Dynamic Data Manipulation Using AVL Trees vs. Red-Black Trees

Source Code

Simon5ei/COMP47500-Assignments

Name	Student ID	Email	Contribution
Jon Eckerth	22209542	jon.eckerth@ucdconnect.ie	20% (Test Data Generation)
Simian Wei	24207892	simian.wei@ucdconnect.ie	20% (ADTs Implementation)
Anthony Salib	20341603	anthony.salib@ucdconnect.ie	20% (Visual Programming)
Daniel Ilyin	18327256	daniel.ilyin@ucdconnect.ie	20% (ADTs Implementation)
Maximilian Schöll	24211713	maximilian.scholl@ucdconnect.ie	20% (Report Writing)

1. Problem Domain Description

Modern programming language standard libraries (such as Java's TreeMap/TreeSet) commonly implement balanced binary trees as efficient data structures for managing ordered collections. **AVL trees** and **Red-Black trees** are two popular balanced binary search tree variants that maintain a logarithmic height of O(log n) through distinct self-balancing mechanisms. This critical property ensures efficient O(log n) time complexity for insertion, search, and deletion operations, making them suitable for applications requiring both dynamic updates and ordered data access.

For example, consider a real-time database indexing system for financial transactions that requires rapid insertions (new transactions), deletions (canceled orders), and lookups (transaction verification). An AVL or Red-Black tree can maintain an ordered index of transaction records by timestamp or ID, ensuring that each operation executes in O(log n) time even as the dataset grows to millions of entries.

This experiment simulates high-frequency transaction processing conditions with both sequential and random access patterns to determine which tree structure better suits the indexing system's needs under varying data distributions and operation mixes.

2. Theoretical Foundations

2.1. AVL Tree

An AVL tree is a strictly balanced binary search tree where the height difference between the left and right subtrees of any node does not exceed [1]. After insertion or deletion, AVL trees restore balance through rotation operations (left, right, left-right, and right-left rotations). Due to its strict balance, AVL trees typically perform better in search operations. However, insertion and deletion operations may incur higher overhead due to more frequent rotations compared to other balanced tree structures, particularly when handling sequences of ordered insertions.

2.2. Red-Black Tree

A Red-Black tree maintains balance through a color system where each node is either red or black, following specific rules: the root is black, red nodes cannot have red children, and all paths from root to leaves contain the same number of black nodes. This approach allows the tree to be up to twice as tall as a perfectly balanced tree but requires fewer rotations during updates. The balancing conditions of Red-Black trees are relatively relaxed compared to AVL trees, allowing the tree height to be slightly higher than that of AVL trees. As a result, Red-Black trees typically perform better for insertions and deletions while still providing good search performance, making them the preferred choice in many standard libraries.

2.3. Balance Breaking Cases

Although the definitions of various balanced binary trees differ, they share fundamental similarities. The key differences lie in the specific information maintained by the nodes and how this information is updated after rotational adjustments. When a binary balanced tree loses its balance, only four distinct scenarios can occur. The operations to restore balance are limited to left and right rotations. We will describe these four imbalance cases in detail below before proceeding to compare various binary balanced tree implementations.

Case LL: The left child's left subtree is too tall, unbalancing node T.
Adjustment: Right-rotate node T (see Figure 1).

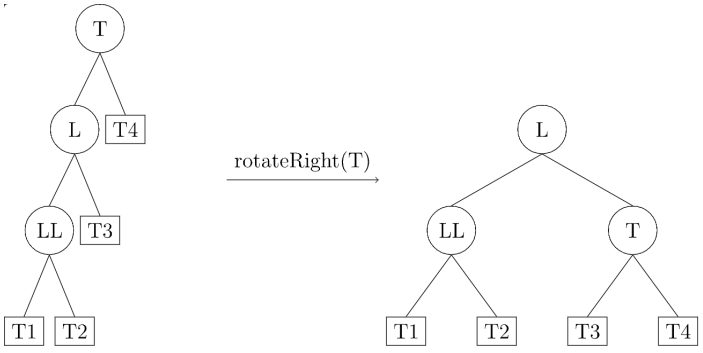

Figure 1: R-rotate (source: [4])

Case RR: The right child's right subtree is too tall, unbalancing node T.
Adjustment: Left-rotate node T (see Figure 2).

