College Of Technology and Built Environment(CTBE)

Data Structures and Algorithm Analysis Assignment

Member Name	ID
Zenebu Melaku	UGR/6058/16
Tsedeniya Fiseha	UGR/9263/16
Elizabeth Abay	UGR/3071/16
Hawi Jarso	UGR/4431/16
Naol Worku	UGR/7914/16

Section : II
Submitted to: Mr. Habib Gemechu
Submission date: Jun 20, 2025

MiniGit: A Lightweight Version Control System

Overview

MiniGit is a custom version control system built with C++, designed to work a lot like Git but on a smaller scale. It lets you track local files, make commits, create branches, and even do merges and diffs — all from the command line. It saves data using the local file system and applies several core data structure and algorithm (DSA) concepts like hashing, trees, directed acyclic graphs (DAGs), and file comparison.

Project Objectives

The main goals of this project are:

- Implement core Git features including repository initialization, staging files, making commits, branching, merging, and viewing the commit history.
- Apply fundamental data structures and algorithms (DSA) in a hands-on software development project.
- Develop proficiency in file input/output operations, hashing with SHA-1, and designing command-line interface (CLI) software.
- Practice software engineering principles like modular design, thorough documentation, and managing versioned development.

Key Data Structures

- **Blob**: This is how file content is stored each file is saved as a blob in the `.minigit/objects/` directory. The file name is a SHA-1 hash based on its content. What makes this interesting is that files aren't duplicated unnecessarily; if the content hasn't changed, the same blob is reused.
- **Tree**: Think of a tree as a folder. It stores references to blobs (files) and other trees (subfolders). Every new commit gets a new tree object representing the current file structure.
- **Commit**: It includes metadata like who made the commit, when, and what files are included. Each commit points to a specific tree, and each is uniquely hashed.
- **Branch**: A branch acts like a named label pointing to a specific commit. It's how Git (and MiniGit) keeps track of different lines of development.
- **Staging Area**: Before committing, files go into the staging area — a sort of prep zone where changes are gathered before they're officially saved.

- **Log History**: You can walk through previous commits by following the links each commit has to its parent. When you run a `git log`-like command, it shows all past commits in reverse order, starting from the latest.

- **Diff**: This command helps compare the current version of a file with an older one (or two commits with each other), highlighting what's been added or removed line by line.

Core DSA Concepts

- **Hashing**: SHA-1 hashing is used for uniquely identifying blobs, trees, and commits.

- **Trees**: The structure reflects directories and is built recursively.

- **DAGs**: Commits form a directed acyclic graph — each commit can have a parent (or multiple, in merges), and this structure helps trace history or find common ancestors.

- **File Diffing**: Files are compared line by line to show differences, useful during commits or merges.

Design Choices

- **Storage**: Everything is stored under a `.minigit/` folder in your project directory.

- **Hashing**: Uses SHA-1 from OpenSSL for generating unique IDs.

- **Trees**: Tree objects help organize folders and files in a structured way.

- **Merging**: A simplified version of Git's three-way merge, including basic conflict handling.

- **Diff Viewer**: Shows clear line-by-line differences for better understanding of changes.

Usage

```
./minigit init  - used to create a repository
./minigit add <path> - used to add files to the index
./minigit commit -m "message" - commits the changes made to the repo
./minigit log - shows all the git commit history
./minigit branch <name> - creates a branch to work on a new feature
./minigit checkout <branch-or-commit> - make the head pointer point to
the current branch or commit
./minigit merge <branch> - will merge your work on your branch with the
main file
```

`./minigit diff <commit1> [<commit2>] - will show the difference between the current version and commit 1.`

Limitations of MiniGit

1. **No Remote Repository**: not able to collaborate remotely,, limiting it to local use.
2. **Manual Conflict Resolution**: Merge conflicts require manual intervention, which can be draining.
3. **Performance**: Inefficient for large files or repositories due to in-memory file reading and basic diff algorithm.
4. **Limited Features**: Missing advanced Git features like stashing, rebasing, or tags.
5. **Error Handling**: Basic checks; lacks robust handling for file permissions or corrupted repositories.
6. **No Binary File Support**: Diffing assumes text files, ignoring binary files.

