Gebze Technical University Computer Engineering Department

CSE 222 – Data Structures And Algorithms

Homework-7 Report

Muhammet Akkurt

1901042644

Purpose and Details of the Assignment:

The inventory data management system implemented in this project utilizes the AVL tree, a self-balancing binary search tree, to efficiently manage and manipulate inventory information. The primary goal was to develop a robust system capable of performing various operations such as adding, removing, searching and updating stock while maintaining optimal performance. To achieve this, the AVL tree ensures that the height difference between the left and right subtrees of any node does not exceed one, thus guaranteeing O(log n) time complexity for these operations. The project includes detailed command processing to read and execute stock operations from an input file, a comprehensive performance analysis to measure the efficiency of each operation, and graphical visualizations to show the results.

Key Methods and Their Functionality:

- **insert(Stock stock):** Inserts a new stock into the AVL tree. If a stock with the same symbol already exists, it updates the existing stock's details. Balances the tree after insertion by calling the rebalance method.
- **delete(String symbol):** Removes a stock from the AVL tree by its symbol. Balances the tree after deletion by calling the rebalance method.
- **search(String symbol):** Searches for a stock by its symbol and returns the stock if found, otherwise returns null.

- **rebalance(Node node):** Checks the balance factor of the node and performs the necessary rotations to maintain the AVL tree properties.
- **rotateLeft(Node node):** Performs a left rotation on the given node to correct right-heavy imbalances.
- **rotateRight (Node node):** Performs a right rotation on the given node to correct left-heavy imbalances.
- **getBalance(Node node):** Calculates and returns the balance factor of the node.
- **height(Node node):** Returns the height of the given node, with null nodes having a height of 0.

Balancing the AVL Tree:

The AVL tree maintains its balance through rotations after every insertion and deletion operation. The balance factor of each node is calculated to determine whether the node is balanced or needs rebalancing.

The balance factor of a node is the height difference between its left and right subtrees.

Balance factor = height(left subtree) - height(right subtree).

Left Rotation -> Applied when a right-heavy imbalance (balance factor < -1) occurs, specifically when the right child is right-heavy or balanced.

Right Rotation -> Applied when a left-heavy imbalance (balance factor > 1) occurs, specifically when the left child is left-heavy or balanced.

Left-Right Rotation -> Applied when a left-heavy imbalance occurs but the left child is right-heavy.

Right-Left Rotation ->Applied when a right-heavy imbalance occurs but the right child is left-heavy.

After performing rotations, the heights of the nodes are updated to reflect the changes.

Command Processing And StockDataManager:

At runtime, the program expects a single argument specifying the name of an input file. Commands are read from the file using a “BufferedReader”. Each line is split into components, and depending on the command type, different actions are taken. Detailed error handling is implemented to manage incorrect formats or missing parameters.

- **addOrUpdateStock:** This method checks if the stock with the given symbol already exists in the AVL tree using the search method. If the stock exists, its price, volume, and market cap are updated directly, leveraging the mutability of the Stock class objects stored in the tree. If the stock does not exist, a new Stock object is created and inserted into the AVL tree using the insert method.
- **removeStock:** Removes a stock entry from the AVL tree based on the symbol. This method calls the delete method of the AVL tree, which handles the removal. If the stock is found, it is removed, and the tree is rebalanced if necessary to maintain the AVL properties, ensuring that the operation times remain logarithmic relative to the number of elements.

- **searchStock:** Searches the AVL tree for a stock with the specified symbol using the AVL tree’s search method. Returns the Stock object if found; otherwise, it returns null.
- **updateStock:** First, it searches for the stock using the given old symbol.If the stock is found, it updates the stock's details with new values provided. Notably, if the stock symbol changes (i.e., newSymbol is different from symbol), the stock is removed from the tree and reinserted with the new symbol to ensure the correct tree indexing.

Issue And Solutions:

It was necessary to ensure that the AVL tree remained balanced after each deletion and insertion. Maintaining equilibrium after each insertion and deletion was necessary to ensure that the tree remained balanced and thus guarantee O(logn) time complexity for search, insertion and deletion operations.

The rebalance method is invoked whenever an insertion or deletion operation might disrupt the balance of the AVL tree. This method ensures that the height difference between the left and right subtrees of any node (known as the balance factor) is within the acceptable range of -1, 0, or 1.

The rebalance method calculates the balance factor using the getBalance method and determines if rotations are needed to restore balance. Four types of rotations can be performed:

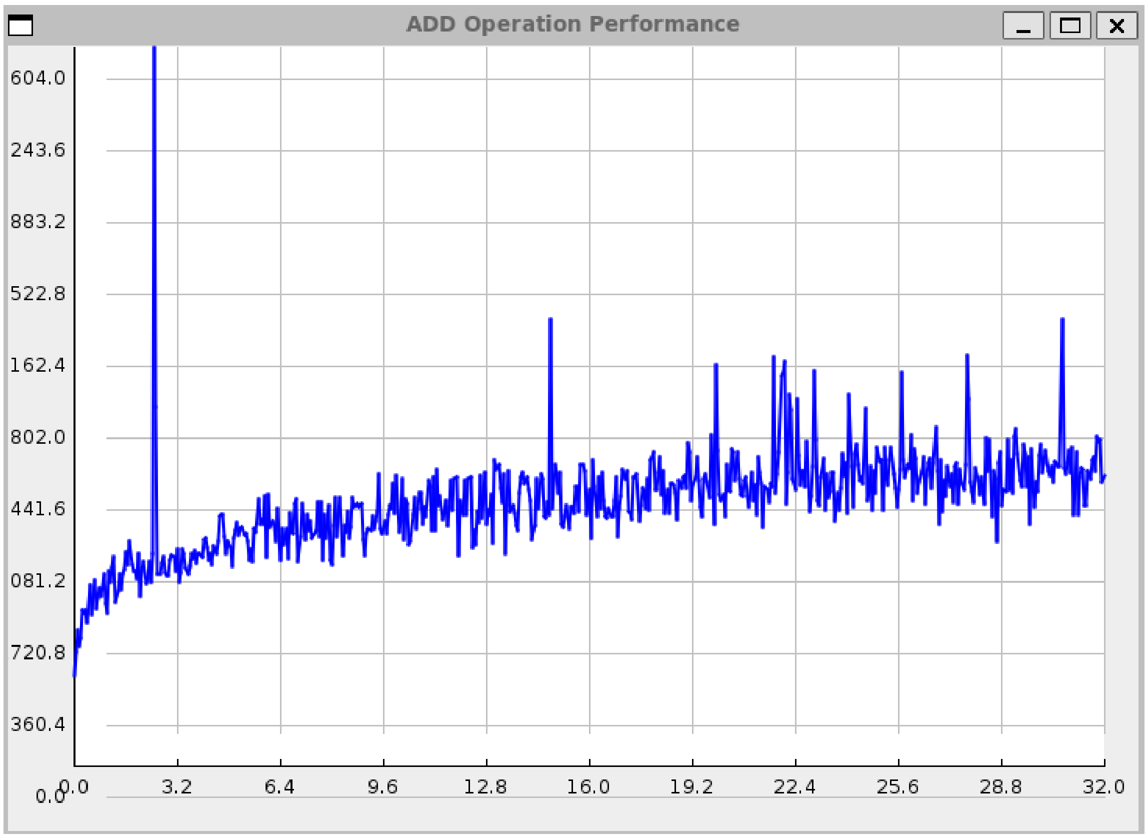
Left Left (LL) Case: A right rotation is performed when the node is unbalanced due to a left child and its left subtree being heavier.
Left Right (LR) Case: A left rotation on the left child followed by a right rotation on the unbalanced node is performed when the node is unbalanced due to a left child and its right subtree being heavier.
Right Right (RR) Case: A left rotation is performed when the node is unbalanced due to a right child and its right subtree being heavier.
Right Left (RL) Case: A right rotation on the right child followed by a left rotation on the unbalanced node is performed when the node is unbalanced due to a right child and its left subtree being heavier.

