**Trek (Version Control System)**



**Session 2023 - 2027**

**Submitted by:**

Javaria Saleem 2023-CS-125

Tayyba Aftab 2023-CS-130

Minahil Imran 2023-CS-134

**Supervised by:**

Sir Nazeef Ul Haq

**Course:**

CSC-200 Data Structures and Algorithms

Department of Computer Science

**University of Engineering and Technology**

Table of Contents

[**● Introduction: 4**](#_gjdgxs)

[**● Repository Initialization: Setting Up the Foundation 4**](#_30j0zll)

[**● Staging Files: Preparing for Commit 4**](#_1fob9te)

[**● Committing Changes: Recording the State of the Repository 5**](#_3znysh7)

[**● Viewing Commit History: Traversing the Past 5**](#_2et92p0)

[**● Branch Creation and Management: Diverging and Merging 5**](#_tyjcwt)

[**● Merging Branches: Bringing Changes Together 6**](#_3dy6vkm)

[**● 8. Performing Diffs: Comparing Changes 6**](#_1t3h5sf)

[**● Conflict Detection: Handling Discrepancies 6**](#_4d34og8)

[**● Ignoring Files Using .gitignore: Excluding Unwanted Files 6**](#_2s8eyo1)

[**● Undo/Redo: 7**](#_17dp8vu)

[**● Pull/Push: 7**](#_3rdcrjn)

[**● Data Structures and Algorithms (DSA) in Trek 7**](#_26in1rg)

[**● Conclusion: 11**](#_2bn6wsx)

Table of Contents

[**● WireFrames: 7**](#_lnxbz9)

[○ Repository initialization: 7](#_35nkun2)

[**○** Staging Files: 7](#_44sinio)

[○ Committing Changes: 8](#_2jxsxqh)

[○ Viewing Commit History 8](#_z337ya)

[○ Branch Creation: 9](#_3j2qqm3)

[○ Merging Branches: 9](#_1y810tw)

[○ Conflict Detection: 10](#_4i7ojhp)

[○ Reset: 10](#_2xcytpi)

[○ Pull/Push 10](#_3whwml4)

# Introduction:

The Trek project serves as a comprehensive model of version control, focusing on repository management, file staging, commit tracking, branch management, and diffing, among other features. By implementing Trek, the project not only serves as a learning tool for understanding the operations behind version control but also provides a deep dive into how Data Structures and Algorithms (DSA) are critical for handling data efficiently in a version control system.

This report explores the core components of the Trek system, providing an in-depth examination of how each feature is implemented, the algorithms used, and the role of data structures in managing data. Additionally, it emphasizes how the creation of a version control system reinforces the understanding of key DSA concepts such as trees, graphs, hashing, and sorting.

# Repository Initialization: Setting Up the Foundation

The first step in any version control system is repository initialization. This process lays the groundwork for the repository’s structure, creating the necessary files and directories to track and manage changes in the codebase. In Trek, this functionality is achieved through the init command, which sets up the .Trek directory that houses all the essential files.

When a user initializes a repository, several things happen behind the scenes:

* A .Trek directory is created, which will hold all metadata about the repository, such as the commit history, staging area, and branch information.
* A HEAD file is created, pointing to the current commit or branch.
* An initial commit file is generated to mark the starting point of the version control system.

At the **data structure level**, **Trek** uses:

* **Hash Tables** to store file paths and metadata efficiently, ensuring that file lookups during commit and staging operations are quick and reliable.
* **Linked Lists** to represent the commit history, where each commit points to its parent, forming a chain of changes.

This setup represents the foundation of **Trek**'s version control system, allowing it to track the history of changes and manage files effectively.

# Staging Files: Preparing for Commit

In version control systems like Git, the staging area (or index) plays a crucial role. Files are first added to the staging area, where they can be reviewed and modified before being committed to the repository. In Trek, the add command stages files by placing them into an index file. This allows the system to track changes between different versions and commit only the desired modifications.

The data structure used here is:

* **Hash Maps** to store file paths and track their modifications, ensuring that the system can quickly identify which files have been modified and need to be staged for commit.

The process of staging a file involves:

* **Adding** the file to the staging area.
* **Storing metadata** about the file (e.g., modification timestamp, hash) to track its changes.

This indexing mechanism ensures that when the user is ready to commit, only the files that have been staged are considered for the commit, allowing for a fine-grained control over the changes being tracked.

# Committing Changes: Recording the State of the Repository

Once files are staged, they are ready to be committed. The commit operation in Trek creates a snapshot of the repository, recording the state of the files and storing it in a commit object. Each commit has several components:

* **Commit Message:** A description of what changes were made.
* **Tree Object:** A reference to the file tree that represents the state of the files at the time of the commit.
* **Parent Commit:** A reference to the previous commit, forming a chain of changes.

