Project Title: Digital Forensics File Reconstruction System

Course Name: Data Structures and Algorithms

Course Code: CS-221

Group Details:

SR.	Name	Registration No.
1	Irab Zara	2024244
2	Mirza Khizar Baig	2024294

Due Date: Sunday, 7th December 2025

Instructor Name: Dr. Zubair Ahmad

1. Executive Summary

The Digital Forensic File Reconstruction System (FRS) is built to handle file metadata efficiently. It uses two main data structures: an AVL Tree for active files and a pure Linked List Stack for recovery history. The AVL Tree keeps file searches and deletions fast with O(log N) performance. The Recovery Stack allows files to be restored immediately in O(1) time following the LIFO approach. This project successfully produced a final version that is optimized, stable, and meets all design requirements using these structures.

2. Introduction and Project Objectives

The project simulates how modern operating systems handle file deletion and recovery. When a file is deleted the system removes only its metadata reference (the pointer). The actual data stays on disk and can still be reconstructed. This final deliverable evaluates the final system architecture and demonstrates that it meets the desired performance goals.

2.1 Project Aims

The FRS was built with the following goals in mind:

- **Performance:** Uses an AVL Tree to ensure search and deletion run in O(log N) time, replacing the slower O(N) behavior of the previously used temporary Linked List.
- **Architecture:** Encapsulate all data structures inside the FileManager class, hence following proper Object-Oriented Programming design.
- **Recovery:** Implement a Stack structure to enforce the LIFO recovery model which has O(1) performance.

2.2 System Architecture Overview

The FileManager class acts as the core “controller”, we may say, of the system. It handles user input and manages interactions between the AVL Tree and the Recovery Stack. By keeping the underlying data structures hidden the system stays neat, easy to manage, and reliable.

3. Data Structures Implementation

The final design is built around two structures: an AVL Tree to manage active files and a Linked List Stack to store deleted file records.

3.1 Active File Table: AVL Tree

We chose an AVL Tree to implement the Active File Table because it guarantees O(log N) time complexity for insertion, search, and deletion. Unlike a basic Binary Search Tree which may get skewed in one direction, the AVL Tree does not degrade into a linear structure as

data grows. This ensures that file access is both fast and reliable. Each file is indexed and sorted by its File ID, which we chose as the primary key. Any other unique metadata attribute could have been used as well.

3.1.1 Insertion and Balancing Logic

Insertion follows the standard recursive Binary Search Tree process. Once a node is inserted, the balance factor is checked as the recursion happens. The balance factor is calculated as the height difference between the left and right subtrees. If this difference becomes greater than 1 or less than -1, the system immediately applies a rotation to restore balance.

3.1.2 Rotation Implementation

All four AVL rotation cases (LL, RR, LR, RL) are handled through two core functions: rotateLeft() and rotateRight().
Single rotations (LL and RR) are resolved using one direct call.
Double rotations (LR and RL) apply two steps. For instance, an LR case first performs rotateLeft() on the child and then rotateRight() on the parent.

3.2 Recovery History: Linked List Stack

The Recovery Stack stores metadata for all logically deleted files. It follows the Last-In, First-Out (LIFO) principle; this directly supports the system's undo and restore functionality. The reason we use a stack is more times than not, we want to restore the most recently deleted record and a Stack works perfectly in this context.

3.2.1 Pure Linked List Design

The stack is implemented as a singly linked list using a single pointer, node* top. We modified the program to avoid built-in C++ containers such as std::vector or std::stack to comply with the requirement of pure DSA implementation.

3.2.2 Constant Time Operations

Both core operations, push() for deletion and pop() for recovery modify only the head pointer. Because no traversal is required, each operation runs in constant time, O(1). This makes the recovery process fast and fault-resistant.

4. Performance Analysis and Proofs

The combined AVL Tree and Linked List Stack deliver guaranteed high performance for all important system functions.