Rotations adjust the tree's structure while preserving in-order traversal properties. In a left rotation, the right child becomes the new root, and the original node becomes the left child. In a right rotation, the left child becomes the new root, and the original node becomes the right child. Heights of the involved nodes are updated accordingly.

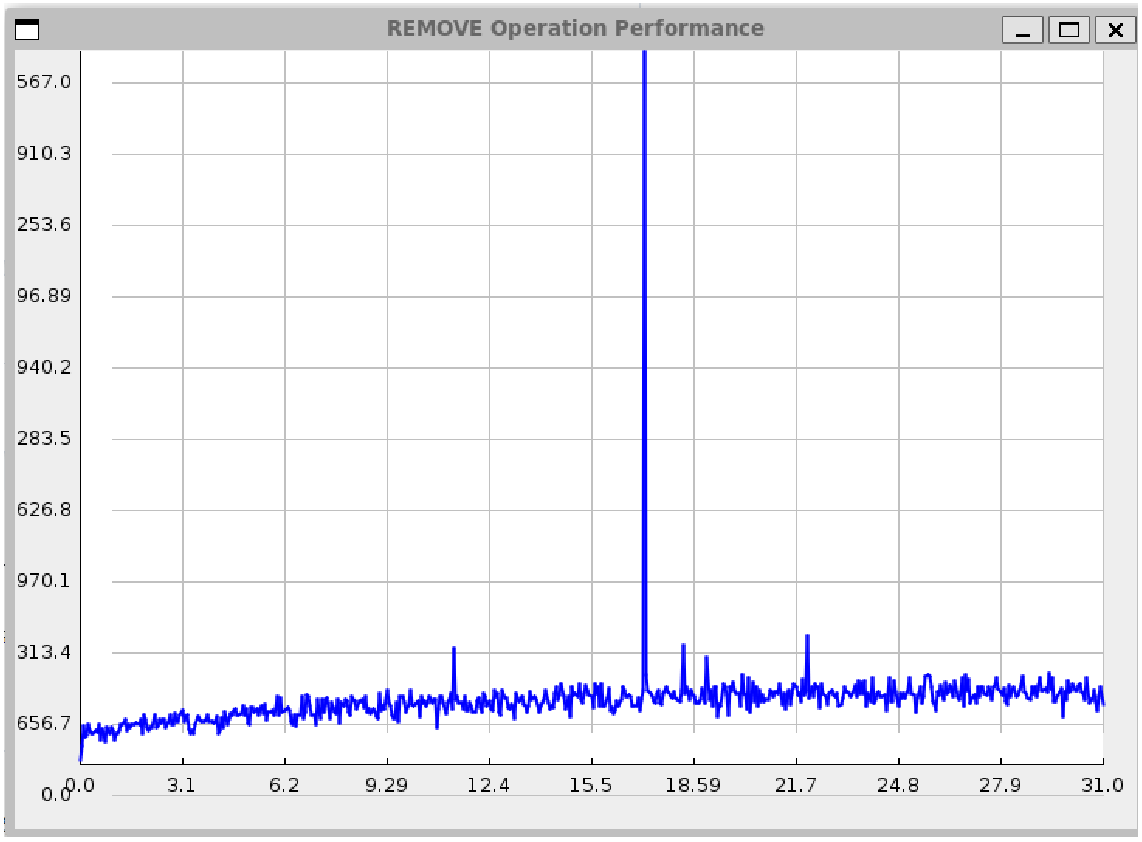
The height of a node is determined using the height method, which returns 0 for null nodes. The getBalance method calculates the balance factor by determining the height difference between the left and right subtrees.

Performance Analysis:

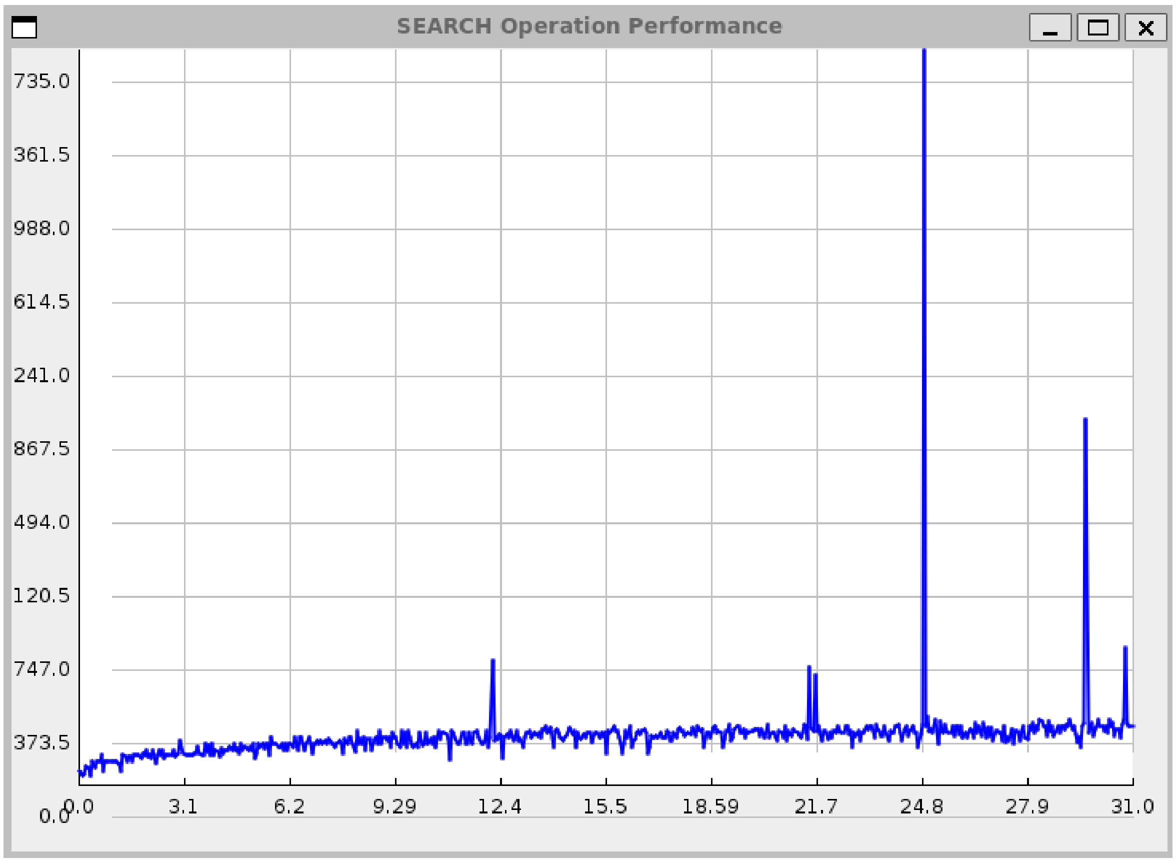
Result graphs with randomly generated 2000 commands:



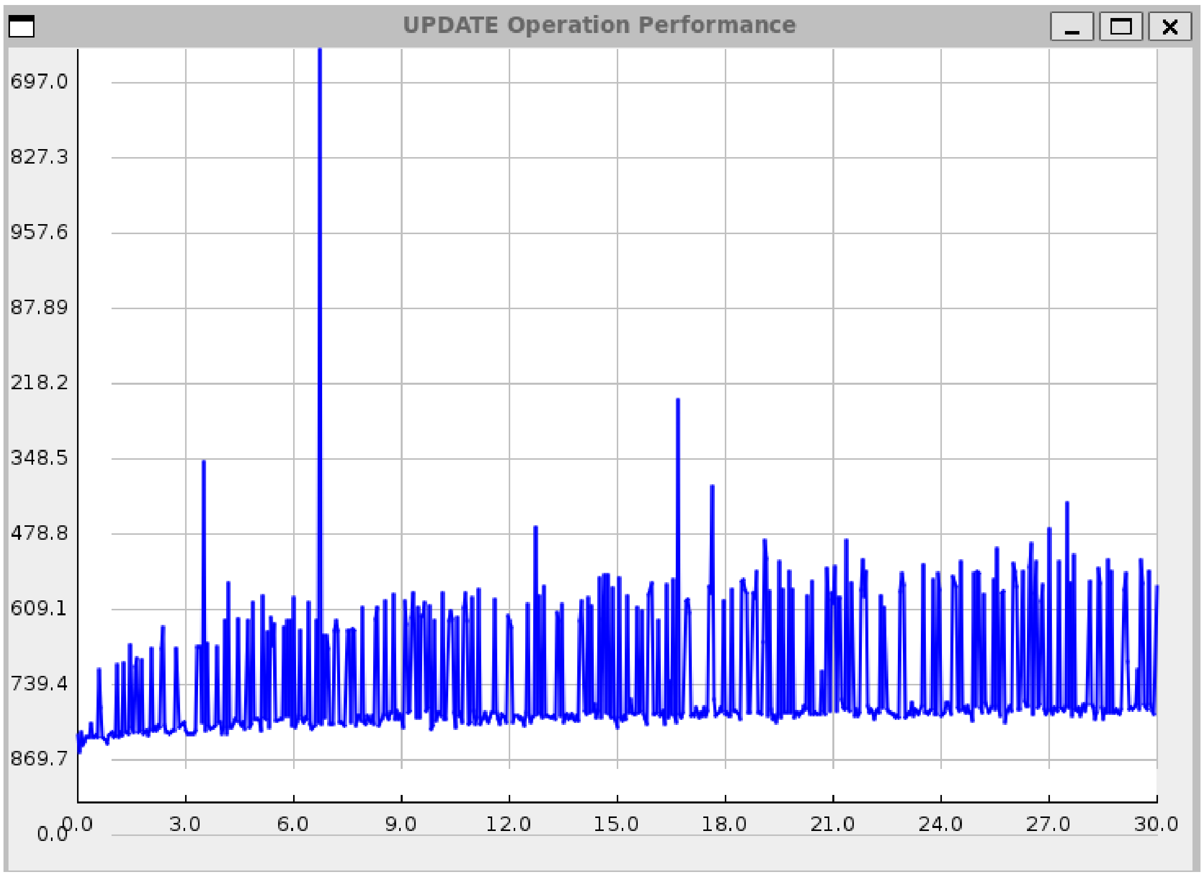
The graph shows the time taken for ADD operations as the number of stocks increases, demonstrating O(log n) complexity.



The graph shows the time taken for REMOVE operations, reflecting the O(log n) efficiency of deletions in an AVL tree.

