2024

CSE222 HW7 - Report

ELIFNUR KABALCI
1801042617

Implementations

Stock.java

```java
public Stock(String symbol, double price, long volume, long marketCap) {
    this.symbol = symbol;
    this.price = price;
    this.volume = volume;
    this.marketCap = marketCap;
}
```

This class helps us create products. It allows us to add a product whose symbol, volume, price and market cap we have to the stock. It consists of getter-setter and print function.

Script.py

It creates an input txt file when run from within Java code. It creates add, remove, search and update methods on the numbers given in the PDF, keeps them in an array and then mixes them. While writing to the file, 1000 add methods are called at first. The mixed commands are then added to the file.

StockDataManager.java

Constructor I creates an avltree from scratch and the read commands are separated here and the relevant part in the avltree file is called depending on its type.

AVLTreePanel.java

This method was not normally available. However, I added it because I wanted to visually see whether the nodes were implemented correctly.

GuiVisualization.java

In the sample code given to us for this class, only plotType was given as a constructor parameter. I added avgTime. This became the y axis of my graphs. A list containing Add, Search or Remove un time data. I added TreeSize. This is also a list. Here, I brought information about how many nodes are in the tree in each speed data. This created the x-axis of the graph. Finally, I brought in an s string. This is the type (add, remove, search) information that will be added to the header so that we can see which data is which, since the graphs appear on the screen at the same time.

According to this code, the sizes of the data arriving at x and y coordinates must be the same. So I got the error for a while. Once I figured this out and edited it, the errors were gone. In the graph that appeared on the first screen, the data was wandering around. I got rid of this by changing the drawGraph method to interpret it logarithmically. Thus, the code took its current form.

AVLTree.java

It contains an inner Node class. This class holds the data held by each node of the tree. I implemented the basic features of AVL tree such as adding and removing data in this class.

InOrder Traversal:

```
private void inOrderTraversal(Node node) {
    if (node == null) {
        return;
    }
    inOrderTraversal(node.left);
    System.out.println(node.stock.getSymbol());
    inOrderTraversal(node.right);
}
```

Traversal methods allow us to view the tree effectively. In the inorder, left, root and then right node are displayed.

PreOrder Traversal:

```
private void preOrderTraversal(Node node) {
    if (node == null) {
        return;
    }
    System.out.println(node.stock.getSymbol());
    preOrderTraversal(node.left);
    preOrderTraversal(node.right);
}
```

Here, viewing is done as root, left and right.

PostOrder Traversal:

```
private void postOrderTraversal(Node node) {
    if (node == null) {
        return;
    }
    postOrderTraversal(node.left);
    postOrderTraversal(node.right);
    System.out.println(node.stock.getSymbol());
}
```

Here, viewing is done as left, right and root.

Left Rotation:

```
private Node leftRotation(Node disBalancedNode) {
    Node newNode = disBalancedNode.right;
    disBalancedNode.right = newNode.left;
    newNode.left = disBalancedNode;
    heightUpdate(disBalancedNode, newNode);
    return newNode;
}
```

Here I am doing left rotation with O(1) complexity value. It's a kind of swap method. Determine the imbalanced node and assign the right side of this node to a new value. We assign the right side of the unbalanced node to the left of this node. We move the unbalanced node to the left of the new node. Then, the height update method mentioned below is called.

Right Rotation:

```
private Node rightRotation(Node disBalancedNode) {
    Node newNode = disBalancedNode.left;
    disBalancedNode.left = newNode.right;
    newNode.right = disBalancedNode;
    heightUpdate(disBalancedNode, newNode);
    return newNode;
}
```

Here I am doing right rotation with O(1) complexity value. We assign a new node to the node to the left of the unstable node. We assign the value on the right of this node to the left of the unbalanced node. We move the unbalanced node to the right of the new node. Then, the height update method mentioned below is called.

```
private void heightUpdate(Node disBalancedNode, Node newNode) {
    disBalancedNode.setHeight(1 + Math.max(getHeight(disBalancedNode.left), getHeight(disBalancedNode.right)));
    newNode.setHeight(1 + Math.max(getHeight(newNode.left), getHeight(newNode.right)));
}
```

This is the method added in both. The Max height value of the node is searched and assigned by increasing it by one.