Future Improvements

- **Remote Support**: Add push/pull to sync with remote repos for collaboration.
- **Interactive Tools**: Improve diff/merge with colored output and conflict markers.
- **Stash Command**: Save and restore uncommitted changes for smoother workflows.
- **Config File**: Store user settings like name/email for personalized commits.
- **Tags**: Add lightweight tags to mark releases or key commits.
- **Undo Options**: Support reset/revert to undo changes or commits.
- **Compression**: Compress objects to save disk space.
- **Binary Files**: Handle non-text files better in add/diff/merge.
- **Logging**: Add operation logs and stats for debugging and insights.
- **Branch Tools**: Include branch deletion, listing, and renaming.

External Requirements:

- **OpenSSL Library**

  Used for `#include <openssl/sha.h>`

Required for SHA-1 hashing (like Git).

Install with:

- Linux: `sudo apt install libssl-dev`
- macOS: `brew install openssl`
- Windows: use MSYS2 or WSL

- **C++17 or Later**

  Required for `<filesystem>` support.

Compile with:

g++ -std=c++17 main.cpp -lssl -lcrypto -o minigit

Code implementation:

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <unordered_map>
```

```cpp
#include <unordered_set>
#include <vector>
#include <filesystem>
#include <ctime>
#include <iomanip>
#include <algorithm>
#include <functional>
#include <openssl/sha.h>


namespace fs = std::filesystem;


class MiniGit {
private:
    // structure of repository directory
    std::string repo_dir = ".minigit";  // Root directory for MiniGit data
    std::string objects_dir = repo_dir + "/objects";   // stores blobs,
commits and trees
    std::string refs_dir = repo_dir + "/refs";  // stores tag references
and branch
    std::string head_file = repo_dir + "/HEAD"; //staging area index
    std::string index_file = repo_dir + "/index";


    struct TreeEntry {
        std::string name;
        std::string hash;   // SHA-1 hash of blob or tree
        bool is_blob;
    };


    struct Commit {
        std::string hash;
        std::string message;
        std::string timestamp;
        std::string parent;
        std::string tree_hash;
    };
```

```
    // it tracks staged directories and filess

    std::unordered_map<std::string, std::string> staging_area; // path ->
blob hash

    std::unordered_map<std::string, std::vector<TreeEntry>> staged_trees;
// dir -> entries
    // Computes SHA-1 hash of a string

    std::string compute_sha1(const std::string& content) {
        unsigned char hash[SHA_DIGEST_LENGTH];
        SHA1(reinterpret_cast<const unsigned char*>(content.c_str()),
content.length(), hash);
        std::stringstream ss;
        for (int i = 0; i < SHA_DIGEST_LENGTH; i++) {
            ss << std::hex << std::setw(2) << std::setfill('0') <<
(int)hash[i];
        }
        return ss.str();
    }
//  To return timestamp of current time and in format of "YYYY-MM-DD
HH:MM:SS"


    std::string get_time() {
        auto now = std::time(nullptr);
        std::stringstream ss;
        ss << std::put_time(std::localtime(&now), "%Y-%m-%d %H:%M:%S");
        return ss.str();
    }


    std::string read_file(const std::string& path) {
        std::ifstream file(path);
        if (!file.is_open()) return "";
        std::stringstream ss;
        ss << file.rdbuf();
        return ss.str();
```

```
    }


    void write_file(const std::string& path, const std::string& content) {
        fs::create_directories(fs::path(path).parent_path());
        std::ofstream file(path);
        file << content;
    }
    // creates tree object for directory and it going to writes it to
objects_dir
    std::string create_tree(const std::string& dir_path) {
        auto entries = staged_trees[dir_path];
    // sorting for deterministic entries
        std::sort(entries.begin(), entries.end(), [](const TreeEntry& a,
const TreeEntry& b) {
            return a.name < b.name;
        });
        std::stringstream ss;
        for (auto& entry : entries) {
            ss << (entry.is_blob ? "blob" : "tree") << " " << entry.name
<< " " << entry.hash << "\n";
        }
        std::string tree_hash = compute_sha1(ss.str());
        write_file(objects_dir + "/" + tree_hash, ss.str());
        return tree_hash;
    }


    void build_tree_hierarchy(const std::string& path, const std::string&
blob_hash = "") {
        fs::path p(path);
        std::string filename = p.filename().string();
        std::string parent_dir = p.parent_path().string().empty() ? "." :
p.parent_path().string();
        if (!blob_hash.empty()) {   // add file to parent directory's tree
            // hash map that links file path to list of entries in the
file
            staged_trees[parent_dir].push_back({filename, blob_hash,
true});
```

