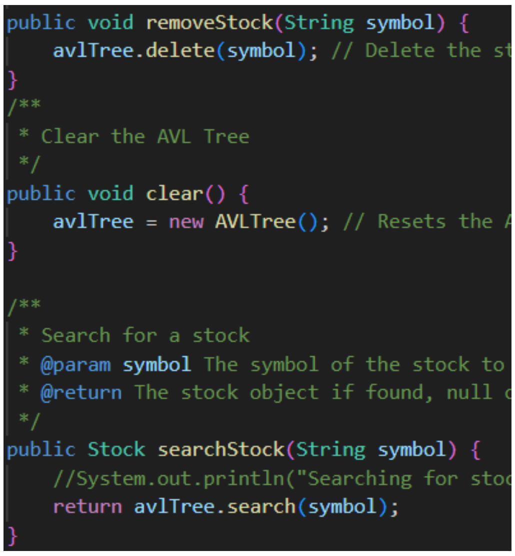
I used StringBuilder to generate random symbols. I produced symbols between 3-5 lengths.

- StockDataManager:



```java
public void addOrUpdateStock(String symbol, double price, long volume, long marketCap) {
    // Search for the stock with the given symbol in the AVL tree
    Stock existingStock = avlTree.search(symbol);
    // If the stock with the given symbol already exists in the AVL tree, update its deta
    if (existingStock != null) {
        existingStock.setPrice(price);
        existingStock.setVolume(volume);
        existingStock.setMarketCap(marketCap);

    }
    // If the stock with the given symbol does not exist in the AVL tree, insert a new st
    else {
        Stock newStock = new Stock(symbol, price, volume, marketCap);
        avlTree.insert(newStock);
    }
}
```

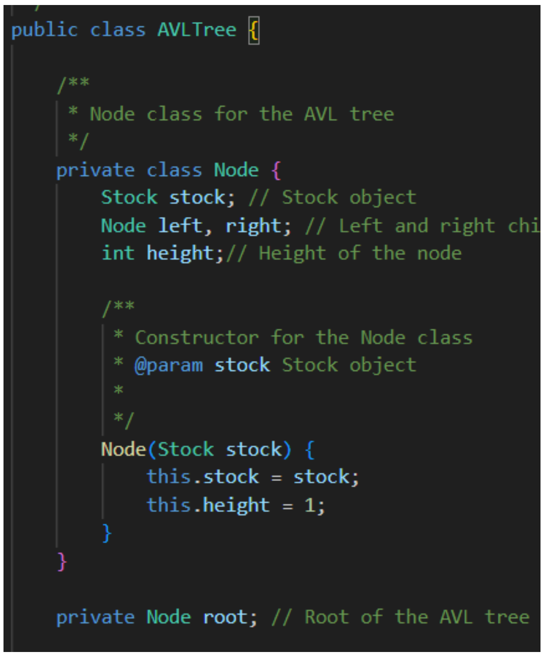
We call the search when adding stock. If there is a product, we update it with new information. Otherwise, we add to the tree.



```java
public void removeStock(String symbol) {
    avlTree.delete(symbol); // Delete the st
}
/**
 * Clear the AVL Tree
 */
public void clear() {
    avlTree = new AVLTree(); // Resets the A
}

/**
 * Search for a stock
 * @param symbol The symbol of the stock to
 * @return The stock object if found, null o
 */
public Stock searchStock(String symbol) {
    //System.out.println("Searching for stoc
    return avlTree.search(symbol);
}
```

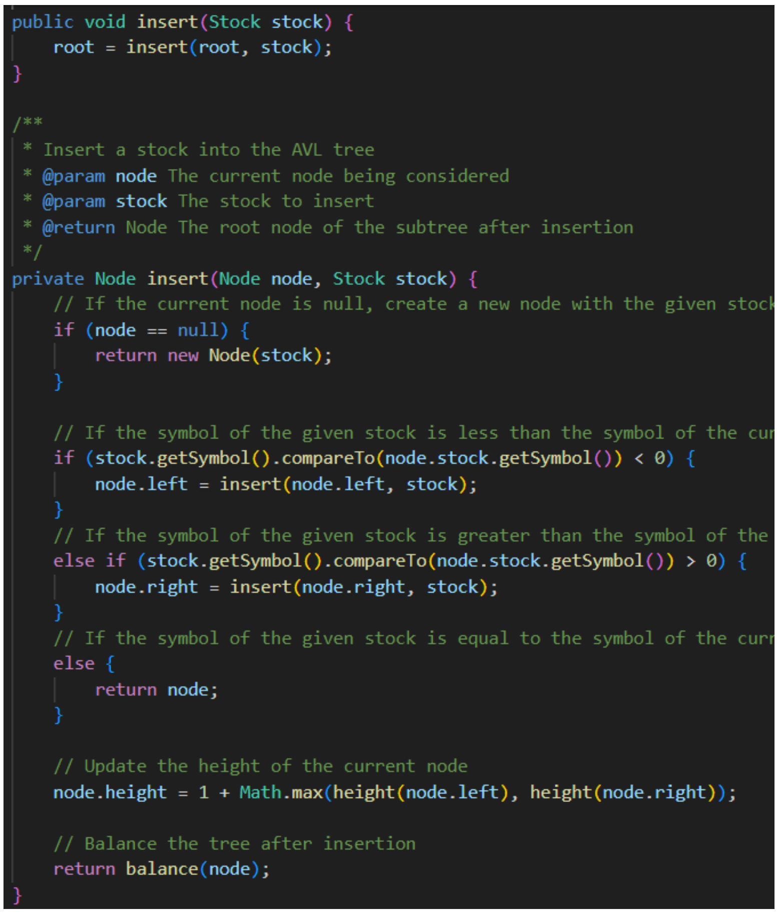
Remove and search are called in StockManager. Our avlTree object calls its own functions.