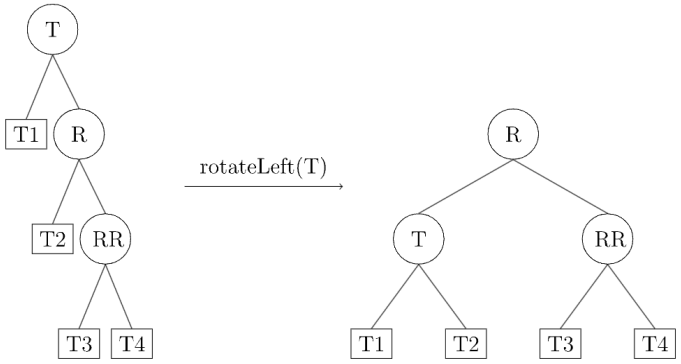

Figure 2: L-rotate (source: [4])

Case LR: The right subtree of T's left child is too tall.
Adjustment: Left-rotate node node L (the left child of T) to convert the imbalance to Case LL, then right-rotate node T (see Figure 3).

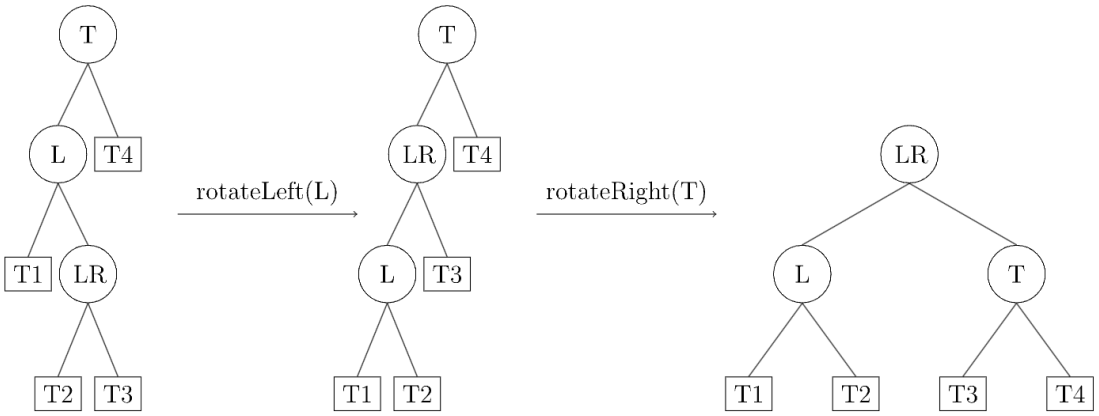

Figure 3: LR-rotate (source: [4])

Case RL: The left subtree of T's right child is too tall.
Adjustment: Right-rotate node R (the right child of T) to convert the imbalance to Case RR, then left-rotate node T (see Figure 4).

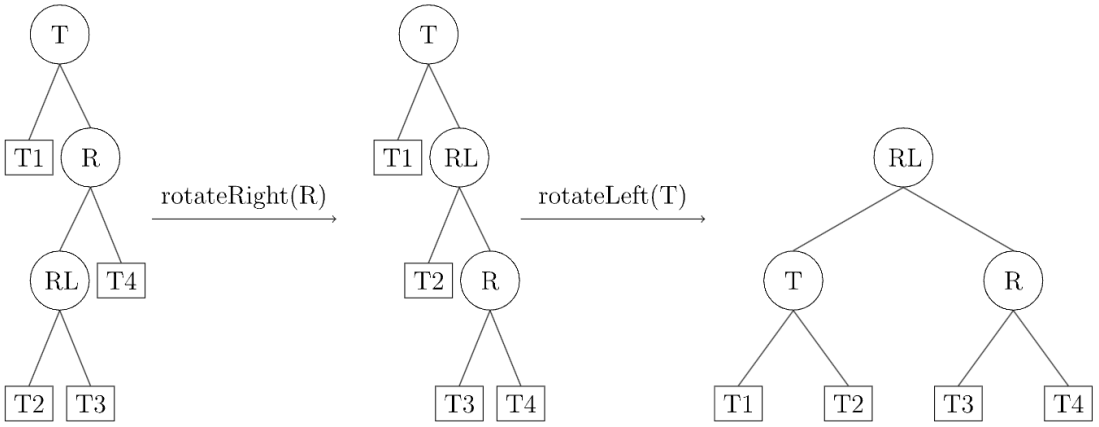

Figure 4: RL-rotate (source: [4])