```
            staging_area[path] = blob_hash; // hash map that links file
path to hash
            return;
        }
    // to process sub directories and create tree objects
        for (auto it = staged_trees.begin(); it != staged_trees.end();
++it) {
            auto& [sub, entries] = *it;
            if (sub == ".") continue;
            std::string tree_hash = create_tree(sub);
            fs::path sp(sub);
            std::string parent = sp.parent_path().string().empty() ? "." :
sp.parent_path().string();
            staged_trees[parent].push_back({sp.filename().string(),
tree_hash, false});
        }
    }



    // collecting files from tree objects recursively
void collect_files(const std::string& tree_hash, const std::string&
prefix, std::unordered_map<std::string, std::string>& files) {
        std::string content = read_file(objects_dir + "/" + tree_hash);
        std::stringstream ss(content);
        std::string line;
        while (std::getline(ss, line)) {
            std::string type = line.substr(0, line.find(" "));
            std::string rest = line.substr(type.length() + 1);
            auto pos = rest.find(" ");
            std::string name = rest.substr(0, pos);
            std::string hash = rest.substr(pos + 1);
            std::string path = prefix.empty() ? name : prefix + "/" +
name;
            if (type == "blob") {
                files[path] = hash;
            } else {
                collect_files(hash, path, files);
            }
```

```
        }
    }


public:
    void init() {       // initializing new mini git repo
        fs::create_directories(objects_dir);
        fs::create_directories(refs_dir);
        write_file(head_file, "ref: refs/master");
        write_file(index_file, "");
        std::cout << "Initialized MiniGit repo\n";
    }

    // add file to staging area
    void add(const std::string& path) {
        if (!fs::exists(path)) {
            std::cout << "Error: Path does not exist\n";
            return;
        }
        if (fs::is_directory(path)) {
        // adding all files of in directory by recursion
            for (auto& entry : fs::recursive_directory_iterator(path)) {
                if (fs::is_regular_file(entry)) {
                    std::string file_path = entry.path().string();
                    std::string content = read_file(file_path);
                    std::string hash = compute_sha1(content);
                    write_file(objects_dir + "/" + hash, content);
                    build_tree_hierarchy(file_path, hash);
                }
            }
        } else { // adding single file
            std::string content = read_file(path);
            std::string hash = compute_sha1(content);
            write_file(objects_dir + "/" + hash, content);
            build_tree_hierarchy(path, hash);
        }
    // update index file with staging area
        std::stringstream index;
        for (auto& [f, h] : staging_area) index << f << " " << h << "\n";
```

```
        write_file(index_file, index.str());
        std::cout << "Added " << path << "\n";
    }


    void commit(const std::string& message) {
        if (staging_area.empty()) {
            std::cout << "Nothing to commit\n";
            return;
        }
        build_tree_hierarchy("");   //finalizing hierarchy of tree
        std::string tree_hash = create_tree(".");
        std::string parent;
        if (fs::exists(head_file)) {
            std::string head = read_file(head_file).substr(5);
            if (fs::exists(refs_dir + "/" + head)) parent =
read_file(refs_dir + "/" + head);
        }
        //create object commit
        std::stringstream commit;
        std::string timestamp = get_time();
        commit << "tree: " << tree_hash << "\n";
        if (!parent.empty()) commit << "parent: " << parent << "\n";
        commit << "timestamp: " << timestamp << "\n";
        commit << "message: " << message << "\n";
        std::string hash = compute_sha1(commit.str());
        write_file(objects_dir + "/" + hash, commit.str());
        std::string ref = read_file(head_file).substr(5);
        write_file(refs_dir + "/" + ref, hash);
        staging_area.clear();
        staged_trees.clear();
        write_file(index_file, "");
        std::cout << "[MiniGit] Commit successful: " << hash << "\n";
    }


      // to display commit history begin form head
void log() {
        std::string head = read_file(head_file).substr(5);
```