- AVLTree



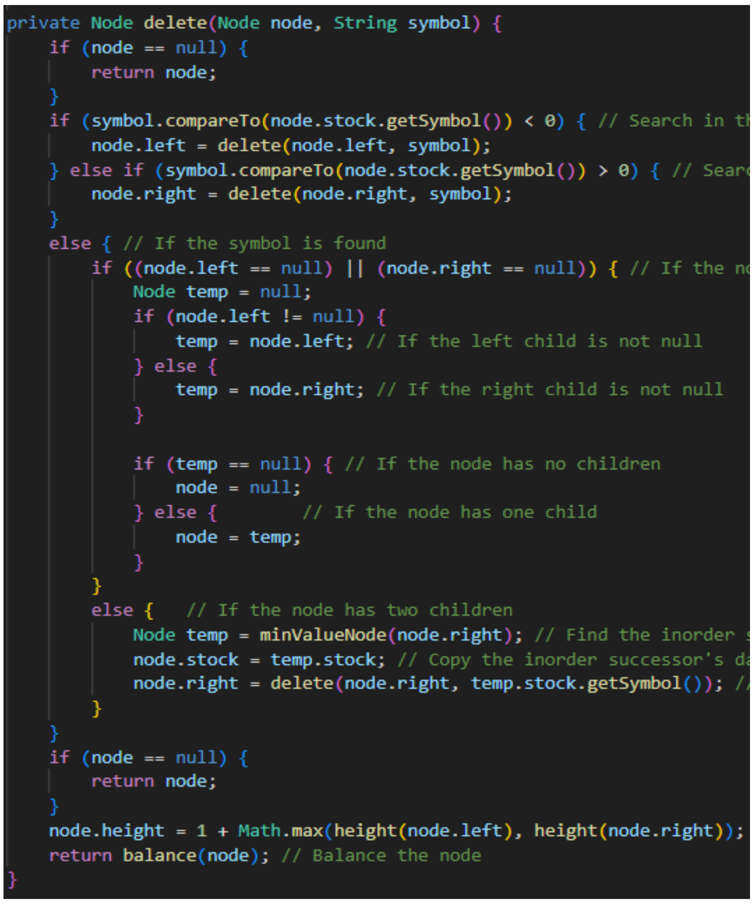
```java
public class AVLTree {

    /**
     * Node class for the AVL tree
     */
    private class Node {
        Stock stock; // Stock object
        Node left, right; // Left and right chi
        int height;// Height of the node

        /**
         * Constructor for the Node class
         * @param stock Stock object
         *
         */
        Node(Stock stock) {
            this.stock = stock;
            this.height = 1;
        }
    }

    private Node root; // Root of the AVL tree
```

There is a Node inner class in AvlTree. Each node holds a stock. And it keeps the left and right nodes connected to it.



