# **PROJECT REPORT**

**Red-Black Tree vs Regular Binary Tree in the context of reading and**

**writing data in a CRUD application**

Xuedinan Gao, Bryce Dunlap, B.P. Fletcher, Anjana Deivasigamani

## **INTRODUCTION**

In dynamic environments where data is constantly changing and must be managed effectively, choosing the right data structure is essential for maintaining optimal performance. Efficient data storage and retrieval are critical in applications where data is frequently accessed and modified. In this project, we explore the application of a Red-Black Tree, a self balancing binary search tree, to store and manage a collection of cooking recipes. The primary objective of this project is to investigate the efficiency of the Red-Black Tree in maintaining sorted order and enabling fast insertion, deletion, and search operations within the context of a recipe database.

The central research question guiding this project is: How does the Red-Black Tree perform in organizing and retrieving cooking recipes compared to other data structures, in particular the regular binary tree, and what are its advantages in this specific application? The rationale for choosing the Red-Black Tree lies in its balanced nature, which guarantees that the height of the tree remains logarithmic with respect to the number of nodes. This balance ensures its effectiveness in managing ordered data efficiently, making it a suitable choice for any CRUD (create, read, update, delete) application where quick access to data is essential. By utilizing a Red-Black Tree for storing recipes, we aim to demonstrate how this data structure can address real-world problems where data management and performance are critical. Through this project, we will analyze the Red-Black Tree’s performance in terms of time complexity and compare it with other data structures to highlight its practical benefits and limitations.

## **ANALYSIS**

### Project Functionality

Core Operations of the Red-Black Tree

The Red-Black Tree implemented in this project supports the following core operations:

* Insertion (put(int key, Object value)): Adds a new node with a specified key and value to the tree, ensuring that the tree remains balanced according to the Red-Black Tree properties.
* Search (find(int key)): Retrieves the node corresponding to the specified key, allowing for quick data retrieval.
* Deletion (remove(int key)): Removes the node with the specified key from the tree, rebalancing the tree as necessary to maintain the Red-Black properties.

Each of these operations is designed to be executed in O(log n) time, where n is the number of nodes in the tree. The implementation details of these operations are discussed in the subsequent sections.

Data Structure Overview

The Red-Black Tree is a type of self-balancing binary search tree where each node contains an additional bit for storing color (either red or black). The tree enforces specific properties to ensure that it remains approximately balanced, resulting in a logarithmic height relative to the number of nodes.   
The properties of the Red-Black Tree are:

1. Every node is either red or black.

2. The root is always black.

3. All leaves (NIL nodes) are black.

4. If a red node has children, then the children are always black.

5. Every path from a node to its descendant NIL nodes has the same number of black nodes.

These properties ensure that the longest path from the root to any leaf is no more than twice the length of the shortest path, thereby maintaining balance and ensuring efficient operations.

### Implementation

Node Structure

Each node in the Red-Black Tree contains the following data:

• Key: An integer that uniquely identifies the node.

• Value: An object representing the data associated with the key, such as a recipe.

• Left and Right Child References: Pointers to the left and right children of the node.

• Parent Reference: A pointer to the parent node.

• Color: An enumeration that can be either RED or BLACK.

Insertion Operation

The put(int key, Object value) method is responsible for inserting a new node into the tree. The process involves:

1. Finding the correct position in the tree where the new node should be inserted, maintaining the binary search tree property.

2. Inserting the node with a default color of RED.

3. Fixing any violations of the Red-Black Tree properties caused by the insertion. This involves checking the color of the parent and uncle nodes, performing rotations, and possibly recoloring nodes.

Search Operation

The find(int key) method searches for a node with the specified key. Starting from the root, the method traverses the tree by comparing the key with the current node’s key, moving left or right accordingly until the node is found or the search reaches a NIL node.

Deletion Operation

The remove(int key) method removes a node from the tree. The process includes:

1. Locating the node to be deleted.
2. Replacing the node with its successor if it has two children, or its child if it has one.
3. Fixing any double-black violations that may arise, ensuring that the Red-Black properties are maintained after deletion.

