

Addis Ababa Institute Of Technology

Data Structures And Algorithms

Group Assignment

**Tittle : MiniGit-Project**

**Group Members ID**

1. Ayantu Lemi UGR/3757/16

2. Fiker Mesfin UGR/1820/16

3. Helina Tadesse UGR/2446/16

4. Krutias Tamere UGR/ 2368/16

5. Pomi Yelma UGR/8500/16

6. Tesnim Mahdi UGR/3759/16

Submitted to: Mr. Beimnet Bekele

Submission Date: June 20 , 2025

Table of content

***[Introduction](#_Toc27775)* [3](#_Toc27775)**

**[1. Features Implemented 3](#_Toc26853)**

**[2. Data Structures Used 6](#_Toc20127)**

**[3. Design Decisions 6](#_Toc24867)**

**[4. Limitations 7](#_Toc8499)**

**[5. Future Improvements 7](#_Toc29211)**

***Introduction***

The MiniGit system is a simplified, command-line based version control application implemented in C++. It replicates core Git functionalities such as initializing a repository, staging files, committing changes, creating branches, checking out past versions, and merging branches. The goal of the system is to give users a deeper understanding of how modern version control systems operate internally, particularly through the use of data structures such as Directed Acyclic Graphs (DAGs), linked lists, hash tables, and file-based I/O.

## Features Implemented

**init** – Initialize a MiniGit repository.  
**add** – Stage files for commit.  
**commit -m** – Commit staged changes with a message.  
**log** – View commit history.  
**branch** – Create and manage branches.  
**checkout** – Switch to a commit or branch.  
**merge** – Merge two branches with basic conflict resolution.

The **'init'** command sets up the base repository folder, **'.minigit/**', and prepares necessary object stores and references. The **'add'** command stages files by recording their hashes and locations in a temporary structure before they are committed. The **'commit'** command creates a new snapshot of the current project state, saves meta-data such as time-stamp and message, and links it to the parent commit in a DAG structure. The **'log'** command traverses this structure from HEAD backwards to list commit history. The **'branch'** command creates a new reference pointer that allows divergent paths of development, and the **'checkout'** command reverts or switches the current directory state to match a specified branch or commit hash. Finally, the **'merge'** command performs a three-way merge by identifying the lowest common ancestor of two branches and attempting to combine changes while resolving conflicts.

**1.1 Initialization**

The initialization process is the first and essential step in setting up the MiniGit version control system. During this step, MiniGit creates the necessary directory structure and files to manage the repository’s data effectively.

Specifically, MiniGit creates a hidden folder named .minigit in the current working directory. This directory serves as the root for all MiniGit meta-data and version control information. Inside .minigit, several sub-directories and files are created:

**objects/:** This folder stores all the file contents (called blobs) and commit objects, each identified uniquely by a generated hash. It acts as a storage area for snapshots of files and commits.

**refs/:** This directory holds references to branches. Each branch is represented as a file within refs/heads/, containing the hash of the latest commit on that branch.

**HEAD file:** This file points to the current branch or commit that is checked out. It helps MiniGit track the active state of the repository, whether it is on a specific branch or a detached commit.

This structured setup mimics the design of popular version control systems like Git but on a smaller scale. By organizing data in this manner, MiniGit ensures that it can efficiently manage commits, track changes over time, and support branching and merging operations.

Overall, the initialization step is a one-time setup performed when creating a new MiniGit repository. It lays the foundation required for all subsequent version control actions, such as adding files, committing changes, switching branches, and merging histories.

**1.2 Staging and Committing**

Staging in MiniGit is the process where files are prepared for inclusion in the next commit. When a file is staged, MiniGit reads its content and generates a unique hash using a simple hashing function. This hash serves as an identifier for the file’s current content snapshot. The content of the staged file is then saved inside the .minigit/objects/ directory using this hash as its filename. This approach enables MiniGit to efficiently track changes by storing only unique content snapshots, avoiding redundancy when files have not changed.

When the user commits, MiniGit creates a new commit object — represented by the CommitNode class — which encapsulates several important pieces of information:

* **Commit message**: A descriptive message explaining the changes or purpose of the commit.
* **Timestamp**: The exact time when the commit was created.
* **Parent commit references**: One or more hashes pointing to the immediate previous commit(s), supporting both linear and merge histories.
* **Tracked files map:** A mapping of filenames to their corresponding blob hashes, effectively capturing the project’s entire state at the commit point.

This commit object is saved to the .minigit/objects/ directory, and the HEAD reference is updated to point to this latest commit. This process ensures that the version history is maintained as a series of snapshots, allowing MiniGit to reproduce the exact state of the project at any commit.

By staging and committing, MiniGit mirrors the fundamental behavior of Git, enabling precise version tracking, efficient storage, and clear historical records of the project’s evolution.

**1.3 Log and Branching**

The log functionality in MiniGit provides users with a clear view of the project’s commit history. It traverses backward through the commit graph, starting from the current HEAD, following parent commit references until it reaches the initial commit. Each commit displayed in the log includes essential metadata such as the commit hash, timestamp, and commit message. This chronological display helps users understand the sequence of changes and track the project's evolution over time.