```
        std::string commit = fs::exists(refs_dir + "/" + head) ?
read_file(refs_dir + "/" + head) : "";
        while (!commit.empty()) {
            std::string content = read_file(objects_dir + "/" + commit);
            std::stringstream ss(content);
            std::string line, msg, time, parent;
            while (std::getline(ss, line)) {
                if (line.find("message: ") == 0) msg = line.substr(9);
                if (line.find("timestamp: ") == 0) time = line.substr(11);
                if (line.find("parent: ") == 0) parent = line.substr(8);
            }
            std::cout << "commit " << commit << "\nDate: " << time <<
"\n\t" << msg << "\n\n";
            commit = parent;
        }
    }
    // creating new brach to point to current head
    void branch(const std::string& name) {
        std::string ref = read_file(head_file).substr(5);
        std::string hash = fs::exists(refs_dir + "/" + ref) ?
read_file(refs_dir + "/" + ref) : "";
        write_file(refs_dir + "/" + name, hash);
        std::cout << "Branch " << name << " created\n";
    }
    // switches to branch or commit
    void checkout(const std::string& target) {
        std::string commit_hash;
        if (fs::exists(refs_dir + "/" + target)) {
            commit_hash = read_file(refs_dir + "/" + target);
            write_file(head_file, "ref: refs/" + target);
        } else {
            commit_hash = target;
            if (!fs::exists(objects_dir + "/" + commit_hash)) {
                std::cout << "Error: Commit not found\n";
                return;
            }
        }
        // Clear working directory
        for (auto& [file, _] : staging_area) {
```

```
            if (fs::exists(file)) fs::remove(file);
        }
        staging_area.clear();
        staged_trees.clear();
        write_file(index_file, "");
        // Restore files from commit
        std::string content = read_file(objects_dir + "/" + commit_hash);
        std::string tree_hash;
        std::stringstream ss(content);
        std::string line;
        while (std::getline(ss, line)) {
            if (line.find("tree: ") == 0) {
                tree_hash = line.substr(6);
                break;
            }
        }
        std::unordered_map<std::string, std::string> files;
        collect_files(tree_hash, "", files);
        for (auto& [path, hash] : files) {
            std::string file_content = read_file(objects_dir + "/" +
hash);
            write_file(path, file_content);
            staging_area[path] = hash;
            build_tree_hierarchy(path, hash);
        }
    // updating file of index
        std::stringstream index;
        for (auto& [f, h] : staging_area) index << f << " " << h << "\n";
        write_file(index_file, index.str());
        std::cout << "Checked out " << target << "\n";
    }
    // finds lowest common ancestor of 2 commits by using depth search
method


    std::string find_lca(const std::string& c1, const std::string& c2) {
        std::unordered_set<std::string> ancestors;
        std::function<void(const std::string&)> collect = [&](const
std::string& commit) {
```

```
            std::string current = commit;
            while (!current.empty()) {
                ancestors.insert(current);
                std::string content = read_file(objects_dir + "/" +
current);
                std::stringstream ss(content);
                std::string line, parent;
                while (std::getline(ss, line)) {
                    if (line.find("parent: ") == 0) {
                        parent = line.substr(8);
                        break;
                    }
                }
                current = parent;
            }
        };
        collect(c1);
        std::string current = c2;
        while (!current.empty()) {
            if (ancestors.count(current)) return current;
            std::string content = read_file(objects_dir + "/" + current);
            std::stringstream ss(content);
            std::string line, parent;
            while (std::getline(ss, line)) {
                if (line.find("parent: ") == 0) {
                    parent = line.substr(8);
                    break;
                }
            }
            current = parent;
        }
        return "";  // if there is no common ancestor being found
    }


    void merge(const std::string& branch) {
        if (!fs::exists(refs_dir + "/" + branch)) {
            std::cout << "Error: Branch not found\n";
            return;
```

```
        }
        std::string head = read_file(head_file).substr(5);
        std::string c1 = read_file(refs_dir + "/" + head);
        std::string c2 = read_file(refs_dir + "/" + branch);
        std::string lca = find_lca(c1, c2);
        if (lca.empty()) {
            std::cout << "Error: No common ancestor\n";
            return;
        }
    // processing file trees for commits
        std::unordered_map<std::string, std::string> files1, files2,
files_lca;
        std::function<void(const std::string&,
std::unordered_map<std::string, std::string>&)> load = [&](const
std::string& commit, auto& files) {
            std::string content = read_file(objects_dir + "/" + commit);
            std::string tree_hash;
            std::stringstream ss(content);
            std::string line;
            while (std::getline(ss, line)) {
                if (line.find("tree: ") == 0) {
                    tree_hash = line.substr(6);
                    break;
                }
            }
            collect_files(tree_hash, "", files);
        };
        load(c1, files1);
        load(c2, files2);
        load(lca, files_lca);
        bool conflict = false;
    //Compare files and head to apply changes
        for (auto& [path, hash2] : files2) {
            auto it1 = files1.find(path);
            auto it_lca = files_lca.find(path);
            std::string hash1 = it1 != files1.end() ? it1->second : "";
            std::string hash_lca = it_lca != files_lca.end() ?
it_lca->second : "";
```

