COSC 520 - A2: Comparison of AVL, Red-Black, and Treap Trees

Qianyu Shang #18991588

March 7, 2025

Abstract

This report summarizes the theoretical and empirical performance of three self-balancing binary search trees: **AVL Tree**, **Red-Black Tree**, and **Treap**. I compare the time and space computational complexities of insertion, search, and deletion on large datasets, demonstrating each structure's $\mathcal{O}(\log n)$ performance in practice.

1 Introduction

Self-balancing binary search trees ensure efficient $\log n$ complexity for insertion, search, and deletion by maintaining a roughly balanced structure. Here, i examine *AVL*, *Red-Black*, and *Treap* trees, each offering a different approach to balancing:

- **AVL Tree**: Maintains strict height-balance (the height of left/right subtrees differs by at most 1).
- **Red-Black Tree**: Uses color-based constraints to balance; widely adopted (e.g., `std::map` in C++).
- **Treap**: Assigns random priorities to nodes; on average, the tree height remains $\log n$.

1.1 Motivation for These Structures

I chose these three because:

1. **AVL** is historically the first self-balancing BST, with rigorous height control.

2. **Red-Black** is commonly used in production libraries for its efficient insertion/deletion.

3. **Treap** is conceptually simpler (random priorities) yet provides average $\log n$ performance.

2 Time & Space Complexities

Table 1 summarizes the asymptotic complexities for each data structure. While AVL and Red-Black guarantee $\mathcal{O}(\log n)$ height, Treap has an *expected* $\log n$ height but can degrade to $\mathcal{O}(n)$ in worst cases.

Structure	**Insertion**	**Search**	**Deletion**	**Space**
AVL Tree	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
Red-Black Tree	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
Treap	Avg $\mathcal{O}(\log n)$	Avg $\mathcal{O}(\log n)$	Avg $\mathcal{O}(\log n)$	$\mathcal{O}(n)$

Table 1: Summary of time/space complexities for AVL, Red-Black, and Treap.

3 Methodology and Experimental Setup

3.1 Implementation Details

All trees (`AVL_Tree.py`, `Red_Black_Tree.py`, `Treap.py`) are written in Python. I then use `main.py` to:

- Generate random datasets in $[0, 10^9]$.
- Measure insertion, search, and deletion times.
- Produce line plots (`.png` files).

3.1.1 Dataset Link

For reproducibility, i provide a script to generate the data dynamically (rather than a fixed file). The dataset is located at: GitHub: Advanced-Algorithm/A2.

3.1.2 Code Link

The code is located at: GitHub: Advanced-Algorithm/A2.

3.2 Benchmark Approach

I tested three data sizes:

1. **100k**
2. **500k**
3. **1M**

For *search*, i sampled 20k queries each time (50% are from the inserted data, 50% newly generated). For *deletion*, i similarly prepared 20k keys (half valid, half new).

4 Results and Analysis

4.1 Insertion Times

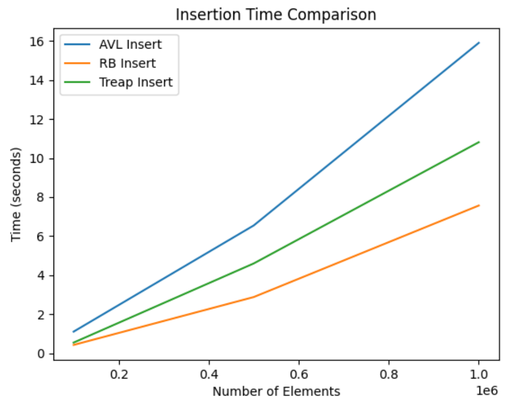

Figure 1: Insertion time comparison for AVL, RB, and Treap.

Figure 1 shows insertion time vs. data size. **Red-Black** is typically faster than AVL due to fewer rotations, and **Treap** tends to lie close to Red-Black in practice, though random priorities may cause minor variations.

4.2 Search Times

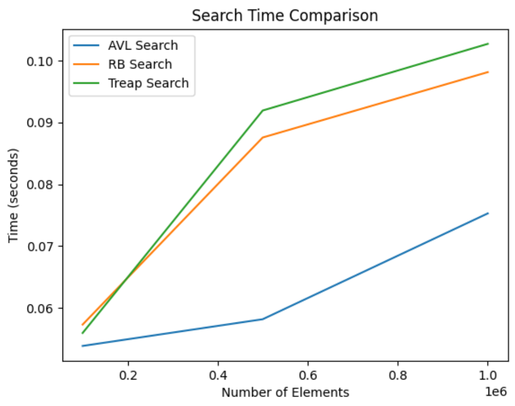

Figure 2: Search time comparison.

All three scale in $\log n$. **AVL** may be marginally faster for searches due to strictly minimal height, but differences remain small overall (Figure 2).

4.3 Deletion Times

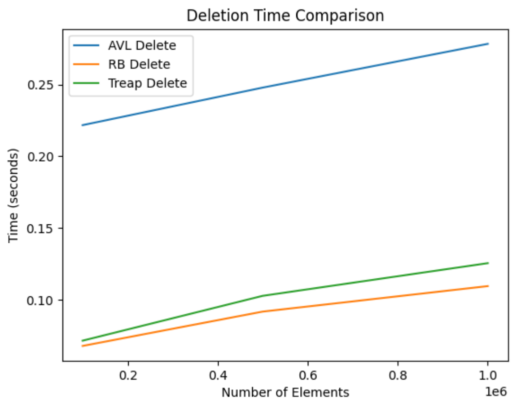

Figure 3: Deletion time comparison.

Figure 3 shows that **Red-Black** often outperforms AVL in deletion, while **Treap** remains competitive. In practice, Red-Black's more flexible balancing yields fewer rotations than AVL's strict approach.

5 Discussion

AVL ensures minimal tree height but can incur more rotation overhead on insert/delete. **Red-Black** often excels at insertion/deletion because it rebalances less aggressively, resulting in fewer rotations to maintain $\log n$. **Treap**, by randomizing priorities, achieves $\log n$ height in expectation and performed well in our tests (rarely hitting worst-case scenarios).

5.1 Practical Insights

Many standard libraries (e.g., `std::map` in C++) adopt Red-Black for a balance of good insertion/deletion times and guaranteed $\log n$. AVL may slightly outperform for read-heavy scenarios (faster searches), while Treap is simpler to implement with average-case $\log n$.

6 Conclusion

This project verifies that all three data structures maintain $\log n$-like performance under large-scale random data. **Red-Black** is typically fastest for insertion/deletion, **AVL** can be marginally faster for searches, and **Treap** remains a strong average-case performer. These findings align with known theory and real-world usage.

References

- G. M. Adelson-Velsky and E. M. Landis, *An algorithm for the organization of information*, 1962.
- Wikipedia: AVL Tree.
- Wikipedia: Red-Black Tree.
- Wikipedia: Treap.
- GeeksforGeeks for BST balancing tutorials.