Search:

```
private Node search(Node node, String symbol) {
    if (node == null) { /* send node(root) or recursive calling, if node is empty */
        return null;
    }
    if (node.stock.getSymbol().equals(symbol)) { /* found */
        return node;
    }
    if (node.stock.getSymbol().compareTo(symbol) > 0) { /* search in min nodes */
        return search(node.left, symbol);
    }
    return search(node.right, symbol); /* search in max nodes */
}
```

Thanks to the recursive design, the time complexity value was achieved with O(logn). It is created by calling the method again by comparing the queried data as a string.

Insert:

```
if (node == null) { /* if node is empty write there the stock */
    nodeCounter++; /* increase number of nodes */
    return new Node(stock);
}
if (node.stock.getSymbol().compareTo(stock.getSymbol()) > 0) { /* node > stock */
    node.left = insert(node.left, stock);
} else if (node.stock.getSymbol().compareTo(stock.getSymbol()) < 0) { /* node < stock */
    node.right = insert(node.right, stock);
} else { /* node == stock */
    return node;
}
```

Similar to what we use in the search method, it is based on recursive comparison of the examined node with the data properties to be added and signing when free space is found.

```
node.setHeight(1 + Math.max(getHeight(node.left), getHeight(node.right)));
int balance = getBalance(node); /* decide which balance will make */

if (balance > 1 && stock.getSymbol().compareTo(node.left.stock.getSymbol()) < 0) { /* ll - right rotation */
    return rightRotation(node);
}
if (balance < -1 && stock.getSymbol().compareTo(node.right.stock.getSymbol()) > 0) { /* rr - left rotation */
    return leftRotation(node);
}
if (balance > 1 && stock.getSymbol().compareTo(node.left.stock.getSymbol()) > 0) { /*
                                                                                    * lr - left and right
                                                                                    * rotations
                                                                                    */
    node.left = leftRotation(node.left);
    return rightRotation(node);
}
if (balance < -1 && stock.getSymbol().compareTo(node.right.stock.getSymbol()) < 0) { /*
                                                                                     * rl - right and left
                                                                                     * rotation
                                                                                     */
    node.right = rightRotation(node.right);
    return leftRotation(node);
}
return node;
```

In the continuation of the method, the height where the node is located is determined and assigned and the balance is questioned. If there is more than 1 height difference between the left and right branches, rotation methods are called to balance them.

Delete:

```
if (node == null) { /* node is empty, maybe tree is empty */
    return node;
}
if (node.stock.getSymbol().compareTo(symbol) > 0) { /* node > symbol */
    node.left = delete(node.left, symbol);
} else if (node.stock.getSymbol().compareTo(symbol) < 0) { /* node < symbol */
    node.right = delete(node.right, symbol);
} else { /* node == symbol */
    if (node.left == null) { /* left node is empty */
        return node.right;
    } else if (node.right == null) { /* right node is empty */
        return node.left;
    }
    /* left and right nodes are not empty */
    Node temp = findMin(node.right); /* dedicate min value's node */
    node.stock = temp.stock; /* write the symbol value, not swap node's */
    node.right = delete(node.right, temp.stock.getSymbol()); /* call for balancing */
    nodeCounter--; /* decrease number of nodes */
}
```

Just like insert, we search for the node recursively. There are 3 types of situations when found. If the left is empty, if the right is empty, if both sides are full. In the first two cases, child direct is assigned to replace the deleted one. In the other case, the smallest value to the right of the unbalanced data is found and the data is transferred there. Then, the starting position of the small data found is deleted from the tree.

Afterwards, it is about establishing balance, similar to the second part of the insert.

Main.java

performPerformanceAnalysis:

In this area, I included the time calculation expression outside the for loop and deleted the divided size part, kept each data in a list, transferred it to the gui class in its own area and had it plotted.