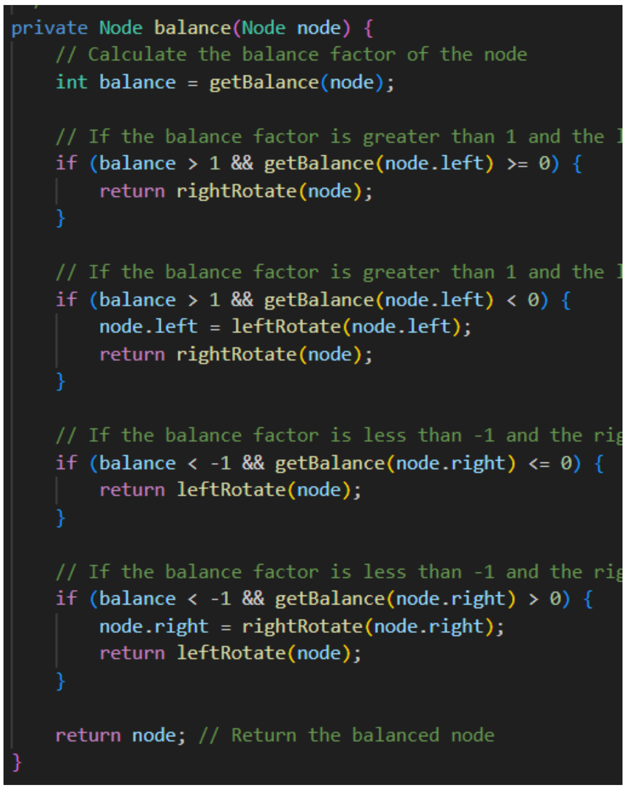
```
public void insert(Stock stock) {
    root = insert(root, stock);
}

/**
 * Insert a stock into the AVL tree
 * @param node The current node being considered
 * @param stock The stock to insert
 * @return Node The root node of the subtree after insertion
 */
private Node insert(Node node, Stock stock) {
    // If the current node is null, create a new node with the given stock
    if (node == null) {
        return new Node(stock);
    }

    // If the symbol of the given stock is less than the symbol of the cur
    if (stock.getSymbol().compareTo(node.stock.getSymbol()) < 0) {
        node.left = insert(node.left, stock);
    }
    // If the symbol of the given stock is greater than the symbol of the
    else if (stock.getSymbol().compareTo(node.stock.getSymbol()) > 0) {
        node.right = insert(node.right, stock);
    }
    // If the symbol of the given stock is equal to the symbol of the curr
    else {
        return node;
    }

    // Update the height of the current node
    node.height = 1 + Math.max(height(node.left), height(node.right));

    // Balance the tree after insertion
    return balance(node);
}
```

In the insert function, first, if the current node is null, it opens a new node. If it is not null, symbol compare is performed. Accordingly, a new recursive function is called for left and right. The height of the node is updated. Rotate functions are then called to balance the tree.



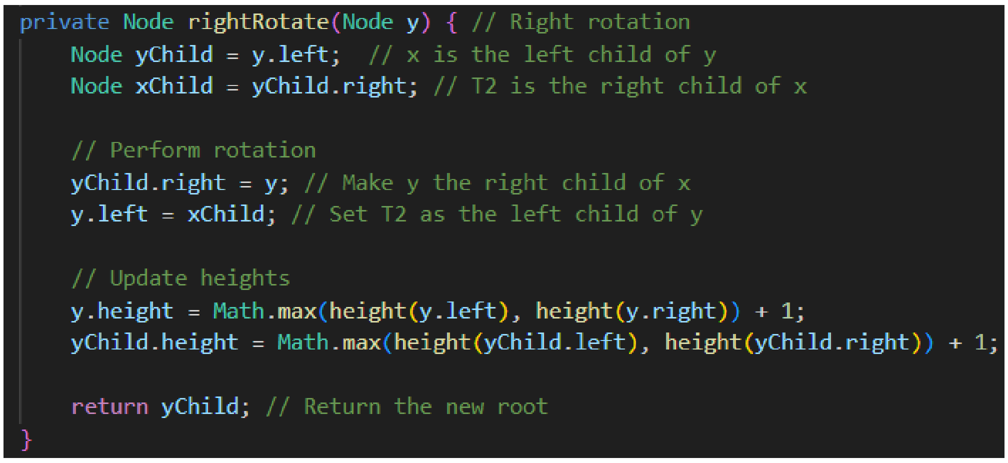
```java
private Node delete(Node node, String symbol) {
    if (node == null) {
        return node;
    }
    if (symbol.compareTo(node.stock.getSymbol()) < 0) { // Search in th
        node.left = delete(node.left, symbol);
    } else if (symbol.compareTo(node.stock.getSymbol()) > 0) { // Searc
        node.right = delete(node.right, symbol);
    }
    else { // If the symbol is found
        if ((node.left == null) || (node.right == null)) { // If the no
            Node temp = null;
            if (node.left != null) {
                temp = node.left; // If the left child is not null
            } else {
                temp = node.right; // If the right child is not null
            }

            if (temp == null) { // If the node has no children
                node = null;
            } else {         // If the node has one child
                node = temp;
            }
        }
        else {   // If the node has two children
            Node temp = minValueNode(node.right); // Find the inorder s
            node.stock = temp.stock; // Copy the inorder successor's da
            node.right = delete(node.right, temp.stock.getSymbol()); //
        }
    }
    if (node == null) {
        return node;
    }
    node.height = 1 + Math.max(height(node.left), height(node.right));
    return balance(node); // Balance the node
}
```

