National University of Computer and Emerging Science

School of Computing — Fall 2024 — Islamabad Campus

CS2001: Data Structures (Fall 2024)

Semester Project

(Deadline: 8th December 2024, 08:00 AM)

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.

Deadline: Deadline to submit project is **8th December 2024, 08:00 AM**. No submission will be considered for grading outside Google Classroom or after **8th December 2024, 08:00 AM**. Correct and timely submission of project is the responsibility of every group; hence no relaxation will be given to anyone.

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.

GitLite: A Simplified Version Control System

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.

Git is a widely-used version control system designed to manage changes to software code and other digital content. It allows developers to track changes, collaborate on projects, and ensure the integrity of their work. Here's a clear definition:

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.

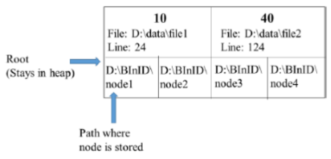
Key Features of Git:

1. **Versioning**: Git maintains a complete history of every change made to the project, allowing developers to go back and review or restore any previous state.
2. **Branching**: Git supports branching, which means developers can create separate lines of work (branches) for different features, bug fixes, or experiments. This enables parallel development and better collaboration.
3. **Merge and Pull Requests**: Git allows for merging branches, enabling multiple developers to integrate their changes into the main codebase. Pull requests make it easy to review and manage code changes before they are merged.
4. **Data Integrity**: Git uses checksums (hashes) to ensure that the data is unchanged and hasn't been tampered with during transfer.
5. **Collaborative Work**: Multiple developers can work on different parts of a project simultaneously without interfering with one another. This makes Git an ideal tool for team environments.

Purpose of this Project
This project aims to build a simplified version control system inspired by Git using tree methods in C++. The main 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. **Initialize Repository (init Command)**
   Command: init <filename>
   What it Does: Initializes the repository and prompts the user to choose a data structure (tree) to store CSV data, such as an **AVL Tree**, **B Tree**, or **Red-Black Tree**. For more detailed information about the working of Red-Black trees, students are recommended to read pages 566 to 576 (i.e., section 12.2.) from the reference book "Data Structures and Algorithm Analysis in C++" by Mark Allen Weiss (edition 4). The book also contains the C++ code for implementing Red-Black tree, however, exact copy will be considered plagiarism.
   Process:
   1. Upload CSV: The system uploads a large CSV file into the system.
   2. Tree Structure Selection: After uploading, the system prompts the user to choose a tree structure (AVL, B Tree, or Red-Black Tree) for organizing the data("if B tree, select the order").
   3. Column Selection: The system asks the user to specify the column of the CSV file on which the tree will be constructed (We can provide you with different types of files, so you must show the user columns names by reading the first line of file, then user will select the column).
   4. File Storage Mechanism:
      - Based on the chosen tree structure, the system initializes the tree.
      - Instead of loading the entire tree into RAM, each tree node is stored in a separate 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.



   This approach ensures that only necessary data is loaded into memory, optimizing performance for large datasets.

2. **Commit Changes (commit Command)**
   Command: commit "message"
   What it Does: Commits the changes to the chosen tree structure, 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: Creates a branch, which is stored in a separate folder. This folder will contain the current version of the tree and data, and users can later switch between branches.
   Process:
   - The system creates a new folder for the branch, like a new server you created by **init**.
   - The files and tree structure are copied into this folder, allowing users to work independently in each branch.

4. **Switch Branch (checkout Command)**
   Command: checkout <branch_name>
   What it Does: Allows the user to switch between branches by navigating to the appropriate folder and loading the corresponding tree structure.
   Process:
   - The system loads the tree structure from the selected branch and makes it active.
   - Changes made in one branch won't affect others until they are 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 the unique identifier, message, and timestamp.

How the System Works

1. **Initialize**:
   - The user runs init, uploads a CSV, and chooses a tree type to store the data (AVL, B, or RB tree).
   - The system then stores the data in the chosen tree structure.
2. **Create Branches**:
   - The user can create multiple branches with branch <name>. Each branch will be represented as a folder.
   - Inside this folder, the current state of the repository (tree and data) is stored.
3. **Switch Between Branches**:
   - To switch to a different branch, the user runs checkout <branch_name>. The system loads the data from the selected branch's folder, allowing the user to continue working independently.
4. **Commit Changes**:
   - After making changes (e.g., adding, modifying, or removing data), the user runs commit "message" to save the new state to the current 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_branch>