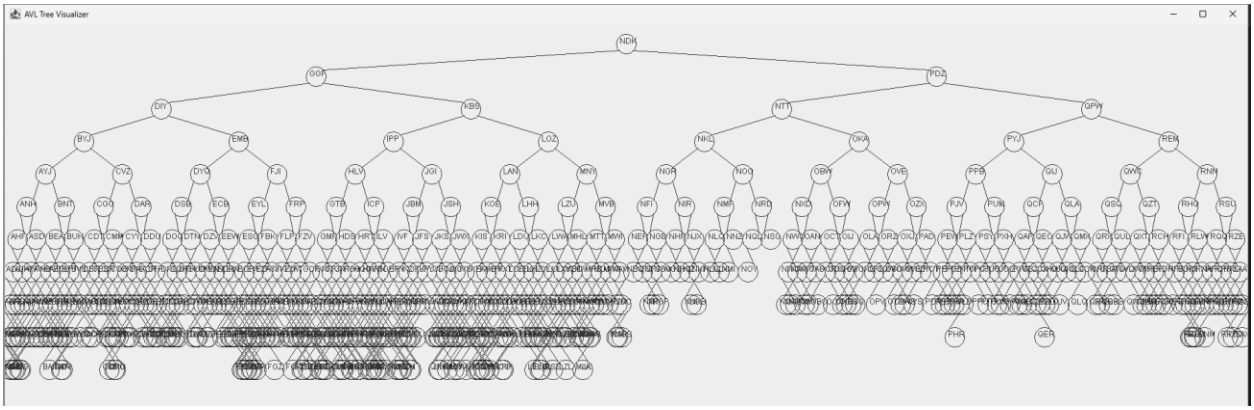
processCommand:

I did not make any changes to this method.

pythonScriptCalling:

Main is also called. It runs the script.py file to randomly prepare an input file each time, without starting any operations. In this section, depending on where the python exe file is located on your computer and the location of your project is important. If this part is not changed when trying to run it on another device, an error will be received.