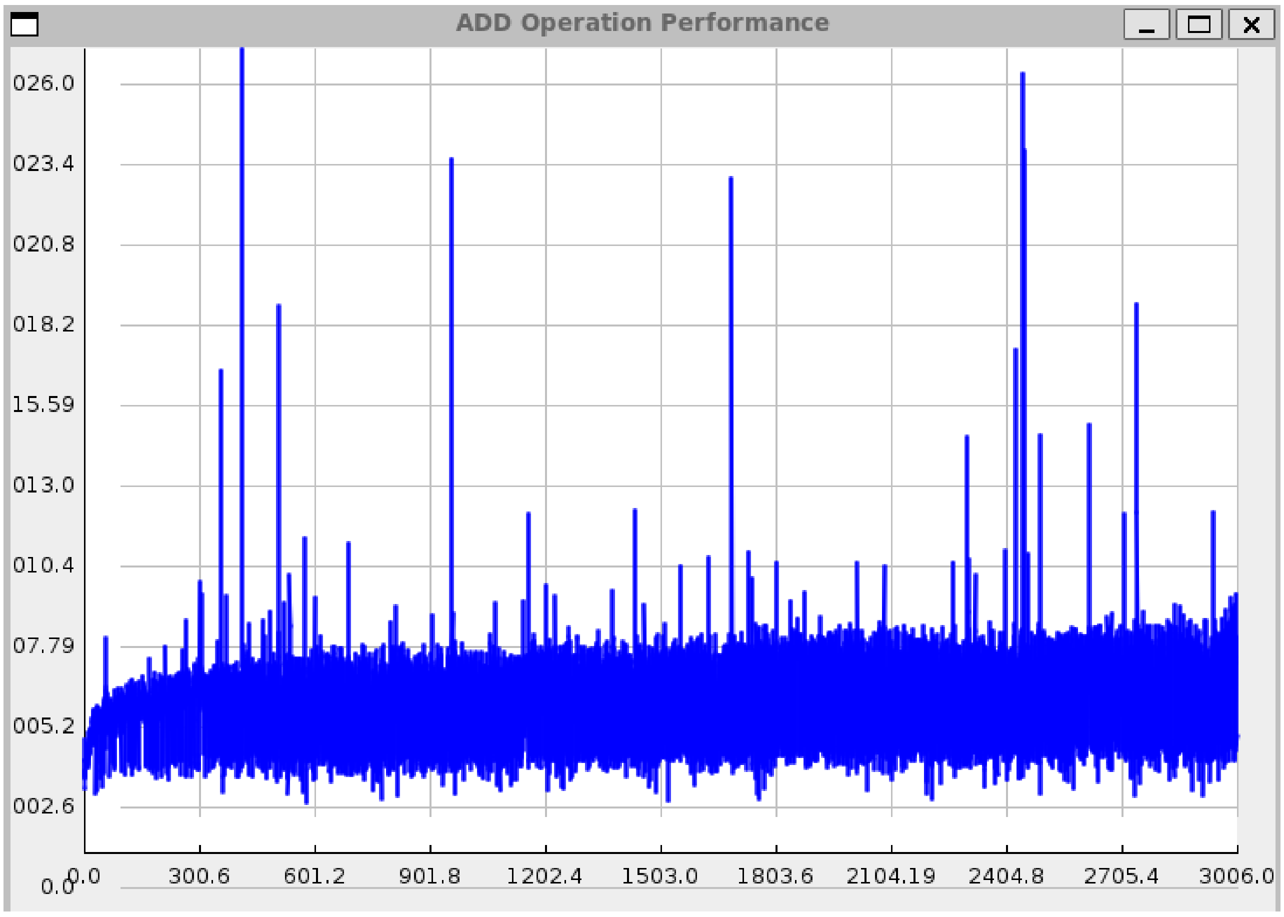


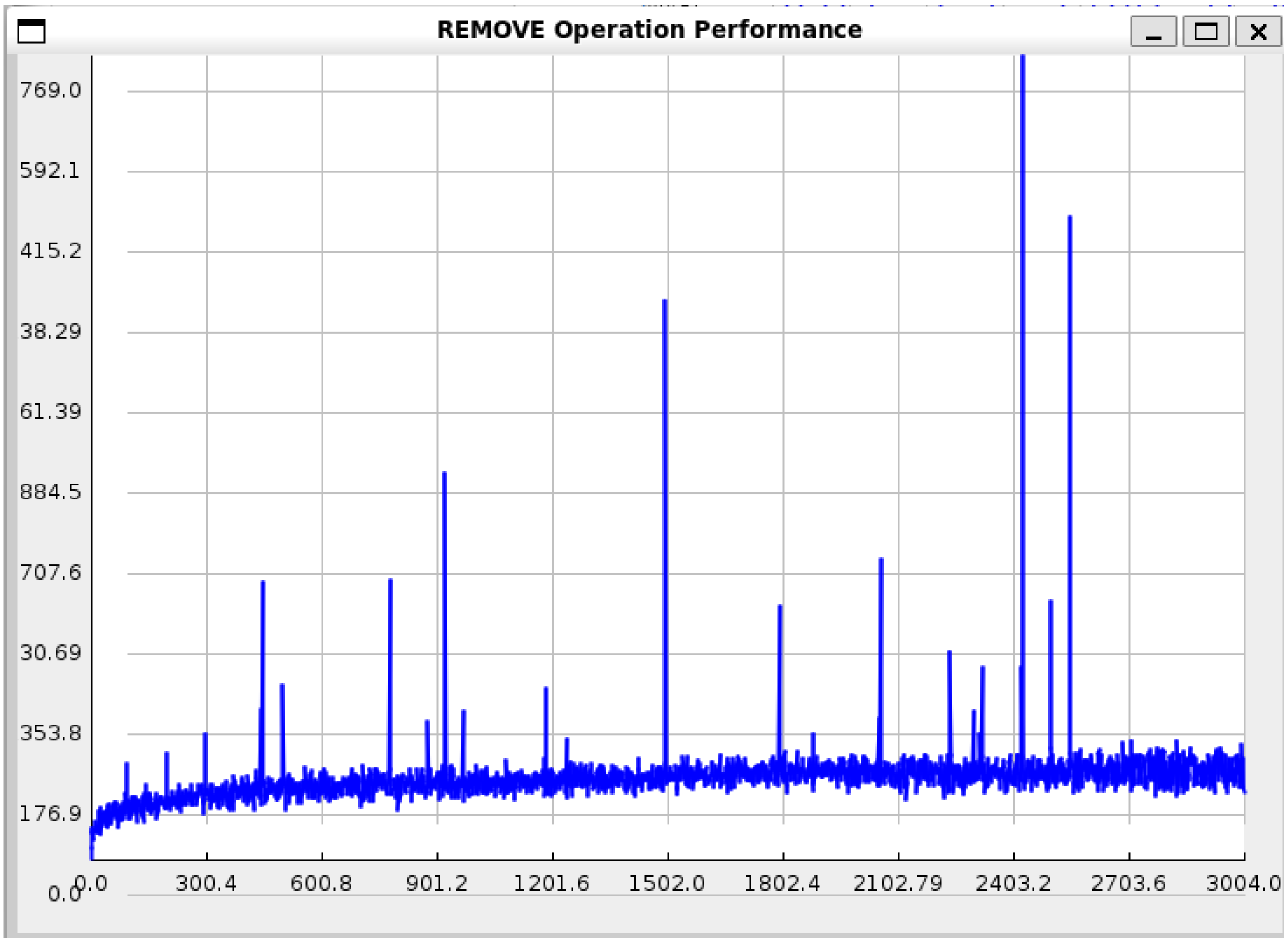
The graph illustrates the time complexity of SEARCH operations, maintaining O(log n) e iciency.

