CSE222-DATA STRUCTURES AND ALGORITHM DESIGN

HW7-REPORT

ADEM MURAT ÖZDEMİR-200104004110

Compiling and Running

Generate random input file:
    javac InputFileGenerator.java
    java InputFileGenerator

After the creation of file , you need to modify makefile for using created file.

makefile
```
1  JC = javac
2  JFLAGS = -g
3  TARGET = Main
4  SOURCES = Main.java AVLTree.java GUIVisualization.java Stock.java StockDataManager.java
5  DOCS = doc
6
7  default: $(TARGET)
8
9  $(TARGET): $(SOURCES)
10     $(JC) $(JFLAGS) $^
11     java -Xint $(TARGET) random1.txt
12
13 doc:
14     javadoc -d $(DOCS) $(SOURCES)
15
16 clean:
17     rm -f *.class
18
19 clean-doc:
20     rm -rf $(DOCS)
21
22 .PHONY: clean default doc clean-doc
```

You need to modify 11th line for using created file or what you want.

Compile and run the code : “make”

Clean the .class files : “make clean”

Create Javadoc documents : “make doc”

Clean the Javadoc : “make clean-doc”

Core Functions:

Explanation of insert function

Public Method: insert(Stock stock)

Purpose:

- To add a new stock to the AVL tree.

How It Works:

- It starts the process of adding the stock from the root of the tree by calling a helper method.

Private Method: insert(Node node, Stock stock)

Purpose:

- To find the correct place in the tree to add the new stock and ensure the tree remains balanced.

How It Works:

1. **Finding the Correct Spot**:
    - If the current spot (node) is empty (null), it means we found where to add the new stock. So, we create a new node here.
    - If the new stock's symbol is less than the current node's symbol, we go to the left child (left subtree).
    - If the new stock's symbol is greater, we go to the right child (right subtree).
    - If the symbols are the same, we update the existing stock information.
2. **Updating the Tree**:
    - After adding the stock, we update the height of the current node. The height helps us keep track of how "tall" the tree is at that point.
    - We then balance the tree if needed to ensure it remains an AVL tree (a balanced binary search tree).

Explanation of delete Function

Public Method: delete(String symbol)

Purpose:

- To remove a stock from the AVL tree based on its symbol.

How It Works:

- It starts the deletion process from the root of the tree by calling a helper method, delete(Node node, String symbol).

Private Method: delete(Node node, String symbol)

Purpose:

- To find and remove the stock with the given symbol from the subtree rooted at the specified node and ensure the tree remains balanced.

How It Works:

1. **Finding the Node to Delete**:
    - If the current node (node) is null, it means the symbol isn't found in the tree, so it returns null.
    - If the symbol is less than the current node's symbol, it recursively calls delete on the left child.
    - If the symbol is greater, it recursively calls delete on the right child.
    - If the symbols match, it means we've found the node to delete.
2. **Deleting the Node**:
    - **Case 1: Node with only one child or no child**:
        - If the node has no left child or no right child, we set temp to the non-null child or null if both are null.
        - If temp is null, the node has no children, so we set the node to null.
        - If temp is not null, we replace the node with temp.
    - **Case 2: Node with two children**:
        - Find the smallest node in the right subtree (minValueNode(node.right)).
        - Replace the current node's stock with the stock of the smallest node in the right subtree.
        - Recursively delete the smallest node in the right subtree.
3. **Updating the Tree**:
    - After deleting the node, update the height of the current node. The height helps us keep track of the depth of the tree.
    - Balance the tree by calling the balance(node) method to ensure the AVL tree properties are maintained.

Explanation of search Function

Public Method: search(String symbol)

Purpose:

- To find and return a stock with the given symbol from the AVL tree.

How It Works:

- Calls a helper method starting from the root.
- Returns the Stock object if found; otherwise, returns null.

Private Method: search(Node node, String symbol)

Purpose:

- To search for a stock within a subtree rooted at a given node.

How It Works:

- If the current node is null or matches the symbol, return the current node.
- If the symbol is less than the current node's symbol, search in the left subtree.
- If the symbol is greater, search in the right subtree.

