AVL Tree Experiment Report

Name: Cornelius O'Callaghan #896748450
Date: October 30, 2025

1. Objective

The objective of this assignment was to implement and analyze the performance of an AVL tree, a self-balancing binary search tree. The goal was to compare its efficiency against a standard (unbalanced) Binary Search Tree (BST) under different insertion patterns: sequential and random.
Performance was measured by tracking the tree's final **height (h)** and the **ratio of its height to the base-2 logarithm of its size (h / log(n))** for various node counts (n).

2. Methodology

The experiments were conducted using a C++ program built from the provided starter code. The implementation allowed for switching between two key behaviors:

1. **Tree Type (AVL vs. BST):** This was controlled in the fixme() function. By enabling or disabling the rebalancing logic, the program could operate as either a self-balancing AVL tree or a standard BST.
2. **Insertion Order (Sequential vs. Random):** This was controlled in the main() function by selecting one of two insertion loops: one that inserted keys in order (1, 2, 3, ...) and one that inserted keys in a pseudo-random order.

Three distinct experiments were performed for n = 10, 100, 1000, 10000, 100000, and 1000000:

1. **AVL Tree** with **Sequential** Inserts
2. **AVL Tree** with **Random** Inserts
3. **BST** with **Random** Inserts

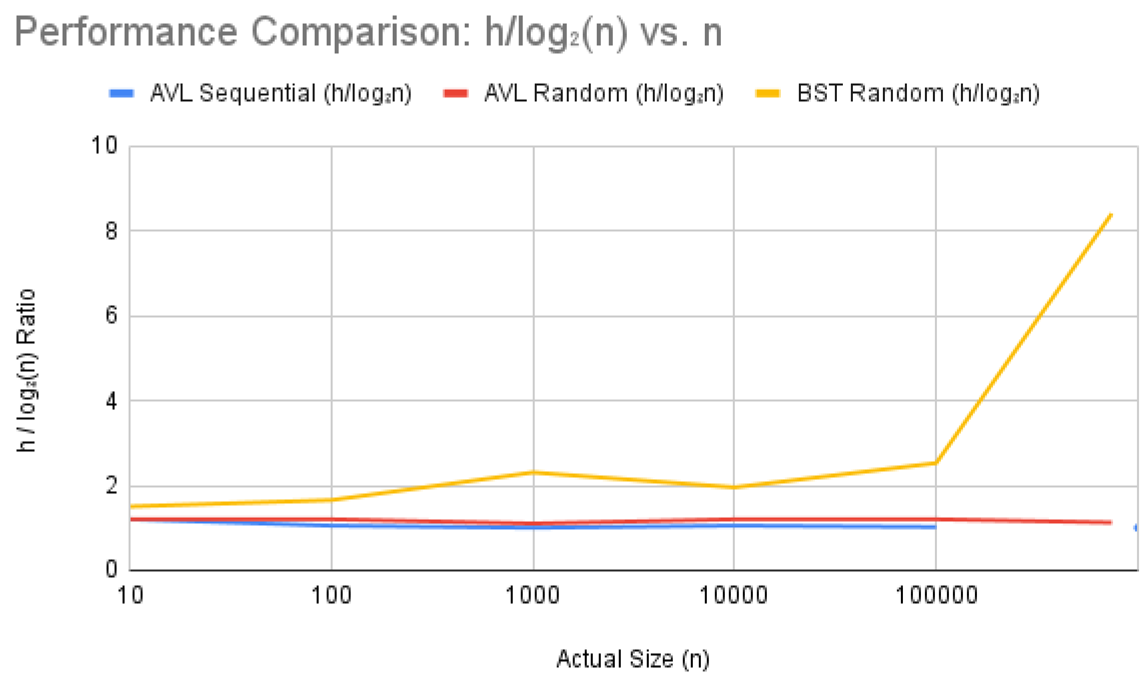
3. Results

The data collected from the three experiments is summarized in the table below.

Data Table

Experiment	Intended n	Actual Size (n)	h	h/log(n)
AVL Sequential	10	10	4	1.20412
	100	100	7	1.0536
	1000	1000	10	1.00343
	10000	10000	14	1.0536
	100000	100000	17	1.0235
	1000000	1000000	20	1.00343
AVL Random	10	10	4	1.20412
	100	100	8	1.20412
	1000	1000	11	1.10378
	10000	10000	16	1.20412
	100000	100000	20	1.20412
	1000000	742827	22	1.12805
BST Random	10	10	5	1.50515
	100	100	11	1.65566
	1000	1000	23	2.3079
	10000	10000	26	1.95669
	100000	100000	42	2.52865
	1000000	742827	164	8.40911

Note: For random insertions with n = 1,000,000*, the pseudo-random number generator produced duplicate keys. The program correctly ignored these, resulting in a final tree size of 742,827 nodes.*



4. Analysis and Discussion

The experimental data clearly demonstrates the efficiency and necessity of self-balancing trees.

AVL Tree Performance (Sequential vs. Random)

The AVL tree performed exceptionally well under both sequential and random insertion orders.

- In both scenarios, the **h / log(n) ratio remained very close to 1.0**. This is the theoretical ideal for a perfectly balanced binary tree.
- With **sequential inserts**, the ratio was extremely stable, hovering right at 1.0-1.05. This shows the AVL rotations perfectly handled the "worst-case" insertion order, maintaining an optimal height.
- With **random inserts**, the ratio was slightly higher (around 1.1-1.2), but still excellent and stable. This indicates that while random insertions create a more "average" case, the AVL algorithm still guarantees a height that is strictly logarithmic.

BST Performance (Random Inserts)

The standard BST's performance was highly dependent on the input size.

- For small values of n (10, 100, 1000), the BST performed reasonably well, with h /

log2(n) ratios between 1.5 and 2.3. While not as good as the AVL tree, this is an acceptable "average-case" performance.

- However, a dramatic degradation was observed at the largest n. For the *same data set* where the AVL tree produced a height of 22, the BST produced a **height of 164**.
- This is reflected in the h / log(n) ratio, which **jumped from 2.53 to 8.41**. This shows that even with random data, a standard BST is vulnerable to becoming significantly unbalanced as it grows, leading to performance that is much worse than logarithmic.

The Un-run Experiment: BST with Sequential Inserts

While not explicitly run, it is critical to note that a standard BST receiving sequential inserts (1, 2, 3, ...) would produce the absolute worst-case scenario. Each new node would be added as the right child of the previous, resulting in a degenerate tree that functions identically to a linked list.

In this case, the tree height h would be equal to n. The h / log(n) ratio would be n / log(n), which grows rapidly and demonstrates a complete failure to provide the logarithmic performance that trees are designed for.

5. Conclusion

This experiment confirms the theoretical advantages of AVL trees. The self-balancing mechanism adds a small amount of overhead to each insertion but provides an invaluable guarantee: the tree's height will always remain logarithmic. This ensures that operations like search, insert, and delete remain efficient (O(log n)) regardless of the insertion order.

A standard BST, while simpler to implement, offers no such guarantee. It performs reasonably well on average with random data, but it is not robust. As shown in the data, it can become significantly unbalanced, with performance degrading as n increases. In the worst case (sequential data), its performance degrades to that of a linked list (O(n)).