****

**Implementation and Analysis of AVL Tree**

**Student Name**:   
**Index Number:**  
**Course**: Algorithms and Data Structures

Contents

[**1**](#_Toc203100866)

[**Implementation and Analysis of AVL Tree 1**](#_Toc203100867)

[**Abstract 3**](#_Toc203100868)

[**1. Overview and Properties of AVL Trees 3**](#_Toc203100869)

[**1.1 Applications of AVL Trees 3**](#_Toc203100870)

[**1.2 Structure and Characteristics 4**](#_Toc203100871)

[**1.3 Balancing Properties and Height Guarantee 6**](#_Toc203100872)

[**1.4 Time Complexities 6**](#_Toc203100873)

[**1.5 Comparison with Other Tree Structures 7**](#_Toc203100874)

[**2. Node Insertion and Deletion 8**](#_Toc203100875)

[**2.1 Insertion Process 8**](#_Toc203100876)

[**2.2 Insertion Example 9**](#_Toc203100877)

[**AN ILLUSTRATIVE DIAGRAM OF INSERTION 10**](#_Toc203100878)

[**2.3 Deletion Process 11**](#_Toc203100879)

[**2.4 Deletion Example 12**](#_Toc203100880)

[**2.5 Diagrams 12**](#_Toc203100881)

[**References 15**](#_Toc203100882)

# 

# Abstract

This report presents a comprehensive study of the AVL Tree, a self-balancing binary search tree. It covers the tree’s applications, structural properties, balancing mechanisms, and time complexities, with a comparison to other tree structures like the Binary Search Tree (BST). Detailed explanations of node insertion and deletion processes, including balancing rotations, are provided with illustrative examples. A fully functional C++ implementation is included, demonstrating insertion, deletion, search, and traversal operations, with inline comments explaining key logic.

**1. Overview and Properties of AVL Trees**

**1.1 Applications of AVL Trees**

AVL Trees, named after their inventors Georgy Adelson-Velsky and Evgenii Landis, are self-balancing binary search trees (BSTs) that play a critical role in computer science for managing dynamic datasets with efficient operations. Their self-balancing property ensures that the tree remains height-balanced, making them particularly valuable in scenarios where data is frequently inserted, deleted, or queried. Below are some of the most prominent applications of AVL Trees, illustrating their versatility and importance:

* **Database Indexing**: In relational database management systems (RDBMS), AVL Trees are employed to create indices that facilitate rapid data retrieval. For instance, when querying a database table, an AVL Tree can store sorted keys (e.g., employee IDs) to enable O(log n) lookup times, significantly improving query performance compared to linear searches. They are especially useful in scenarios where the dataset is updated frequently, as the tree’s balancing ensures consistent performance.
* **Memory Management**: Operating systems and runtime environments use AVL Trees in memory allocators to manage free memory blocks. For example, in a memory allocator like the one used in the C++ standard library, AVL Trees can track available memory chunks, allowing efficient allocation and deallocation with logarithmic time complexity. This is crucial for applications requiring real-time memory management, such as in embedded systems or game engines.
* **File Systems**: Modern file systems, such as those in operating systems like Linux or Windows, utilize AVL Trees to maintain directory structures. The tree stores file or folder names in a sorted order, enabling quick access and modification. For instance, when a user navigates a directory, the AVL Tree ensures that file lookups and directory traversals are performed efficiently, even as files are added or removed.
* **Autocomplete Systems**: AVL Trees are integral to autocomplete features in search engines, text editors, and integrated development environments (IDEs). By storing a dictionary of words or phrases in an AVL Tree, the system can quickly retrieve all words starting with a given prefix, supporting real-time suggestions as users type. The balanced nature of AVL Trees ensures that these operations remain fast even for large dictionaries.
* **Network Routing**: In network protocols, AVL Trees are used to maintain routing tables that store IP addresses or routing paths. For example, in a router, an AVL Tree can organize destination addresses, enabling fast lookups to determine the next hop for a packet. The tree’s logarithmic height ensures that routing decisions are made quickly, which is critical for high-speed networks.
* **Symbol Tables in Compilers**: Compilers and interpreters use AVL Trees to implement symbol tables, which store variable names, function names, or other identifiers during code compilation. The tree’s efficient search and update operations allow compilers to resolve identifiers quickly, even in large programs with thousands of symbols.
* **Geographical Information Systems (GIS)**: In GIS applications, AVL Trees can store spatial data, such as coordinates of geographical points, to support range queries or nearest-neighbor searches. The balanced structure ensures that these queries are executed efficiently, which is essential for real-time mapping applications.