The commit history is organized using:

* **Trees** to represent the structure of files in the repository. Each commit contains a reference to a "tree object" that holds the file structure at the time of the commit.
* **Linked Lists** to represent the commit chain, where each commit points to its parent, allowing easy traversal of the commit history.

Committing is not just about saving a snapshot—it also ensures that each change is uniquely identifiable using hashing. The commit object is hashed using a SHA1 function, providing a unique identifier for each commit. This hash ensures that the commit can be traced back and referenced throughout the version history.

# Viewing Commit History: Traversing the Past

One of the core features of Trek is the ability to view commit history. By executing the log command, users can see the sequence of commits that have been made in the repository. Each commit is displayed with its hash, commit message, and the parent commit hash.

The log function traverses the commit history using:

* **Depth-First Search (DFS) or Breadth-First Search (BFS)** algorithms to traverse the linked list of commits. The DFS approach is commonly used to retrieve the commit history from the most recent commit to the oldest.
* **Trees** to organize and visualize the commit history, with branches pointing to different lines of development.

This allows users to follow the development timeline of a project and gain insight into the changes made at each step.

# Branch Creation and Management: Diverging and Merging

Branches are a fundamental part of version control, enabling developers to work on separate features or fixes without affecting the main codebase. In Trek, branches are created by creating new pointers to the current commit. Each branch has its own pointer to the commit history, and switching between branches simply means updating the HEAD file to point to a different commit or branch.

Branches in Trek are managed using:

* + **Graphs** to represent different branches in the repository, with each branch pointing to a specific commit.
  + **Linked Lists** to manage the commit history for each branch, ensuring that commits can be traced from the root to the current state.

Branch switching and management in Trek enable the user to diverge and merge their changes with ease, facilitating parallel development without interference.

# Merging Branches: Bringing Changes Together

When two branches have diverged and changes have been made to the same part of the repository, a merge is required to bring these branches back together. The merge process involves:

* + **Identifying common ancestors:** The system traces back to the last common commit between the branches.
  + **Merging changes:** The differences between the two branches are combined, and if no conflicts exist, the result is a new commit that represents the merged state.

Merging in Trek utilizes:

* + **Trees** to represent the file structure in each branch.
  + **Hashing** to track changes and ensure that the correct state is merged.
  + **Conflict Detection:** When two branches modify the same file differently, a conflict occurs. Trek identifies these conflicts by comparing the changes made to the file in both branches.

While Trek can detect conflicts, it does not yet resolve them, leaving that to the user to manually address.

# 8. Performing Diffs: Comparing Changes

Another essential feature of Trek is the ability to view differences between commits or branches. The diff function compares two versions of the repository and outputs the changes that were made between them. These differences are displayed with:

* + **Additions** marked by a plus sign (+).
  + **Deletions** marked by a minus sign (-).

Diffing works by:

* + **Traversing the file tree:** The system compares the file contents at different commits and identifies changes.
  + **Arrays or Lists** to store the file contents and compare them line by line.

# Conflict Detection: Handling Discrepancies

When two branches have modifications to the same section of code, a merge conflict arises. Trek can detect conflicts by comparing the changes made in both branches. However, it does not yet provide automatic conflict resolution, leaving that to the user.

The conflict detection relies on:

* + **Hashmaps** to track changes in specific files.
  + **Graph** structures to represent how branches diverge and merge.

The detection is performed by checking if the changes made in both branches overlap at the same location in the file.

# Ignoring Files Using .gitignore: Excluding Unwanted Files

The .gitignore file in Trek is used to exclude certain files or directories from being tracked by the version control system. This is essential for ignoring temporary files, build artifacts, or other files that should not be included in the repository.

The .gitignore functionality relies on:

* + **Sets or Lists** to store ignored file paths.
  + **Pattern Matching** to determine which files to ignore based on user-defined rules

By ignoring these files, the repository remains clean and focused on the essential parts of the codebase.

# Undo/Redo:

Functionality: The undo() and redo() functions simulate a rollback and forward mechanism in the version control system. Users can revert their repository to a previous state (using undo()) or move forward to a more recent state (using redo()). These functions provide an intuitive way to navigate through changes and correct mistakes.

**How It Works:** These functions maintain two stacks—one for undo operations and one for redo operations. When a commit is undone, it is pushed onto the redo stack, and the repository state is rolled back to a previous commit. Similarly, when a redo is performed, the previous state is restored from the redo stack. The reset() function is used to reset the repository to a specific commit, enabling precise control over the version history.