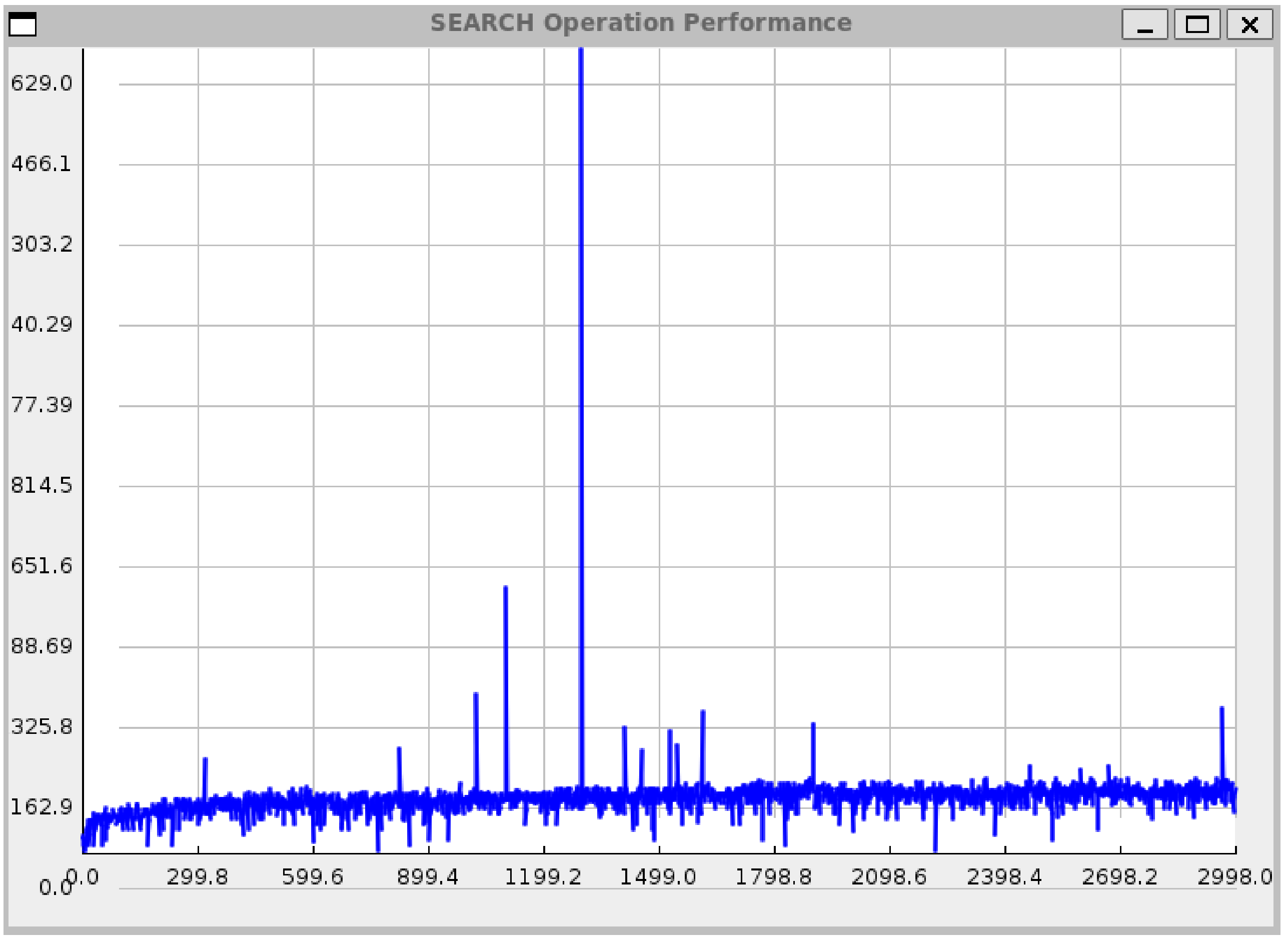


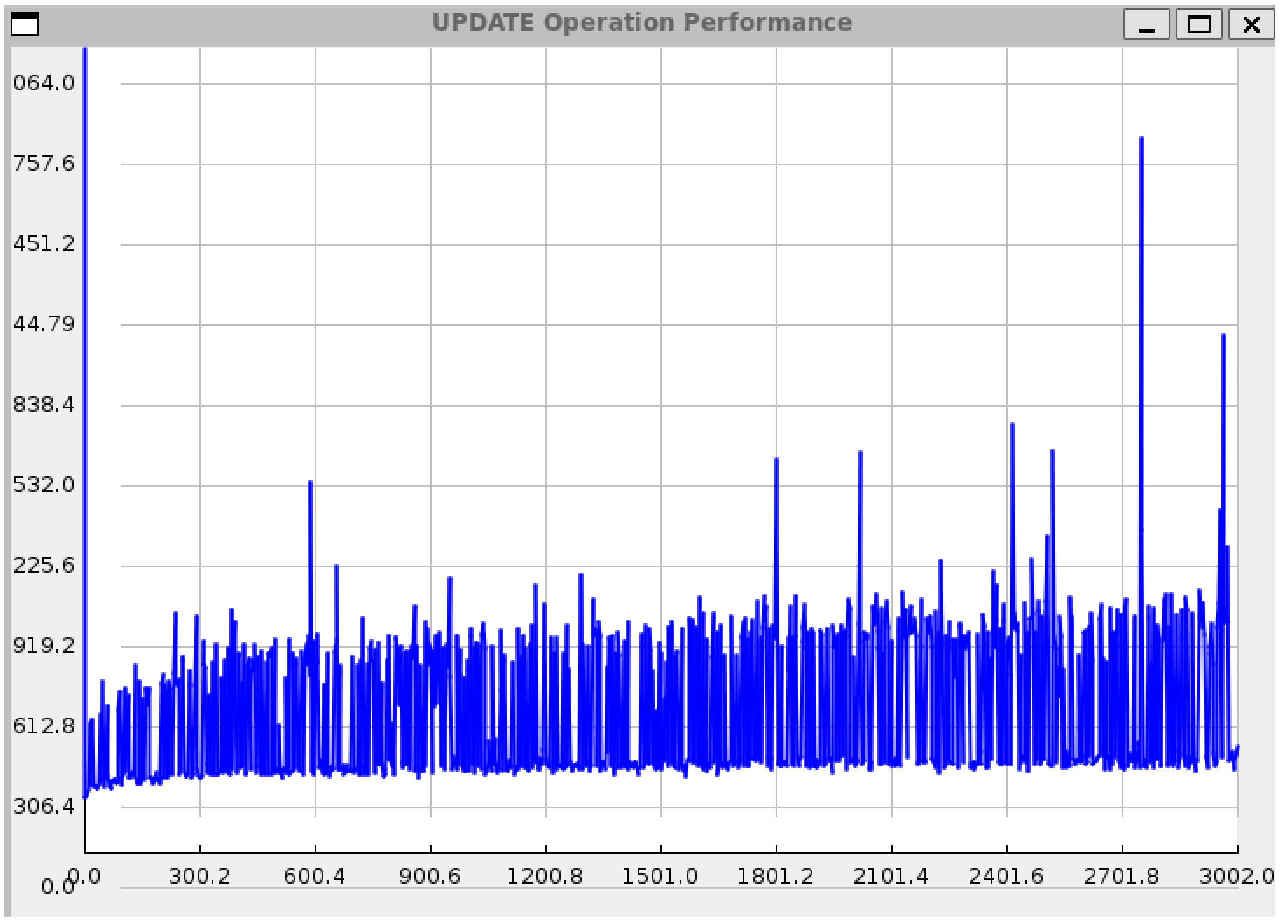
The graph shows the time taken for UPDATE operations, reflecting the O(log n) efficiency of updates in an AVL tree.