Program Integration

The Red-Black Tree is integrated into a program designed to manage a database of cooking recipes. Each node in the tree stores a recipe’s ingredients and metadata, allowing for efficient addition, deletion, modification, and querying of recipes. The tree’s balanced nature ensures that these operations remain performant even as the database grows.

#### Theoretical Foundation: Red-Black Tree vs. Normal Binary Tree

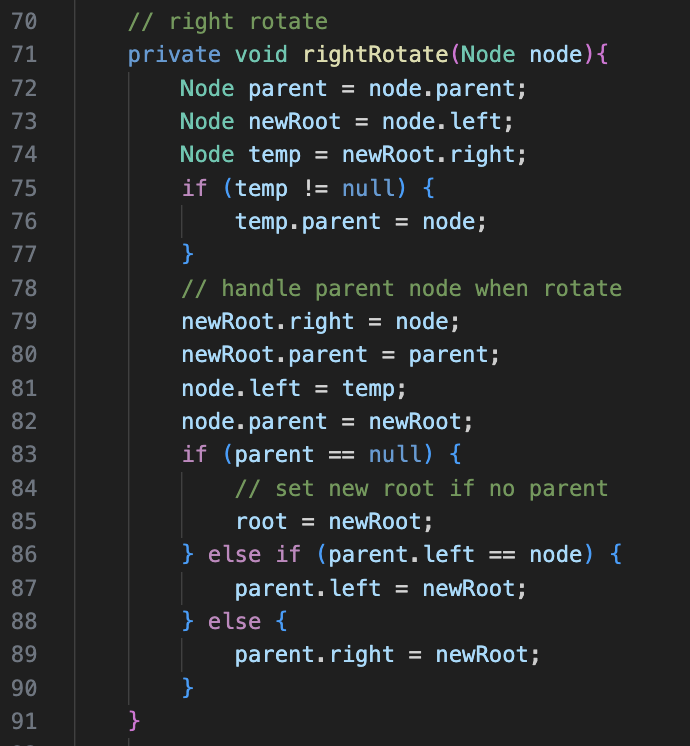
A Red-Black Tree maintains a balanced structure through its color properties and rotations. This balance ensures that the tree’s height is logarithmic relative to the number of nodes, providing efficient O(log n) operations for insertion, deletion, and search. The Red-Black Tree has key advantages in guaranteeing balance, predictable time complexity, and robustness without degrading performance as the data set gets larger.

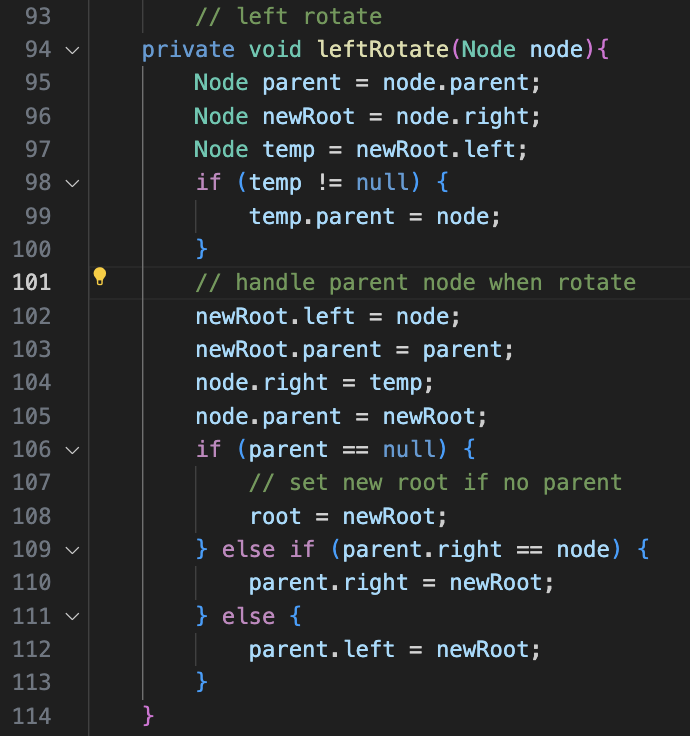
Normal binary trees on the other hand do not enforce any balancing properties. As a result, they can become skewed, with one side significantly deeper than the other, leading to a worst-case time complexity of O(n) for basic operations. This degradation in performance is particularly problematic in applications that require frequent data modification and retrieval, such as recipe management in our case.

