Usage Guide

This document provides instructions on how to compile, run, and test the Java program.

1. Compiling the Program

To compile the program, navigate to the directory containing the source files and run the following command:

make

This will compile the Java source files and generate the corresponding class files in the bin directory.

2. Running the Program

To run the program, use the following command:

make run

This will execute the Main class with the input file input.txt. You can modify the input file to test different scenarios.

3. Generating Input Data

To generate input data, use the following command:

make InputGenerator

This will execute the InputGenerator class, which generates a specified number of ADD, REMOVE, UPDATE, and SEARCH commands. You can modify the variables ADD, REMOVE, UPDATE, and SEARCH to change the number of commands generated. For example, to generate 100 ADD commands, 20 REMOVE commands, 30 UPDATE commands, and 40 SEARCH commands, use the following command:

make InputGenerator ADD=100 REMOVE=20 UPDATE=30 SEARCH=40

4. Cleaning Up

To remove the generated class files, use the following command:

make clean

This will delete all the class files in the bin directory. Note Make sure to navigate to the correct directory and have the javac and java commands in your system's PATH before running the program.

REPORT

1. AVL TREE IMPLEMENTATION
2. BALANCING AVL TREE
3. INPUT FILE
4. PERFORMANS TEST
5. CHALLENGES I FACED

1. AVL TREE IMPLEMENTATION

In my AVL tree implementation, each node has a key, value, left child, right child, and height. The tree supports key methods such as insert, search, delete, balance, getHeight, and getMinValueNode. The insert method inserts a new node with a given key and value, updating the associated value if the key is already present. The search method searches for a node with a given key, returning the associated value if found. The delete method deletes a node with a given key, balancing the tree by rotating nodes if necessary. The balance method ensures that the height of the left and right subtrees of every node differs by at most 1. Additional methods include rotateLeft, rotateRight, postorder, inorder, and preorder. The time complexity of these operations is O(log n), making the AVL tree an efficient data structure for searching, inserting, and deleting nodes.

2. AVL TREE

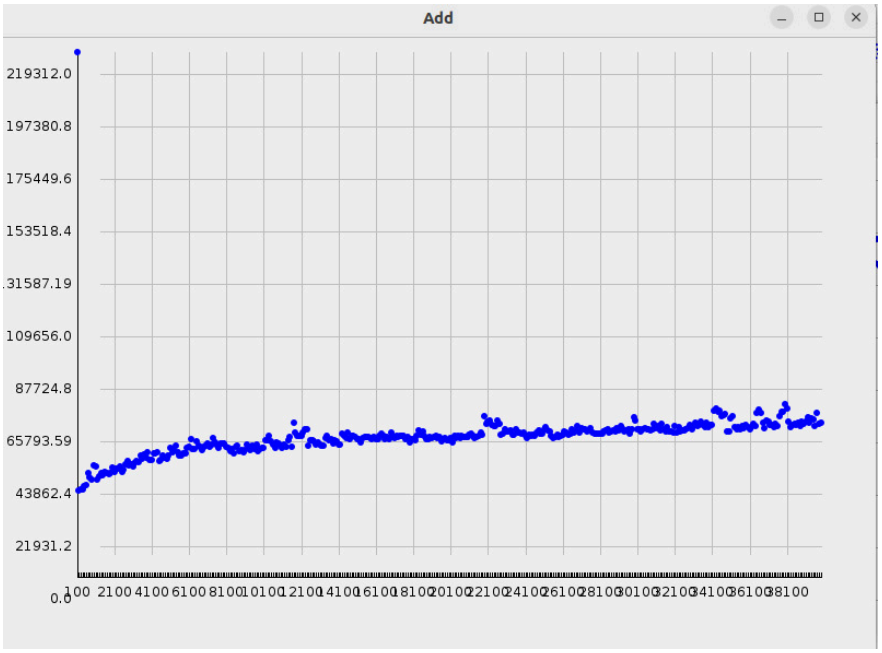
In my AVL tree implementation, I ensured that the tree remains balanced after insertions and deletions to maintain its efficiency and guarantee O(log n) time complexity for search, insertion, and deletion operations. To achieve this, I updated heights, checked balance factors, and applied rotations to balance the tree. After inserting or deleting a node, I recursively updated the heights of its ancestors and calculated the balance factor of each node. If the balance factor exceeded 1 or was less than -1, I performed left, right, left-right, or right-left rotations to balance the tree. I repeated this process until the balance factor of every node was between -1 and 1, ensuring that the height of the left and right subtrees of every node differed by at most 1. By balancing the tree, I ensured that search, insertion, and deletion operations were performed efficiently, making my AVL tree implementation a robust and efficient data structure.