# Pull/Push:

Simulated push and pull functionalities provide the capability to sync changes between branches. These actions replicate the process of transferring changes between local repositories, helping users manage updates across different branches within a repository. Though basic, the push and pull commands can be expanded for full remote repository integration, facilitating team collaboration in future versions.

# Data Structures and Algorithms (DSA) in Trek

The Trek project provides an excellent opportunity to apply core DSA principles to a real-world problem. From commit history management to conflict detection, the project reinforces concepts like:

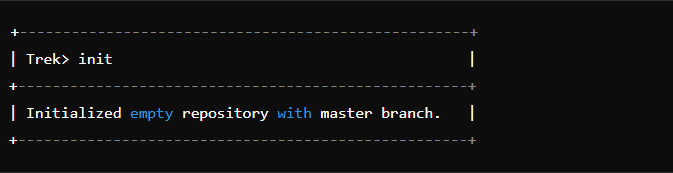
* + **Linked Lists** for representing commit history.
  + **Hash Tables** for efficient file tracking and hashing.
  + **Trees** for managing the hierarchical structure of files and commits.
  + **Graphs** for representing branches and their relationships.
  + **Hashing** for ensuring data integrity and uniqueness.
  + **Stacks** to undo redo commits

By implementing these data structures and algorithms, the project deepens the understanding of how data is stored, manipulated, and accessed in a version control system.

# WireFrames:

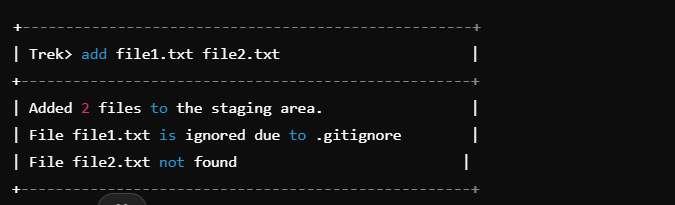
## Repository initialization:

When the init command is run, the program initializes a new Git-like repository. It creates necessary folders such as .Trek, objects, and refs/heads. It sets up an empty master branch and a .gitignore file. If the repository already exists, it will notify the user.



## Staging Files:

The add command stages the specified files for commit. It checks if each file is ignored by the .gitignore file. If the file is found and not ignored, it gets added to the staging area. Otherwise, the system reports errors like files not found or being ignored.



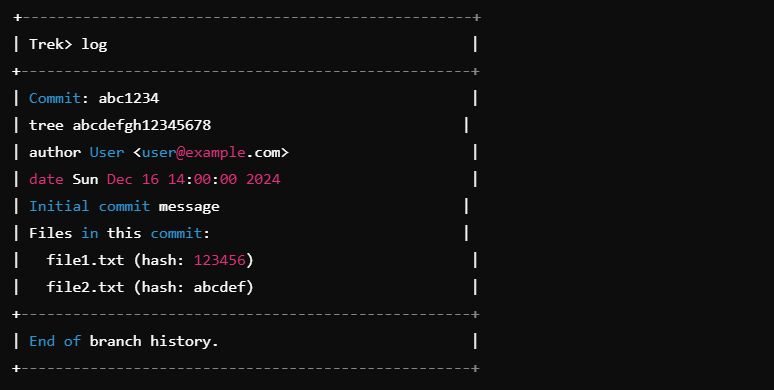
## Committing Changes:

The commit command commits the staged changes with a message. It generates a commit hash and stores information about the commit, including the author, date, message, and tree hash. The HEAD is updated to point to the new commit.



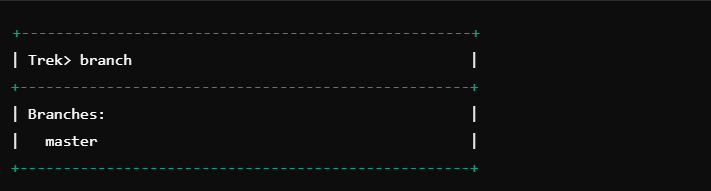
## Viewing Commit History

The log command displays the commit history for the current branch, showing each commit’s unique hash, tree hash, author, date, and commit message. It also lists the files modified, added, or deleted in each commit, along with their hashes. The log is shown in reverse chronological order, with the most recent commit at the top. This command allows users to track changes and review the project’s history, with options to filter results or customize the output format for more detailed

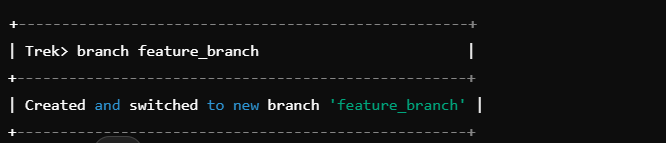


## Branch Creation:

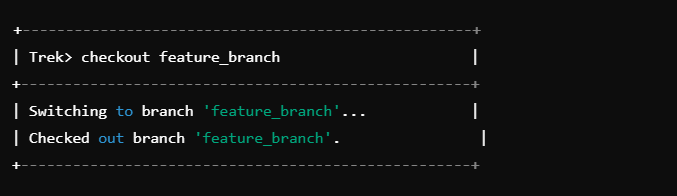
The branch command, when run without any arguments, lists all the available branches in the repository. In this case, it shows that only the master branch exists.