Performance analysis of a 10000 instruction file with more ADD commands and a similar proportion of other commands:

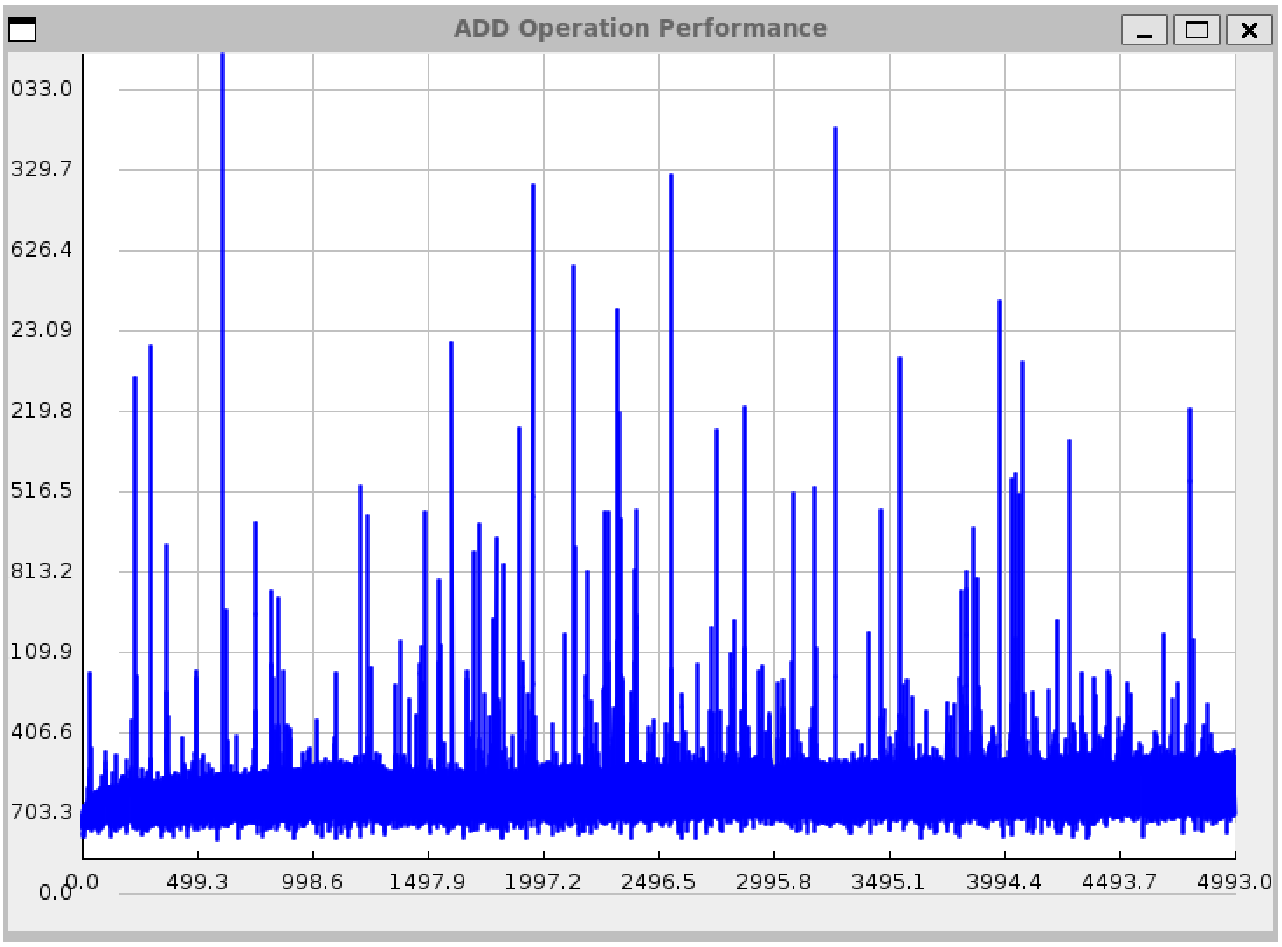


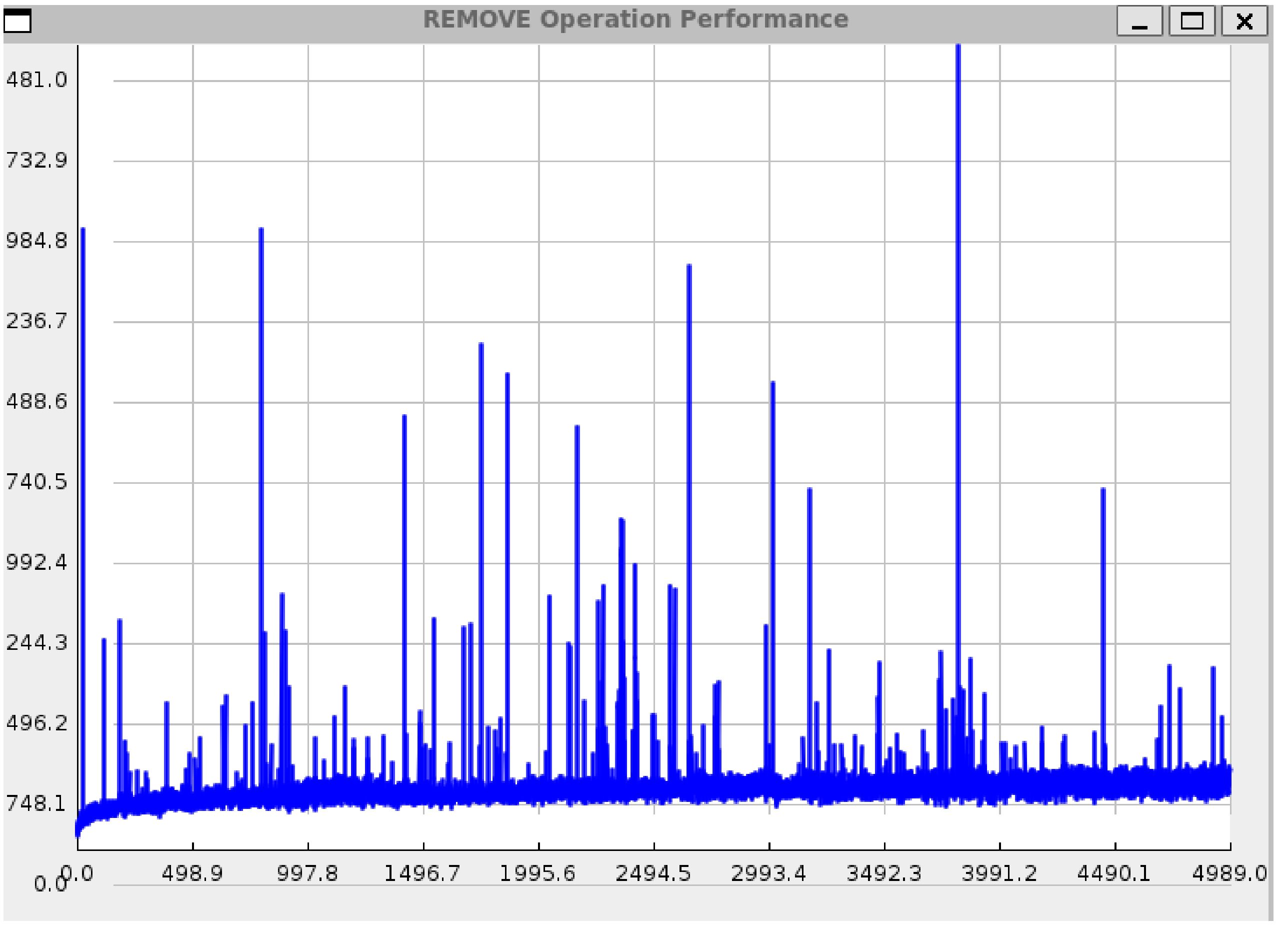