4.1 Time Complexity Proof Table

Operation	Structure Used	Complexity	Justification
File Search	AVL Tree	O(log(n))	The AVL property prevents height imbalance. This guarantees the traversal depth is always logarithmic.
Logical Delete	AVL Tree / Stack	O(log(n))	The search time to **locate** the file in the AVL Tree determines complexity. The O(1) stack push operation does not change this outcome.
File Recovery (Undo)	Linked List Stack	O(1)	The pop() operation requires a constant number of pointer updates at the head. It's instantaneous.
File Listing (Sorted)	AVL Tree	O(n)	The system must visit every one of the n nodes to perform an in-order traversal and print the file list in ascending ID order.

5. Challenges and Solutions

We faced certain critical challenges while finalizing system compliance and performance.

5.1 Constraint Violation in Stack Implementation

Challenge: The initial Deliverable 2 approach used the std::vector internally for the RecoveryStack. While it achieved O(1) complexity, it violated the project constraint. The project requires using raw pointers to implement the core data structures.

Solution: We redesigned the RecoveryStack as a plain singly linked list. The push() and pop() methods are implemented with manual memory management using new and delete updating pointers directly.

5.2 Complexity of AVL Rotation Logic

Challenge: Correctly implementing the AVL rotations involved intricate pointer re-parenting and height updating. Errors in this complex logic could easily corrupt the tree.

Solution: We reduced the risk by breaking the four rotation cases into two base functions: rotateLeft() and rotateRight(). This made the double rotations easier to follow and verify, ensuring the tree stayed balanced right after each insertion.

6. Conclusion and Future Development

The Digital Forensic File Reconstruction System successfully meets all objectives. We established a high-performance system that provides logarithmic-time file search and constant-time file recovery. The transition from the initial Linked List to the AVL Tree, and the structural correction of the Stack, delivers a robust system that fully meets all performance goals and core DSA application requirements.

6.1 Future Work

To evolve the DF FRS from an academic project to a forensically sound and actual recovery tool, we recommend implementing the following innovative features:

- **Permanent Deletion Algorithm:** Presently, deletion is logical only. A future version should implement full AVL deletion to permanently remove file entries from the Active File Table while preserving tree balance and complexity.

- **Persistence Layer:** Allow the program to remember files by saving data to storage and restoring it when reopened.

- **Forensic Integrity Layer:** Include cryptographic hashing like SHA-256, in the file metadata. When a file is recovered, the system recomputes and checks its hash to make sure the data hasn’t been tampered with. This keeps recovered files reliable and suitable as authentic forensic evidence.

- **Weighted Recovery Strategy:** Swap the simple LIFO recovery stack for a Priority Queue using a heap. This lets files be recovered based on priority instead of just the order they were deleted, allowing smarter, more flexible retrieval.

- **Graphical User Interface (GUI):** Replacing the command-line interface with a GUI would improve usability. It would also allow visualisation of the AVL Tree structure and make the recovery process easier to understand.

7. Application of Class Topics

The final FRS architecture is a practical demonstration of the key Data Structures and Algorithms principles we learned in the course.

Class Topic	Application in FRS
Trees (AVL)	The Active File Table uses an AVL Tree to apply automatic self-balancing. This ensures O(log N) complexity even in the worst case for file search and insertion.
Linked Lists	The Recovery Stack is built as a pure singly linked list. This design highlights direct pointer handling and manual memory management while allowing push() and pop() to run in constant time.
Stacks	The Stack's LIFO behavior directly supports the undo and recovery feature. This simple design allows files to be restored in constant time.
Recursion	The AVL Tree insertion process is built on recursion. It is used to traverse the tree and as each recursive call returns, the system checks the balance factor and performs rotations when needed.
Time Complexity	The entire system design was driven by complexity analysis, specifically optimizing the most frequent operations to achieve O(log(n)) for searching and O(1) for recovery.
Object-Oriented Programming (OOP)	The project is built around the **FileManager** class, which encapsulates the AVLTree and RecoveryStack objects. This demonstrates key OOP concepts like encapsulation and modular design.