Branching enables the creation of separate development paths within the project. Each branch acts as a named pointer to a specific commit node, allowing multiple lines of development to exist simultaneously without interfering with each other. When a new branch is created, MiniGit saves the current commit hash under the new branch’s name. Switching branches updates the HEAD reference to point to the selected branch’s latest commit, effectively changing the working context. This structure supports isolated development, experimentation, and easy context switching between features or fixes.

**1.4 Checkout**

The checkout command in MiniGit allows users to switch between different branches or specific commits within the project’s history. When a checkout is performed, MiniGit restores the files in the working directory to match the state of the selected commit by reading the stored blob hashes and writing the corresponding file contents. This operation ensures the working directory accurately reflects the chosen snapshot.

Additionally, the checkout operation updates the HEAD pointer to reference the checked-out commit or branch, maintaining consistency in the project state tracking. This functionality is fundamental in version control, as it allows users to safely navigate through different points in the project’s timeline, revert to earlier versions, or switch between feature branches for development or testing.

**1.6 Merging Branches**

Merging in MiniGit replicates a core Git capability by combining divergent development histories into a unified commit. The process begins by identifying the Lowest Common Ancestor (LCA) — the most recent common commit shared by both the current branch and the target branch being merged. This LCA acts as the baseline for comparison.

MiniGit then performs a three-way merge by comparing file states across the LCA, the current branch’s commit, and the target branch’s commit. Files modified only in one branch are incorporated directly into the merge result. However, if the same file is modified differently in both branches, MiniGit flags this as a conflict and inserts conflict markers into the file to highlight areas requiring manual resolution.

After merging all changes, MiniGit creates a new merge commit referencing both parent commits, effectively reconciling the two histories into a single, coherent branch. This process maintains a comprehensive project history while allowing collaborative and parallel development workflows.

## Data Structures Used

- **DAG (Directed Acyclic Graph)**: Represents commit history and relationships.  
- **Linked List**: For linear traversal of commits and history.  
- **Hash Table / Map**: For staged files, commit refs, branches.  
- **Custom Hashing Function**: Simulates SHA-1.

The MiniGit system leverages several fundamental data structures to efficiently manage and represent version control operations. At the core is the **Directed Acyclic Graph (DAG)**, which models the commit history and the relationships between commits. Each commit node points to its parent(s), allowing the system to maintain a precise and non-circular record of changes over time. To facilitate sequential traversal of commits, such as when displaying commit logs, **linked lists** are employed, providing a straightforward method to iterate through the linear history of a branch. **Hash tables or maps** play a crucial role in managing the staging area, branch references, and commit metadata by enabling fast lookups and preventing duplication. Additionally, a **custom hashing function**, designed to simulate the behavior of the SHA-1 algorithm used by Git, uniquely identifies file contents and commits, ensuring integrity and efficient storage within the .minigit directory structure. Together, these data structures enable MiniGit to replicate essential Git functionalities while maintaining performance and accuracy.

We have created a detailed video demo script and storyboard that showcased the MiniGit system in action. The video begins by explaining the purpose of MiniGit, followed by practical demonstrations of each command. The flow of the demo mirrors a typical software development workflow — starting with repository initialization, then making commits, branching out, checking out versions, and finally merging. Conflicts during merges are intentionally introduced to display how the system identifies and notifies the user. The narration also includes technical background about the underlying data structures, such as how DAG traversal enables commit history and how hashing ensures file uniqueness.

## Design Decisions

- Store blobs in .minigit/objects/ using hash.  
- Commit nodes include metadata and parent reference.  
- Each branch points to a commit node.  
- Staging area uses maps to avoid duplicates.

In designing MiniGit, several key architectural choices were made to balance simplicity, functionality, and performance. File contents, or blobs, are stored in the .minigit/objects/ directory and are identified using hashes generated by our custom hashing function. This approach ensures that identical files are stored only once, optimizing storage space. Each commit is represented as a node containing important meta-data such as the commit message, time-stamp, and a reference to its parent commit, thereby enabling the formation of a commit history graph. Branches are implemented as pointers that reference specific commit nodes, allowing for flexible navigation and parallel lines of development. To efficiently manage the staging area, hash maps are employed, which prevent duplicate file entries and enable quick lookup during the commit preparation process. These design decisions collectively help MiniGit achieve core Git-like features in a clean, maintainable manner.

## Limitations

While MiniGit successfully implements many fundamental Git features, it does have certain limitations due to its simplified design. Notably, it **lacks support for remote repository functionality**, meaning users cannot push or pull changes to and from external servers or collaborators. The **merge conflict resolution** is basic and primarily identifies conflicts without offering advanced automatic merging or sophisticated conflict resolution strategies. Additionally, MiniGit does **not support binary files**, as it is designed to work primarily with text-based source code files, limiting its usability for projects that include images, executable, or other binary data. These constraints highlight areas for potential future enhancements.

Each functionality was separated into distinct source files with appropriate function names and comments. The CLI interface was tested thoroughly to ensure users received helpful error messages and feedback. A help command was included for user assistance, and special care was taken to make outputs like logs and commit messages easy to read.

## Future Improvements

- GUI support.  
- More advanced merge algorithm.  
- DAG caching for performance.  
- Diff viewer integration.

To move closer to Git’s full potential, MiniGit could be enhanced with automatic merge conflict resolution, binary file tracking, a graphical user interface, and remote repository support. These improvements would make MiniGit more robust and suitable for real-world use cases.