Structures de Données - Avancé

A Comparative Study of AVL and Red-Black Trees

TRAN Minh Duong
NGUYEN Hoang Nhat
PHAM Ngan Ha

Fall 2025

Abstract

This report presents the implementation of a generic Red-Black Tree and a comparative performance analysis against an AVL Tree. The operations of insertion, search, and deletion were tested on datasets of varying sizes. The analysis of the results, presented graphically, confirms the logarithmic complexity of both structures and highlights the subtle performance differences related to their respective rebalancing strategies.

1 Introduction

Self-balancing binary search trees are fundamental data structures in computer science, guaranteeing a time complexity of $O(\log N)$ for basic operations. Among the most well-known implementations, AVL trees and Red-Black Trees offer distinct rebalancing strategies with different performance trade-offs.

The objective of this project is to implement a functional Red-Black Tree and empirically compare it to an AVL tree. By measuring the execution time of key operations on increasingly large datasets, we aim to validate their theoretical complexities and analyze the practical advantages of each structure.

2 Data Structures and Algorithms

Both structures are built on the same foundation (a binary search tree), but they use different methods to stay balanced.

2.1 AVL Trees

An AVL tree is a **strictly** balanced tree. Its main rule is that for any node, the heights of its left and right subtrees can only differ by one. It keeps track of this using a "balance factor" (usually -1, 0, or 1) stored in each node. If an insertion or deletion breaks this rule (making the balance -2 or +2), the tree uses single or double rotations to fix itself.

2.2 Red-Black Trees

A Red-Black Tree is a **less strictly** balanced tree. It uses a simpler idea where every node is colored either RED or BLACK. It follows five basic rules to stay balanced:

- Every node is either red or black.
- The root node is always black.
- All leaf nodes (NIL) are black.
- If a node is red, then both its children are black.
- Every simple path from a node to a descendant leaf contains the same number of black nodes.

If an insert or delete breaks these rules (especially rules 4 or 5), the tree fixes itself using rotations and by "flipping" node colors.

2.3 Deletion in Red-Black Trees

Deleting a node from a Red-Black Tree is much trickier than inserting one. It is very easy to break the color rules, especially the black-height property. The process involves first performing a standard BST deletion and then fixing any rules that were broken.

First, we find the node x to be deleted. Just like in a normal BST, there are three cases:

- **Case 1:** x is a leaf. We just remove it.
- **Case 2:** x has 1 child. We replace x with its child.
- **Case 3:** x has two children. We find its in-order successor (the smallest node in the right subtree), copy its data into x, and then delete the successor node instead.

Here's the problem: if the node we removed was RED, all the rules are still fine. But if the node we removed was BLACK, we have broken the black-height rule. The paths that went through that node now have one less black node than all the other paths. This creates a "double black" problem on the node x that replaced it.

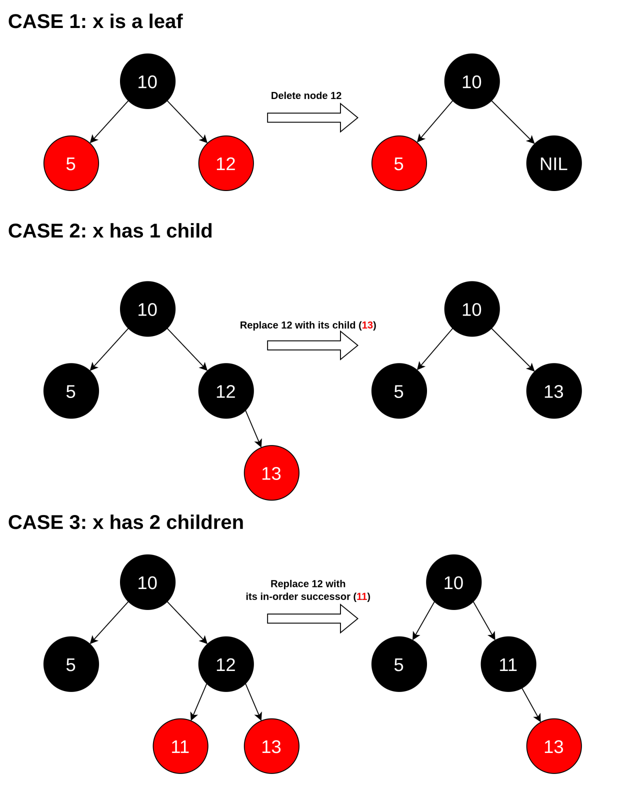

Figure 1: Standard BST deletion before rebalancing.