```
            if (hash1 != hash_lca && hash2 != hash_lca && hash1 != hash2)
{
                std::cout << "CONFLICT: both modified " << path << "\n";
                conflict = true;
                continue;
            }
            if (hash2 != hash_lca) {
                std::string content = read_file(objects_dir + "/" +
hash2);
                write_file(path, content);
                staging_area[path] = hash2;
                build_tree_hierarchy(path, hash2);
            }
        }
        if (conflict) {
            std::cout << "Merge failed due to conflicts\n";
            return;
        }
    // create merge commit
        build_tree_hierarchy("");
        std::string tree_hash = create_tree(".");
        std::stringstream commit;
        std::string timestamp = get_time();
        commit << "tree: " << tree_hash << "\n";
        commit << "parent: " << c1 << "," << c2 << "\n";
        commit << "timestamp: " << timestamp << "\n";
        commit << "message: Merge branch '" << branch << "'\n";
        std::string hash = compute_sha1(commit.str());
        write_file(objects_dir + "/" + hash, commit.str());
        write_file(refs_dir + "/" + head, hash);
        staging_area.clear();
        staged_trees.clear();
        write_file(index_file, "");
        std::cout << "Merged " << branch << "\n";
    }
    // display differences between commits
    void diff(const std::string& c1, const std::string& c2 = "") {
        std::string commit1 = c1;
```

```
        if (fs::exists(refs_dir + "/" + c1)) commit1 = read_file(refs_dir
+ "/" + c1);
        std::string commit2 = c2;
        if (!c2.empty() && fs::exists(refs_dir + "/" + c2)) commit2 =
read_file(refs_dir + "/" + c2);
        std::unordered_map<std::string, std::string> files1, files2;
        if (!fs::exists(objects_dir + "/" + commit1)) {
            std::cout << "Error: Commit not found\n";
            return;
        }
        std::string tree1;
        std::string content = read_file(objects_dir + "/" + commit1);
        std::stringstream ss(content);
        std::string line;
        while (std::getline(ss, line)) {
            if (line.find("tree: ") == 0) {
                tree1 = line.substr(6);
                break;
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <unordered_map>
#include <unordered_set>
#include <vector>
#include <filesystem>
#include <ctime>
#include <iomanip>
#include <algorithm>
#include <functional>
#include <openssl/sha.h>


namespace fs = std::filesystem;


class MiniGit {
private:
      // structure of repository directory
```

```
    std::string repo_dir = ".minigit";      // Root directory for MiniGit
data
    std::string objects_dir = repo_dir + "/objects";   // stores blobs,
commits and trees
    std::string refs_dir = repo_dir + "/refs";    // stores tag references
and branch
    std::string head_file = repo_dir + "/HEAD";   //staging area index
    std::string index_file = repo_dir + "/index";


    struct TreeEntry {
        std::string name;
        std::string hash;      // SHA-1 hash of blob or tree
        bool is_blob;
    };


    struct Commit {
        std::string hash;
        std::string message;
        std::string timestamp;
        std::string parent;
        std::string tree_hash;
    };

      // it tracks staged directories and filess

    std::unordered_map<std::string, std::string> staging_area; // path ->
blob hash

    std::unordered_map<std::string, std::vector<TreeEntry>> staged_trees;
// dir -> entries
      // Computes SHA-1 hash of a string

    std::string compute_sha1(const std::string& content) {
        unsigned char hash[SHA_DIGEST_LENGTH];
        SHA1(reinterpret_cast<const unsigned char*>(content.c_str()),
content.length(), hash);
        std::stringstream ss;
```

```
        for (int i = 0; i < SHA_DIGEST_LENGTH; i++) {
            ss << std::hex << std::setw(2) << std::setfill('0') <<
(int)hash[i];
        }
        return ss.str();
    }
//    To return timestamp of current time and in format of "YYYY-MM-DD
HH:MM:SS"


    std::string get_time() {
        auto now = std::time(nullptr);
        std::stringstream ss;
        ss << std::put_time(std::localtime(&now), "%Y-%m-%d %H:%M:%S");
        return ss.str();
    }


    std::string read_file(const std::string& path) {
        std::ifstream file(path);
        if (!file.is_open()) return "";
        std::stringstream ss;
        ss << file.rdbuf();
        return ss.str();
    }


    void write_file(const std::string& path, const std::string& content) {
        fs::create_directories(fs::path(path).parent_path());
        std::ofstream file(path);
        file << content;
    }
      // creates tree object for directory and it going to writes it to
objects_dir
    std::string create_tree(const std::string& dir_path) {
        auto entries = staged_trees[dir_path];
      // sorting for deterministic entries
        std::sort(entries.begin(), entries.end(), [](const TreeEntry& a,
const TreeEntry& b) {
```

