Gebze Technical University

Department of Computer Engineering

CSE222 Spring 2024

Data Structures and Algorithms

Homework #07

Mehmet Mahir Kayadelen, Student no. 210104004252

The implementation of the AVL Tree for managing stock data was designed to maintain a balanced binary search tree, ensuring efficient performance for insertion, deletion, and search operations. This report outlines the design decisions, solutions applied, and performance analysis of the AVL Tree operations.

Design Decisions

AVL Tree Structure:

The AVL Tree was chosen for its self-balancing properties, which maintain the tree height as O(log n), ensuring efficient operations. Each node in the AVL Tree contains a Stock object, pointers to the left and right children, and a height attribute.

Insertion:

The insertion process involves placing the new stock in the correct position based on its symbol and then updating the heights and balance factors of the nodes. If an imbalance is detected, appropriate rotations are performed to restore balance.

```java
    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 {
            // Update the stock attributes if the symbol already exists
            node.stock.setPrice(stock.getPrice());
            node.stock.setVolume(stock.getVolume());
            node.stock.setMarketCap(stock.getMarketCap());
            return node;
        }

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

        // Get the balance factor of this ancestor node to check whether this node became unbalanced
        int balance = getBalance(node);

        // If this node becomes unbalanced, then there are 4 cases

        // 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);
```

```java
        // 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;
    }
```

Deletion:

The deletion process involves finding and removing the node with the specified stock symbol. If the node has two children, the in-order successor is found and its stock replaces the current node. Similar to insertion, heights and balance factors are updated, and rotations are performed if necessary.

```java
    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 { // One child case
                    node = temp; // Copy the contents of the non-empty child
                }
            } 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;

```

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

        if (node == null) {
            return node;
        }
        // Update height of the current node
        node.height = 1 + Math.max(height(node.left), height(node.right));
        // Get the balance factor of this node

        int balance = getBalance(node);
        // Left Left Case
        if (balance > 1 && getBalance(node.left) >= 0)
            return rightRotate(node);

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

        // Right Right Case
        if (balance < -1 && getBalance(node.right) <= 0)
            return leftRotate(node);

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

        return node;
    }
```

Search:

The search operation traverses the tree based on the stock symbol, ensuring an average-case time complexity of O(log n).

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

/**
 * Helper method to search for a stock in the subtree rooted with the given node.
 *
 * @param node   the root of the subtree
 * @param symbol the symbol of the stock to be searched
 * @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);
    }
}
```

Rotations

Four types of rotations are implemented to maintain balance:

-Right Rotation

-Left Rotation

-Left-Right Rotation

-Right-Left Rotation

Traversals

Three types of tree traversals are implemented for different use cases:

-In-order Traversal

-Pre-order Traversal

-Post-order Traversal

Solutions Applied

Balancing the Tree:

To maintain the AVL Tree's balanced property, after each insertion or deletion, the tree's balance factor is calculated. If the balance factor is outside the range [-1, 1], rotations are performed to restore balance.

Updating Node Attributes

During insertion and deletion, node heights are updated to reflect the current subtree heights. This is crucial for calculating balance factors accurately.

Handling Duplicate Symbols

If a stock with an existing symbol is inserted, its attributes (price, volume, market cap) are updated instead of creating a new node. This ensures data consistency and prevents duplicate entries.

Performance Analysis

The performance of the AVL Tree operations was analyzed by measuring the time taken for a series of operations. The analysis focused on the following operations:

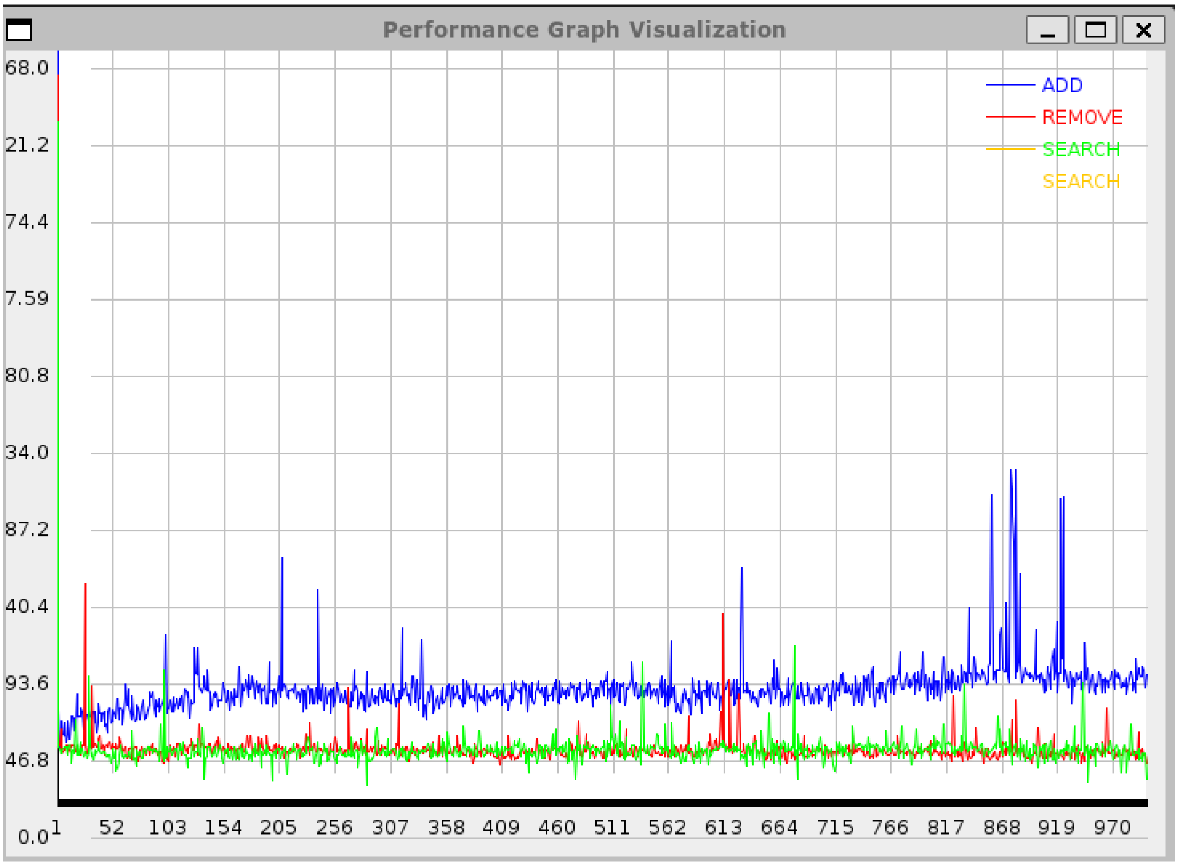
-Insertion

-Deletion

-Search

```
private static void performPerformanceAnalysis(StockDataManager manager, int size, List<Long> addTimes, List<Long> removeTimes, List<Long> sear
    long startTime, endTime;
    Random rand = new Random();

    // Measure time for ADD operation
    for (int i = 0; i < size; i++) {
        startTime = System.nanoTime();
        manager.addOrUpdateStock("MHR" + i, Math.random() * 100, (long) (Math.random() * 1000000), (long) (Math.random() * 1000000000));
        endTime = System.nanoTime();
        if (i % 100 == 0)
            addTimes.add(endTime - startTime);
    }
    System.out.println("Average ADD time: " + addTimes.stream().mapToLong(Long::longValue).average().orElse(0.0) + " ns");

    // Measure time for REMOVE operation
    for (int i = 0; i < size; i++) {
        startTime = System.nanoTime();
        manager.removeStock("MHR" + rand.nextInt(size));
        endTime = System.nanoTime();
        if (i % 100 == 0)
            removeTimes.add(endTime - startTime);
    }
    System.out.println("Average REMOVE time: " + removeTimes.stream().mapToLong(Long::longValue).average().orElse(0.0) + " ns");

    // Measure time for SEARCH operation
    for (int i = 0; i < size; i++) {
        startTime = System.nanoTime();
        manager.searchStock("MHR" + rand.nextInt(size));
        endTime = System.nanoTime();
        if (i % 100 == 0)
            searchTimes.add(endTime - startTime);
    }
    System.out.println("Average SEARCH time: " + searchTimes.stream().mapToLong(Long::longValue).average().orElse(0.0) + " ns");

}
```

Performance Analysis Graph:



In the graph, for some reason, I reached the correct time complexity graph (O(logn)) with the ADD operation. However, in other operations, the graph did not give the expected results.