3. System Design and Implementation

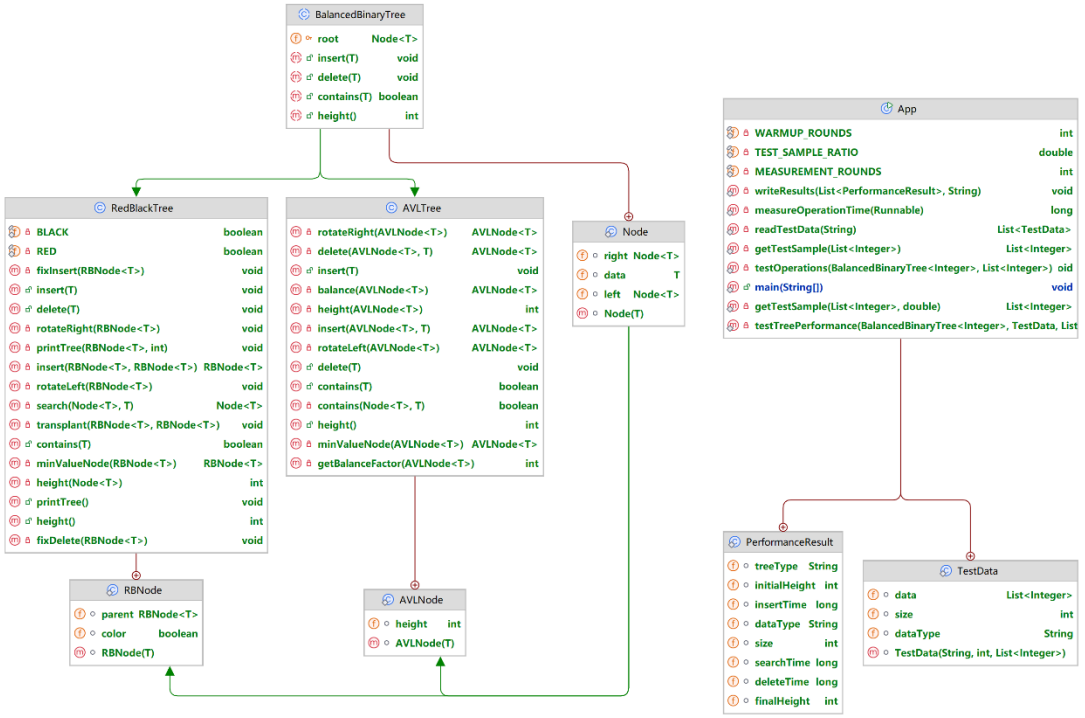

Figure 5: System Architecture

The code structure of this experiment is divided into the following parts:

- **Base Class Design**: The BalancedBinaryTree abstract class is defined, containing common methods such as insertion, deletion, search, and tree height calculation.
- **AVL Tree Implementation**: Inheriting from BalancedBinaryTree, it implements AVL tree insertion, deletion, and rotation operations to ensure strict balance.
- **Red-Black Tree Implementation**: Inheriting from BalancedBinaryTree, it implements Red-Black tree insertion, deletion, and color-fixing operations to maintain approximate balance.
- **Testing Framework**: The App class reads test data, generates test samples, and tests the performance of AVL trees and Red-Black trees on different datasets. The test results are saved in CSV format for further analysis.

Test data is simplified to single integers representing data entry IDs (e.g., process IDs or entry numbers), focusing solely on indexing efficiency rather than on data content processing.

4. Analysis

By analyzing the dataset with random data distribution and the dataset with ordered data distribution, the graphs visualized by running the program are as follows:

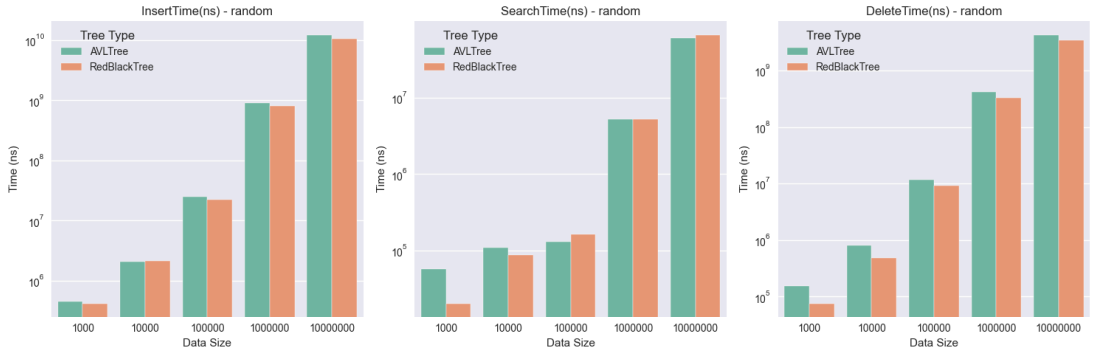

Figure 6: Random Dataset Result

For random data, the insertion and deletion efficiency of the red-black tree is consistent with the theory and slightly higher than that of the AVL tree. Unlike the theory, the query time of the red-black tree is also smaller than that of the AVL tree under smaller data.

