NAME: ALI FAROOQ,

MUHAMMAD ALI REHMAN

RED NO. 2023094, 2023372

COUSRE CODE: ES-221

COURSE NAME:

DATA STRUCTURE AND ALGORITHM

Initial Project Report: GIT-Lite represents a simplified version control system built from Git principles.

Project Title: The Data Structure Design and Implementation Phase of GIT-Lite is the initial focus of this report.

Team Members:

ALI FAROOQ 2023094

MUHAMMAD ALI REHMAN 2023372

Project Overview:

In collaborative environments, especially in cases involving large files, the user-ID hearsay of needing to re-share entire files after doing so once or twice gets substantial each modification takes both time and resources. For instance, if you're working with your team. If you have a 10 GBs file, every change will have to copy the whole file from one box to the other, which is wasted bandwidth and storage. Furthermore, if data becomes corrupted the whole database is compromised. This project aims A solution to these problems by building the Git-like repository system based on tree structures to store the data versioning and data synchronization. The system's main features will consist of configuring several servers (Fourier example, you would have a server1, server2, etc. to manage your file versions and syncing changes. The servers count can be Auto-adjustable based on user needs, allowing ability to add share servers when needed. The main goal is to make a safe, dependable & accurate, efficient and economical best version control system for data. flows, maintains the highest level of integrity, promotes effortless collaboration, and eliminates duplicate data transfers. As part of this project, you will learn better knowledge of concepts such as versioning, hashing, tree structures and branching of a new, stripped-down version control system. Git by default is a source control system that's widely used to track changes in software code, as well as by other digital content. It enables developers to continue to keep track of moves, teamwork in addition to personnel make good on the job.

What is Git?

Git is a distributed version control system which facilitates software teams to manage source code. It enables multiple jobs to create a system that supports having people edit

the same thing at the same time without fighting over changes. Git keeps track of all the changes in codebase and keep history. So, it is easy to navigate, coming back to courses prior. The project will hold multiple versions and consolidate updates from different contributors.

Key Features of Git:

1. Versioning: Git keeps a record of every single change to the project, letting developers to go back and resize to revisit or restore past states.

2. Branching: Git has the concept of branching, essentially that developers can create offshoots (branches) for other features, bugfixes or experiments. This allows for parallel development and better collaboration.

3. Merge and Pull Requests: Git permits branches to merge that, therefore, allow multiple developers to integrate. their patches into the main tree. Pull requests allow you to review, manage and discuss code changes before they are merged.

4. Data Integrity: Git uses checksums (hashes) to prove the data has not changed and hasn't been tampered with during transfer.

5. Collaborative Work: So, multiple developers can work on different aspects of that project in parallel. without interfering with one another. For this reason, Git is perfect for team-based environments.

Purpose of this Project:

This project is about creating a simple version control system similar to Git using tree related C++ methods. The main objective is in setting up a reliable, affordable, and secure process in which to get data managed as well as transferred between servers. Objective is to create less redundancy, more accurate data and make collaboration easier when you do. handling large files.

Key Functionalities:

1. Initialize Repository (init Command)

Command: init <filename>

What it Does: Initializes the repository, and prompts the user for a datastructure (tree) to use for the CSV Data, such as AVL Tree, B Tree, and Red-Black Tree. Please see the following for more information about For the working of Red-Black trees, pages 566 to 576 (i.e., section 12.2) has to be read by the students. as from reference book “Data Structures and Algorithm Analysis in C++” by Mark Allen Weiss (edition 4). Additionally, the book provides C++ code for implementing Red-Black tree, but exact copy will be considered plagiarism.

Process:

1. Upload CSV: The system loads big csv files in the system.

2. Tree Structure Selection: User is asked for after uploading to choose tree structure system prompts the user (AVL, B Tree, or Red-Black Tree) to manage the data (“if B tree, pick the order”).

3. Column Selection: The system prompts the user to set the csv column for which the user wants to select the input. will be run by tree (We can share the type of different files, so you must how the user columns names by reading the first line of file, then user will pick a column).

4. File Storage Mechanism:

- According to the preselected tree structure the system starts the tree.

- Instead of loading the whole tree into RAM, each tree node is stored in its own file.

- The file name will be based on the value of the selected column.

- Each file contains the node's data and references to its parent and child nodes.

2. Commit Changes (commit Command)

Command: commit "message"

What it Does: Saves the changes to the specified tree structure, thus creating a version of the tree with the user's data and a message.

3. Branching with Folders (branch Command)

Command: branch <branch_name> What it Does: Make a branch, which is placed in a new directory. This folder will hold the current copy of the tree and data, and later on users can switch branches. Process:

• The system generates a new branch folder, as new server, you allocated an init.

• The files and tree structure are spat into this dir so users can work alone. each branch.

4. Switch Branch (checkout Command)

Command: checkout <branch_name> What it Does: Liquidates user from branches by changing navigation to the appropriate directory and loading the corresponding tree structure. Process:

• The system loads tree structure of the selected branch and makes it active.