#### Performance Analysis and Proof Of Correctness

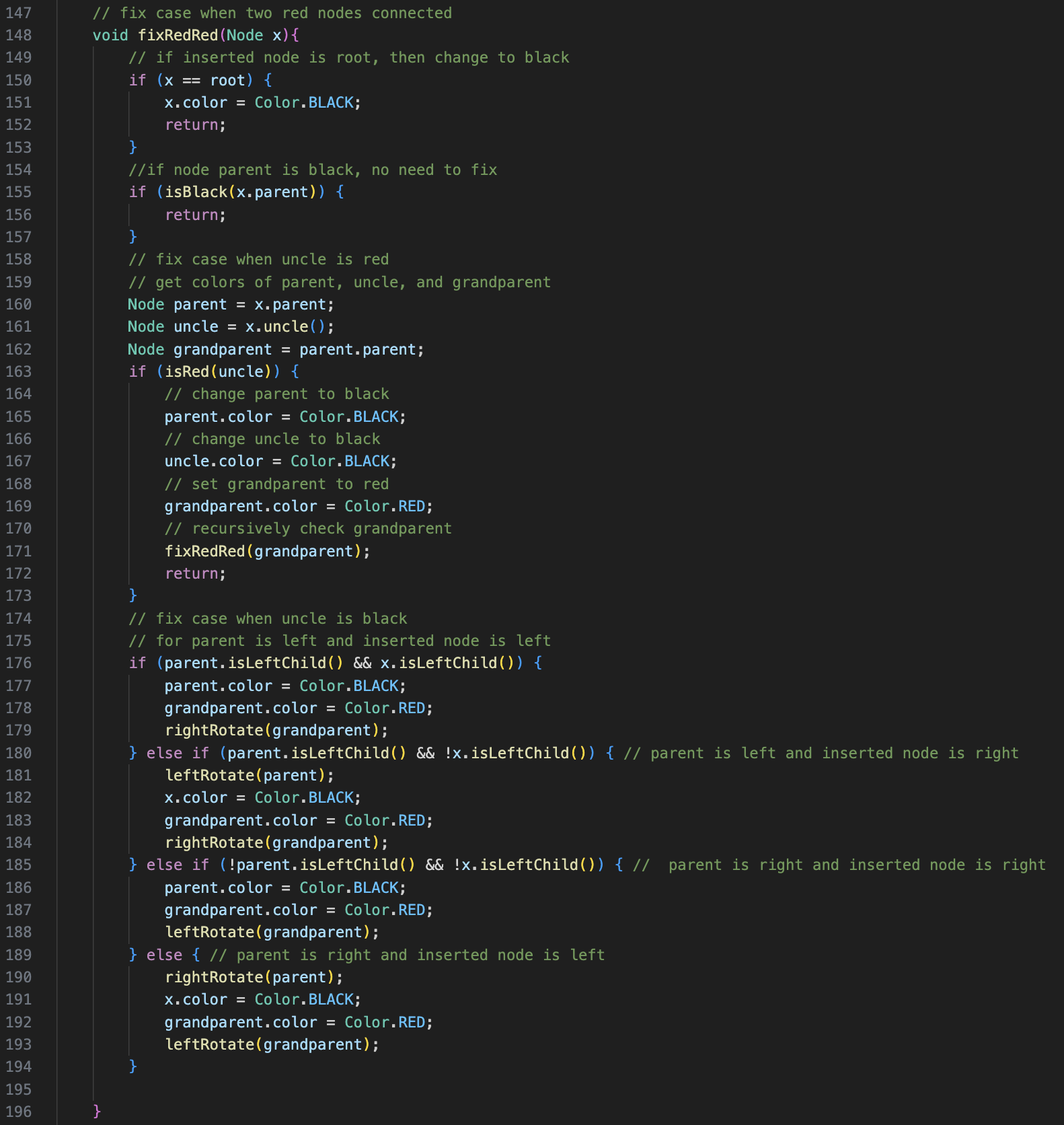
Due to the characteristics of a Red-Black Tree, certain adjustments or rotations are necessary to maintain the tree's self-balance.

