Red/Black Tree Implementation Report

Made by: Will Frautschy // Aasama Prabhakar

# Implementation of Red Black Tree and Test Suite

**GitHub Source**: <https://github.com/wfrautschy4/CSE-2331-Final-Project>

For this project, we used java to code it while using classes like the setup of software 2. The library includes two classes, one for binary tree, and another for the implementation of Red/Black Tree. The implementation includes functions for insertion, deletion, searching, and traversing. The test suite is also in the repo and ensures that each method works correctly.

# A number lines and numbers on a black background AI-generated content may be incorrect.Visualization

For our project, to visualize and understand how the tree is changing given any operations are being acted on it, we use a library called TreePrinter which simply prints the tree into the command line. While it is not perfect since you cannot animate changes, it does help you understand the whole structure of the tree without manually traversing it at any given instance. It also had to ability to display red or black versions of the nodes to accurately depict the colors of the red/black tree.

# Documentation and Presentation

|  |  |
| --- | --- |
| Insert | The way that insert is implemented is by recursively climbing down the trees and picking the child whose is appropriate for the data being inserted. Once the node has been added, it checks if the node needs balanced, then traverses back upwards to the parent and checks if it needs balanced until it reaches the top of the tree. |
| Delete | Delete works similarly by recursively climbing down the tree until the node being removed is found. Once it is found it is replaced by another appropriate node nearby and then balances its way back up the tree if required. |
| Search | Recursively climbs down the tree and returns true if the node is found |
| Traverse | Prints all the elements in order from left to right. Basically, prints out the list from min -> max. This would be the same as converting the tree into a sorted list and being able to traverse it in that manner outside of the tree. But this would hurt runtimes |
| Red-Black Insert | Checks all conditions of Red-Black Tree while inserting a new node: Makes sure root is black, no red nodes have red children, is perfectly balanced throughout all paths, every node is either red or black and that all leaf nodes are black |
| Red-Black Delete | Makes sure that all conditions of Red-Black Tree remain intact while deleting mainly: by making sure that no red nodes have red children and that all paths are perfectly balanced. |

# Performance Analysis

We implemented and measured the runtime of 3 different functions (Insert, Delete, and Search). In an attempt to measure the runtime of a single insertion with varying sizes of trees, I found that most of my insertions were insignificant like 1 ns at 10000 items. So instead, I timed how long it took to insert n items. Therefore, instead of expecting log(n) time for a single insert. We expect nlog(n) time for n inserts. Since ∑(1->n)(log(i) = nlog(n)

Link to Desmos Graphs: <https://www.desmos.com/calculator/boxhkrjl3p>

|  |  |
| --- | --- |
| Insert |  |
| Delete |  |
| Search |  |

**Insert –** Runtime appears to be approximately nlog(n) within some constant

**Delete –** Runtime also appear to show a clear nlog(n) relation which makes sense if it is deleting n items and each iteration takes log(n) time

**Search –** The first 7 iterations are super fast and form a curve relative to log^3(n) but around the 8th iteration, I’m assuming that the cache on my computer runs out and it slows down to around 20x for a 10x size increase. That would fit the curve around nlog(n) (which is what it should be)

**Comparison Chart**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Tree** | **Insertion** | **Deletion** | | | **Traversal** | |
| **Red-Black Tree** | O(log n) | O(log n) | | | O(log n) | |
| **Binary Search Tree** | O(n) | O(n) | O(n) | | | |
| **AVL Tree** | O(log n) | O(log n) | | | O(log n) |
| **Min/Max Heap** | O(log n) | O(log n) | | O(n) | | | |

When comparing different tree-based data structures, the Red-Black Tree stands out for its consistently balanced time complexities across a wide range of operations. Although structures like a perfectly balanced Binary Search Tree (BST) can occasionally produce faster insertion, deletion, or search times under ideal conditions, they lack guarantees against imbalance as data grows unpredictably. In contrast, the Red-Black Tree maintains near-optimal performance in all input scenarios, thanks to its dynamic self-balancing through color properties and rotations. This ensures that operations such as insertion, deletion, traversal, and search remain efficient, typically running in O(log n) time regardless of input order or distribution. Its ability to balance operational speed and structural stability makes it one of the most reliable and versatile choices among tree-based data structures, especially when consistent performance is critical. The AVL tree also has similar runtimes as red/black tree but has slightly stricter balancing operations, which can cost more even though it may not affect its asymptotic runtime. Red/Black tree’s more lenient balancing can make it better for larger datasets.

# Documentation and Implementation

The Red-Black Tree implementation is structured around maintaining balance through color properties and efficient pointer manipulation. Each node contains a color attribute (red or black), along with references to its parent, left child, and right child. Insertion begins by placing a new node as a red leaf, followed by a correction phase (insertRedBlackValidity) that uses rotations and recoloring to restore Red-Black Tree properties. Deletion is handled through a similar balance-preserving process, where the node is removed, and if necessary, a fix-up (deleteRedBlackValidity) is triggered to resolve violations caused by black node removal. These algorithms guarantee that the tree height remains logarithmic relative to the number of nodes, ensuring that insertion, deletion, and search operations consistently perform in O(log n) time even under worst-case input distributions. All core operations modify tree pointers directly, minimizing overhead and promoting faster rebalancing compared to full subtree copying.

Several optimizations were incorporated to enhance both performance and structural reliability. Rotations (rotateLeft and rotateRight) were implemented using direct pointer adjustments rather than subtree transfers, allowing rotations to complete with minimal overhead. During deletions, successor tracking was carefully managed to ensure fix-up operations applied to the correct replacement node rather than a cleared successor, addressing a common deletion pitfall. The design also explicitly maintains parent pointers during every modification, ensuring upward navigation is always accurate. Special cases, such as deleting the root node or removing nodes with zero children, are handled by updating parent-child links directly. Throughout the project, clarity and robustness were prioritized, favoring clear logic and minimal memory footprint without sacrificing the logarithmic time complexity guarantees of the Red-Black Tree. These design decisions result in an implementation that is both theoretically sound and highly practical for real-world ordered data operations.