Explanation of balance Function

Purpose:

- To ensure the AVL tree remains balanced after insertion or deletion operations.

How It Works:

- Calculates the balance factor of the given node.
- Performs rotations based on the balance factor to restore balance if needed.

Steps:

1. **Calculate Balance Factor**:
    - The balance factor is the difference in height between the left and right subtrees of the node.
2. **Check for Left-Heavy Case** (balanceFactor > 1):
    - If the node is left-heavy, it checks the balance factor of the left child to determine the type of rotation needed.

    - **Left-Left Case** (getBalanceFactor(node.left) >= 0):
        - Performs a right rotation on the current node.
    - **Left-Right Case** (getBalanceFactor(node.left) < 0):
        - Performs a left rotation on the left child, followed by a right rotation on the current node.

3. **Check for Right-Heavy Case** (balanceFactor < -1):
    - If the node is right-heavy, it checks the balance factor of the right child to determine the type of rotation needed.
    - **Right-Right Case** (getBalanceFactor(node.right) <= 0):
        - Performs a left rotation on the current node.
    - **Right-Left Case** (getBalanceFactor(node.right) > 0):
        - Performs a right rotation on the right child, followed by a left rotation on the current node.
4. **Return the Balanced Node**:
    - If the node is already balanced (-1 <= balanceFactor <= 1), it returns the node as is.
    - After performing the necessary rotations, it returns the new root of the subtree.

Explanation of leftRotate and rightRotate Functions

Private Method: leftRotate(Node x)

Purpose:

- To perform a left rotation on a given node to help balance the AVL tree.

How It Works:

1. **Identify Nodes**:
    - y is set to x's right child.
    - T2 is set to y's left child.
2. **Perform Rotation**:
    - y's left child is set to x.
    - x's right child is set to T2.
3. **Update Heights**:
    - Update the height of x and y.
4. **Return New Root**:
    - Return y as the new root of the subtree.

Private Method: rightRotate(Node y)

Purpose:

- To perform a right rotation on a given node to help balance the AVL tree.

How It Works:

1. **Identify Nodes**:
   - x is set to y's left child.
   - T2 is set to x's right child.
2. **Perform Rotation**:
   - x's right child is set to y.
   - y's left child is set to T2.
3. **Update Heights**:
   - Update the height of y and x.
4. **Return New Root**:
   - Return x as the new root of the subtree.

Explanation of getBalanceFactor Function

Purpose:

- To calculate the balance factor of a given node in the AVL tree.

How It Works:

1. **Check for Null Node**:
   - If the node is null, the balance factor is 0 because an empty node is considered balanced.
2. **Calculate Balance Factor**:
   - The balance factor is the difference in height between the left and right subtrees of the node.
   - Specifically, it is calculated as height(node.left) - height(node.right).

Returns:

- An integer representing the balance factor of the node.

Test Results

First Part Analysis (reads from a .txt file and do operations)