I first checked for null in the delete function. I compared the symbol variable that came as a parameter, starting from the root node. In string comparison, I called left and right recursive functions according to size and size. When the string is synchronized, the node is found. The first if check determines whether the node we are deleting has no child nodes or only one child node. If so, we need to determine which child node it is. I did this again with the if control. If the node has a left child node, the temp is set to that left child node. If there is no left child node, temp is set to the right child node. If temp is still null, it means it has no child nodes. In this case, we can make the node null. If temp is not null, it means that node has a child node. In this case, we set equal to its child node instead of the current node. If there are two child nodes, the minValueNode function finds the node with the smallest value in the given subtree. This is the leftmost node of the right subtree. So why did we send the right node to minValueNode? => because the smallest node at the bottom right is the closest node to the node we will delete. It is the most logical node to replace the node to be deleted. It seems to preserve the order of the tree. Then, we replace the data of the node with the smallest value we found with the data of the node that is being deleted. I then deleted the inorder successor node after replacing it. I updated the height of the node. And finally, I called the balance function to restore balance to the tree.



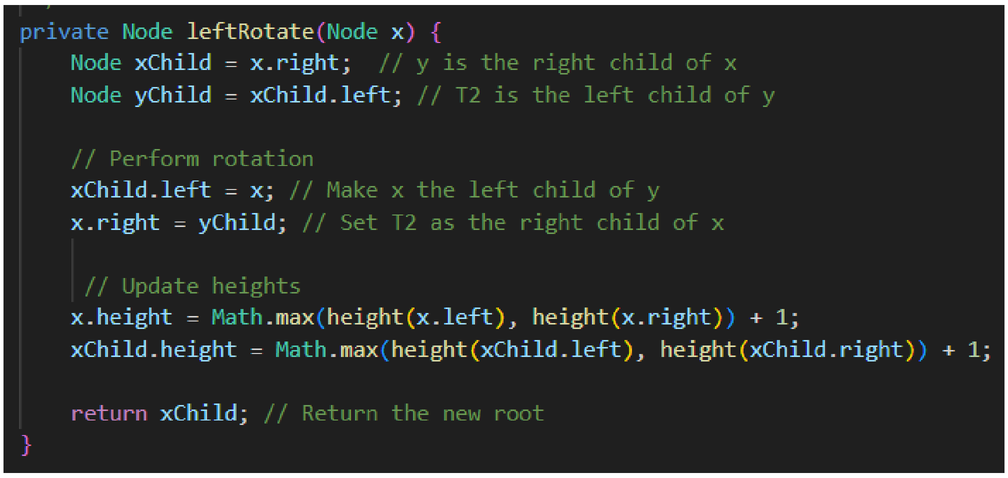
```
private Node balance(Node node) {
    // Calculate the balance factor of the node
    int balance = getBalance(node);

    // If the balance factor is greater than 1 and the l
    if (balance > 1 && getBalance(node.left) >= 0) {
        return rightRotate(node);
    }

    // If the balance factor is greater than 1 and the l
    if (balance > 1 && getBalance(node.left) < 0) {
        node.left = leftRotate(node.left);
        return rightRotate(node);
    }

    // If the balance factor is less than -1 and the rig
    if (balance < -1 && getBalance(node.right) <= 0) {
        return leftRotate(node);
    }

    // If the balance factor is less than -1 and the rig
    if (balance < -1 && getBalance(node.right) > 0) {
        node.right = rightRotate(node.right);
        return leftRotate(node);
    }

    return node; // Return the balanced node
}
```

With the getBalance function, the difference between the heights of the left and right subtrees of a given node is calculated. If this balance variable is greater than 1 or less than -1, it is necessary to rebalance the tree. The balance function rebalances the tree based on this balance factor. If the balance variable is greater than 1 and the balance variable of the left subtree is non-negative, a right rotation is performed. If the balance variable is greater than 1 and the balance variable of the left subtree is negative, first a left and then a right rotation is performed. Similarly, if the balance variable is less than 1 and the balance variable of the right subtree is non-positive, a left rotation is performed. If the balance factor is less than -1 and the balance factor of the right subtree is positive, first a right and then a left turn is made. Thus, we calculate search, insertion, deletion and update operations in LogN time.



```java
private Node rightRotate(Node y) { // Right rotation
    Node yChild = y.left;  // x is the left child of y
    Node xChild = yChild.right; // T2 is the right child of x

    // Perform rotation
    yChild.right = y; // Make y the right child of x
    y.left = xChild; // Set T2 as the left child of y

    // Update heights
    y.height = Math.max(height(y.left), height(y.right)) + 1;
    yChild.height = Math.max(height(yChild.left), height(yChild.right)) + 1;

    return yChild; // Return the new root
}
```

By rotating the y node to the right, the yChild node becomes the new root of the tree. yChild's right child becomes y, while xChild becomes y's left child. After this process, the heights of the nodes are updated and the new root is returned.