2.4 Deletion Fix-Up in Red-Black Trees (Double Black Problem)

A "double black" node x means that path is "missing" one black node. To fix this, we look at x's sibling, w. There are four cases.

Case 1: Sibling w is Red If the sibling w is RED, we can't fix the problem directly. We do a rotation at the parent and swap the colors of the parent and w. This doesn't solve the problem, but it guarantees that x's new sibling is BLACK, which turns the problem into one of the next three cases.

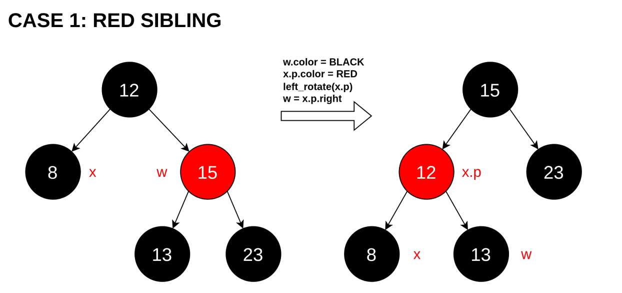

Figure 2: Case 1 — Red sibling rotation.

Case 2: Sibling w is Black (and its children are Black) If w is BLACK and both of its children are BLACK, we can safely recolor w to RED. This passes the extra black from x and w up to their parent p. We then re-run the whole fix-up process starting from p.

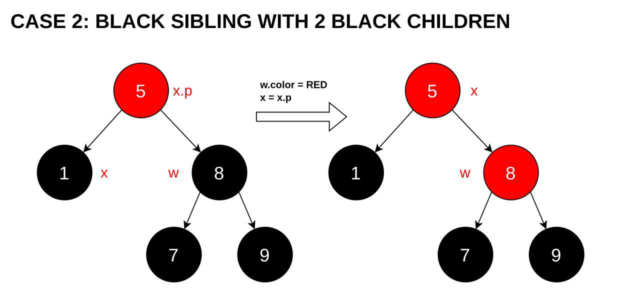

Figure 3: Case 2 — Black sibling with two black children.

Case 3: Sibling w is Black (with Red left child) If w is BLACK, its right child (the one on the opposite side of x) is BLACK, but its left child is RED. We do a rotation at w and swap the colors of w and its red child. This is a setup move that turns the problem into Case 4.

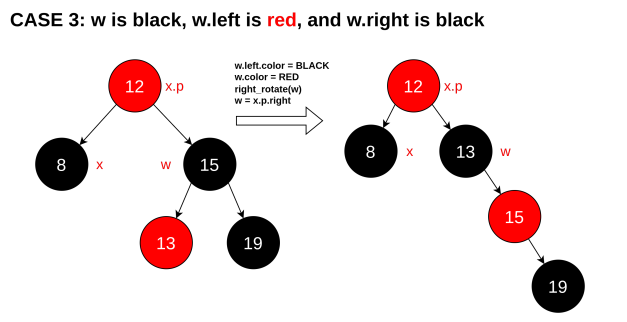

Figure 4: Case 3 — Black sibling with red near child.

Case 4: Sibling w is Black (with Red "far" child) If w is BLACK and its "far" child is RED, we can solve the problem for good. We do a final rotation at the parent and recolor three nodes (the parent, w, and w's red child). This balances the black-heights, and the "double black" problem is gone.

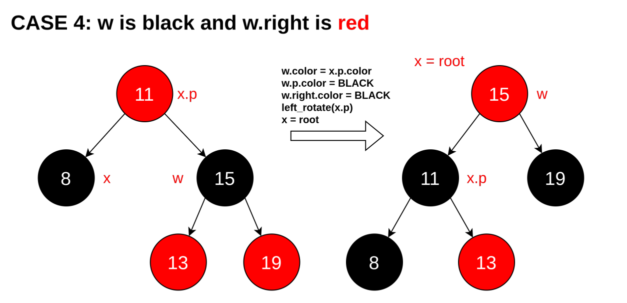

Figure 5: Case 4 — Black sibling with red far child.

