The project includes several key components:

1. **Stock Class**: Represents a single stock with attributes such as symbol, price, volume, and market capitalization. This class includes constructors, getters, and setters for these attributes.
2. **AVLTree Class**: Implements an AVL tree data structure to store and manage **Stock** objects. The tree is balanced based on the stock symbol attribute. This class includes methods for insertion, deletion, searching, and balancing (rotations).
3. **StockDataManager Class**: Manages the AVL tree and provides methods for data management such as adding stocks, removing stocks, searching for stocks, and updating stock information. This class utilizes the **AVLTree** class for data management.
4. **GUIVisualization Class**: Uses Java Swing to create a graphical user interface that displays performance graphs. These graphs show the relationship between the number of nodes and the running time of different operations (ADD, REMOVE, SEARCH, UPDATE).
5. **Main Class**: The entry point of the application. It initializes the **StockDataManager**, loads data from the input file, retrieves performance data, and displays the performance graphs using the **GUIVisualization** class.
6. **GenerateInputFile** Class: Generates random input files with different sizes and operation mixes to test the performance of the stock data management system. This class helps in creating test scenarios for adding, removing, searching, and updating stock data.

The input file contains commands for managing stock data, such as ADD, REMOVE, SEARCH, and UPDATE. The program reads these commands, processes them, and measures the time taken for each operation to analyze and visualize the performance.

My Delete Funtion

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

    private Node delete(Node node, String symbol) {
        if (node == null) {
            return node;
        }

        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 temp = null;
                if (temp == node.left) {
                    temp = node.right;
                } else {
                    temp = node.left;
                }

                // No child case
                if (temp == null) {
                    temp = node;
                    node = null;
                } else {
                    node = temp; // One child case
                }
            } else {
                // node with two children: Get the inorder successor (smallest
in the right subtree)
                Node temp = minValueNode(node.right);

                // Copy the inorder successor's data to this node
                node.stock = temp.stock;
```

```
                // Delete the inorder successor
                node.right = delete(node.right, temp.stock.getSymbol());
            }
        }
```

Instert Function

The **insert** method in the **StockDataManager** class inserts a new stock into the AVL tree or updates an existing stock. It ensures the tree remains balanced by performing necessary rotations (left, right, left-right, right-left) based on the balance factor.

```
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 {
            node.stock = stock; // Update the existing stock
            return node;
        }
        node.height = 1 + Math.max(height(node.left), height(node.right));
        int balance = getBalance(node);
        // Left Left Case
        if (balance > 1 && stock.getSymbol().compareTo(node.left.stock.getSymbol()) < 0) {
            return rightRotate(node);
        }
        // Right Right Case
        if (balance < -1 && stock.getSymbol().compareTo(node.right.stock.getSymbol()) > 0)
{
            return leftRotate(node);
        }
        // Left Right Case
        if (balance > 1 && stock.getSymbol().compareTo(node.left.stock.getSymbol()) > 0) {
            node.left = leftRotate(node.left);
            return rightRotate(node);
        }
        // Right Left Case
        if (balance < -1 && stock.getSymbol().compareTo(node.right.stock.getSymbol()) < 0)
{
            node.right = rightRotate(node.right);
            return leftRotate(node);
        }
        return node;
```

```
    }
```

My Search Function

The search method in the StockDataManager class searches for a stock with a given symbol in the AVL tree. It recursively traverses the tree, comparing the symbol to find and return the corresponding Stock object or null if not found.

```
public Stock search(String symbol) {
        return search(root, symbol);
    }

    private Stock search(Node node, String symbol) {
        if (node == null) {
            return null;
        }

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

The **StockDataManager** class processes stock data operations by reading commands from an input file. For each command (ADD, REMOVE, SEARCH, UPDATE), it performs the corresponding operation on an AVL tree and measures the time taken. The measured times are stored in separate lists for each type of operation. This data is then used to analyze and visualize the performance of the AVL tree-based stock management system.