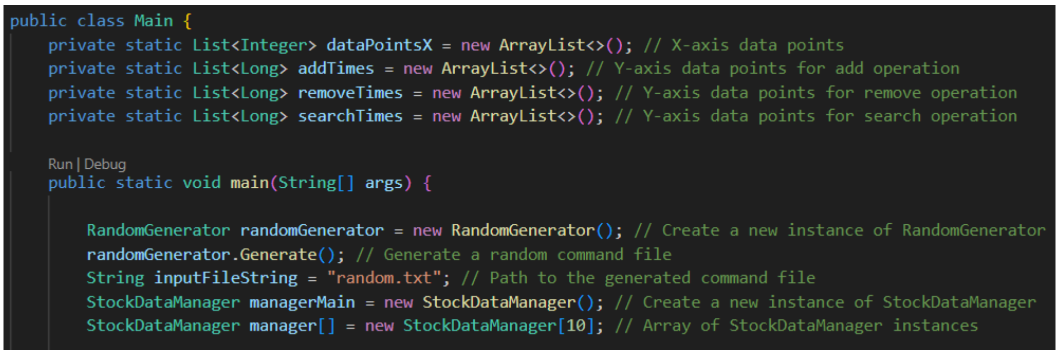


```java
private Node leftRotate(Node x) {
    Node xChild = x.right;  // y is the right child of x
    Node yChild = xChild.left; // T2 is the left child of y

    // Perform rotation
    xChild.left = x; // Make x the left child of y
    x.right = yChild; // Set T2 as the right child of x

     // Update heights
    x.height = Math.max(height(x.left), height(x.right)) + 1;
    xChild.height = Math.max(height(xChild.left), height(xChild.right)) + 1;

    return xChild; // Return the new root
}
```

By rotating the x node to the left, the xChild node becomes the new root of the tree. While xChild's left child becomes x, yChild becomes x's right child. After this process, the heights of the nodes are updated and the new root is returned.

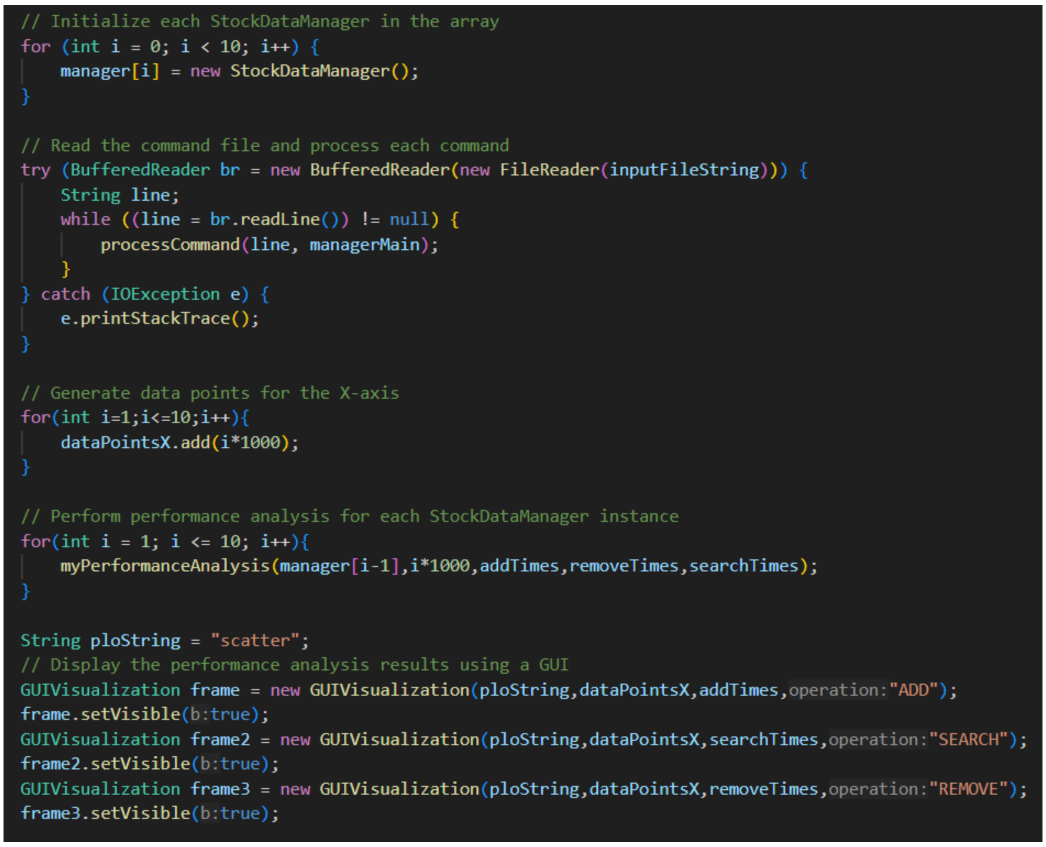
- MAIN

In the main code, I first kept a list for x points. And I kept separate lists for other transactions. When the program started, I called the randomGenerator object and generated random inputs. And to create a graph, I created an array with 10 elements, took the average times and tried to reach LogN time in the graph.