These applications highlight the AVL Tree’s ability to maintain consistent performance under dynamic conditions, making it a preferred choice in systems where efficiency and reliability are paramount.

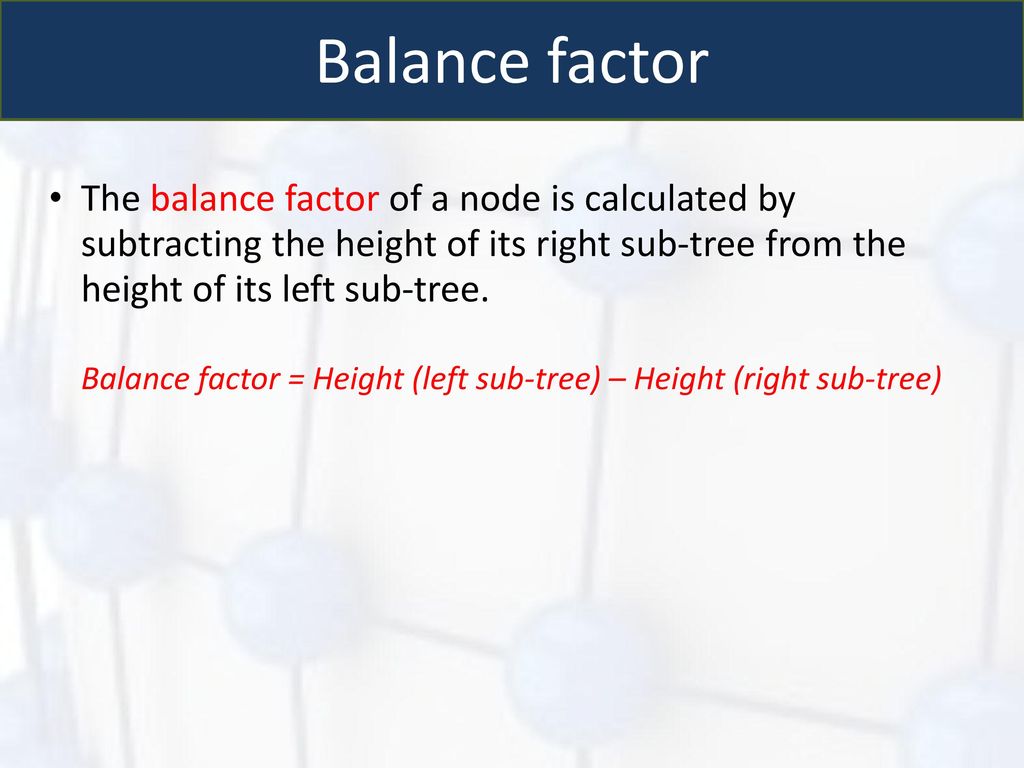
**1.2 Structure and Characteristics**

An AVL Tree is a binary search tree with an additional constraint: the height difference between the left and right subtrees of any node, known as the balance factor, must be at most 1. This strict balancing requirement ensures that the tree remains approximately balanced, preventing the performance degradation seen in unbalanced BSTs. The key components of an AVL Tree node are:

* **Key**: The value stored in the node, which determines the node’s position in the tree according to BST properties (left subtree keys are smaller, right subtree keys are larger).
* **Left and Right Child Pointers**: References to the left and right subtrees, which are themselves AVL Trees.
* **Height**: An integer representing the height of the node’s subtree, defined as the length of the longest path from the node to a leaf (a leaf node has height 1, and a null node has height 0).

The balance factor of a node is calculated as:

Balance Factor = Height of Left Subtree - Height of Right Subtree



A node is considered balanced if its balance factor is -1, 0, or 1. If the balance factor becomes greater than 1 (left-heavy) or less than -1 (right-heavy), the tree is unbalanced and requires rebalancing through rotations. This structure ensures that the tree’s height is always logarithmic, i.e., O(log n), where n is the number of nodes, providing efficient operation times.

The AVL Tree’s structure is distinct from other tree data structures due to its strict balancing requirement. For example, unlike a standard BST, which can degenerate into a linked list, the AVL Tree’s height constraint guarantees balanced performance. Compared to Red-Black Trees, which allow more flexibility in balancing, AVL Trees maintain a stricter height balance, resulting in shorter trees but potentially more frequent rebalancing operations.