• Changes made in one branch will not affect other branches until they get committed.

5. View Commit History (log Command)

Command: log

What it Does: Displays a log of all commits for the current branch. Each commit shows a unique identifier, message, and timestamp.

How the System Works

1. Initialize:

- The user calls init, uploads a CSV, and wishes to store the data on a tree of type AVL, B, or RB tree.

- The system then stores the information in the chosen tree type.

2. Create Branches:

- The user can create multiple branches of branch <name>. Every branch will be shown as a folder.

- Inside of this directory lies the current state of the repository (tree and data).

3. Switch Between Branches:

- To switch to a different branch, the user runs checkout. The system loads the data from the selected branch’s folder, allowing the user to continue working independently

4. Commit Changes:

- When changes (e.g. adding modifying or removing data) are made the user performs commit "message" to save the new state to working branch.

Advanced Example of branch and checkout Workflow:

1. Create a Repository

> init

• Description: Initialize a new repository.

• Input: Choose the tree type (AVL, B, or Red-Black).

• Example:

> init

Choose tree type (AVL/B/Red-Black): AVL

2. Create a Branch

> branch <branch_name>

• Description: Create a new branch.

• Input: Name of the branch.

• Example:

> branch feature-1

Branch 'feature-1' created successfully.

3. Switch Branch

> checkout <branch_name>

• Description: Switch to an existing branch.

• Input: Name of the branch.

• Example:

> checkout feature-1

Switched to branch 'feature-1'.

4. Make Changes & Commit

> commit "<message>"

• Description: Commit changes with a message.

• Input: Commit message.

• Example:

> commit "Added new feature to branch"

Changes committed with message: "Added new feature to branch".

5. Display All Branches

> branches

• Description: List all branches with their names.

• Example:

> branches

- main

- feature-1

- bugfix-2

6. Delete a Branch

> delete-branch <branch_name>

• Description: Delete an existing branch.

• Input: Name of the branch.

• Example:

> delete-branch feature-1

Branch 'feature-1' deleted successfully.

7. Merge Branches

> merge <source_branch> <target_bbranch>

• Description: Merge the changes from one branch into another.

• Input: Source and target branch names.

• Example:

> merge feature-1 main

Merged 'feature-1' into 'main' successfully.

8. Visualize the Tree Structure (Bonus Activity)

> visualize-tree <branch_name>

• Description: Display a visual representation of the tree for a specific branch.

• Input: Branch name.

• Output: A graphical/textual representation of the tree structure.

• Example:

> visualize-tree main

Visualization of tree for 'main': [10] / \ [5] [15]

9. Display Commit History

> log

• Description: Show a history of all commits in the current branch.

• Example:

> log

Commit History for 'feature-1':

- Commit #3: "Refactored feature implementation."

- Commit #2: "Added new feature to branch."

- Commit #1: "Initialized branch."

10. Display Current Branch

> current-branch

• Description: Show the name of the branch you’re currently on.

• Example:

> current-branch

You are on branch: 'main'.

11. Save Repository to File

> save

• Description: Save the current repository state to a file for persistence.

• Example:

> save

Repository saved successfully to 'repo_data.txt'.

12. Load Repository from File

> load <file_name>

• Description: Load a previously saved repository state.

• Input: File name containing the repository data.

• Example:

> load repo_data.txt

Repository loaded successfully from 'repo_data.txt'.

To clarify, the functionalities that can be performed on the tree, and how they relate to a file system, are as follows:

1. Add a Node (File/Directory):

• When you add a new node to the tree, it represents the addition of a new file or directory.
• This operation will change the structure of the tree, potentially affecting the hierarchy or the order of the tree.

• File System Impact: The new file or directory will be added to the file system, and its placement in the tree should be reflected accordingly.

2. Delete a Node (File/Directory):

• When you delete a node from the tree, you remove an existing file or directory.

• File System Impact:

The corresponding file or directory is removed from the file system, and this change must be reflected in the tree. The removal may affect other parts of the system if there are dependencies or relations between the nodes.

3. Update a Node (File/Directory):

• Updating a node means changing the properties of a file or directory, like renaming it, changing its metadata (e.g., size, permissions), or modifying its contents.

• File System Impact: The file or directory is updated in the file system, and the changes should be reflected in the tree structure.

Reflection Across the System:

After performing any of these operations, the tree structure must be updated to reflect the changes in the file system. This means:

• Consistency: The tree and the file system must always be consistent. If a file is added, deleted, or updated in the tree, those changes must be mirrored in the actual file system, and vice versa.

• Order: The structure and order of the tree may change based on these operations. For example, adding a new node might change the order in which files are listed or accessed.

• Rebalancing (if needed): If the tree has specific properties (e.g., a balanced tree or B-tree), it might need to be rebalanced after additions or deletions to ensure efficient performance.

In short, any operation performed on the tree (add, delete, update) must not only reflect in the tree structure but also in the underlying file system, ensuring synchronization and integrity between both.

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.