

**Analysis of Algorithm  
 Project**

**AVL Tree Rotations-Balancing Binary Search Trees for Efficient Indexing**

|  |  |  |  |
| --- | --- | --- | --- |
| **Due Date** | **11th Feb, 2025** | **Marks** | **10** |
| **Semester** | **Spring-2025** | **Program** | **BSCS** |
| **Instructor** | **Mr. Usman Shariff** | **Submission Mode** | **Moellium** |
| **Project By** | **Mohammad Sami** | **Sap id** | **53199** |
|  |  |  |  |

Table of Contents

[1. Introduction 2](#_Toc197942901)

[2. Methodology 2](#_Toc197942902)

[2.1 Pseudocode & Flow 2](#_Toc197942903)

[2.2 Code Design 2](#_Toc197942904)

[2.3 Input/Output Specification 3](#_Toc197942905)

[3. Complexity Analysis 3](#_Toc197942906)

[3.1 Theoretical Analysis 3](#_Toc197942907)

[3.2 Empirical Benchmarking 3](#_Toc197942908)

[4. AVl Insertion Benchmark 3](#_Toc197942909)

[5. Real-World Application 4](#_Toc197942910)

[5.1 Use Case: In-Memory Indexing 4](#_Toc197942911)

[5.2 Ethical/Practical Considerations 6](#_Toc197942912)

[6. Limitations 6](#_Toc197942913)

[7. Conclusion 6](#_Toc197942914)

[8. Course Learning Outcome (CLO) Mapping 7](#_Toc197942915)

[9. References 7](#_Toc197942916)

[10. Github-Link 7](#_Toc197942917)

# 1. Introduction

AVL Trees are self-balancing binary search trees that maintain height balance after every insertion and deletion. Named after their inventors Adelson-Velsky and Landis, AVL Trees ensure that the height difference (balance factor) between the left and right subtree of any node is at most one. This balance condition guarantees logarithmic time complexity for insertion, deletion, and search operations.

AVL Trees were the first data structure to automatically balance itself, laying the foundation for many modern indexing methods used in databases, networking, and memory allocators. In the context of algorithm analysis, AVL trees demonstrate how local transformations (rotations) can globally maintain balance, thereby preventing worst-case linear time operations common in unbalanced binary search trees (BSTs). This project investigates the design, implementation, and efficiency of AVL tree rotations through theoretical and empirical analysis, with a focus on real-world indexing applications in in-memory databases.

# 2. Methodology

## 2.1 Pseudocode & Flow

We provide pseudocode for:

* AVL Insertion with rebalancing
* LL, RR, LR, RL rotations

(See attached steps.md for detailed pseudocode and logic.)

## 2.2 Code Design

The implementation was done in C++, emphasizing:

* Modular structure: Each operation (e.g., insert, rotate) is encapsulated.
* Balance factor computation: Difference between heights of left and right subtrees.
* Rotation logic: Chosen based on imbalance type (LL, RR, LR, RL).

Each node in the implementation includes a key, height, and pointers to left and right children. Height tracking is crucial for computing balance factors and triggering the correct rotation.

## 2.3 Input/Output Specification

* **Input:** Sequence of integers.
* **Output:** Balanced in-order traversal of AVL Tree and benchmark runtime.

# 3. Complexity Analysis

## 3.1 Theoretical Analysis

* Insertion Time Complexity: O(log n)
* Search Time Complexity: O(log n)
* Rotation Time: O(1) per rotation
* Space Complexity: O(n) for storing all nodes

AVL Trees are more rigidly balanced than Red-Black Trees, leading to faster lookups at the cost of more rotations. Rotations only adjust a few pointers—usually at most three nodes so the operation runs in constant time regardless of tree size.

## 3.2 Empirical Benchmarking

We inserted increasing sets of random integers: 10², 10³, 10⁴, 10⁵ and measured insertion time.