* Right Rotate and Left Rotate operations are performed when the height of a subtree is impacted, helping to restore balance.
* FixRedRed and FixDoubleBlack procedures are used to correct invalid configurations, such as when two red nodes or two black nodes are not equal in the subtrees, which violates the tree's properties.

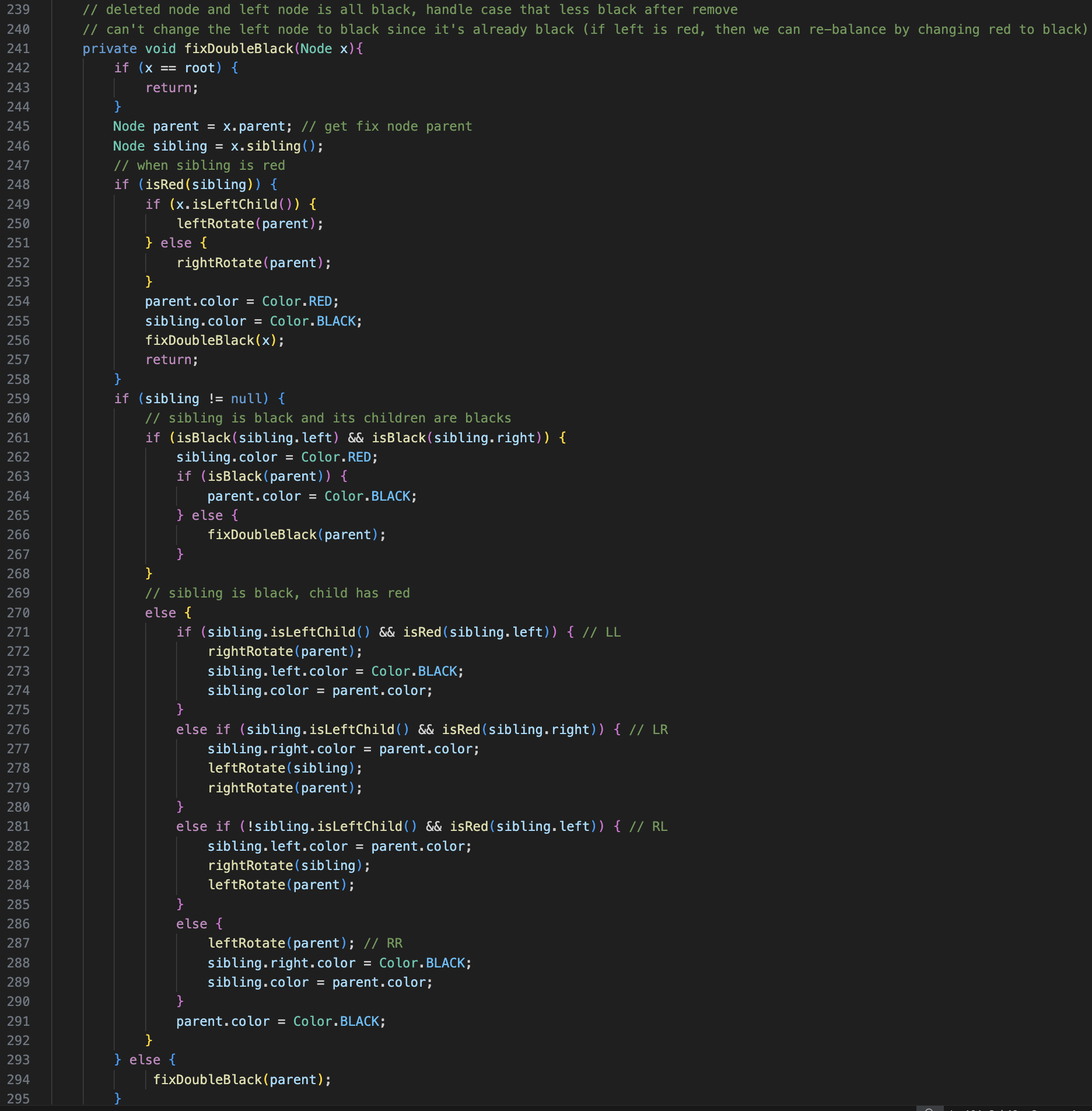




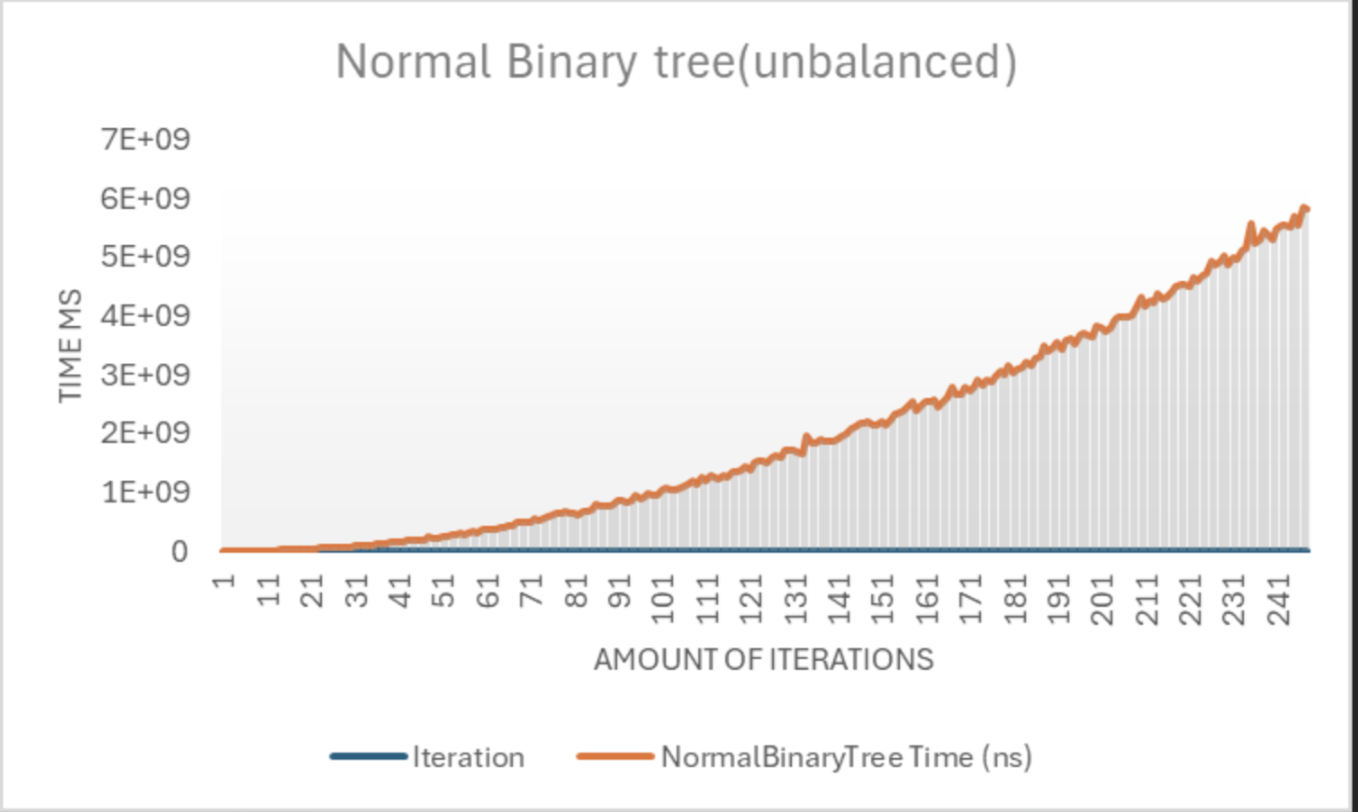
* Once we manipulate the Red-Black Tree with insertion or deletion, the height of the subtree will be impacted. The above rebalancing code helps to ensure that the tree remains balanced, thereby maintaining the logarithmic time complexity for search, insert, and delete operations.