After these steps, the tree is fully balanced again and follows all the Red-Black rules. Even though this seems complex, it only requires a few checks and at most three rotations. This is why the delete operation is still very fast, with a worst-case time of $O(\log N)$.

3 Methodology

3.1 Project Structure

The project directory is organized as follows, illustrating the separation between source files, unit tests, benchmark and analysis.

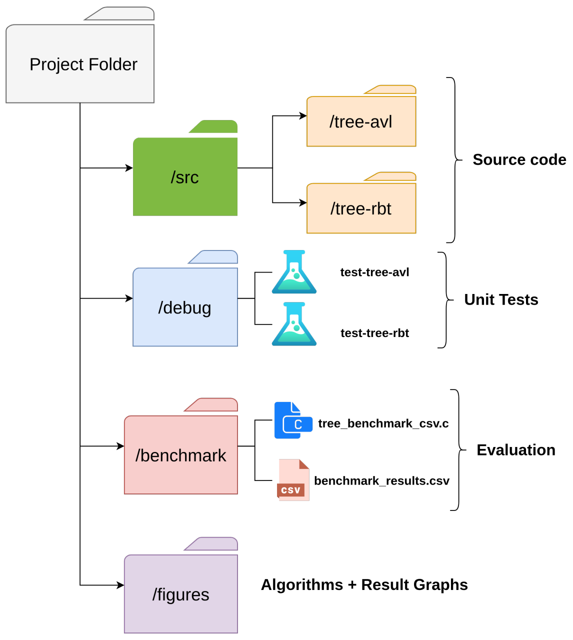

Figure 6: Folder hierarchy of the project

3.2 Experimental Design

Timing Method

Execution time is measured using `clock_gettime()` with `CLOCK_MONOTONIC` for high precision. The elapsed time (in milliseconds) is computed as:

$$t_{ms} = (s_{end} - s_{start}) \times 10^3 + \frac{(ns_{end} - ns_{start})}{10^6}$$

Workload and Process

To compare the two trees, the experiment was run for data sizes $N = 50, 100, 150, \ldots, 1000$.

For each value of N, the following three-stage workload was timed for both the AVL tree and the Red-Black tree:

1. **Measure Insertion:** The total time to insert N randomly generated integers into an empty tree.
2. **Measure Search:** The total time to search for all N elements that were just inserted.
3. **Measure Deletion:** The total time to delete all N elements, which were pre-shuffled into a random order.

Output Format

The averaged results for each N were saved to a CSV file `benchmark_results.csv` for plotting:

4 Results and Analysis

We measured the operation time in milliseconds for AVL tree and Red-Black tree as N grew from 100 to 1000. A dashed O(N) line is shown for reference.