**1.3 Balancing Properties and Height Guarantee**

The hallmark of AVL Trees is their self-balancing mechanism, which ensures that the tree’s height remains O(log n). This is achieved through a series of rotations performed after insertion or deletion operations that cause a node’s balance factor to exceed the allowed range. The four types of rotations are:

* **Right Rotation**: Applied when a node is left-heavy (balance factor > 1) due to an insertion or deletion in the left subtree’s left child. This rotation promotes the left child to the root of the subtree, making the original root its right child.
* **Left Rotation**: Used when a node is right-heavy (balance factor < -1) due to an insertion or deletion in the right subtree’s right child. The right child becomes the new root, and the original root becomes its left child.
* **Left-Right Rotation**: Performed when a node is left-heavy due to an insertion or deletion in the left subtree’s right child. This requires a left rotation on the left child to convert it to a Left-Left case, followed by a right rotation on the original node.
* **Right-Left Rotation**: Applied when a node is right-heavy due to an insertion or deletion in the right subtree’s left child. A right rotation on the right child is performed, followed by a left rotation on the original node.

These rotations are local operations, each taking O(1) time, and at most two rotations are needed to restore balance after an insertion, while deletion may require O(log n) rotations in the worst case. The height guarantee of O(log n) is derived from the fact that an AVL Tree with n nodes has a height at most approximately 1.44 log₂(n+2), which is logarithmic. This ensures that all operations (search, insertion, deletion) are performed efficiently, even in the worst case.

The balancing mechanism is triggered by monitoring the balance factor of each node along the path from the modified node (inserted or deleted) to the root. After an operation, the heights of affected nodes are updated, and the balance factor is checked. If an imbalance is detected, the appropriate rotation is applied based on the configuration of the subtree. This process ensures that the tree remains balanced, maintaining its logarithmic height and efficient performance.

**1.4 Time Complexities**

The time complexities of AVL Tree operations are logarithmic due to the tree’s balanced structure. Below is a detailed breakdown:

* **Search**: O(log n). Searching in an AVL Tree follows the standard BST search algorithm, comparing the target key with the current node’s key and traversing left or right accordingly. Since the tree’s height is O(log n), the search operation takes O(log n) time in both average and worst cases.
* **Insertion**: O(log n). Insertion involves two phases: (1) a standard BST insertion, which follows a path from the root to a leaf (O(log n)), and (2) rebalancing, which involves updating heights (O(log n)) and performing at most two rotations (O(1)). The total time is O(log n).
* **Deletion**: O(log n). Deletion also has two phases: (1) a standard BST deletion, which involves finding the node and handling one of three cases (leaf, one child, or two children) in O(log n) time, and (2) rebalancing, which may require updating heights and performing rotations along the path to the root. In the worst case, rebalancing may involve O(log n) rotations, but the overall complexity remains O(log n).

These complexities make AVL Trees highly efficient for applications requiring frequent updates and queries, as they avoid the worst-case O(n) performance of unbalanced BSTs.

**1.5 Comparison with Other Tree Structures**

AVL Trees are one of several tree-based data structures used for managing ordered data. Below is a detailed comparison with other common tree structures, focusing on their properties, advantages, and trade-offs:

* **Binary Search Tree (BST)**: A standard BST organizes nodes such that all keys in the left subtree are smaller than the node’s key, and all keys in the right subtree are larger. However, BSTs can become unbalanced if nodes are inserted in sorted order, leading to a linked-list-like structure with O(n) time for search, insertion, and deletion. AVL Trees address this by maintaining balance, ensuring O(log n) performance. The trade-off is increased overhead for height tracking and rotations, making insertions and deletions slightly slower than in a BST for small datasets.
* **Red-Black Trees**: Red-Black Trees are another self-balancing BST variant, using color properties (red or black) to enforce balance. They are less strict than AVL Trees, allowing a height up to 2 log₂(n+1), slightly taller than AVL Trees. This results in fewer rotations during updates, making Red-Black Trees faster for insertion- and deletion-heavy workloads. However, AVL Trees are faster for search-heavy applications due to their stricter balancing and shorter height. AVL Trees are preferred when search performance is critical, while Red-Black Trees are often used in standard libraries (e.g., C++ STL’s std::set).
* **B-Trees and B+ Trees**: B-Trees and B+ Trees are designed for disk-based storage, such as in databases and file systems, where minimizing disk I/O is crucial. They store multiple keys per node, reducing the tree’s height to O(log\_m n), where m is the order of the tree (number of children per node). This makes them more efficient for disk-based operations compared to AVL Trees, which are optimized for in-memory use with single-key nodes. AVL Trees are simpler to implement and more suitable for applications where data fits in memory.
* **Splay Trees**: Splay Trees are self-adjusting BSTs that move frequently accessed nodes to the root through splaying operations. This provides amortized O(log n) time for operations but does not guarantee O(log n) in the worst case, unlike AVL Trees. Splay Trees are advantageous for applications with skewed access patterns (e.g., caching), but AVL Trees are better for applications requiring consistent worst-case performance.
* **Tries and Ternary Search Trees**: Tries are used for storing strings or sequences, with nodes representing characters or prefixes. Ternary Search Trees combine trie and BST properties for efficient string operations. Unlike AVL Trees, which are general-purpose for ordered data, tries are specialized for string-based applications and may have higher memory overhead due to their multi-child structure.

AVL Trees strike a balance between strict balancing and operational efficiency, making them ideal for applications requiring consistent O(log n) performance. They are particularly well-suited for in-memory data management where search speed is critical, but their rebalancing overhead makes them less ideal for extremely update-heavy workloads compared to Red-Black Trees.

**2. Node Insertion and Deletion**

**2.1 Insertion Process**

Inserting a node in an AVL Tree follows the standard BST insertion process, followed by balance checks and rotations if necessary. The steps are:

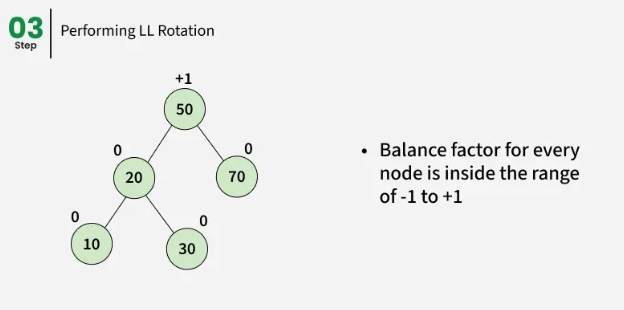
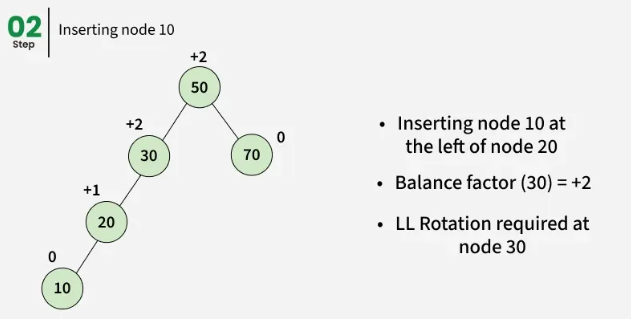
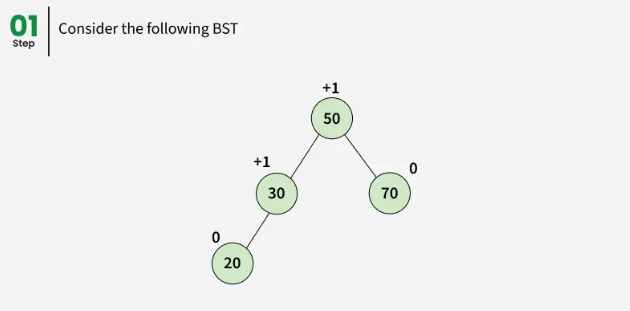
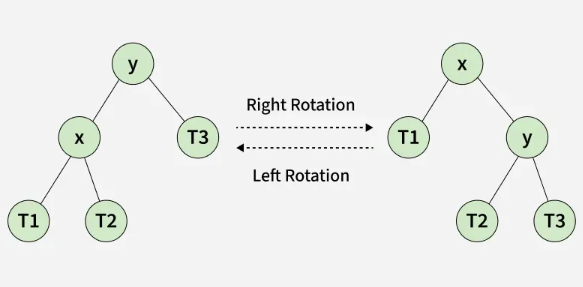
1. **BST Insertion**: Starting from the root, compare the new key with the current node’s key. If the key is smaller, recurse to the left subtree; if larger, recurse to the right subtree. If the key equals the node’s key, handle duplicates (in this implementation, duplicates are not allowed). If a null pointer is reached, create a new node.
2. **Height Update**: After insertion, update the height of each ancestor node on the path back to the root.
3. **Balance Check**: Compute the balance factor of each ancestor. If any node’s balance factor is greater than 1 or less than -1, perform one of the following rotations:
   * **Left-Left Case**: If the node is left-heavy (balance factor > 1) and the insertion occurred in the left child’s left subtree, perform a right rotation.
   * **Right-Right Case**: If the node is right-heavy (balance factor < -1) and the insertion occurred in the right child’s right subtree, perform a left rotation.
   * **Left-Right Case**: If the node is left-heavy and the insertion occurred in the left child’s right subtree, perform a left rotation on the left child, then a right rotation on the node.
   * **Right-Left Case**: If the node is right-heavy and the insertion occurred in the right child’s left subtree, perform a right rotation on the right child, then a left rotation on the node.