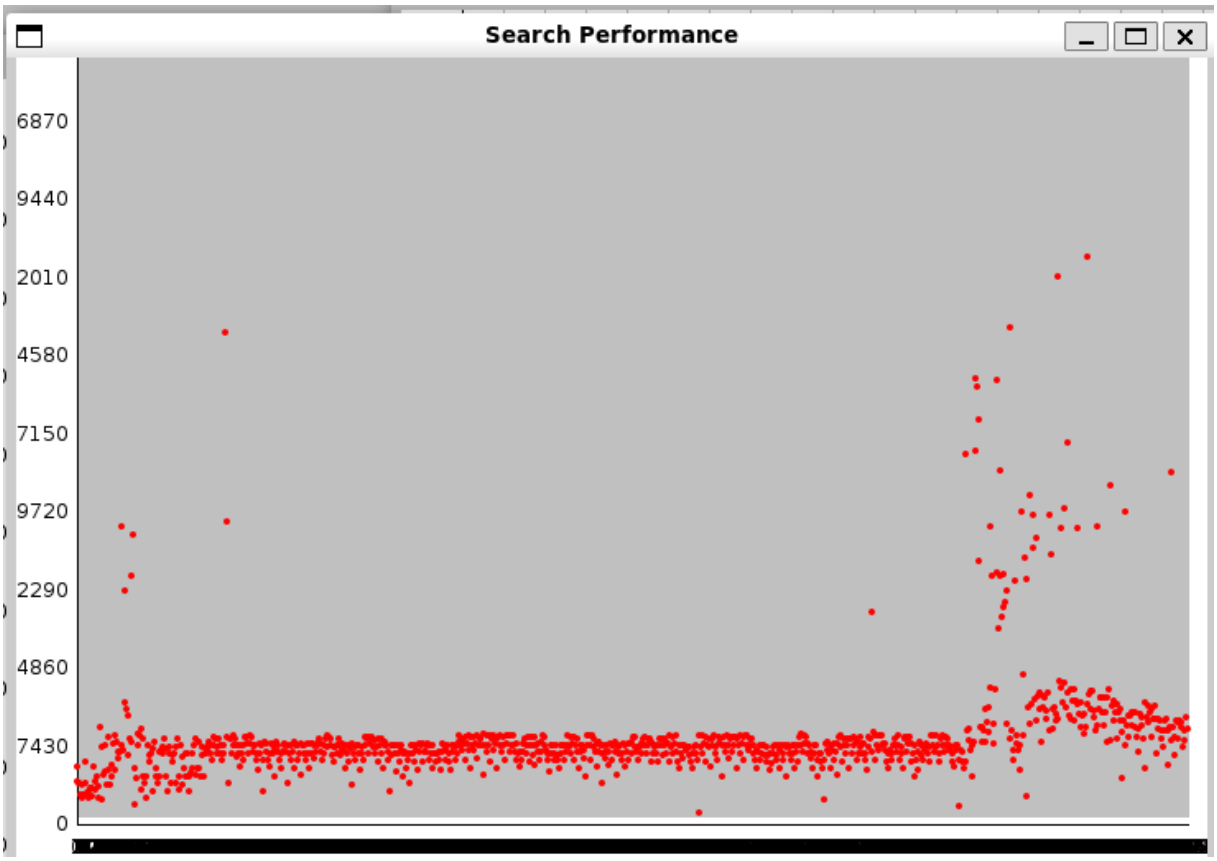
```
public class StockDataManager {
    private List<Integer> addTimes;
    private List<Integer> removeTimes;
    private List<Integer> searchTimes;
    private List<Integer> updateTimes;

    public StockDataManager() {
        addTimes = new ArrayList<>();
        removeTimes = new ArrayList<>();
        searchTimes = new ArrayList<>();
        updateTimes = new ArrayList<>();
    }
```