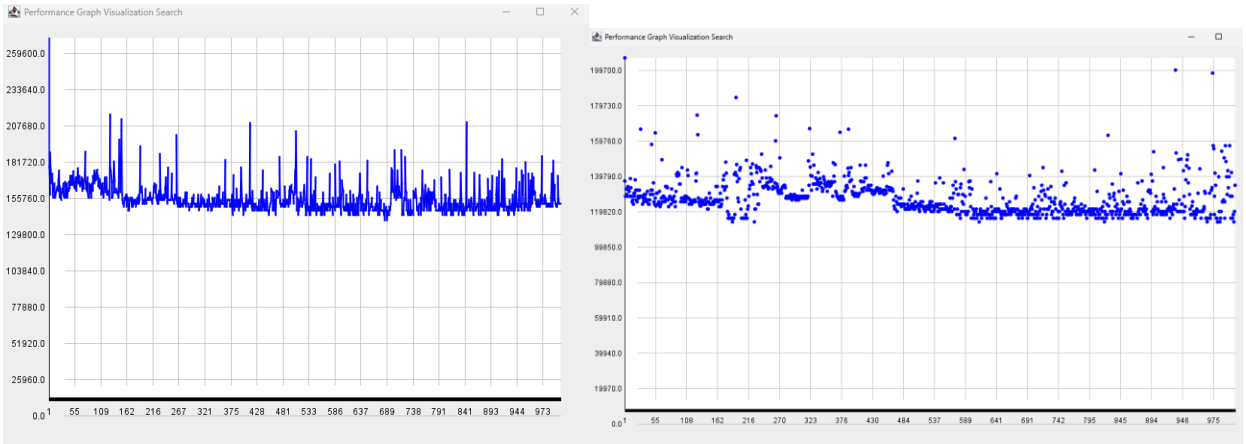
Outputs – Graphs

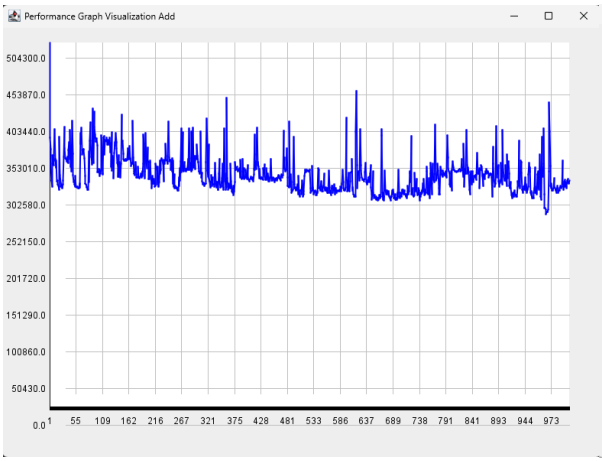


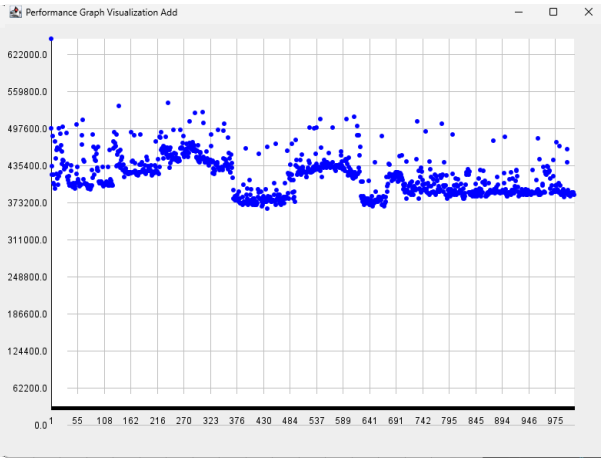
This part is not included in the assignment. It has been added to more easily examine the status of the data in the input file.

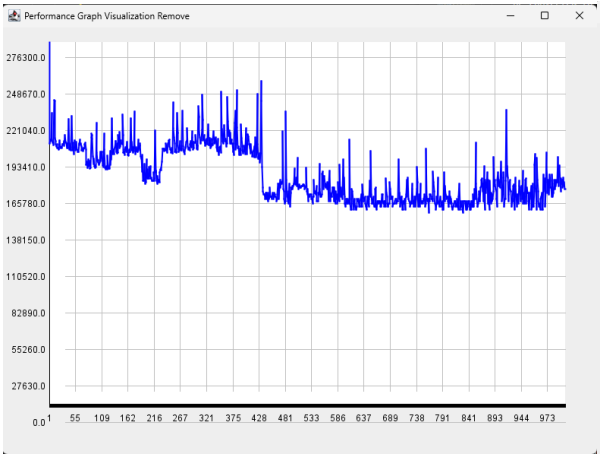
The graphics belong to the same tree vertically. Since the tree occurs randomly every time the code runs, the data on the left and right sides are different from each other. However, I added it to show how the graph looks on average with scatter and line.

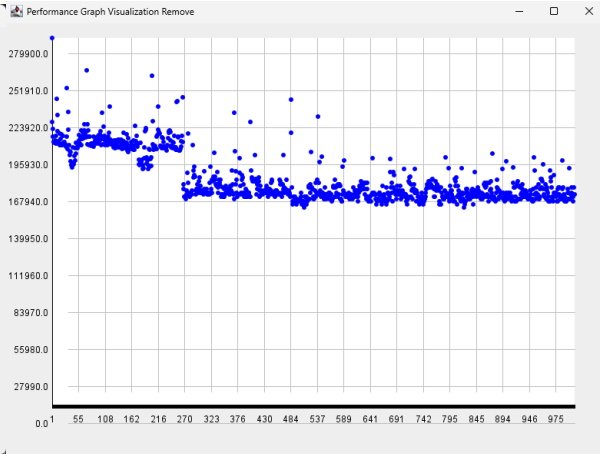
```
PS C:\Users\e.kabalci2018\Desktop\HW7\HW7_Demo\src> javac Main.java
PS C:\Users\e.kabalci2018\Desktop\HW7\HW7_Demo\src> java Main "input.txt"
```