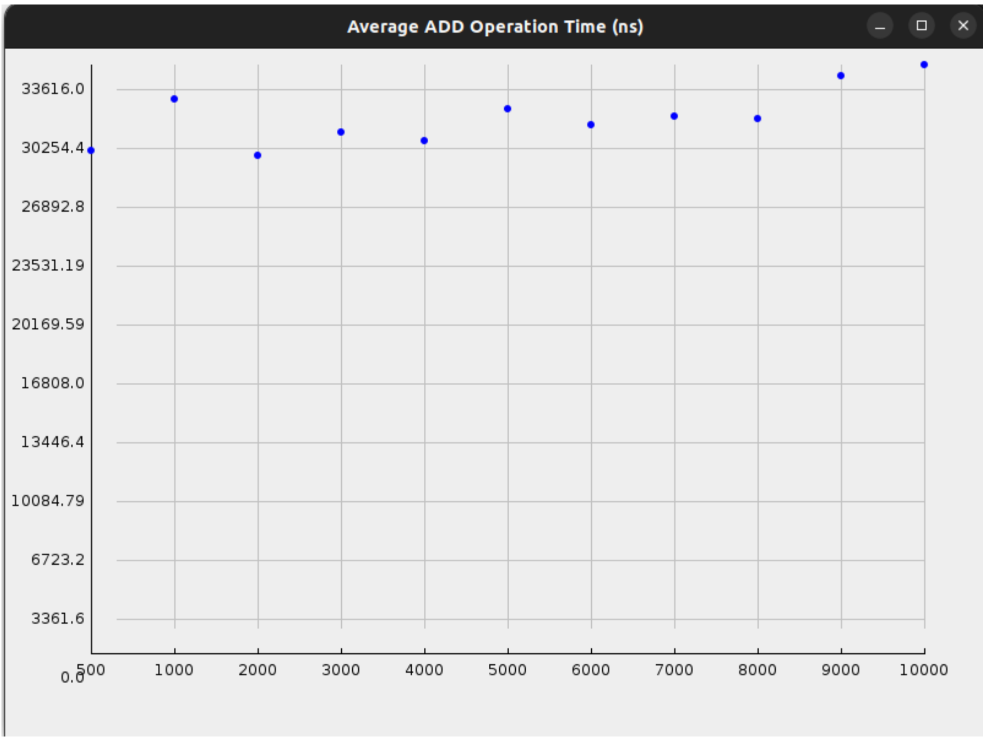
```
ademmurat@ademmurat-GL553VD:~/Desktop/DataStructures2024/homeworks/datahw7/HW7/HW7/HW7_Demo/src$ make
javac -g Main.java AVLTree.java GUIVisualization.java Stock.java StockDataManager.java
java -Xint Main random1.txt

First Part Analysis:
Average ADD time: 20329 ns
Average SEARCH time: 2378 ns
Average REMOVE time: 4969 ns
Average UPDATE time: 17490 ns
```

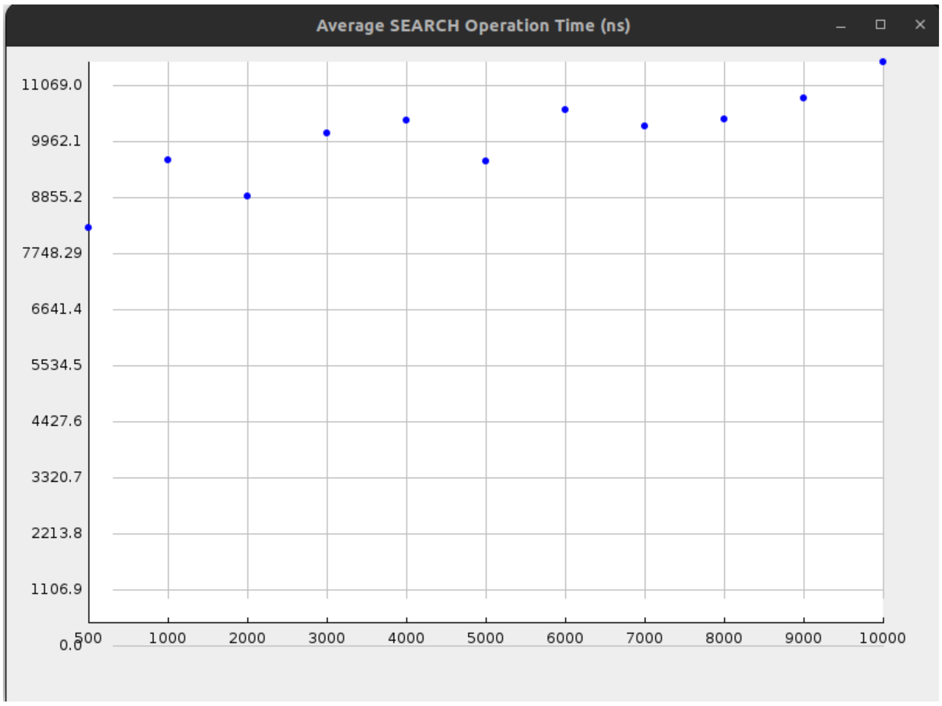
Second Part Analysis (creates a tree that created with random stocks to different sizes)

