# MiniGit: View Log and Branching Implementation in C++

Implementation Overview

While implementing the `log` and `branch` features of the MiniGit version control system in C++. These features allow users to view commit history and create/manage branches.

Data Structures Used

1. Commit Node Structure:

struct CommitNode {

std::string hash; // Unique identifier for the commit

std::string message; // Commit message

std::string timestamp; // Time of commit

std::string parentHash; // Parent commit hash

std::vector<std::string> fileHashes; // Hashes of files in this commit

};

```

2. Branch Structure:

struct Branch {

std::string name; // Branch name

std::string latestCommit; // Hash of the latest commit on this branch

};

```

3. Repository State:

class MiniGitRepository {

private:

std::string currentBranch; // Currently active branch

std::map<std::string, Branch> branches; // Map of branch names to Branch objects

std::map<std::string, CommitNode> commitHistory; // Map of commit hashes to CommitNode

// ... other repository state ...

};

```

Implementation Code

1. View Log Implementation

```cpp

void MiniGitRepository::log() {

if (commitHistory.empty()) {

std::cout << "No commits yet." << std::endl;

return;

}

std::string currentCommitHash = branches[currentBranch].latestCommit;

while (!currentCommitHash.empty()) {

CommitNode commit = commitHistory[currentCommitHash];

// Print commit information

std::cout << "commit " << commit.hash << std::endl;

std::cout << "Author: Your Name <your.email@example.com>" << std::endl;

std::cout << "Date: " << commit.timestamp << std::endl;

std::cout << std::endl;

std::cout << " " << commit.message << std::endl;

std::cout << std::endl;

// Move to parent commit

currentCommitHash = commit.parentHash;

}

}

```

2. Branching Implementation

void MiniGitRepository::createBranch(const std::string& branchName) {

if (branches.find(branchName) != branches.end()) {

std::cout << "A branch named '" << branchName << "' already exists." << std::endl;

return;

}

Branch newBranch;

newBranch.name = branchName;

newBranch.latestCommit = branches[currentBranch].latestCommit;

branches[branchName] = newBranch;

std::cout << "Created new branch '" << branchName << "'" << std::endl;

}

void MiniGitRepository::listBranches() {

std::cout << "Available branches:" << std::endl;

for (const auto& pair : branches) {

if (pair.first == currentBranch) {

std::cout << "\* " << pair.first << std::endl;

} else {

std::cout << " " << pair.first << std::endl;

}

}

}

```

Design Decisions may include:-

1. Commit Storage:

- Used a map to store commits for O(1) access by hash

- Each commit maintains a reference to its parent for history traversal

2. Branch Implementation:

- Branches are lightweight pointers to commits

- Current branch is tracked separately for easy referenced

- Branch creation is cheap as it just creates a new pointer

3. Log Traversal:

- Follows parent pointers from HEAD to initial commit.

- Linear history assumed (no merge commits in this basic implementation)

Limitations or disadvantages may include :-

1. Performance:

- The log traversal is linear time O(n) where n is number of commits

- For very large histories, this could be slow

2. Branch Visualization:

- Doesn't show the branching/merging structure visually

- Only shows linear history from current branch

3. Memory Usage:

- All commits are kept in memory

- For very large repositories, this could be memory intensive

Future Improvements

1. Graphical Log Output

- Implement ASCII art to show branch/merge history

- Similar to `git log --graph`

2. Pagination:

- Add pagination support for long commit histories

- Similar to `git log --oneline | less`

3. Filtering Options:

- Add filtering by author, date range, or message content

- Similar to “git log --author="name"`

4. Commit Hashing:

- Currently using simple hashes, could implement SHA-1 for more realistic behavior

5. Persistent Storage:

- Currently in-memory, could serialize to disk for persistence

Short Report

Data Structures Used

The implementation uses several key data structures:

1. Maps (Hash Tables): For O(1) access to commits by hash and branches by name

2. Linked List: The commit history forms a linked list through parent pointers

3. Vectors: To store file hashes associated with each commit

Design Decisions concept-

The design prioritizes:

- Simple traversal of commit history

- Fast branch switching

- Easy addition of new commits

- Minimal memory overhead for branches

The choice of maps for commit and branch storage provides fast access while maintaining flexibility. The linear commit history with parent pointers matches Git's internal model while keeping the implementation simple.

Limitations of the design:-

The current implementation has several limitations that differ from real Git:

1. No support for merge commits (non-linear history)

2. All data is kept in memory (no disk persistence)

3. Simplified hashing mechanism

4. No handling of detached HEAD state

Future Improvements

To make this more Git-like, we could:

1. Implement disk-based storage of commits and objects

2. Add proper SHA-1 hashing of commit objects

This implementation provides a solid foundation for understanding version control systems while keeping the codebase manageable for educational purposes.