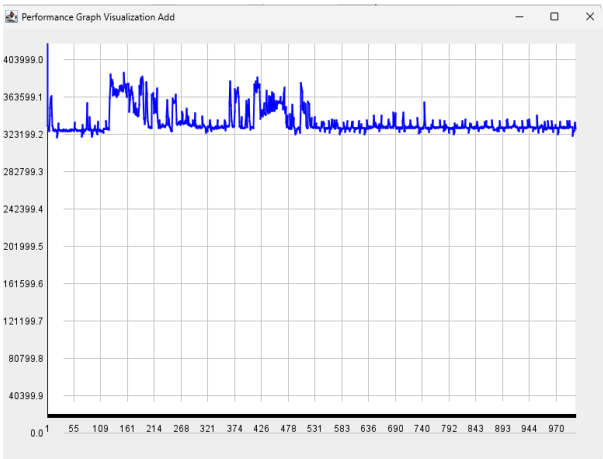


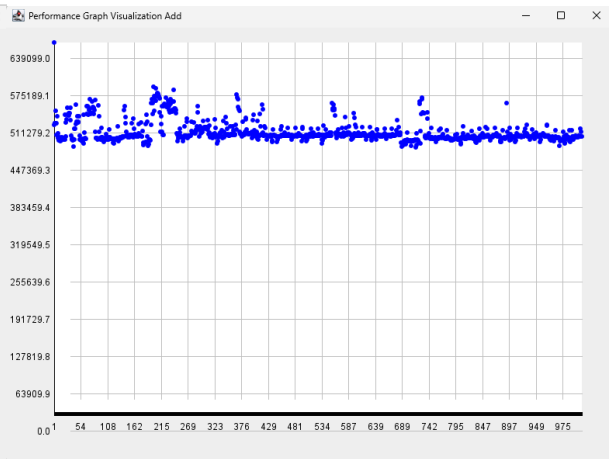


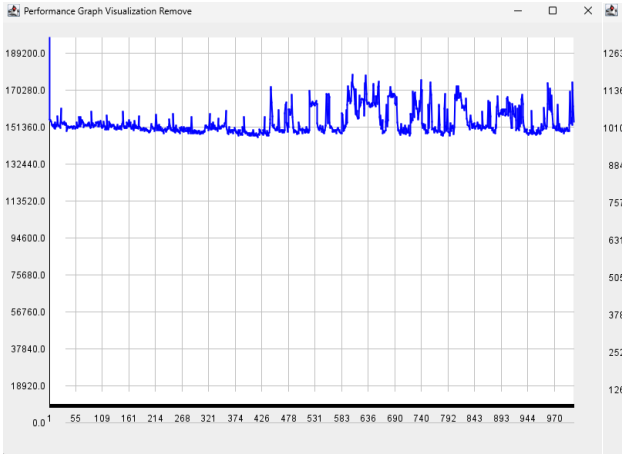


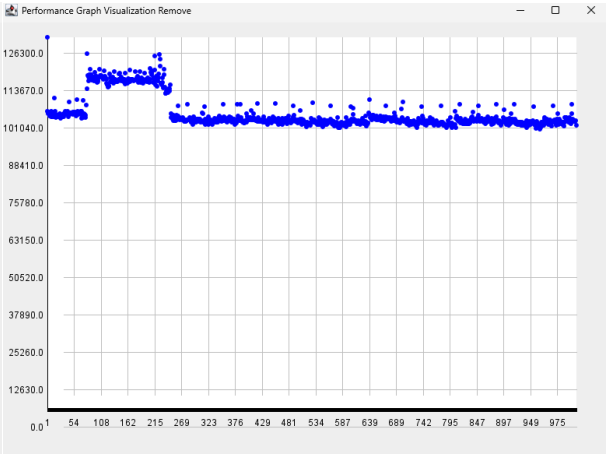

```
PS C:\Users\e.kabalci2018\Desktop\HW7\HW7_Demo\src> javac Main.java
PS C:\Users\e.kabalci2018\Desktop\HW7\HW7_Demo\src> java -Xint Main "input.txt"
```