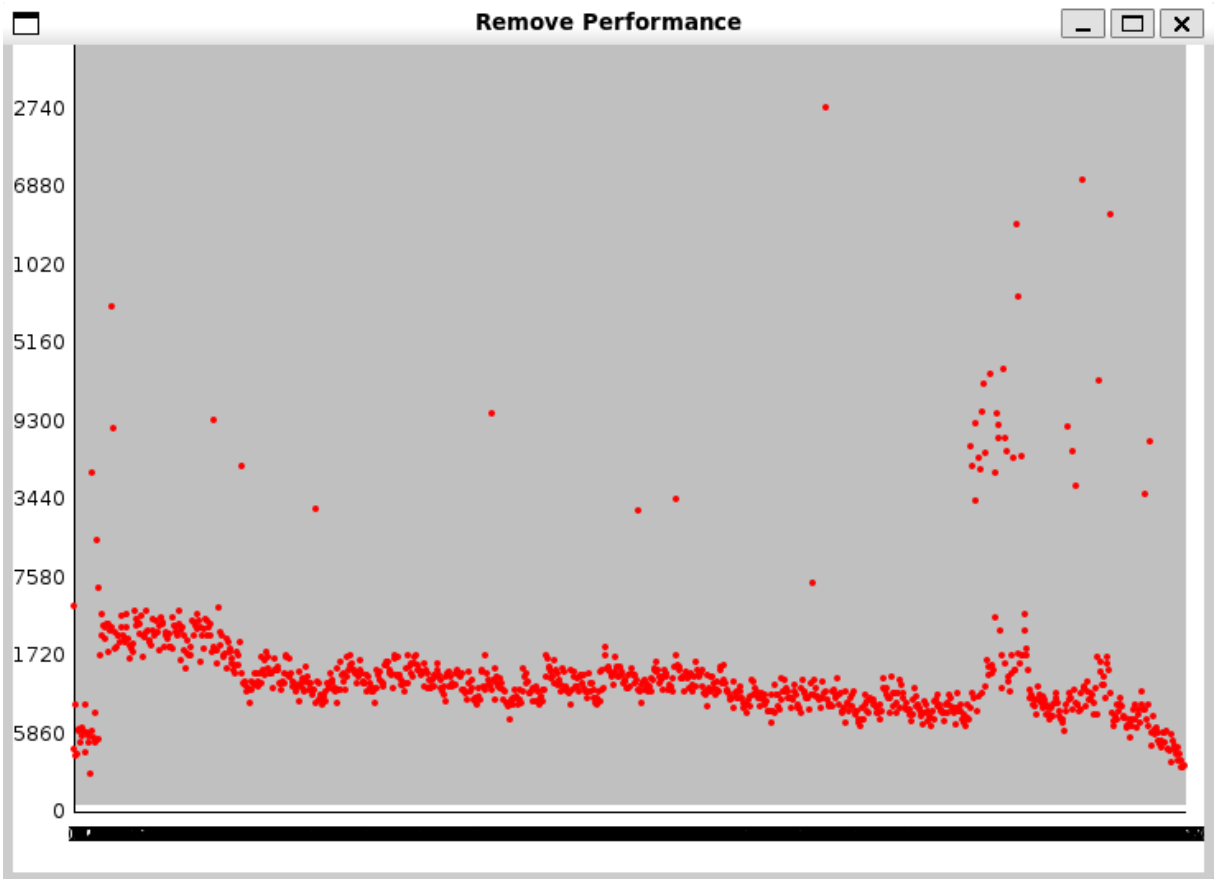
Time Complexity

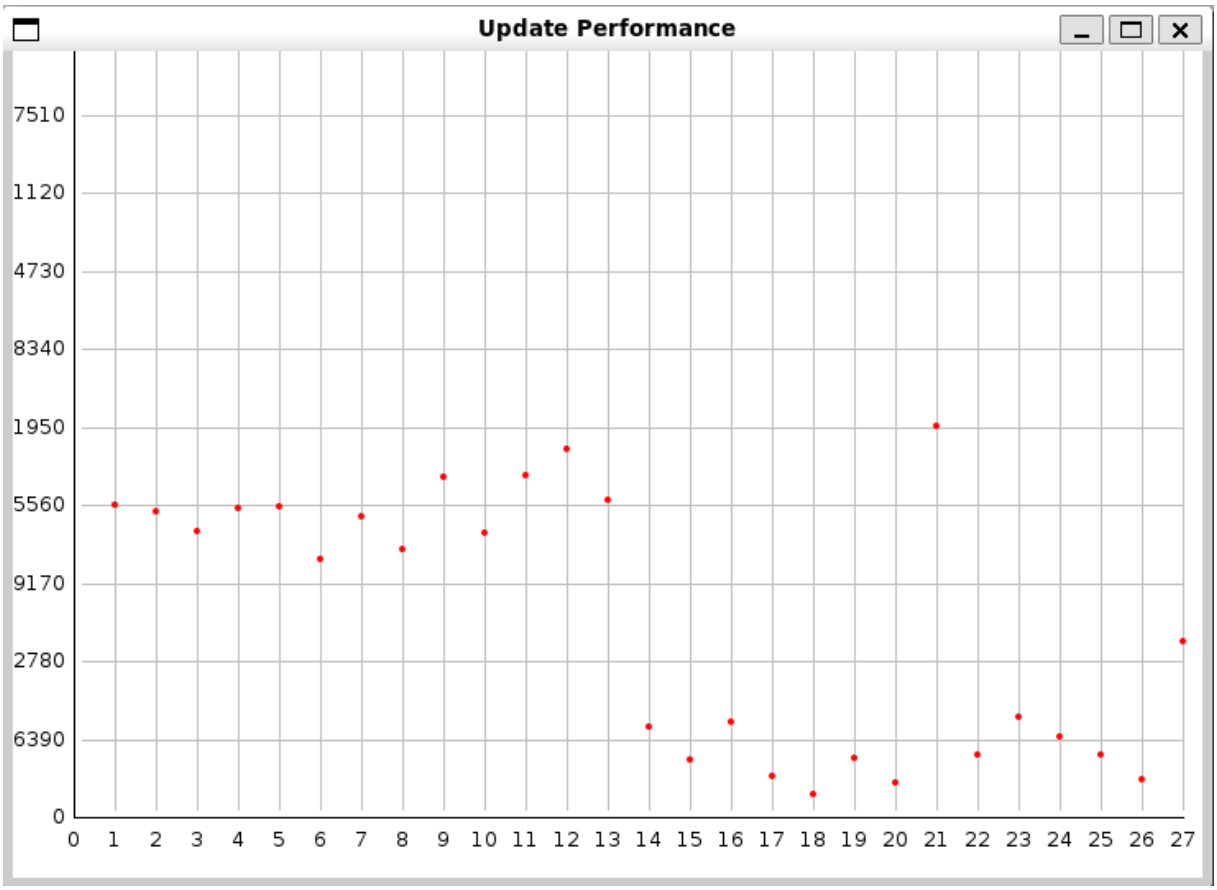
The time complexity for each operation in the AVL tree is as follows:

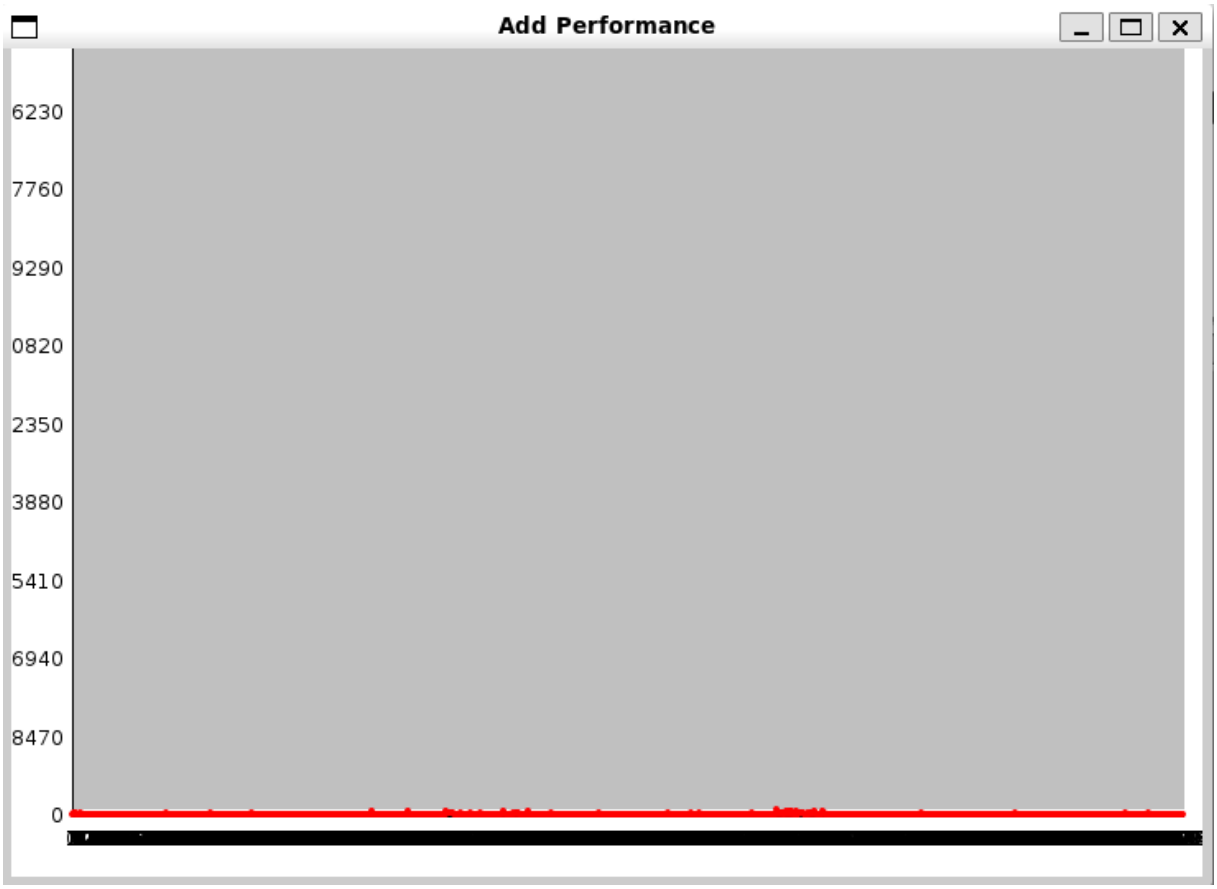
- **Add (Insertion)**: The time complexity for inserting a node into an AVL tree is O(log n) because the tree is balanced and the height of the tree is maintained to be logarithmic relative to the number of nodes.
- **Remove (Deletion)**: The time complexity for deleting a node from an AVL tree is O(log n). Similar to insertion, the tree remains balanced, ensuring that deletions also operate in logarithmic time.
- **Update**: Updating a stock in the AVL tree involves searching for the node, which takes O(log n), and then updating its values, which is O(1). Therefore, the overall time complexity for updating is O(log n).
- **Search**: The time complexity for searching for a node in an AVL tree is O(log n) due to the balanced nature of the tree, which ensures that each search operation is efficient.

```
Stock [symbol=GOOGL, price=657.24, volume=88931, marketCap=432777078]
Stock not found: GOOGL
Stock [symbol=AAPL, price=1683.92, volume=504959, marketCap=2861722962]
Stock not found: TSLA
Stock [symbol=NFLX, price=1853.48, volume=974581, marketCap=4086113060]
Stock not found: V
Stock [symbol=AAPL, price=1108.42, volume=160995, marketCap=1519243538]
Stock [symbol=BABA, price=2090.85, volume=689627, marketCap=2157095141]
Stock not found: AMZN
Stock not found: FB
Stock [symbol=BABA, price=2090.85, volume=689627, marketCap=2157095141]
Average ADD time: 42974 ns
Average SEARCH time: 13434 ns
Average REMOVE time: 14996 ns
```


Search Performance
6870
9440
2010
4580
7150
9720
2290
4860
7430
0


Remove Performance
2740
6880
1020
5160
9300
3440
7580
1720
5860
0


Update Performance
7510
1120
4730
8340
1950
5560
9170
2780
6390
0
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27


Add Performance
6230
7760
9290
0820
2350
3880
5410
6940
8470
0