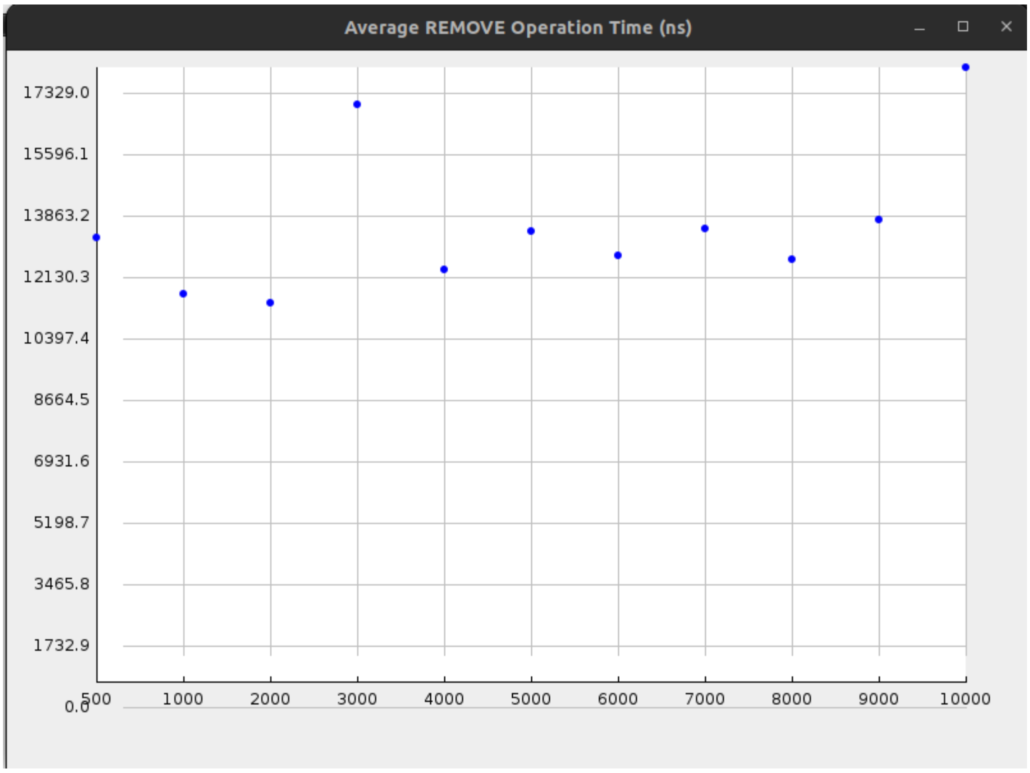
```
Second Part Analysis for size 500:
Average ADD time: 28716 ns
Average SEARCH time: 7798 ns
Average REMOVE time: 12547 ns
Average UPDATE time: 30275 ns

Second Part Analysis for size 1000:
Average ADD time: 31724 ns
Average SEARCH time: 9139 ns
Average REMOVE time: 10967 ns
Average UPDATE time: 31501 ns

Second Part Analysis for size 2000:
Average ADD time: 28503 ns
Average SEARCH time: 8427 ns
Average REMOVE time: 10706 ns
Average UPDATE time: 26624 ns

Second Part Analysis for size 3000:
Average ADD time: 29823 ns
Average SEARCH time: 9677 ns
Average REMOVE time: 16309 ns
Average UPDATE time: 29918 ns

Second Part Analysis for size 4000:
Average ADD time: 29278 ns
Average SEARCH time: 9917 ns
Average REMOVE time: 11633 ns
Average UPDATE time: 32029 ns

Second Part Analysis for size 5000:
Average ADD time: 31097 ns
Average SEARCH time: 9130 ns
Average REMOVE time: 12712 ns
Average UPDATE time: 29602 ns
```

```
Second Part Analysis for size 5000:
Average ADD time: 31097 ns
Average SEARCH time: 9130 ns
Average REMOVE time: 12712 ns
Average UPDATE time: 29602 ns

Second Part Analysis for size 6000:
Average ADD time: 30186 ns
Average SEARCH time: 10139 ns
Average REMOVE time: 12026 ns
Average UPDATE time: 30131 ns

Second Part Analysis for size 7000:
Average ADD time: 30726 ns
Average SEARCH time: 9820 ns
Average REMOVE time: 12802 ns
Average UPDATE time: 30100 ns

Second Part Analysis for size 8000:
Average ADD time: 30594 ns
Average SEARCH time: 9944 ns
Average REMOVE time: 11943 ns
Average UPDATE time: 30363 ns

Second Part Analysis for size 9000:
Average ADD time: 33041 ns
Average SEARCH time: 10360 ns
Average REMOVE time: 13039 ns
Average UPDATE time: 31942 ns

Second Part Analysis for size 10000:
Average ADD time: 33616 ns
Average SEARCH time: 11069 ns
Average REMOVE time: 17329 ns
Average UPDATE time: 32463 ns
```

