CL-2001
Data Structures

Project (Maximum 3/2 Group Members)

Note: Carefully read the following instructions (*Each instruction contains a weightage*)

- Use understandable name of variables.
- Write a code in C++ language.
- **Submit .cpp files.**
- Code the problem statement on MS Visual Studio C++ compiler, It is a console based project. No graphics are included.
- Please submit a zip file in this format **Group members rollno_DSLabProject.**
- Do not submit your project after deadline. Late and email submission is not accepted.
- Do not copy code from any source otherwise you will be penalized with **ZERO** marks in the Project

Semester Project: GitLite: A Simplified Version Control System

Subject: Data Structures Lab

Instructor: Maham Saleem

Project Overview

In collaborative environments, especially when working with large files, the process of resharing entire files after each change can be both time-consuming and resource-intensive. For example, if you and your team are working on a 10 GB file, every change would require copying the entire file from one system to another, which wastes bandwidth and storage. Additionally, if data gets corrupted, the entire database could be at risk. This project aims to address these issues by building a Git-like repository system that uses tree structures to efficiently handle versioning and data synchronization. The system's core functionality will include setting up multiple servers (e.g., server1, server2, etc.) to manage the file versions and synchronize changes. The number of servers can be dynamically adjusted based on user requirements, allowing flexibility to add or share servers as needed. The primary objective is to create a secure, efficient, and cost-effective version control system that ensures data integrity, allows easy collaboration, and reduces redundancy in data transfers. Through this project, you will gain a deeper understanding of key concepts such as versioning, hashing, tree structures, and branching in the context of a simplified version control system.

What is Git?

Git is a distributed version control system that helps software teams manage source code. It enables multiple people to work on a project simultaneously without interfering with each other's changes. Git keeps track of all modifications to the codebase and maintains a history of changes. This makes it easy to navigate, revert to previous versions, and merge updates from different contributors.

Key Features of Git:

1. **Versioning**: Maintains a complete history of every change made to the project
2. **Branching**: Supports creating separate lines of work for different features or experiments
3. **Merging**: Allows integrating changes from different branches
4. **Data Integrity**: Uses checksums (hashes) to ensure data hasn't been tampered with
5. **Collaborative Work**: Multiple developers can work simultaneously without conflicts
6. **Purpose of this Project :** This project aims to build a simplified version control system inspired by Git using tree methods in C++. The focus is on creating an efficient, cost-effective, and secure system to manage and transfer data across servers. The goal is to reduce redundancy, enhance data integrity, and facilitate easy collaboration, especially when handling large files.

KEY FUNCTIONALITIES

1. USER AUTHENTICATION

SIGN UP (SIGNUP COMMAND)

Command: signup <username> <password> <email>

What it Does: Creates a new user account in the system.

Process:

- Validates username uniqueness using hash table lookup
- Hashes password using the Instructor Hash method for security
- Stores user credentials in a hash table (username as key)

Data Structures Used:

- Hash table for user storage and O(1) lookup
- Linked list for maintaining user registration order

LOGIN (LOGIN COMMAND)

Command: login <username> <password>

What it Does: Authenticates user and starts a session.

Process:

- Looks up username in hash table
- Hashes entered password and compares with stored hash
- Maintains login state using a stack (for session history)

LOGOUT (LOGOUT COMMAND)

Command: logout

What it Does: Ends the current user session.

CHANGE PASSWORD (CHANGE-PASSWORD COMMAND)

Command: change-password <old_password> <new_password>

What it Does: Updates user password.

2. INITIALIZE REPOSITORY (INIT COMMAND)

Command: init <filename>

What it Does: Initializes the repository and prompts the user to choose a data structure to store CSV data.

Process:

1. **Upload CSV**: The system uploads a CSV file into the system
2. **Tree Structure Selection**: User chooses between AVL Tree or BST
3. **Column Selection**: System displays column names from the CSV file, user selects which column to build the tree on
4. **File Storage Mechanism**:
    - Each tree node is stored in a separate file
    - File name is based on the value of the selected column
    - Each file contains the node's data and references to its parent and child nodes
    - This ensures only necessary data is loaded into memory

3. FILE AND FOLDER MANAGEMENT

ADD FILE (ADD-FILE COMMAND)

Command: add-file <filepath> <filename>

What it Does: Adds a new file to the repository tracking system.

Process:

- Reads file content from specified path
- Stores file metadata in BST/AVL tree (filename as key)
- Creates file entry with metadata: name, size, hash, timestamp
- Marks file as "staged" for next commit
- Uses queue to manage multiple file additions

DELETE FILE (DELETE-FILE COMMAND)

Command: delete-file <filename>

What it Does: Removes a file from repository tracking.

CREATE FOLDER (CREATE-FOLDER COMMAND)

Command: create-folder <foldername>

What it Does: Creates a new folder/directory in the repository.