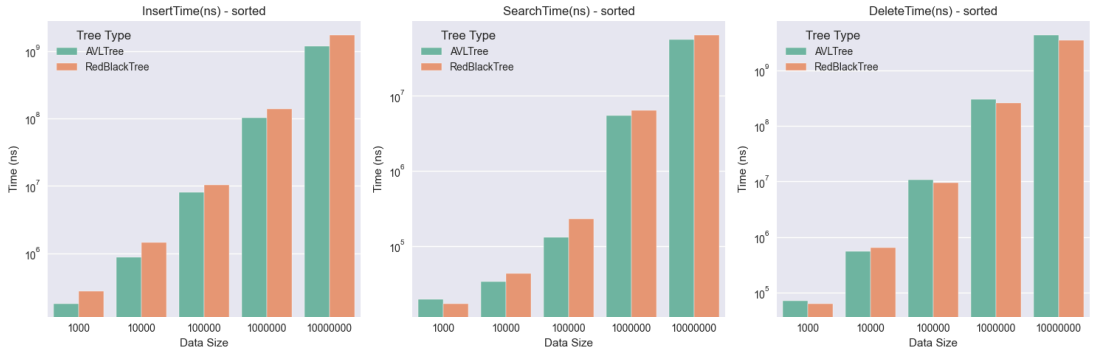

Figure 7: Sorted Dataset Result

For our ordered dataset, the performance aligns with the theoretical expectations. However, our analysis revealed an interesting anomaly: red-black trees demonstrate unexpectedly high efficiency with smaller data sets, which appears to contradict theoretical predictions.

This difference might be due to a different usage of cache by the red-black-tree. First, red-black trees typically require much less rotation operations than AVL trees due to their more relaxed balancing conditions. While AVL trees strictly limit height differences between subtrees to 1, red-black trees only require consistent black node counts from root to leaf. This reduction in rotations directly translates to fewer memory accesses during insertion and deletion operations.

The node structure of red-black trees is also more compact, storing only a single color bit (red or black) rather than the integer balance factors needed in AVL trees. This compactness enables better utilization of cache lines and reduces cache misses during traversal operations.

Red-black trees maintain a more stable overall structure despite allowing slightly higher tree

heights than AVL trees. This structural stability minimizes frequent memory reordering and enhances cache locality, which proves particularly beneficial for smaller datasets.

Red-black trees exhibit more favorable memory access patterns during modifications. While AVL trees often require global adjustments, insertion and deletion operations in red-black trees typically affect only localized subtrees. This localized access pattern aligns more effectively with modern CPU cache mechanisms, further explaining their superior performance with smaller datasets.

5. Conclusions

Our analysis of the experimental data shows that AVL trees and Red-Black trees each have their own strengths and weaknesses. We found that AVL trees tend to perform a bit worse than Red-Black trees when it comes to insertion operations, especially as datasets get larger. This makes sense since AVL trees need more rotations to keep their strict balance. In theory, AVL trees should be better for searching because they're more balanced, but surprisingly, our tests showed that Red-Black trees often did better in real-world scenarios, probably because they work better with the computer's cache. For deletions, Red-Black trees were more consistent performers since they don't need as many rotations due to their more relaxed balancing rules. Looking at the big picture, Red-Black trees generally performed better in our practical tests, particularly with datasets that change frequently, where the extra work needed for insertions and deletions really starts to matter.

6. Video Demonstration

Google Drive:
https://drive.google.com/file/d/14w6WHpRBAQxWmE11VSoDAbx-oVJg1HKw/view?usp=drive_link

7. References

[1] “DSA AVL Trees.” Accessed: Feb. 16, 2025. [Online]. Available: https://www.w3schools.com/dsa/dsa_data_avltrees.php

[2] “Introduction to Red-Black Tree.” Accessed: Feb. 16, 2025. [Online]. Available: https://www.geeksforgeeks.org/introduction-to-red-black-tree/

[3] R. Sedgewick and K. D. Wayne, Algorithms, 4th ed. Upper Saddle River: Addison-Wesley, 2011.

[4] “欢迎来到 OI Wiki.” Accessed: Feb. 16, 2025 [Online]. Available: https://github.com/OI-wiki/OI-wiki