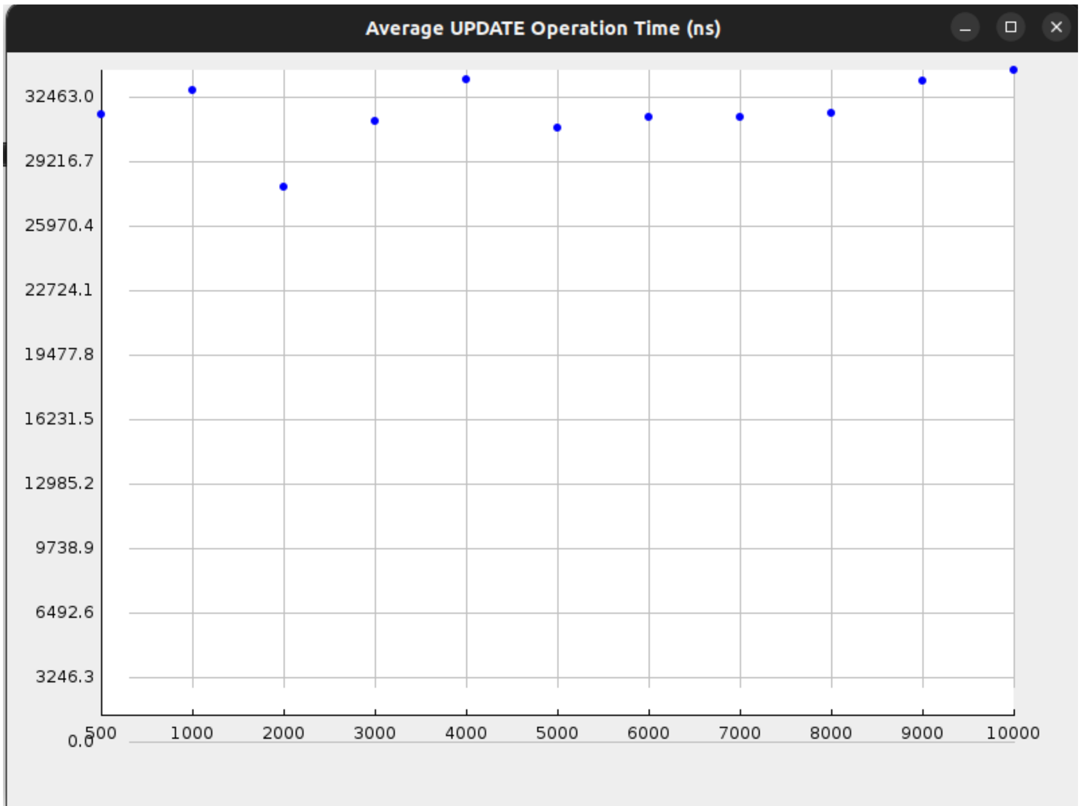
Plot Graphs for Second Part Average Time Analyses







Since the CPU frequency was constantly changing, sometimes I could not get the required values. But as seen from the graphs, I got a graph similar to the logn graph it should be.



Challenges Faced :

I had a hard time trying to get the graphs to appear properly. I used Xint for this.