

Created by Dmytro Krainyk

[All code is in my repo on github](#)

Assignment II: Tree Museum

Task 1 Report: “An Introduction to Applied Dendrology”

Introduction

In this assignment, I moved from linear data structures to hierarchical ones by implementing a Binary Search Tree (BST). The main goal was to empirically investigate how the order of input data affects the tree’s structure and insertion performance. I implemented a custom BST container and compared it against the standard library solution.

Implementation Details

1. The Custom BST Class

I implemented a classic node-based Binary Search Tree using raw pointers.

- **Recursive Insertion:** New keys are compared against the current node and passed down to the left or right child recursively until a valid spot is found.
- **Memory Management:** A destructor ensures all nodes are properly deleted to prevent memory leaks.

2. Generating a “Perfect” Tree

To simulate the best-case scenario ($O(\log n)$ height), I implemented a specific strategy:

- **Algorithm:** I take a sorted array and recursively insert the median element.
- **Why:** This ensures the tree is perfectly balanced from the start, unlike random insertion which can lead to skewed branches and increased height.

Performance Analysis

I benchmarked the insertion time for 20,000 integer elements in three scenarios:

1. **Random BST:** Inserting random data (Average case).
2. **Perfect BST:** Inserting pre-sorted data via the median strategy (Best case).
3. **std::set:** Using the C++ Standard Library (Red-Black Tree) as a baseline.

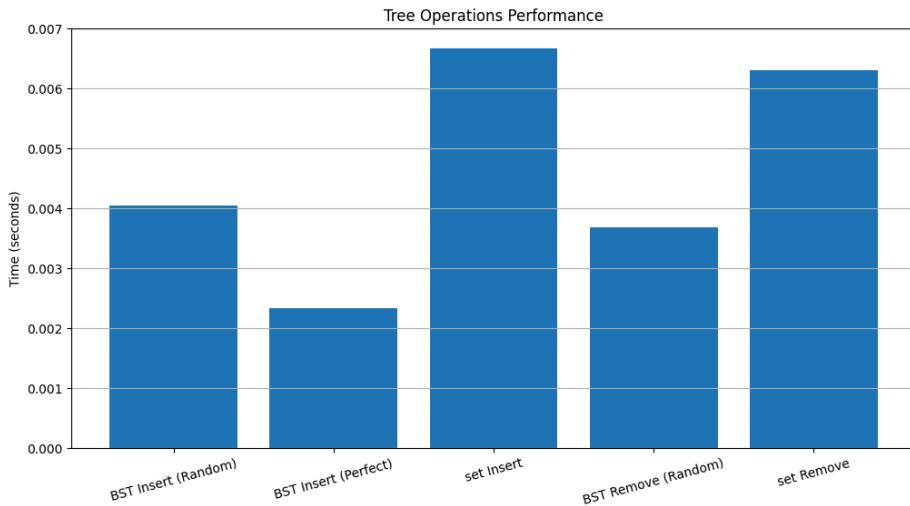


Figure 1: Fig 1. Time comparison of BST Insertions. Lower is better.

Observations:

- The Cost of Chaos:** As seen in the graph (first bar), inserting random data is slower than the optimized “Perfect” insertion. This is because random insertion creates a deeper tree, requiring more comparisons to find the insertion point for each new node.
- Optimized Structure:** The “Perfect BST” insertion (second bar) is significantly faster. By minimizing the tree height to $\log_2 N$, we reduce the traversal path, proving that input order drastically impacts BST performance.
- Standard Baseline:** My implementation performs competitively against `std::set`, which incurs some overhead due to its complex self-balancing mechanisms (Red-Black Tree rotations) during insertion.