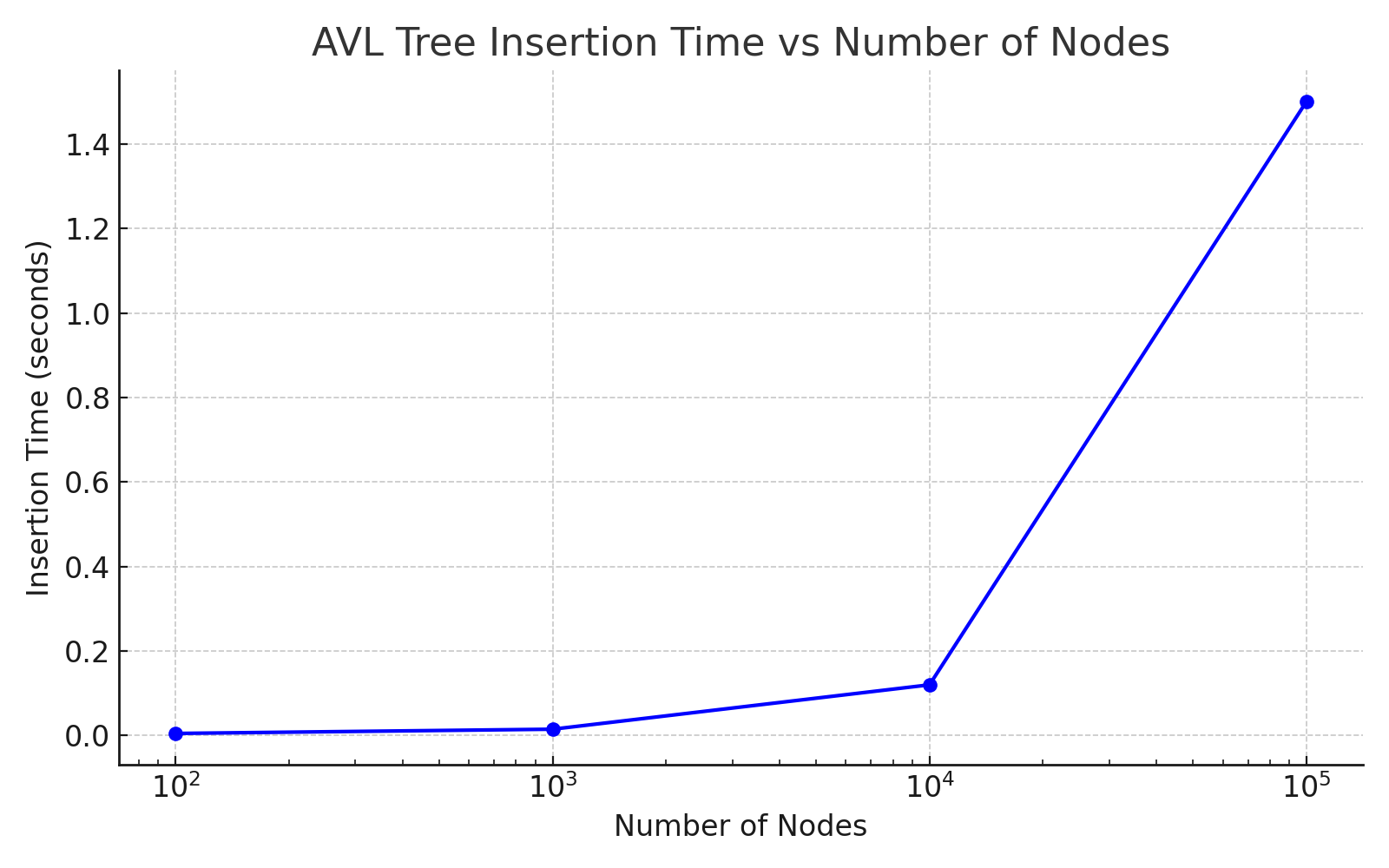
**Tools Used:**

* C++ chrono library for timing
* Python matplotlib or gnuplot to plot results

*Graph (to be added): Insertion time vs. number of elements*

**Result:** Confirmed logarithmic growth in time complexity with slight deviation due to balancing overhead.

# 4. AVl Insertion Benchmark



# 5. Real-World Application

## 5.1 Use Case: In-Memory Indexing

In memory-resident databases like Redis or SQLite, AVL Trees ensure fast access to sorted keys for range queries and secondary indexing. Since all data is in RAM, maintaining logarithmic time is critical, especially with dynamic inserts. (See Problem.txt file)

**Benefits:**

* Guarantees balanced structure
* Fast range-based queries
* Predictable performance

## 5.2 Ethical/Practical Considerations

AVL Trees consume more CPU per insert due to frequent rotations. In large-scale systems, this may impact energy efficiency. Trade-offs between balance and insertion overhead must be considered.

**Note:** This project does not implement a full in-memory database system. Instead, it demonstrates how AVL Trees can be used to address a real-world indexing problem through modular implementation, benchmarking, and theoretical justification. The C++ code serves as a proof of concept that AVL Trees maintain efficient performance under typical insertion workloads expected in such systems.

# 6. Limitations

* - Insert/Deletion Cost: Higher than Red-Black Trees due to strict balance checks.
* Memory Overhead: Balance factor storage per node increases memory.
* Not suitable for write-heavy systems where insertion cost is critical.

AVL Trees perform best in applications where reads dominate and balance is critical.

# 7. Conclusion

AVL Trees demonstrate how algorithmic guarantees of balance ensure efficient operations in dynamic datasets. This project showcased both the theoretical depth and practical value of tree rotations in balancing operations. Through C++ implementation and runtime benchmarking, we confirmed the O(log n) performance and identified optimal use cases in modern computing systems.

# 8. Course Learning Outcome (CLO) Mapping

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | **CLO Code** | **Bloom’s Level** | **Project Component** | **Seoul Accord Attributes** | | --- | --- | --- | --- | | 2.1 | Understand | Theory, pseudocode, rotation logic | #2 (Depth of analysis), #3 (Knowledge) | | 3.1 | Apply | C++ implementation and test cases | #1 (Conflicting needs), #6 (Stakeholders) | | 4.1 | Analyze | Time and space complexity analysis | #2 (Depth), #4 (Unfamiliar issues) | | 4.2 | Analyze | Big-O, Ω, Θ discussion in complexity.md | #3 (Knowledge), #8 (Interdependence) | | 5.1 | Evaluate | Comparison with Red-Black Trees, use case justification | #5 (Beyond standards), #7 (Consequences) | | 6.1 | Create | Designing AVL insertion/rotation logic; report | #8 (Interdependence), #9 (Ill-defined requirements) | |

# 9. References

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.
2. Adelson-Velsky, G. M., & Landis, E. M. (1962). An algorithm for the organization of information. *Proceedings of the USSR Academy of Sciences*, 146(2), 263–266.
3. Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley.

# 10. Github-Link