The branch <branch\_name> command creates a new branch with the specified name and switch to it. It writes the current commit hash to the new branch's file under refs/heads. If the branch already exists, it switches to it.

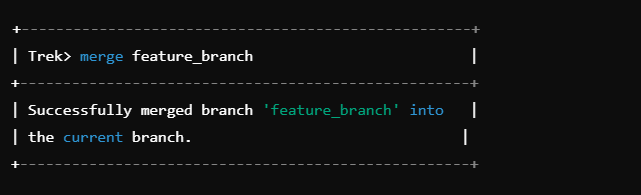


The checkout <branch\_name> command switches to the specified branch. It updates the HEAD file to point to the selected branch. It simulates the process of changing the working directory to reflect the contents of the branch.



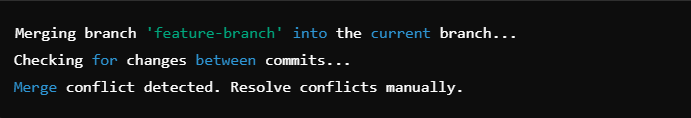
## Merging Branches:

The merge <branch\_name> command merges the specified branch into the current branch. If there are conflicts, it will notify the user. If the merge is successful without conflicts, it updates the branch reference.



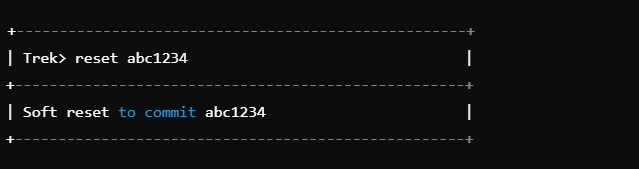
## Conflict Detection:

Conflict detection in Trek works by reading the content of the files from both commits and comparing them. If the contents are different, the conflict is flagged, and the user is asked to manually resolve it, typically by editing the file to reconcile the changes.



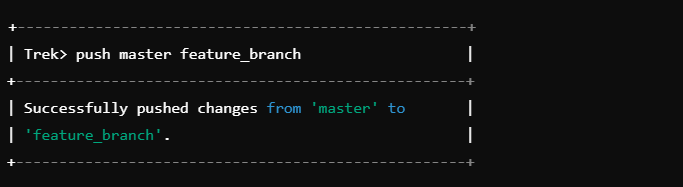
## Reset:

The reset <commit\_hash> command resets the repository to a specific commit. A "soft" reset updates the HEAD to point to the specific commit, but leaves the working directory unchanged. If the hard option is specified, the working directory is also reset.

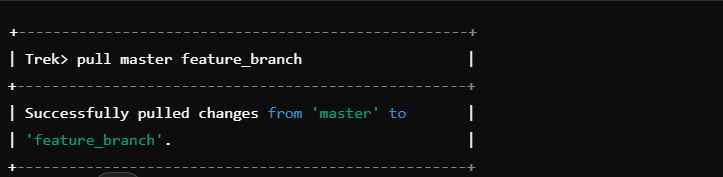


## Pull/Push

The push <source\_branch> <target\_branch> command is used to transfer the changes made in one branch (the source branch) to another branch (the target branch). It works by copying the commit history, including the commit hash, from the source branch and applying it to the target branch. This operation essentially updates the target branch with all the commits that are present in the source branch but not yet in the target. The commit hash ensures that the changes are linked to the correct point in the history, preserving the integrity of the version control system. If the target branch already has changes that conflict with the source, the push operation may result in conflicts, which will need to be resolved. This command is useful for pushing updates from a feature branch to a main or development branch.



The pull <source\_branch> <target\_branch> command pulls changes from one branch to another. It copies the commit hash from the source branch and writes it to the target branch.



# Conclusion:

The Trek project provides a deep understanding of both version control and DSA. By implementing features like branching, merging, and conflict detection, the project demonstrates how core data structures and algorithms are applied in real-world systems. This project reinforces the importance of efficient data handling in software development and offers valuable lessons in building complex systems from the ground up.