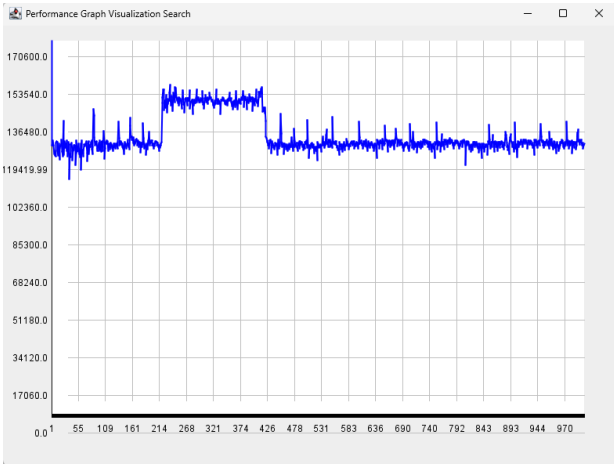
This was added to show the effect of the -Xint plugin on drawing the graph.

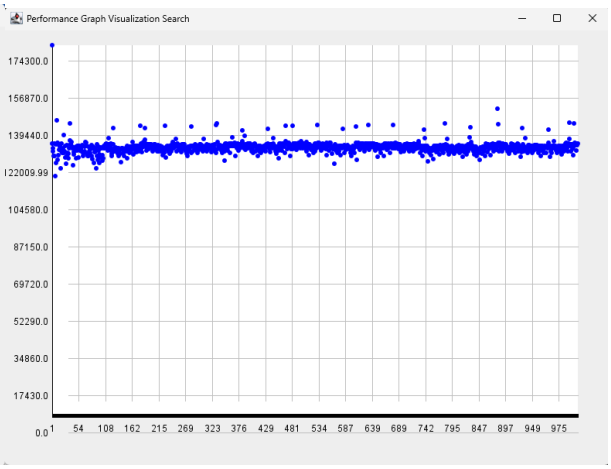




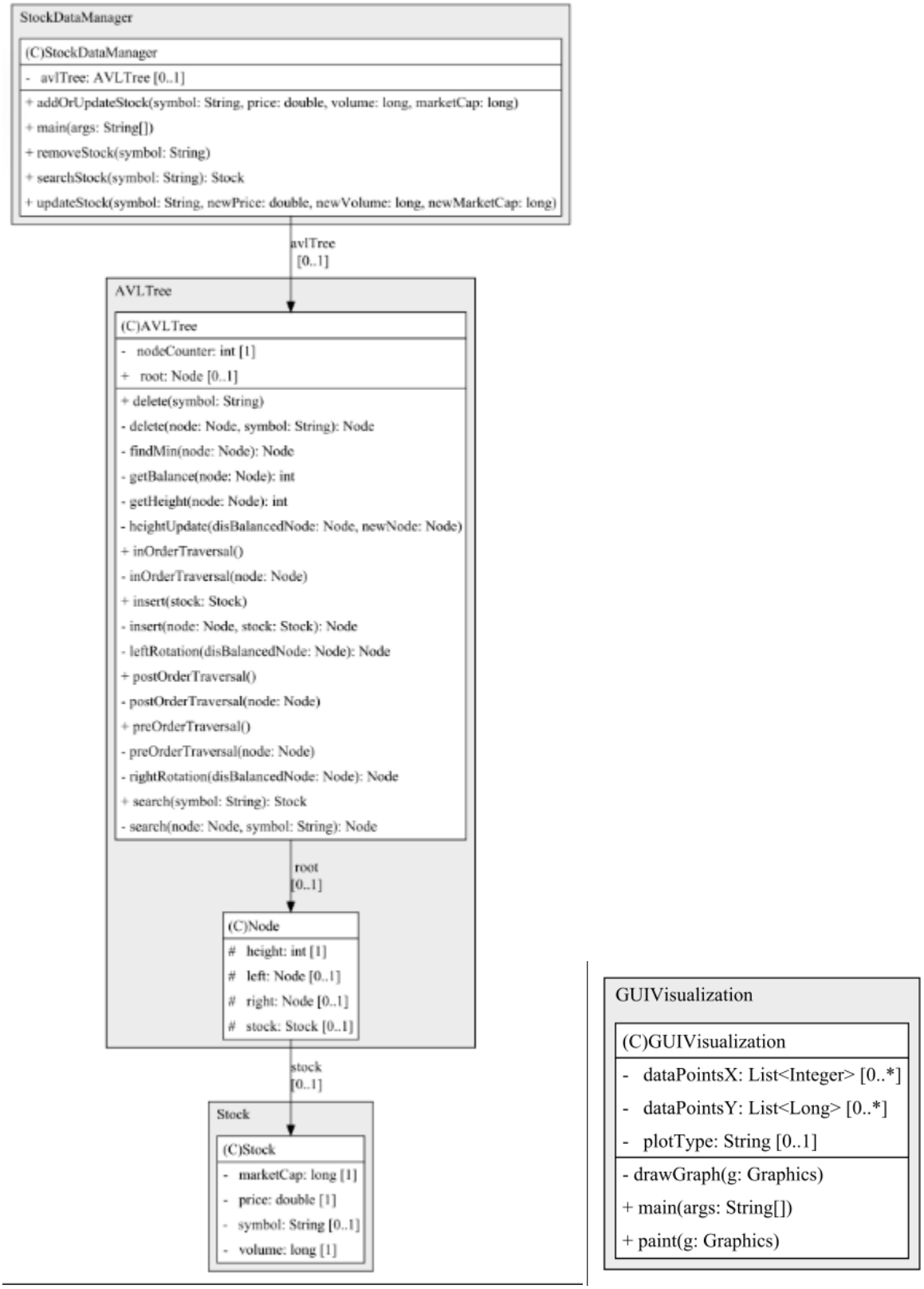

Performance Graph Visualization Remove
189200.0
170280.0
151360.0
132440.0
113520.0
94600.0
75680.0
56760.0
37840.0
18920.0
0.0
1 55 109 161 214 268 321 374 426 478 531 583 636 690 740 792 843 893 944 970