```
            return a.name < b.name;
        });
        std::stringstream ss;
        for (auto& entry : entries) {
            ss << (entry.is_blob ? "blob" : "tree") << " " << entry.name
<< " " << entry.hash << "\n";
        }
        std::string tree_hash = compute_sha1(ss.str());
        write_file(objects_dir + "/" + tree_hash, ss.str());
        return tree_hash;
    }


    void build_tree_hierarchy(const std::string& path, const std::string&
blob_hash = "") {
        fs::path p(path);
        std::string filename = p.filename().string();
        std::string parent_dir = p.parent_path().string().empty() ? "." :
p.parent_path().string();
        if (!blob_hash.empty()) {    // add file to parent directory's
tree
            // hash map that links file path to list of entries in the
file
            staged_trees[parent_dir].push_back({filename, blob_hash,
true});
            staging_area[path] = blob_hash; // hash map that links file
path to hash
            return;
        }
      // to process sub directories and create tree objects
        for (auto it = staged_trees.begin(); it != staged_trees.end();
++it) {
            auto& [sub, entries] = *it;
            if (sub == ".") continue;
            std::string tree_hash = create_tree(sub);
            fs::path sp(sub);
            std::string parent = sp.parent_path().string().empty() ? "." :
sp.parent_path().string();
```

```
            staged_trees[parent].push_back({sp.filename().string(),
tree_hash, false});
        }
    }



      // collecting files from tree objects recursively
void collect_files(const std::string& tree_hash, const std::string&
prefix, std::unordered_map<std::string, std::string>& files) {
        std::string content = read_file(objects_dir + "/" + tree_hash);
        std::stringstream ss(content);
        std::string line;
        while (std::getline(ss, line)) {
            std::string type = line.substr(0, line.find(" "));
            std::string rest = line.substr(type.length() + 1);
            auto pos = rest.find(" ");
            std::string name = rest.substr(0, pos);
            std::string hash = rest.substr(pos + 1);
            std::string path = prefix.empty() ? name : prefix + "/" +
name;
            if (type == "blob") {
                files[path] = hash;
            } else {
                collect_files(hash, path, files);
            }
        }
    }


public:
    void init() {        // initializing new mini git repo
        fs::create_directories(objects_dir);
        fs::create_directories(refs_dir);
        write_file(head_file, "ref: refs/master");
        write_file(index_file, "");
        std::cout << "Initialized MiniGit repo\n";
    }
```

```
// add file to staging area
void add(const std::string& path) {
    if (!fs::exists(path)) {
        std::cout << "Error: Path does not exist\n";
        return;
    }
    if (fs::is_directory(path)) {
         // adding all files of in directory by recursion
        for (auto& entry : fs::recursive_directory_iterator(path)) {
            if (fs::is_regular_file(entry)) {
                std::string file_path = entry.path().string();
                std::string content = read_file(file_path);
                std::string hash = compute_sha1(content);
                write_file(objects_dir + "/" + hash, content);
                build_tree_hierarchy(file_path, hash);
            }
        }
    } else { // adding single file
        std::string content = read_file(path);
        std::string hash = compute_sha1(content);
        write_file(objects_dir + "/" + hash, content);
        build_tree_hierarchy(path, hash);
    }
  // update index file with staging area
    std::stringstream index;
    for (auto& [f, h] : staging_area) index << f << " " << h << "\n";
    write_file(index_file, index.str());
    std::cout << "Added " << path << "\n";
}


void commit(const std::string& message) {
    if (staging_area.empty()) {
        std::cout << "Nothing to commit\n";
        return;
    }
    build_tree_hierarchy("");    //finalizing hierarchy of tree
    std::string tree_hash = create_tree(".");
    std::string parent;
```