Process:

- Creates folder node in tree structure
- Establishes parent-child relationship using tree hierarchy
- Creates physical directory in file system
- Initializes empty file list for the folder

DELETE FOLDER (DELETE-FOLDER COMMAND)

Command: delete-folder <foldername>

What it Does: Removes a folder and its contents.

Process:

- Uses DFS to traverse folder tree and find all nested files
- Recursively deletes all contents
- Removes folder node from tree
- Deletes physical directory

LIST FILES (LS COMMAND)

Command: ls <foldername> (optional folder parameter)

What it Does: Lists all files and folders in current or specified directory.

MOVE FILE/FOLDER (MOVE COMMAND)

Command: move <source> <destination>

What it Does: Moves files or folders to different location.

4. COMPRESS() METHOD (RLE)

Purpose:
To reduce file size by replacing consecutive repeating characters with a count-character pair.

Method Signature:
Compress_RLE(inputData)

Inputs:

- inputData: A string or byte stream to be compressed.

Process:

1. Initialize an empty output string compressed.
2. Traverse the input data sequentially.
3. Count consecutive occurrences of the same character.
4. Append the count followed by the character to the compressed output.
5. Continue until the entire input is processed.

Output:

- compressed: The RLE-encoded representation of the input.

Example:
Input: AAAAABBBCC
Output: 5A3B2C

DECOMPRESS() METHOD (RLE)

Method Signature:
Decompress_RLE(compressedData)

Inputs:

- compressedData: The RLE-encoded data sequence.

Process:

1. Initialize an empty string original.
2. Read numeric values (repeat count) from the input.
3. Read the next character following the numeric value.
4. Append the character repeated count times to original.
5. Repeat until all compressed data is expanded.

Output:

- original: The fully restored uncompressed data.

Example:
Input: 5A3B2C
Output: AAAAABBBCC

5. COMMIT CHANGES (COMMIT COMMAND)

Command: commit "message"

What it Does: Commits the changes to the chosen tree structure, creating a version with a message and timestamp.

Process:

- Generates a unique commit ID using hashing
- Stores commit metadata (message, timestamp, hash)
- Updates the commit history using a linked list structure

6. BRANCHING WITH FOLDERS (BRANCH COMMAND)

Command: branch <branch_name>

What it Does: Creates a branch stored in a separate folder.

7. SWITCH BRANCH (CHECKOUT COMMAND)

Command: checkout <branch_name>

What it Does: Switches between branches by navigating to the appropriate folder.

Process:

- Uses graph traversal (DFS/BFS) to locate the target branch
- Loads the tree structure from the selected branch

8. VIEW COMMIT HISTORY (LOG COMMAND)

Command: log

What it Does: Displays commit history for the current branch.

Process:

- Traverses the linked list of commits
- Displays commit ID (hash), message, and timestamp

9. DISPLAY ALL BRANCHES (BRANCHES COMMAND)

Command: branches

What it Does: Lists all branches with their names.

Implementation:

- Uses graph structure to traverse all branch nodes
- Displays branch names with indicators for current branch

10. DELETE A BRANCH (DELETE-BRANCH COMMAND)

Command: delete-branch <branch_name>

What it Does: Deletes an existing branch.

11. MERGE BRANCHES (MERGE COMMAND)

Command: merge <source_branch> <target_branch>

What it Does: Merges changes from source branch into target branch.

Process:

- Identifies differences and integrates changes
- Creates a new commit for the merge

12. DISPLAY CURRENT BRANCH (CURRENT-BRANCH COMMAND)

Command: current-branch

What it Does: Shows the name of the current active branch.

13. ADD NODE (ADD COMMAND)

Command: add <key> <value>

What it Does: Adds a new node to the tree structure.

14. DELETE NODE (DELETE COMMAND)

Command: delete <key>

What it Does: Removes a node from the tree structure.

15. UPDATE NODE (UPDATE COMMAND)

Command: update <key> <new value>

What it Does: Updates the value of an existing node.

16. SEARCH NODE (SEARCH COMMAND)

Command: search <key>

What it Does: Searches for a node in the tree.

Hash Generation Instructions

This program generates a hash for values in a selected column using one of two methods chosen by the user at the start:

1. Instructor Hash

- For Integers: Multiply all digits of the number and take the result modulo 29. Example: For 1523, Hash = (1 × 5 × 2 × 3) % 29.
- For Strings: Multiply ASCII values of all characters and take the result modulo 29. Example: For "Owais", Hash = (ASCII(O) × ASCII(w) × ASCII(a) × ASCII(i) × ASCII(s)) % 29.

2. SHA-256:

- Uses the built-in SHA-256 hashing function to generate a secure 64-character hexadecimal hash.

Project Viva Guidelines

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.

Evaluation Process:

- The Lab Instructor will review the self-evaluation sheet, verify the claimed functionalities, assign marks, and highlight the implemented features.
- The Course Instructor will then ask the group to demonstrate the highlighted functionalities.
- The group will perform live testing to showcase the accuracy and reliability of their implementation.

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.

Best of Luck