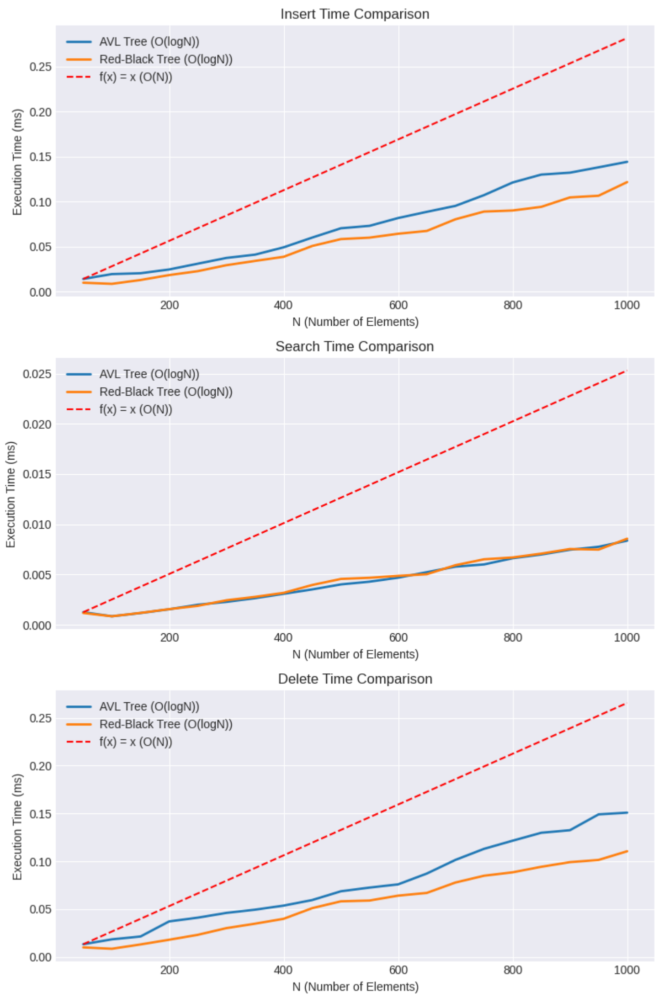

Figure 7: Comparison of Execution Times for AVL and Red-Black Trees.

The Red-Black tree is observed to be faster than the AVL tree considering their insertion and deletion time, with the time complexity of O(logN)). Searching time of both types of tree is nearly identical across all N, time complexity stays O(logN).

Analysis of the graph (Figure 7) reveals several key points:

1. The Red-Black tree is better for updating operations (insertion, deletion), which can be attributed to the fact that it has fewer rotations and more recoloring. Meanwhile, the AVL tree enforces stricter balance with more rotations on updating.
2. For search operation, performance is nearly identical because search time is defined by tree height, and both trees have similar height.

5 Conclusion

This project successfully implemented and validated the functionality of a generic Red-Black tree and compared it to an AVL tree. The empirical performance analysis confirmed that while both structures offer logarithmic complexity, they present distinct trade-offs.

The Red-Black tree proved more efficient for insertions, while the AVL tree holds a slight theoretical advantage for searches. In practice, the choice between the two depends on the expected workload: applications requiring frequent insertions and deletions (e.g., schedulers) will benefit from a Red-Black tree, whereas those dominated by searches (e.g., static dictionaries) might prefer the more rigid structure of an AVL tree.

6 Member Contribution

As per the project requirements, members provided a cross-evaluation of participation, based on the following scale:

- **A:** Participated well in the group's work.
- **B:** Participated moderately.
- **C:** Did not participate.

Table 1: Cross-Evaluation of Group Members

Evaluator	Duong	Ha	Nhat
Duong	—	A	A
Ha	A	—	A
Nhat	A	A	—

We believe that everyone has devoted their time and contributed equally to the completion of this project.

References

[1] Tran Minh Duong, Nguyen Hoang Nhat, and Pham Ngan Ha. Project source code: Avl vs. red-black trees. https://github.com/tmdeptrai/AVLvsRedBlack, 2025.

[2] Russell A. Brown. Comparative performance of the avl tree and three variants of the red-black tree. *Software: Practice and Experience*, 55(9):1607–1615, June 2025.

[3] Svetlana Strbac-Savic and Milo Tomasevic. Comparative performance evaluation of the avl and red-black trees. pages 14–19, 09 2012.

[4] Zegour Djamel Eddine and Lynda Bounif. Avl and red black tree as a single balanced tree. pages 65–68, 03 2016.

[5] Deepa Balasubramaniam. A survey on different method of balancing a binary search tree. 4:660–664, 10 2017.

[6] GeeksforGeeks. Introduction to red-black tree. https://www.geeksforgeeks.org/introduction-to-red-black-tree/. Accessed: 2025-11-09.

[7] Wikipedia. Red–black tree — wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Red%E2%80%93black_tree. Accessed: 2025-11-09.

[8] Wikipedia. Avl tree — wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/AVL_tree. Accessed: 2025-11-09.

[9] GeeksforGeeks. Avl tree data structure. https://www.geeksforgeeks.org/introduction-to-avl-tree/. Accessed: 2025-11-09.

[10] Richard E. Pattis. Avl trees. https://ics.uci.edu/~pattis/ICS-23/lectures/notes/AVL.txt. Course Notes for ICS-23, University of California, Irvine. Accessed: 2025-11-09.