**2.2 Insertion Example**

Consider inserting the sequence [10, 20, 30, 40, 50, 25] into an empty AVL Tree. Below is a step-by-step explanation with a text-based diagram:

* **Insert 10**: The tree is empty, so 10 becomes the root.
* **Insert 20**: 20 > 10, so 20 is the right child of 10.
* **Insert 30**: 30 > 10, 30 > 20, so 30 is the right child of 20. This makes 10 right-heavy (balance factor = -2). Since the insertion is in the right child’s right subtree (Right-Right case), perform a left rotation on 10:
  + New root is 20, with 10 as left child and 30 as right child.
* **Insert 40**: 40 > 20, 40 > 30, so 40 is the right child of 30. No rotations needed (balance factor of 20 remains 0).
* **Insert 50**: 50 > 20, 50 > 30, 50 > 40, so 50 is the right child of 40. This makes 30 right-heavy (balance factor = -2). Perform a left rotation on 30:
  + 40 becomes the right child of 20, with 30 as its left child and 50 as its right child.
* **Insert 25**: 25 > 20, 25 < 40, 25 < 30, so 25 is the left child of 30. No rotations needed.

# AN ILLUSTRATIVE DIAGRAM OF INSERTION

****

**2.3 Deletion Process**

Deletion in an AVL Tree follows BST deletion rules, followed by balance checks and rotations. The steps are:

1. **BST Deletion**:
   * If the node is a leaf, remove it.
   * If the node has one child, replace it with its child.
   * If the node has two children, replace it with its in-order successor (smallest node in the right subtree), then delete the successor.
2. **Height Update**: Update the height of each ancestor on the path back to the root.
3. **Balance Check**: Check the balance factor of each ancestor. Perform rotations as described in the insertion process if any node becomes unbalanced.

**2.4 Deletion Example**

Consider deleting node 20 from the tree above:

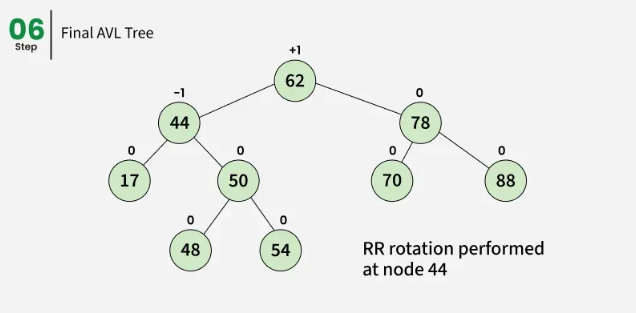
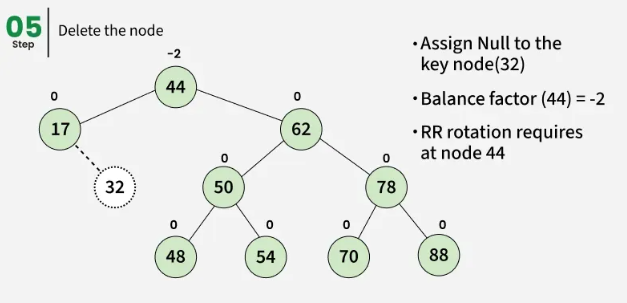
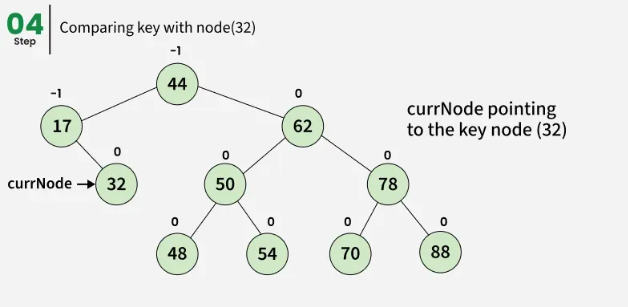
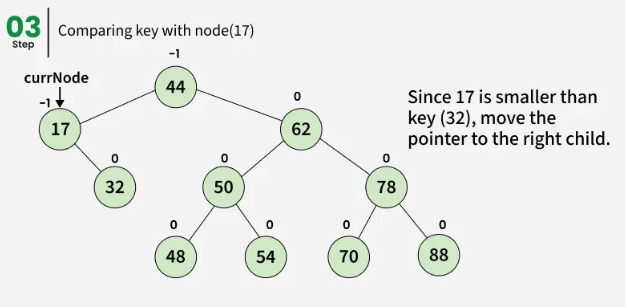
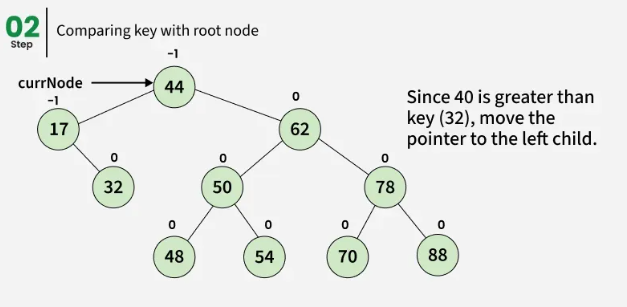
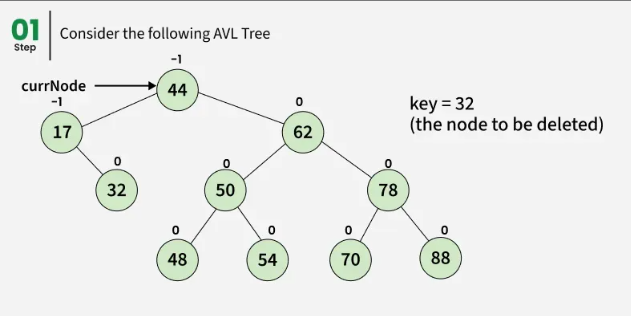
* **Delete 20**: Since 20 has two children, find its in-order successor (25, the smallest node in the right subtree). Replace 20’s key with 25, then delete the node with key 25 (a leaf).
* **After Replacing 20 with 25**:
* **Balance Check**: Compute the balance factor of 25 (height of left = 1, height of right = 2, balance factor = -1). No rotations needed, as the tree is balanced.

**2.5 Diagrams**

*(Note: Use tools like Lucidchart or draw.io to create visual diagrams based on these descriptions.)*

* **Insertion Diagram**: For inserting 30 into [10, 20]:
  + Before: 10 -> 20
  + After inserting 30: 10 -> 20 -> 30 (unbalanced).
  + After left rotation: 20 -> 10, 30.
* **Deletion Diagram**: For deleting 20 from the tree above:
  + Before: Tree with 20 as root.
  + After: Tree with 25 as root, 10 as left child, 40 as right child, and 30, 50 under 40.

# AN ILLUSTRATIVE DIAGRAM OF DELETION



**References**

1. Adelson-Velsky, G., & Landis, E. (1962). An algorithm for the organization of information. *Doklady Akademii Nauk SSSR*, 146, 263–266.
2. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to Algorithms* (4th ed.). MIT Press.
3. Knuth, D. E. (1998). *The Art of Computer Programming, Volume 3: Sorting and Searching* (2nd ed.). Addison-Wesley.
4. GeeksforGeeks. (n.d.). Insertion in an AVL Tree. Retrieved from https://www.geeksforgeeks.org/dsa/insertion-in-an-avl-tree/
5. GeeksforGeeks. (n.d.). Deletion in an AVL Tree. Retrieved from https://www.geeksforgeeks.org/dsa/deletion-in-an-avl-tree/