```
        if (fs::exists(head_file)) {
            std::string head = read_file(head_file).substr(5);
            if (fs::exists(refs_dir + "/" + head)) parent =
read_file(refs_dir + "/" + head);
        }
             //create object commit
        std::stringstream commit;
        std::string timestamp = get_time();
        commit << "tree: " << tree_hash << "\n";
        if (!parent.empty()) commit << "parent: " << parent << "\n";
        commit << "timestamp: " << timestamp << "\n";
        commit << "message: " << message << "\n";
        std::string hash = compute_sha1(commit.str());
        write_file(objects_dir + "/" + hash, commit.str());
        std::string ref = read_file(head_file).substr(5);
        write_file(refs_dir + "/" + ref, hash);
        staging_area.clear();
        staged_trees.clear();
        write_file(index_file, "");
        std::cout << "[MiniGit] Commit successful: " << hash << "\n";
    }


      // to display commit history begin form head
void log() {
        std::string head = read_file(head_file).substr(5);
        std::string commit = fs::exists(refs_dir + "/" + head) ?
read_file(refs_dir + "/" + head) : "";
        while (!commit.empty()) {
            std::string content = read_file(objects_dir + "/" + commit);
            std::stringstream ss(content);
            std::string line, msg, time, parent;
            while (std::getline(ss, line)) {
                if (line.find("message: ") == 0) msg = line.substr(9);
                if (line.find("timestamp: ") == 0) time = line.substr(11);
                if (line.find("parent: ") == 0) parent = line.substr(8);
            }
            std::cout << "commit " << commit << "\nDate: " << time <<
"\n\t" << msg << "\n\n";
```

```
            commit = parent;
        }
    }
      // creating new brach to point to current head
    void branch(const std::string& name) {
        std::string ref = read_file(head_file).substr(5);
        std::string hash = fs::exists(refs_dir + "/" + ref) ?
read_file(refs_dir + "/" + ref) : "";
        write_file(refs_dir + "/" + name, hash);
        std::cout << "Branch " << name << " created\n";
    }
      // switches to branch or commit
    void checkout(const std::string& target) {
        std::string commit_hash;
        if (fs::exists(refs_dir + "/" + target)) {
            commit_hash = read_file(refs_dir + "/" + target);
            write_file(head_file, "ref: refs/" + target);
        } else {
            commit_hash = target;
            if (!fs::exists(objects_dir + "/" + commit_hash)) {
                std::cout << "Error: Commit not found\n";
                return;
            }
        }
        // Clear working directory
        for (auto& [file, _] : staging_area) {
            if (fs::exists(file)) fs::remove(file);
        }
        staging_area.clear();
        staged_trees.clear();
        write_file(index_file, "");
        // Restore files from commit
        std::string content = read_file(objects_dir + "/" + commit_hash);
        std::string tree_hash;
        std::stringstream ss(content);
        std::string line;
        while (std::getline(ss, line)) {
            if (line.find("tree: ") == 0) {
                tree_hash = line.substr(6);
```

```
                break;
            }
        }
        std::unordered_map<std::string, std::string> files;
        collect_files(tree_hash, "", files);
        for (auto& [path, hash] : files) {
            std::string file_content = read_file(objects_dir + "/" +
hash);
            write_file(path, file_content);
            staging_area[path] = hash;
            build_tree_hierarchy(path, hash);
        }
      // updating file of index
        std::stringstream index;
        for (auto& [f, h] : staging_area) index << f << " " << h << "\n";
        write_file(index_file, index.str());
        std::cout << "Checked out " << target << "\n";
    }
      // finds lowest common ancestor of 2 commits by using depth search
method


    std::string find_lca(const std::string& c1, const std::string& c2) {
        std::unordered_set<std::string> ancestors;
        std::function<void(const std::string&)> collect = [&](const
std::string& commit) {
            std::string current = commit;
            while (!current.empty()) {
                ancestors.insert(current);
                std::string content = read_file(objects_dir + "/" +
current);
                std::stringstream ss(content);
                std::string line, parent;
                while (std::getline(ss, line)) {
                    if (line.find("parent: ") == 0) {
                        parent = line.substr(8);
                        break;
                    }
                }
```