* This code fixes the case when the inserted node is red and would connect two red nodes.

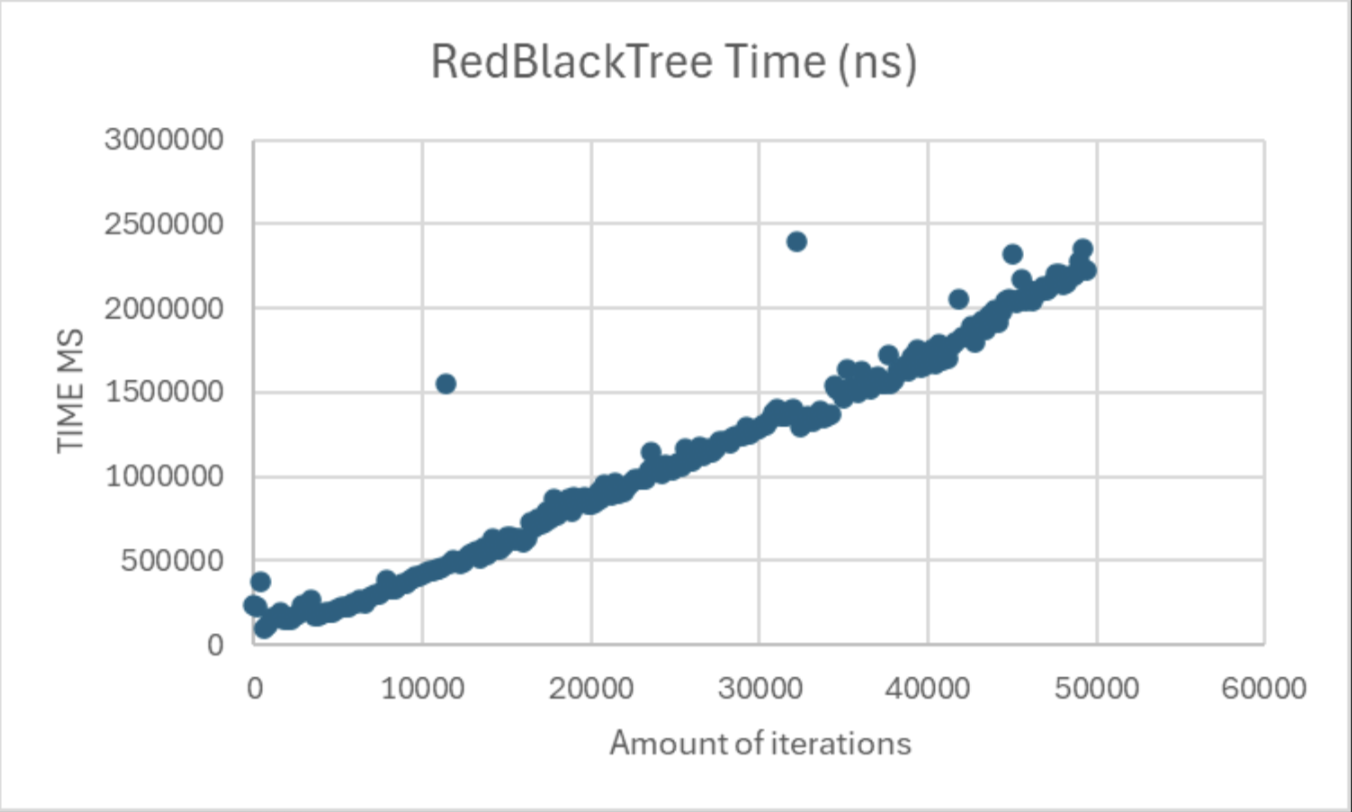


* This code fixes the case when we delete a node from the tree and leaves us with an odd number of black nodes.



*Fig.1 Performance Analysis of Unbalanced Binary Tree Insertion Operations*

In the above figure(Fig.1), you will notice an exponential rise in how long it takes to iterate with an unbalanced binary tree. However, this time scale is a bit misleading as the time complexity for an unbalanced binary tree to iterate is O(n). The reason you see exponential growth is due to other processes in the background causing the JIT compiler to stumble. The main takeaway here is to see how extreme of a difference it is compared to the picture below that uses a self balancing binary tree (The Red Black Tree).