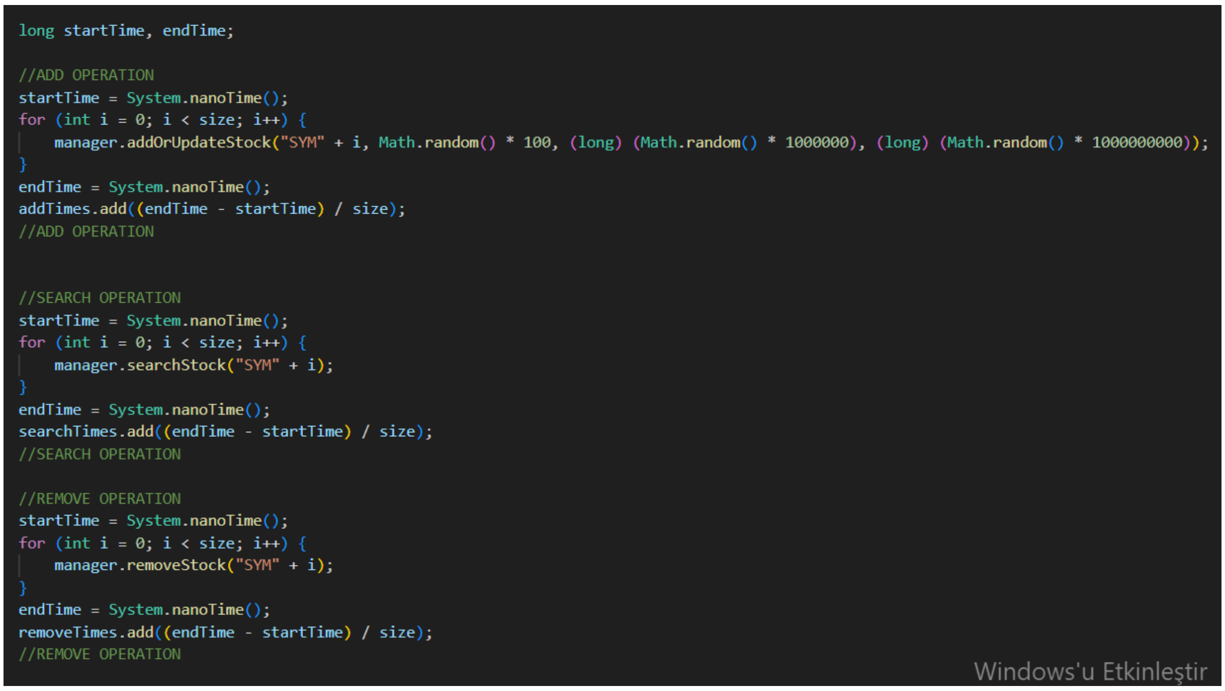
```
public class Main {
    private static List<Integer> dataPointsX = new ArrayList<>(); // X-axis data points
    private static List<Long> addTimes = new ArrayList<>(); // Y-axis data points for add operation
    private static List<Long> removeTimes = new ArrayList<>(); // Y-axis data points for remove operation
    private static List<Long> searchTimes = new ArrayList<>(); // Y-axis data points for search operation

    Run | Debug
    public static void main(String[] args) {

        RandomGenerator randomGenerator = new RandomGenerator(); // Create a new instance of RandomGenerator
        randomGenerator.Generate(); // Generate a random command file
        String inputFileString = "random.txt"; // Path to the generated command file
        StockDataManager managerMain = new StockDataManager(); // Create a new instance of StockDataManager
        StockDataManager manager[] = new StockDataManager[10]; // Array of StockDataManager instances
```

In this part of Main, I took the data from the randomly generated input and sent it to my managerMain object, which does not have a test tree. For testing, I created a manager array with 10 elements and created 3 gui screens to see the graphics.



```
// Initialize each StockDataManager in the array
for (int i = 0; i < 10; i++) {
    manager[i] = new StockDataManager();
}

// Read the command file and process each command
try (BufferedReader br = new BufferedReader(new FileReader(inputFileString))) {
    String line;
    while ((line = br.readLine()) != null) {
        processCommand(line, managerMain);
    }
} catch (IOException e) {
    e.printStackTrace();
}

// Generate data points for the X-axis
for(int i=1;i<=10;i++){
    dataPointsX.add(i*1000);
}

// Perform performance analysis for each StockDataManager instance
for(int i = 1; i <= 10; i++){
    myPerformanceAnalysis(manager[i-1],i*1000,addTimes,removeTimes,searchTimes);
}

String ploString = "scatter";
// Display the performance analysis results using a GUI
GUIVisualization frame = new GUIVisualization(ploString,dataPointsX,addTimes,operation:"ADD");
frame.setVisible(b:true);
GUIVisualization frame2 = new GUIVisualization(ploString,dataPointsX,searchTimes,operation:"SEARCH");
frame2.setVisible(b:true);
GUIVisualization frame3 = new GUIVisualization(ploString,dataPointsX,removeTimes,operation:"REMOVE");
frame3.setVisible(b:true);
```

