CSE 222 HOMEWORK 7: Balanced Tree-based Stock Data Management

Gebze Technical University Computer Engineering Department

Submission Deadline: 20th May 2024 23:59

1 Introduction

This assignment requires you to implement a balanced tree data structure, specifically an AVL tree, to manage stock data. The goal is to efficiently store, retrieve, and manipulate stock information while maintaining a balanced structure for optimal performance.

2 Assignment Requirements

2.1 Stock Data Structure

Each node in the AVL tree must represent a stock and contain the following attributes:

- **symbol** (String) - The unique stock symbol (e.g., AAPL for Apple Inc.). This attribute will be used as the key for inserting nodes into the AVL tree and must be unique for each stock.
- **price** (double) - The current stock price.
- **volume** (long) - The trading volume of the stock.
- **marketCap** (long) - The market capitalization of the company.

3 OOP Design Specifications

3.1 Class Hierarchy

You must develop the following classes for the stock data management system:

- **Stock**: Represents a single stock with the attributes mentioned above. Implement appropriate constructors, getters, and setters for this class.
- **AVLTree**: Implements an AVL tree data structure to store and manage Stock objects. The tree should be balanced based on the stock symbol attribute. Implement methods for insertion, deletion, and searching.
- **StockDataManager**: 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 should utilize the AVLTree class for data management.

4 Core Features

You must implement the following functionalities:

- Add and remove stocks from the AVL tree. The tree should remain balanced after each operation. When adding a stock, use the stock symbol as the key for insertion. If a stock with the same symbol already exists in the tree, update its attributes instead of adding a new node.
- Search for a stock by its symbol and retrieve its information. The search operation should have a time complexity of O(log n).
- Perform rotations (left rotation, right rotation, left-right rotation, right-left rotation) to balance the AVL tree after insertions and deletions. You must implement these rotations within the AVLTree class.
- Implement in-order, pre-order, and post-order traversals of the AVL tree. These traversals should be methods within the AVLTree class.

5 Input Format and Testing

The program should read input from a file that contains commands for managing the stock data. The input file can include the following commands:

- **ADD symbol price volume marketCap**: Adds a new stock to the AVL tree or updates an existing stock's attributes if a stock with the same symbol already exists.
- **REMOVE symbol**: Removes the stock with the specified symbol from the AVL tree.
- **SEARCH symbol**: Searches for the stock with the specified symbol in the AVL tree.
- **UPDATE symbol newname newprice newvolume newmarketCap**: Updates the attributes of the stock with the specified symbol in the AVL tree.

You must write a script to randomly generate input files with different sizes and operation mixes. For example, an input file may contain 100 NODE commands, 100 ADD commands, 30 REMOVE commands, 1000 SEARCH commands, and 0 UPDATE commands.

Here's an example of a randomly generated input file:

```
ADD XYZ 25.50 1000000 5000000
ADD ABC 10.25 500000 2500000
ADD DEF 15.75 750000 3750000
SEARCH XYZ
ADD GHI 20.00 1250000 6250000
REMOVE ABC
SEARCH DEF
UPDATE GHI JKL 22.50 1500000 7500000
ADD MNO 12.80 600000 3000000
SEARCH JKL
```

In this example, the stock symbols (XYZ, ABC, DEF, GHI, JKL, MNO) are randomly generated and do not represent real-life stock symbols. The prices, volumes, and market capitalizations are also randomly generated values.

Using the generated input files, you should test each operation (ADD, REMOVE, SEARCH, UPDATE) multiple times with different input sizes and measure the running time. Plot graphs showing the relationship between the problem size (number of nodes in the tree) and the running time for each operation. The graphs should demonstrate the logarithmic time complexity of the operations.

6 Performance Analysis

You must conduct performance tests to analyze the time complexity of each operation (ADD, REMOVE, SEARCH, UPDATE). For each operation, follow these steps:

1. Start with a tree containing a specific number of nodes (e.g., 1000 nodes).
2. Perform the operation a fixed number of times (e.g., 100 ADD operations) and measure the running time for each operation.
3. Calculate the average running time for the operation.
4. Repeat steps 1-3 with increasing tree sizes (e.g., 10000 nodes, 100000 nodes, etc.).
5. Plot a graph showing the relationship between the tree size and the average running time for the operation.

Repeat this process for each operation (ADD, REMOVE, SEARCH, UPDATE) and provide separate graphs for each operation.

7 GUI Visualization

Implement a simple graphical user interface (GUI) using a library such as Java Swing or JavaFX to visualize the performance graphs. The GUI should display the graphs showing the relationship between the problem size and the running time for each operation.

8 Submission Guidelines

Your submission must include the following:

- Complete source code for the Stock, AVLTree, and StockDataManager classes.
- The script used to generate random input files.
- Sample input files used for testing.
- Comprehensive JavaDoc documentation for all classes and methods.
- A PDF report describing your implementation, including any design decisions, challenges faced, and solutions applied. The report should also include the performance analysis graphs for each operation.

- A Makefile to compile and run your program.

Your code should be well-commented, clearly explaining the purpose, inputs, outputs, and side effects of each method.

9 Evaluation Criteria

Your project will be evaluated based on the following criteria:

- Correctness and completeness of the AVL tree implementation, including proper insertion, deletion, and balancing operations.
- Efficiency of search, insertion, and deletion operations, ensuring a time complexity of O(log n) for these operations.
- Proper handling of duplicate stock symbols during insertion.
- Correct implementation of the input file parsing and processing.
- Thoroughness of the performance analysis, including graphs demonstrating the logarithmic time complexity of each operation.
- Adherence to object-oriented programming principles, including encapsulation, inheritance, and polymorphism.
- Code quality, readability, and proper documentation using JavaDoc.
- Completeness and clarity of the PDF report, explaining the implementation details, design choices, and performance analysis.
- Functionality of the GUI for visualizing the performance graphs.

10 Frequently Asked Questions (FAQs)

1. **Q:** How should I handle duplicate stock symbols when inserting nodes into the AVL tree? **A:** If a stock with the same symbol already exists in the tree, update its attributes (name, price, volume, marketCap) instead of adding a new node. The stock symbol serves as the unique identifier for each node in the tree.
2. **Q:** Can I use additional libraries or frameworks for this assignment? **A:** You are allowed to use the standard Java libraries for this assignment. If you wish to use any external libraries or frameworks, please consult with the instructor first and provide proper justification in your PDF report.
3. **Q:** How should I format the input files? **A:** The input files should follow the format specified in the "Input Format and Testing" section. Each command should be on a separate line, and the attributes should be space-separated. Ensure that your program can parse and process the input files correctly.
4. **Q:** What should I include in the PDF report? **A:** Your PDF report should cover the following aspects of your implementation:

- An overview of your AVL tree implementation, including the key methods and their functionality.
- An explanation of how you handled the balancing of the AVL tree after insertions and deletions.
- A description of the input file format and how you processed the commands.
- The performance analysis graphs for each operation (ADD, REMOVE, SEARCH, UPDATE), along with an explanation of the observed time complexity.
- Any challenges you faced during the implementation and how you addressed them.

5. **Q:** How should I handle error scenarios, such as trying to remove a non-existent stock or searching for a stock that doesn't exist in the tree? **A:** Your implementation should gracefully handle error scenarios. For example, if a REMOVE command is issued for a stock that doesn't exist in the tree, you can choose to either ignore the command or display an appropriate error message. Similarly, if a SEARCH command is performed for a non-existent stock, your program should indicate that the stock was not found.