*Fig.2 Performance Analysis of Red-Black Tree Insertion Operations*

The Red-Black tree consistently outperforms the normal binary tree in use due to its balanced nature, preventing the tree from becoming skewed and ensuring that insertion times remain logarithmic. The normal binary tree shows significant performance degradation as the number of nodes increases, with insertion times increasing dramatically.

#### Practical Usability

The Red-Black Tree is well suited for this application and any other application that requires frequent data insertion and removal. It is also very well scalable so if one were to expand this application to a networked collection of recipes with thousands of users able to access and add their own, the performance of the tree would still remain stable and efficient. The additional memory overhead required to store color in each node is completely outweighed by the performance benefits.

The AVL Tree is also a viable option with its own trade offs. Similar to the Red-Black Tree, it is a highly efficient self-balancing binary search tree, but with a different balancing approach. It ensures that the heights of the two child subtrees of any node differ by no more than one, thereby maintaining a more rigid balance compared to Red-Black Trees. This strict balancing makes AVL Trees particularly efficient for read-heavy applications where search operations are more frequent than insertions or deletions. In scenarios like this, the AVL Tree can offer slightly faster lookup times compared to a Red-Black Tree because it tends to be more balanced.

However, the AVL Tree’s strict balancing comes with a trade-off. It requires more rotations during insertion and deletion operations to maintain its balance. This can lead to a higher performance overhead in scenarios where insertions and deletions are frequent. Despite this, the AVL Tree remains highly efficient for datasets where reading is the most common function, and the cost of the additional balancing is justified by the enhanced search speed. Like the Red-Black Tree, the AVL Tree is also scalable and can efficiently manage large datasets. However, its more complex balancing might make it less preferable in environments like this where both insertion and deletion performance is a critical factor. Since write operations are frequent in this recipe book application, we took the Red-Black Tree as being the better option, although with the size of our data set the actual performance difference in practice is marginal.

## **CONCLUSION**

Both AVL and Red-Black Trees are excellent data structures, developed as self-balancing enhancements of the binary tree. While they theoretically share the same time and space complexity, subtle differences in performance emerged during speed testing, likely due to nuances in the testing and implementation methods. These differences, though minor, highlight the distinct characteristics of each tree structure, emphasizing the importance of considering these unique qualities in real-world applications. Both types of Trees demonstrated impressive performance, particularly in search operations, but their implementation complexity can make them challenging to maintain. Each tree structure has its own strengths and is highly valuable in specific application areas, making it essential to account for their unique characteristics when selecting the most appropriate data structure.

One significant weakness identified in the project is the complexity of implementing and maintaining self-balancing trees, such as Red-Black and AVL Trees. The code becomes more complex due to the sophisticated balancing procedures needed to maintain efficiency, which makes it challenging to extend and debug. Additionally, the increased memory overhead associated with maintaining balance, particularly in Red-Black Trees, poses a challenge in environments where memory is limited. These factors make these data structures less accessible for those new to the concepts and can complicate their use in memory-constrained systems.

Limitations in the performance testing were also encountered. It's likely that the testing environment and implementation assumptions don't accurately represent how these trees behave in real-world situations, where factors like different workload forms and data distributions could affect the outcomes. The results might not accurately reflect how these trees perform in more complicated or extensive applications due to the need for simplifying assumptions.

Looking ahead, there are several avenues for future research. More detailed comparisons of Red-Black and AVL Trees in various datasets and practical applications may offer more insightful information on how they perform differently. Additionally, there is potential for exploring optimizations that reduce memory overhead and simplify balancing algorithms, making these data structures more practical for various applications. To further understand these trees' practical utility, more research could apply them in various contexts, like network routing or database indexing.

Overall, we have gained some insights about the strengths and challenges of implementing these tree data structures. While Red-Black and AVL Trees offer notable benefits in terms of performance, their complexity and memory requirements need to be carefully considered when choosing the best structure for a specific use case. These structures can be used in a variety of real-world situations even more successfully with more research and optimization.