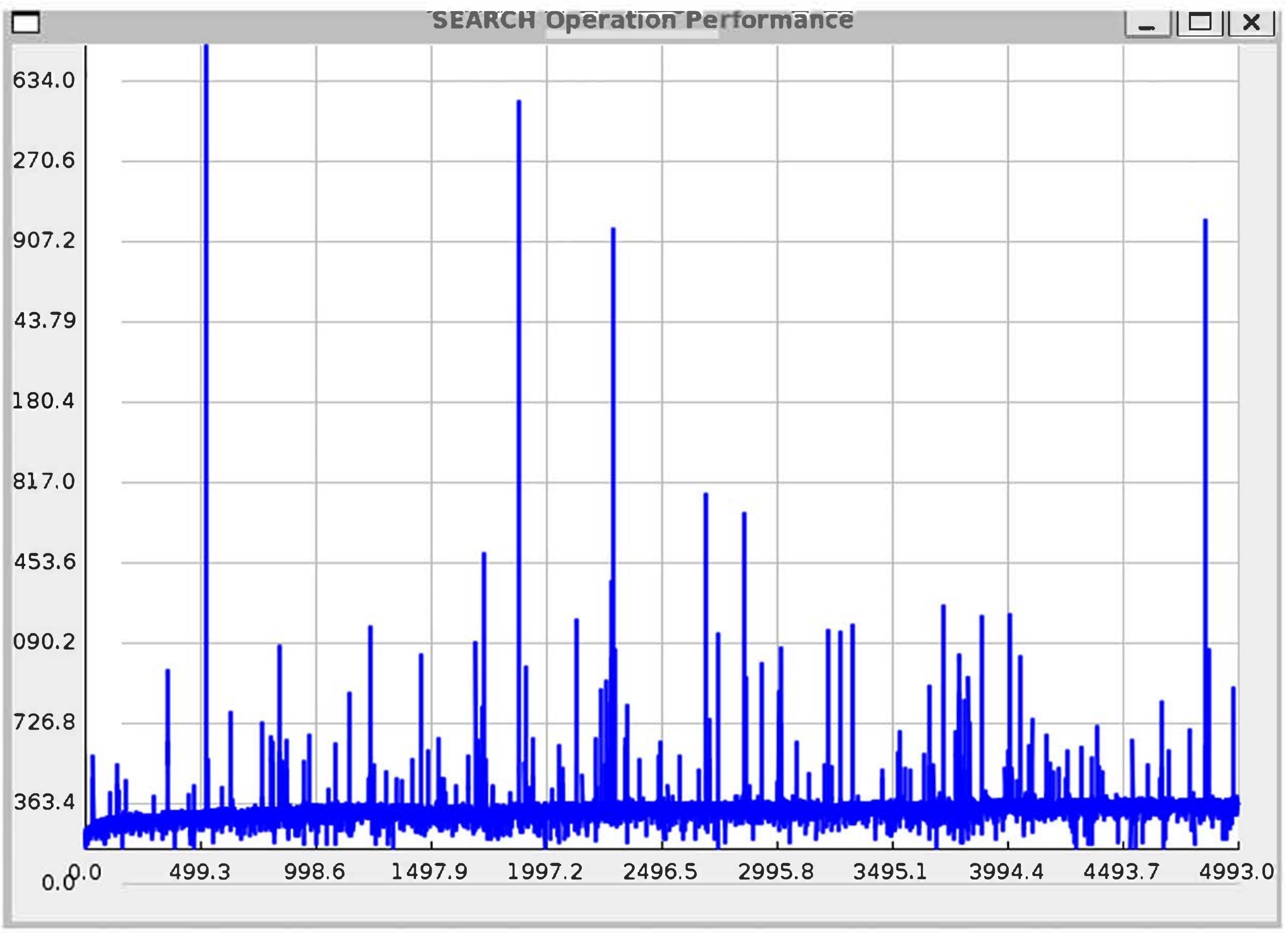

SEARCH Operation Performance
629.0
466.1
303.2
40.29
77.39
814.5
651.6
88.69
325.8
162.9
0.0
0.0
299.8
599.6
899.4
1199.2
1499.0
1798.8
2098.6
2398.4
2698.2
2998.0