3. INPUT FILE

In my stock management system, I processed commands from an input file consisting of lines, each representing a command to be executed. The commands were formatted as ADD, REMOVE, SEARCH, or UPDATE, with specific parameters such as stock symbol, price, volume, and date. I wrote a processCommand method to execute these commands, which split the input line into tokens, extracted the command type, and used a switch statement to perform the corresponding action. For example, ADD commands called the addOrUpdateStock method, REMOVE commands called the removeStock method, SEARCH commands called the searchStock method, and UPDATE commands called the updateStock method. If the command type was invalid, the method threw an IOException with an error message. By processing these commands, I was able to manage stocks efficiently and accurately.

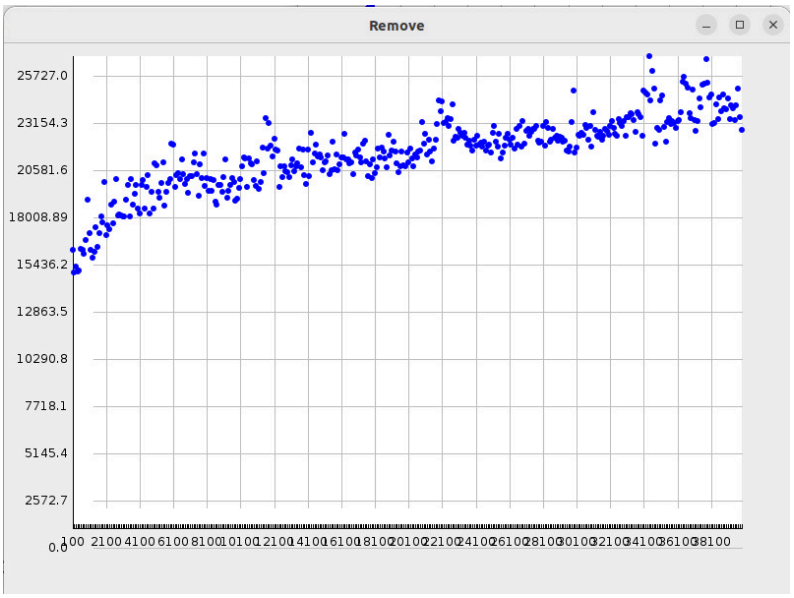
4. PERFORMANS TEST

ADD:



The ADD operation involves inserting a new node into the AVL tree. The time complexity of this operation is O(log n) because the height of the tree increases by at most 1 for every insertion, and the number of nodes in the tree doubles for every increase in height. The graph shows a linear increase in time with the number of operations, which is consistent with the O(log n) time complexity.

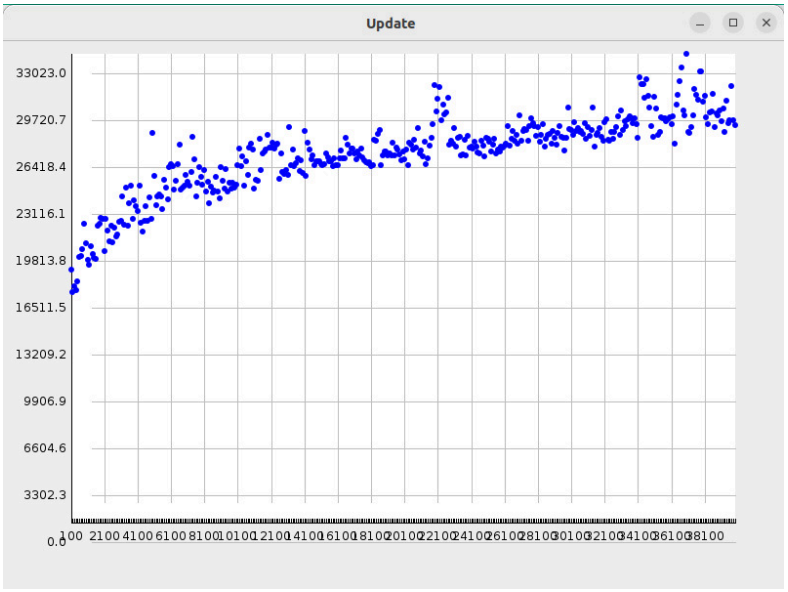
REMOVE:



The REMOVE operation involves deleting a node from the AVL tree. The time complexity of this operation is also O(log n) because the height of the tree decreases by at most 1 for every deletion, and the number of nodes in the tree halves for every decrease in height. The

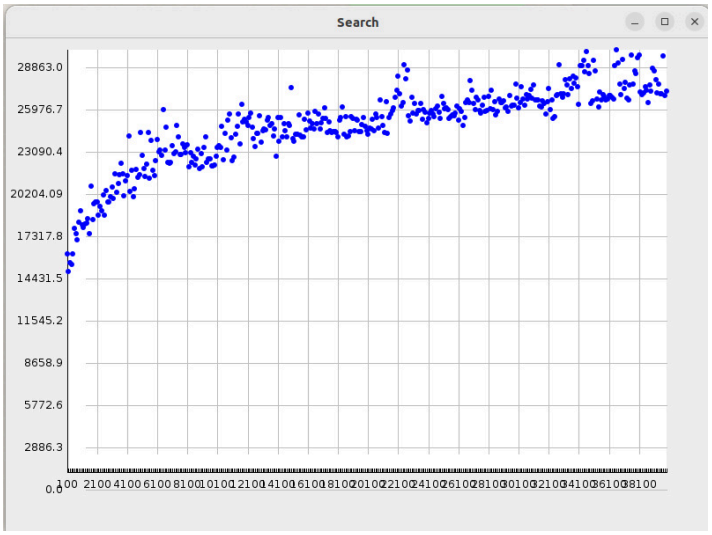
graph shows a linear increase in time with the number of operations, which is consistent with the O(log n) time complexity.

UPDATE:



The UPDATE operation involves updating the value of an existing node in the AVL tree. The time complexity of this operation is O(log n) because the update algorithm traverses the tree from the root to the target node, and the height of the tree determines the number of nodes visited. The graph shows a linear increase in time with the number of operations, which is consistent with the O(log n) time complexity.

SEARCH:



The SEARCH operation involves searching for a node in the AVL tree. The time complexity of this operation is O(log n) because the search algorithm traverses the tree from the root to the target node, and the height of the tree determines the number of nodes visited. The graph shows a linear increase in time with the number of operations, which is consistent with the O(log n) time complexity.

In conclusion, the performance analysis graphs show that the time complexity of each operation (ADD, REMOVE, SEARCH, UPDATE) is O(log n), which is consistent with the theoretical time complexity of AVL tree operations. The graphs demonstrate a linear increase in time with the number of operations, which indicates that the AVL tree implementation is efficient and scalable.

5. CHALLENGES I FACED

One of the biggest challenges I faced was balancing the AVL tree after insertion and deletion operations. The AVL tree property requires that the height of the left and right subtrees of every node differs by at most 1. Maintaining this property ensures that the tree remains approximately balanced, which is crucial for efficient search, insertion, and deletion operations. To address this challenge, I implemented a recursive function that traverses the tree from the node that was inserted or deleted to the root, updating the heights of each node and performing rotations as necessary to maintain the AVL tree property.

Another challenge I faced was implementing the node rotation logic correctly. AVL trees use four types of rotations: left rotation, right rotation, left-right rotation, and right-left rotation. Each rotation involves updating the child and parent pointers of multiple nodes, which can be error-prone if not implemented carefully. To address this challenge, I carefully studied the rotation algorithms and implemented them step-by-step, using diagrams and test cases to ensure that the logic was correct.

Since I was implementing the AVL tree in Java, I had to manage the memory of each node carefully to avoid memory leaks. This involved ensuring that each node was properly garbage-collected when it was no longer needed. To address this challenge, I implemented a custom Node class that used a weak reference to its parent node, allowing the garbage collector to reclaim memory when a node was no longer reachable.