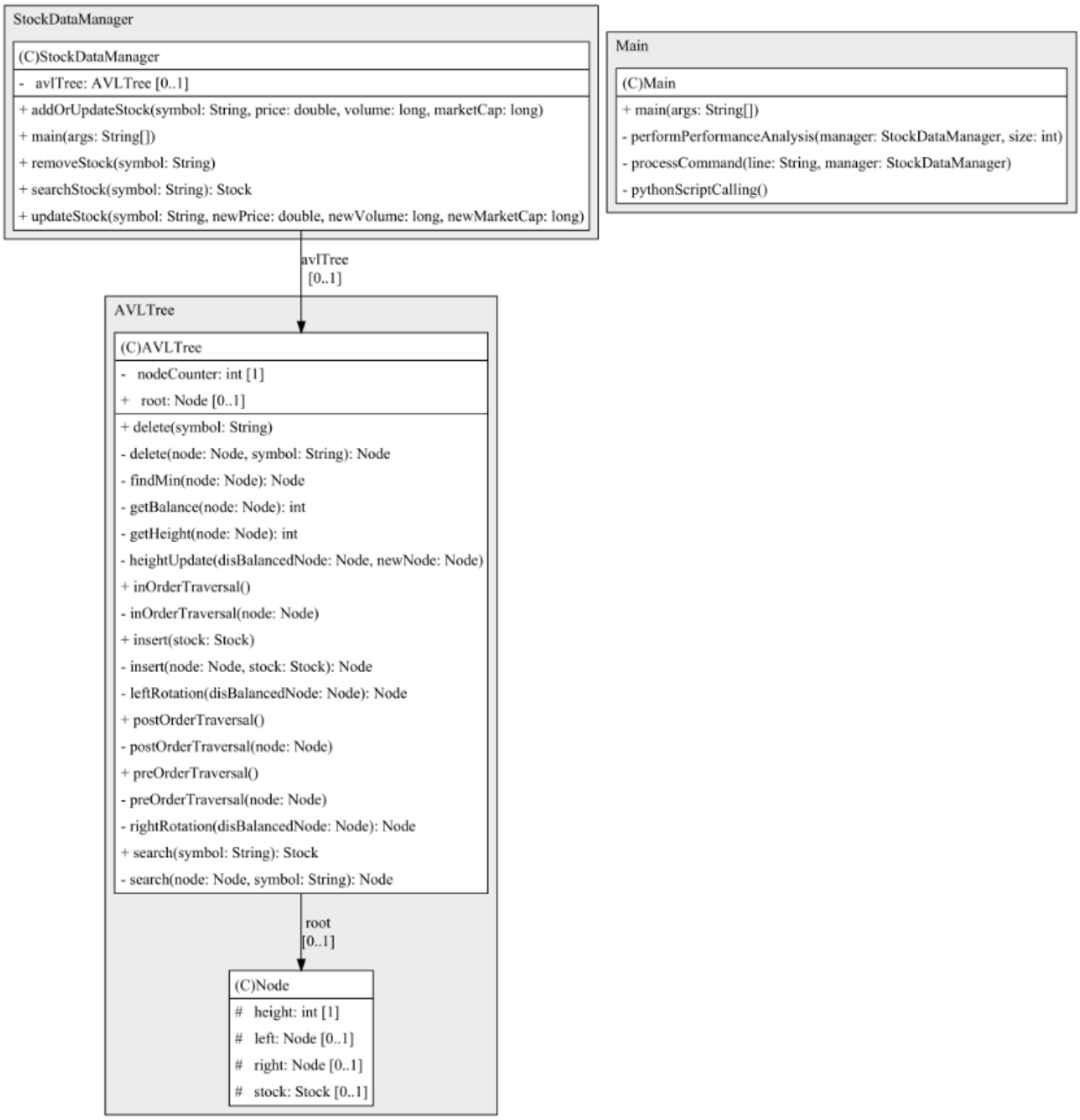

Performance Graph Visualization Remove
126300.0
113670.0
101040.0
88410.0
75780.0
63150.0
50520.0
37890.0
25260.0
12630.0
0.0
1 54 108 162 215 269 323 376 429 481 534 587 639 689 742 795 847 897 949 975


Performance Graph Visualization Search
170600.0
153540.0
136480.0
119419.99
102360.0
85300.0
68240.0
51180.0
34120.0
17060.0
0.0
1 55 109 161 214 268 321 374 426 478 531 583 636 690 740 792 843 893 944 970


Performance Graph Visualization Search
174300.0
156870.0
139440.0
122009.99
104580.0
87150.0
69720.0
52290.0
34860.0
17430.0
0.0
1 54 108 162 215 269 323 376 429 481 534 587 639 689 742 795 847 897 949 975

Class Diagrams:




StockDataManager
(C)StockDataManager
- avlTree: AVLTree [0..1]
+ addOrUpdateStock(symbol: String, price: double, volume: long, marketCap: long)
+ main(args: String[])
+ removeStock(symbol: String)
+ searchStock(symbol: String): Stock
+ updateStock(symbol: String, newPrice: double, newVolume: long, newMarketCap: long)
Main
(C)Main
+ main(args: String[])
- performPerformanceAnalysis(manager: StockDataManager, size: int)
- processCommand(line: String, manager: StockDataManager)
- pythonScriptCalling()
avlTree
[0..1]
AVLTree
(C)AVLTree
- nodeCounter: int [1]
+ root: Node [0..1]
+ delete(symbol: String)
- delete(node: Node, symbol: String): Node
- findMin(node: Node): Node
- getBalance(node: Node): int
- getHeight(node: Node): int
- heightUpdate(disBalancedNode: Node, newNode: Node)
+ inOrderTraversal()
- inOrderTraversal(node: Node)
+ insert(stock: Stock)
- insert(node: Node, stock: Stock): Node
- leftRotation(disBalancedNode: Node): Node
+ postOrderTraversal()
- postOrderTraversal(node: Node)
+ preOrderTraversal()
- preOrderTraversal(node: Node)
- rightRotation(disBalancedNode: Node): Node
+ search(symbol: String): Stock
- search(node: Node, symbol: String): Node
root
[0..1]
(C)Node
height: int [1]
left: Node [0..1]
right: Node [0..1]
stock: Stock [0..1]