- 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'.
  ```

These commands should offer a comprehensive and interactive experience for your Git-like system while covering essential features and some advanced options for enhanced functionality.

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

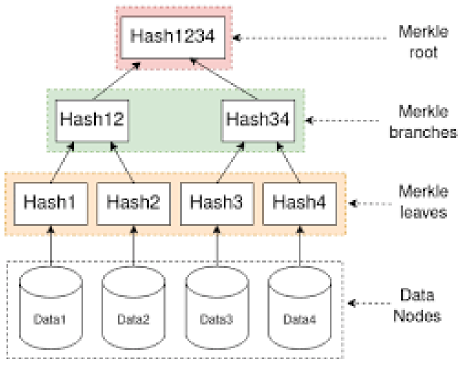
This program generates a hash for values in a selected column using one of two methods chosen by the user at the start:

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.

Choose the desired method at program start.

How It Will Work:

Upon initialization, the system will build a tree structure (AVL, B Tree, or Red-Black Tree, based on the user's choice). Each node in this tree will have a cryptographic hash computed using the node's data. These hashes will then be combined in pairs at higher levels of the tree to generate a new hash for the parent node. This forms a Merkle Tree, where the integrity of the tree can be verified by checking the root hash.

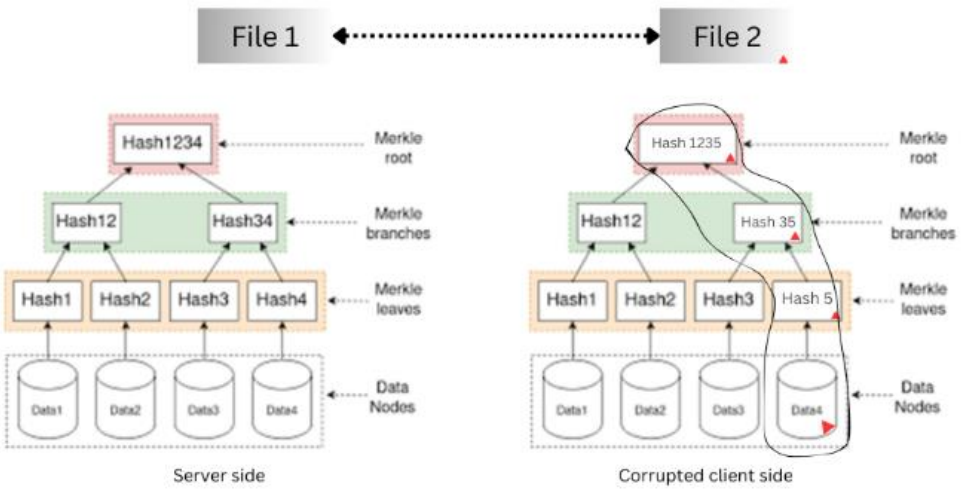


How Data Corruption Will Be Detected:

The Merkle Tree structure allows for easy detection of data corruption. If any node in the tree is altered, its hash changes, which will propagate upwards, causing the root hash to differ from the expected value. By comparing the root hash with a stored reference, the system can quickly identify if any data has been corrupted.

How It Will Transfer Changes Efficiently:

By using Merkle Trees, only the hashes of the affected nodes need to be transmitted during updates. Instead of transferring the entire dataset, only the modified parts (or branches) of the tree, along with their updated hashes, are transmitted. This minimizes the data transfer cost and improves efficiency. The decentralized storage (node data stored in separate files) further optimizes this, as each node is independently accessible, avoiding the need to load the entire tree into memory.



The image above explains the transfer of changes.

When detecting data corruption, Merkle trees provide an efficient solution. Both the server and client start by comparing the root hashes of their respective trees. If the root hashes match, the data is intact; otherwise, it indicates that some part of the file is corrupted.

To pinpoint the corrupted chunk, the comparison proceeds from the root down to the leaf nodes. At each level, the client and server compare their hashes. The first mismatch identifies the branch containing the corrupted data. This process continues until the exact corrupted leaf node is found.

Instead of requesting the entire file from the server which could be extremely large, this method allows the client to request only the corrupted chunk, usually about one-quarter of the file or less, depending on the tree's depth. Once the corrupted chunk is retrieved, the client replaces the faulty part, restoring the file's integrity efficiently.

This approach minimizes data transfer and speeds up the recovery process, making it particularly useful for large files.

Dataset Description

This dataset provides a comprehensive view of healthcare operations, including patient details, medical conditions, financial aspects, and hospital services. It serves as an excellent resource for simulating real-world medical data management and analysis.

Key Features:

1. **Patient Demographics**: Captures essential information such as **Name**, **Age**, **Gender**, and **Blood Type**, allowing analysis of trends based on patient profiles.
2. **Medical Conditions**: Covers a wide range of diagnoses (e.g., Cancer, Diabetes, Obesity), providing opportunities for health outcome analysis.
3. **Hospital Services**:
   - **Date of Admission** and **Discharge Date**: Useful for evaluating average treatment durations.
   - **Doctor** and **Hospital**: Helps study healthcare provider performance.
   - **Admission Type** (Urgent/Elective/Emergency): Highlights patient severity levels.
4. **Billing Information**:
   - **Billing Amount**: Enables financial analysis of treatments.
   - **Insurance Provider**: Useful for understanding patient access to healthcare funding.
5. **Medications and Test Results**: Tracks prescribed treatments (e.g., Aspirin, Ibuprofen) and test outcomes (Normal/Abnormal/Inconclusive).

National University of Computer and Emerging Science

School of Computing Fall 2024 Islamabad Campus

Project Demo Schedule and Guidelines

The project demos will be conducted on 9th and 10th December. The evaluation process will follow a structured format to ensure consistency and fairness:

1. **Setup**:

   All instructors (Lab Instructors and the Course Instructor) will be seated in the same classroom.

2. **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.

3. **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.

4. **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.

Each group must bring a completed **self-evaluation sheet**, which will be provided on **5th December**.

Best of Luck ☺