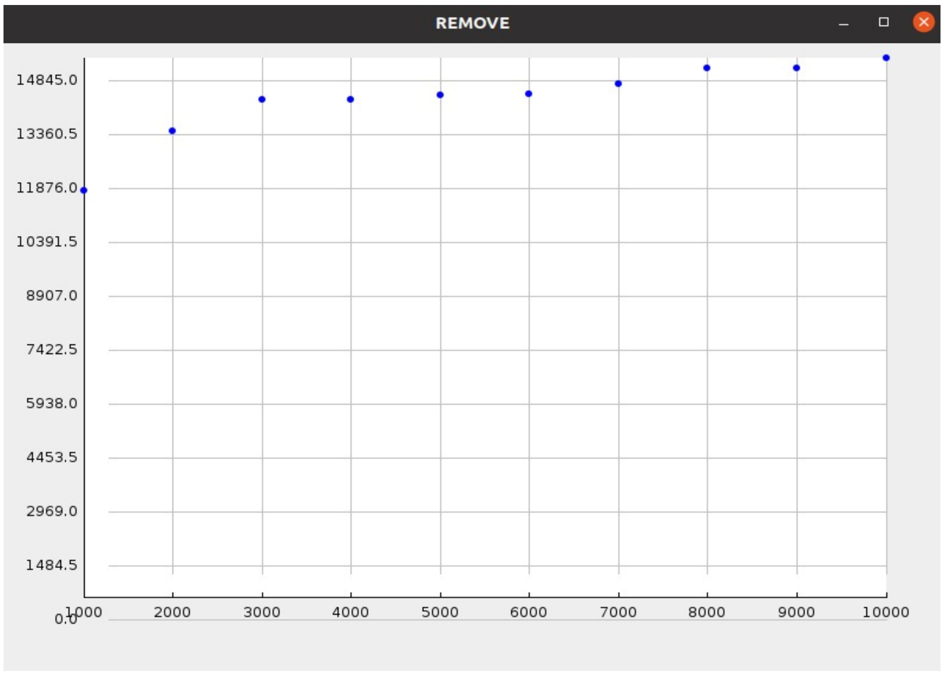
In the PerformanceAnalysis function, I set a time variable for the start and end of the processes and took an average of the last measured time for each tree. I added this to the list of that process. Then I sent these lists as parameters to the gui object.



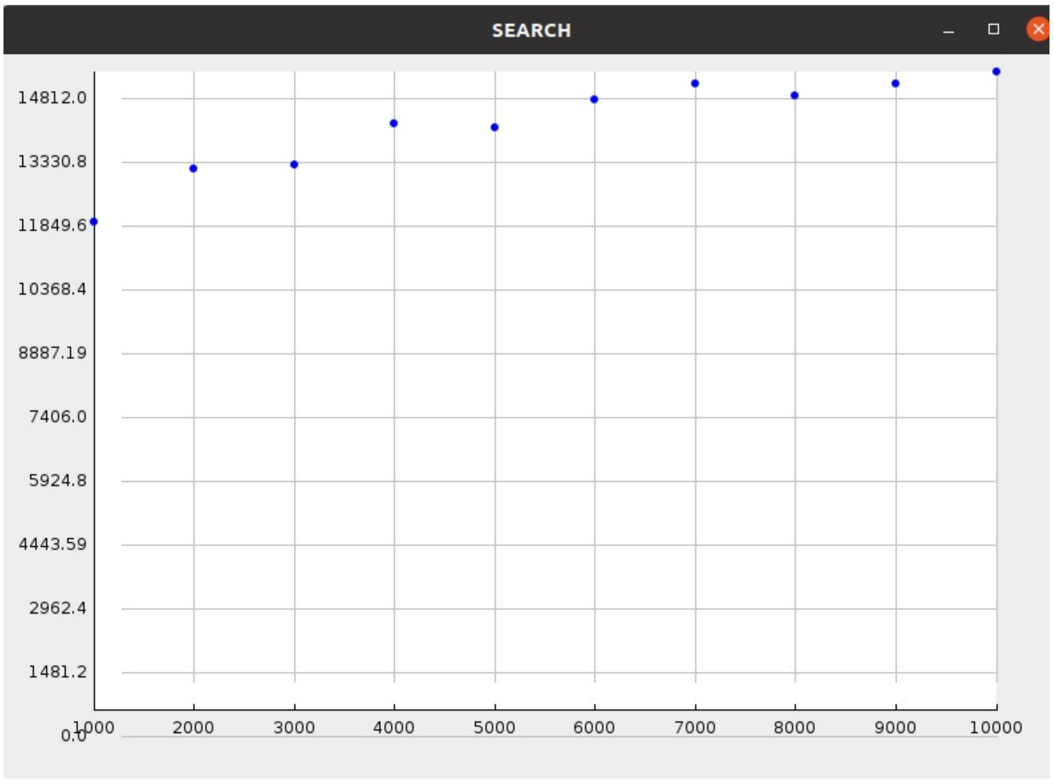
```java
long startTime, endTime;

//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();
addTimes.add((endTime - startTime) / size);
//ADD OPERATION


//SEARCH OPERATION
startTime = System.nanoTime();
for (int i = 0; i < size; i++) {
    manager.searchStock("SYM" + i);
}
endTime = System.nanoTime();
searchTimes.add((endTime - startTime) / size);
//SEARCH OPERATION

//REMOVE OPERATION
startTime = System.nanoTime();
for (int i = 0; i < size; i++) {
    manager.removeStock("SYM" + i);
}
endTime = System.nanoTime();
removeTimes.add((endTime - startTime) / size);
//REMOVE OPERATION
```

