**CS2001: Data Structures**
Semester Project

---

**Project groups:** This project can be done within a group of three (3) students. There is no restriction on the selection of group members. Students are allowed to make groups according to their preferences. However, the group members must belong to the same data structure section.
**Submission:** All submissions MUST be uploaded on Google Classroom. Solutions sent to the emails will not be graded. To avoid last minute problems (unavailability of Google Classroom, load shedding, Internet down etc.), you are strongly advised to start working on the project from day one.
Combine all your work (solution folder) in one .zip file after performing "Clean Solution". Submit zip file on Google Classroom within given deadline. If only .cpp file is submitted, it will not be considered for evaluation.
**Plagiarism: -100% marks** in the project if any significant part of project is found plagiarized. A code is considered plagiarized if **more than 20%** code is not your own work.
**Deadline:** Deadline to submit project is 11th May 2025, 11:59 PM. No submission will be considered for grading outside Google Classroom or after deadline. Correct and timely submission of project is the responsibility of every group; hence no relaxation will be given to anyone.
Note: Only one Group member will submit on GCR.

---

# Google Drive File System

## Introduction

The Google Drive File System is a console-based application designed to simulate a robust file management system, leveraging fundamental data structures to ensure optimized storage, retrieval, and recovery mechanisms. This project manual provides a detailed outline of the system's architecture, functionalities, and implementation guidelines.

---

## Data Structures Used

### 1.1 Tree (Folder Structure)

A tree data structure represents the hierarchical organization of directories and subdirectories. Each folder is a node, and subfolders/files are children, enabling depth-first and breadth-first traversal. This structure facilitates efficient navigation and management of files and folders.

---

### 1.2 Hash Table (File Metadata & Fast Lookup)

A hash table stores file metadata (size, type, date, owner) for rapid access, allowing quick file indexing and retrieval with O(1) complexity. This data structure prevents duplication and supports hashing-based conflict resolution.

---

### 1.3 Stack (Recently Deleted Files - Recycle Bin)

A stack implements LIFO (Last-In-First-Out) storage for deleted files, enabling efficient restoration of the most recently deleted files. This mechanism is used for implementing the Recycle Bin feature.

---

### 1.4 Queue (Recent Files & Access History)

A queue stores recently accessed files in a FIFO (First-In-First-Out) manner, implementing a Least Recently Used (LRU) cache for optimized access.

---

### 1.5 Graph (File Sharing & User Connections)

A graph represents users as nodes and shared file relationships as edges, implementing an adjacency list/matrix for efficient file-sharing operations.

---

### 1.6 Linked List (File Versioning & Updates)

A doubly linked list maintains a version history for each file, enabling rollback to previous versions and efficient tracking of changes.

---

# Functionalities by Level

## 2.1 Basic Functionalities

**Folder & File Management (Tree + Hash Table):**

Users can create, delete, and navigate through folders. Each file is indexed using hashing for efficient retrieval.

**File Operations (CRUD System):**

Create File: Initializes file with metadata.

Read File: Displays the content stored in a file.

Update File: Modifies the contents and stores version history.

Delete File: Moves file to Recycle Bin (stack).

**Recycle Bin (Stack-based Recovery System):**

Allows retrieval of the most recently deleted file. Implements time-based auto-deletion of old entries.

**Recent Files (Queue-based Access Tracking):**

Tracks the last accessed files and prioritizes frequently used files. The oldest file (least recently used) is removed when the queue reaches its capacity, ensuring that the list only contains the most relevant files.

**User Authentication (Graph for Login/sign in /signup/logout):**

Securely manages user credentials with passwords.

Incase of forgotten password security question must be save for every user.

Save current time and date of logout of a user.

Each **user is a node**, and **edges store login relationships** (who last logged in, last logout time, etc.).

---

## 2.2 Intermediate Functionalities

**File Search (Tree Traversal & Hashing):**

Uses pre-order/post-order traversal for directory searches. Hash table enables O(1) complexity for file lookups.

**Graph-based File Sharing System:**

Users (nodes) can share files using directed edges. Implements BFS/DFS traversal for file access.

**File Versioning (Linked List):**

Stores multiple versions of a file in a linked list. Allows rollback to a previous state efficiently.

**Error Handling & Validation:**

Prevents duplication, invalid inputs, and unauthorized access. Implements exception handling to prevent system crashes.

---

## 2.3 Advanced Functionalities

**Access Control & Permissions (Graph-Based User Roles):**

Defines read, write, and execute permissions. Implements user groups with hierarchical access control.

Users are **nodes**, and access levels are assigned based on user roles.

- A **directed edge** between users means permission is granted.
- **Hierarchical control** allows admins to modify access.

**Example:**

- Admin → Can read, write, execute files.
- Editor → Can read and write but not execute.
- Viewer → Can only read files.

**Compression Algorithm (Advanced Storage Optimization):**

Implements run-length encoding (RLE) or dictionary-based compression. Reduces storage requirements for large files.

The system reduces file size using **Run-Length Encoding (RLE)** or **Dictionary-Based Compression**.

- **What is RLE?**
  - If a file contains AAAAABBBCC, it is stored as 5A3B2C to save space.
- **What is Dictionary-Based Compression?**
  - Common words/symbols are replaced with **shorter codes** to reduce file size

**Cloud Synchronization (Queue-based Background Sync):**

Implements background tasks to sync local files with cloud storage. Ensures data consistency between online and offline states.

**Scalability & Optimization:**

Uses balanced tree structures (AVL ) for improved performance. Implements efficient garbage collection for file system optimization.

---

## Best of luck!

**Project Viva Guidelines**

1. **Self-Evaluation Sheet:**
   Each group must bring a completed self-evaluation sheet listing their implemented functional requirements. This sheet will serve as a reference for the evaluation process.

2. **Evaluation Process:**
   The Lab Instructor will review the self-evaluation sheet, verify the claimed functionalities, assign marks, and highlight the implemented features.
   The group will perform live testing to showcase the accuracy and reliability of their implementation.

3. **Expectations:**
   Groups should be well-prepared and have all necessary materials, including the project and its documentation.
   Functionalities should be tested thoroughly beforehand to ensure a smooth demonstration.