AVL vs Red-Black Tree Performance Analysis

Student Names and IDs:

- Ahmed Ehab (ID: 22010494)
- Samaa Ibrahim (ID: 22010820)
- Sama Yosri (ID: 22010819)

Introduction

This report presents a comprehensive analysis of two self-balancing binary search tree implementations: AVL Trees and Red-Black Trees. The analysis focuses on the time complexity of key operations (insertion, deletion, and search) and the tree height at different input sizes. The performance metrics were collected through rigorous testing with varying tree sizes ranging from 100 to 100,000 nodes.

Implementation Overview

Both tree implementations follow the `SelfBalancedBTS<T>` interface and include the following operations:

- Insertion of single elements and batches
- Deletion of single elements and batches
- Search operations
- Tree height calculation
- In-order traversal

Performance Analysis

1. Insertion Time

Size	AVL (ms)	Red-Black (ms)	Ratio (AVL/RB)
100	0.0038	0.0032	1.19
200	0.0074	0.0098	0.76
500	0.0233	0.0123	1.89
1,000	0.0298	0.0152	1.96
10,000	0.4428	0.2174	2.04
100,000	8.4420	4.3415	1.94

Observations:

- Red-Black Trees consistently outperform AVL Trees for insertions as the tree size increases
- At size 100,000, Red-Black Trees are approximately twice as fast as AVL Trees
- The performance gap widens more significantly as the dataset grows larger
- The only anomaly occurs at size 200, where AVL performs better than Red-Black

2. Deletion Time

Size	AVL (ms)	Red-Black (ms)	Ratio (AVL/RB)
100	0.0031	0.0042	0.74
200	0.0053	0.0067	0.79
500	0.0177	0.0100	1.77
1,000	0.0318	0.0174	1.83
10,000	0.3346	0.2098	1.59
100,000	9.8485	5.5745	1.77

Observations:

- AVL Trees perform better for smaller datasets (100-200 nodes) during deletion
- For medium to large datasets (500+ nodes), Red-Black Trees are significantly faster
- At size 100,000, Red-Black Trees are approximately 1.77 times faster than AVL Trees

3. Search Time

Size	AVL (ms)	Red-Black (ms)	Ratio (AVL/RB)
100	0.0005	0.0004	1.25
200	0.0010	0.0013	0.77
500	0.0046	0.0051	0.90
1,000	0.0108	0.0105	1.03
10,000	0.1819	0.2194	0.83
100,000	5.2779	6.1443	0.86

Observations:

- Search performance is quite comparable between both trees
- AVL Trees generally perform slightly better for search operations in larger datasets
- This advantage can be attributed to AVL's stricter balancing requirements, resulting in shorter paths

4. Tree Height

Size	AVL Height	Red-Black Height	Difference (RB-AVL)
100	8	9	1
200	9	9	0
500	11	11	0
1,000	12	12	0

Size	AVL Height	Red-Black Height	Difference (RB-AVL)
10,000	16	17	1
100,000	20	20	0

Observations:

- Tree heights are remarkably similar between both implementations
- In a few cases, Red-Black Trees have a height one greater than AVL Trees
- This aligns with theoretical expectations: AVL Trees maintain stricter balance (difference ≤ 1 between subtrees)
- Red-Black Trees allow slightly more imbalance but gain performance benefits for modifications

Theoretical Analysis

Time Complexity

Operation	AVL Tree	Red-Black Tree
Search	O(log n)	O(log n)
Insert	O(log n)	O(log n)
Delete	O(log n)	O(log n)

While both trees have the same asymptotic complexity, our empirical results show that:

1. **Insertion & Deletion**: Red-Black Trees are generally faster
    - AVL Trees may require more rotations to maintain balance
    - Red-Black Trees have a maximum of 2 rotations for insertion and 3 for deletion
    - AVL Trees may perform up to O(log n) rotations in worst case
2. **Search**: AVL Trees are slightly more efficient
    - Due to their more strictly balanced nature, they typically maintain shorter paths

Space Complexity

Tree Type	Space Requirements
AVL Tree	Each node requires 1 extra byte for height/balance factor
Red-Black Tree	Each node requires 1 extra bit for color

Red-Black Trees have a slight advantage in memory usage, requiring only a single bit per node for color compared to AVL Trees which typically need a byte to store the height or balance factor.

Implementation Insights

AVL Tree Implementation

The AVL Tree implementation uses the following balancing technique:

- Balance factor (bf) = right height - left height
- Rotations are triggered when |bf| > 1
- Four cases: Left-Left, Left-Right, Right-Right, Right-Left

Key implementation aspects:

- Height information stored in each node
- Balance factor calculated during updates
- Maximum of O(log n) rotations for insertion/deletion

Red-Black Tree Implementation

The Red-Black Tree implementation maintains these properties:

- Every node is either red or black
- The root and NIL leaves are black
- Red nodes can't have red children
- All paths from a node to descendant NIL nodes have the same number of black nodes

Key implementation aspects:

- Color information stored in each node
- NIL leaf nodes simplify boundary conditions
- Maximum of 2 rotations for insertion, 3 for deletion

Conclusion

Based on our empirical analysis, we can make the following recommendations:

1. **For write-heavy applications** (frequent insertions and deletions):
    - **Red-Black Trees** offer better performance
    - Examples: Database indexes with frequent updates, real-time data structures
2. **For read-heavy applications** (frequent searches):
    - **AVL Trees** perform slightly better
    - Examples: Static dictionaries, read-only databases
3. **For balanced workloads**:
    - **Red-Black Trees** provide good overall performance
    - Less overhead for rebalancing operations
    - Comparable search performance to AVL Trees
4. **Memory-constrained environments**:
    - **Red-Black Trees** have a slight advantage in memory usage

In the specific context of our English Dictionary implementation, the choice between AVL and Red-Black depends on the expected usage pattern. If the dictionary will primarily be used for lookups with infrequent updates, an AVL Tree would be preferable. If the dictionary will be frequently modified, a Red-Black Tree would offer better overall performance.

The empirical results confirm theoretical expectations: Red-Black Trees sacrifice a small amount of balance (and thus search performance) for significantly improved modification performance, especially for large datasets.