- GRAPHS:

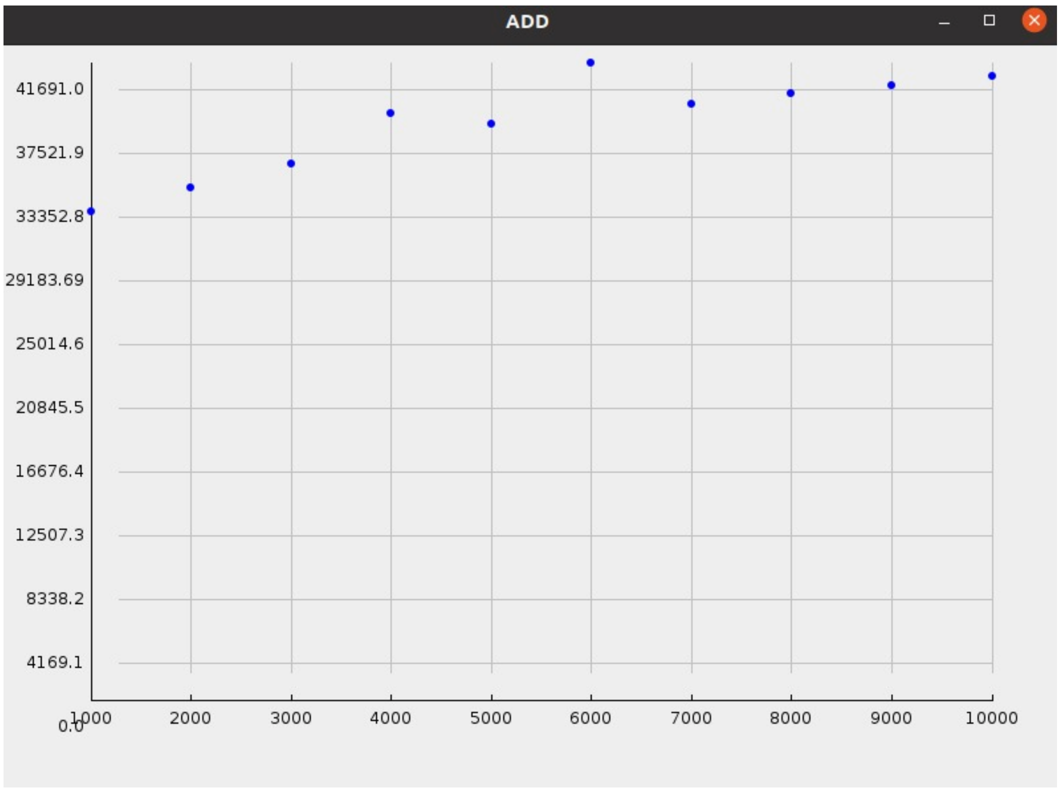
REMOVE GRAPH



SEARCH GRAPH


SEARCH
14812.0
13330.8
11849.6
10368.4
8887.19
7406.0
5924.8
4443.59
2962.4
1481.2
0.0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000

ADD GRAPH


ADD
41691.0
37521.9
33352.8
29183.69
25014.6
20845.5
16676.4
12507.3
8338.2
4169.1
0.0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000