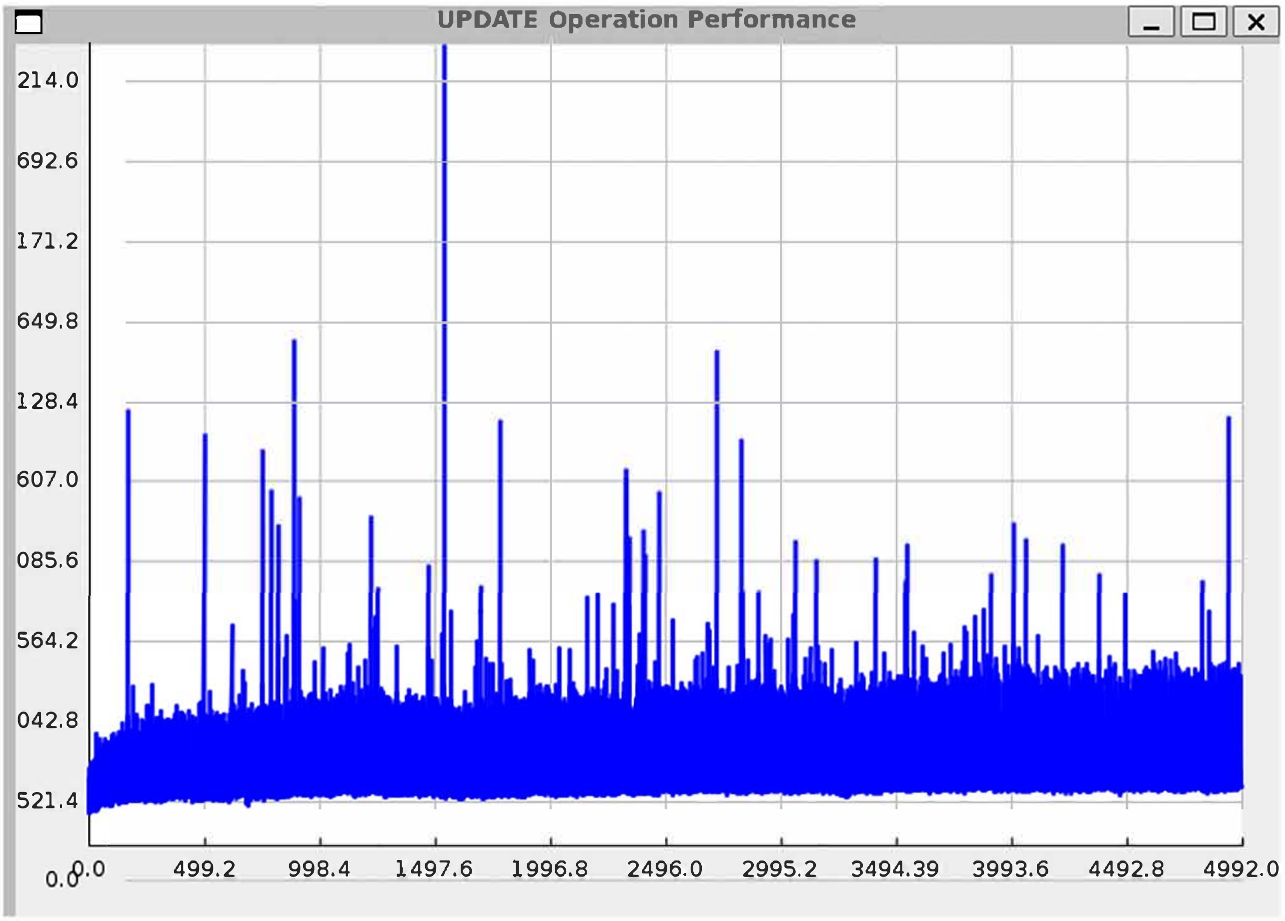

UPDATE Operation Performance
064.0
757.6
451.2
44.79
838.4
532.0
225.6
919.2
612.8
306.4
0.0
0.0
300.2
600.4
900.6
1200.8
1501.0
1801.2
2101.4
2401.6
2701.8
3002.0

Performance analysis of a 50000 instruction file with more SEARCH commands and a similar proportion of other commands:






SEARCH Operation Performance
634.0
270.6
907.2
43.79
180.4
817.0
453.6
090.2
726.8
363.4
0.0
0.0
499.3
998.6
1497.9
1997.2
2496.5
2995.8
3495.1
3994.4
4493.7
4993.0


UPDATE Operation Performance
214.0
692.6
171.2
649.8
128.4
607.0
085.6
564.2
042.8
521.4
0.0
0.0
499.2
998.4
1497.6
1996.8
2496.0
2995.2
3494.39
3993.6
4492.8
4992.0

ADD Operation Performance:

Gradual increase in operation time as the tree grows, reflecting the logarithmic complexity.
Occasional higher spikes due to rebalancing operations when the tree becomes unbalanced.

REMOVE Operation Performance:

Gradual increase in operation time as the tree grows, reflecting the logarithmic complexity.
Occasional spikes likely due to the removal of nodes that require rebalancing.

SEARCH Operation Performance:

Gradual increase in operation time as the tree grows, reflecting the logarithmic complexity.

UPDATE Operation Performance:

The graph exhibits more fluctuations compared to the other operations. The updates might involve both search and modify operations, leading to variations in time taken based on the tree structure at the time of the update.

Finally,the logarithmic time complexity of AVL trees is reflected in their runtimes, with small fluctuations due to rebalancing. Occasional spikes in the graphs indicate the cost of maintaining the stable state of the tree.