```
                current = parent;
            }
        };
        collect(c1);
        std::string current = c2;
        while (!current.empty()) {
            if (ancestors.count(current)) return current;
            std::string content = read_file(objects_dir + "/" + current);
            std::stringstream ss(content);
            std::string line, parent;
            while (std::getline(ss, line)) {
                if (line.find("parent: ") == 0) {
                    parent = line.substr(8);
                    break;
                }
            }
            current = parent;
        }
        return ""; // if there is no common ancestor being found
    }


    void merge(const std::string& branch) {
        if (!fs::exists(refs_dir + "/" + branch)) {
            std::cout << "Error: Branch not found\n";
            return;
        }
        std::string head = read_file(head_file).substr(5);
        std::string c1 = read_file(refs_dir + "/" + head);
        std::string c2 = read_file(refs_dir + "/" + branch);
        std::string lca = find_lca(c1, c2);
        if (lca.empty()) {
            std::cout << "Error: No common ancestor\n";
            return;
        }
      // processing file trees for commits
        std::unordered_map<std::string, std::string> files1, files2,
files_lca;
```

```
        std::function<void(const std::string&,
std::unordered_map<std::string, std::string>&)> load = [&](const
std::string& commit, auto& files) {
            std::string content = read_file(objects_dir + "/" + commit);
            std::string tree_hash;
            std::stringstream ss(content);
            std::string line;
            while (std::getline(ss, line)) {
                if (line.find("tree: ") == 0) {
                    tree_hash = line.substr(6);
                    break;
                }
            }
            collect_files(tree_hash, "", files);
        };
        load(c1, files1);
        load(c2, files2);
        load(lca, files_lca);
        bool conflict = false;
      //Compare files and head to apply changes
        for (auto& [path, hash2] : files2) {
            auto it1 = files1.find(path);
            auto it_lca = files_lca.find(path);
            std::string hash1 = it1 != files1.end() ? it1->second : "";
            std::string hash_lca = it_lca != files_lca.end() ?
it_lca->second : "";
            if (hash1 != hash_lca && hash2 != hash_lca && hash1 != hash2)
{
                std::cout << "CONFLICT: both modified " << path << "\n";
                conflict = true;
                continue;
            }
            if (hash2 != hash_lca) {
                std::string content = read_file(objects_dir + "/" +
hash2);
                write_file(path, content);
                staging_area[path] = hash2;
                build_tree_hierarchy(path, hash2);
            }
```

```
        }
        if (conflict) {
            std::cout << "Merge failed due to conflicts\n";
            return;
        }
       // create merge commit
        build_tree_hierarchy("");
        std::string tree_hash = create_tree(".");
        std::stringstream commit;
        std::string timestamp = get_time();
        commit << "tree: " << tree_hash << "\n";
        commit << "parent: " << c1 << "," << c2 << "\n";
        commit << "timestamp: " << timestamp << "\n";
        commit << "message: Merge branch '" << branch << "'\n";
        std::string hash = compute_sha1(commit.str());
        write_file(objects_dir + "/" + hash, commit.str());
        write_file(refs_dir + "/" + head, hash);
        staging_area.clear();
        staged_trees.clear();
        write_file(index_file, "");
        std::cout << "Merged " << branch << "\n";
    }
      // display differences between commits
    void diff(const std::string& c1, const std::string& c2 = "") {
        std::string commit1 = c1;
        if (fs::exists(refs_dir + "/" + c1)) commit1 = read_file(refs_dir
+ "/" + c1);
        std::string commit2 = c2;
        if (!c2.empty() && fs::exists(refs_dir + "/" + c2)) commit2 =
read_file(refs_dir + "/" + c2);
        std::unordered_map<std::string, std::string> files1, files2;
        if (!fs::exists(objects_dir + "/" + commit1)) {
            std::cout << "Error: Commit not found\n";
            return;
        }
        std::string tree1;
        std::string content = read_file(objects_dir + "/" + commit1);
        std::stringstream ss(content);
        std::string line;
```

```
        while (std::getline(ss, line)) {
            if (line.find("tree: ") == 0) {
                tree1 = line.substr(6);
                break;
            }
        }
        collect_files(tree1, "", files1);
        if (c2.empty()) {
            for (auto& [path, hash] : staging_area) files2[path] = hash;
        } else {
            if (!fs::exists(objects_dir + "/" + commit2)) {
                std::cout << "Error: Commit not found\n";
                return;
            }
            std::string tree2;
            content = read_file(objects_dir + "/" + commit2);
            ss.clear();
            ss.str(content);
            while (std::getline(ss, line)) {
                if (line.find("tree: ") == 0) {
                    tree2 = line.substr(6);
                    break;
                }
            }
            collect_files(tree2, "", files2);
        }
       // collect all unique paths of file
        std::unordered_set<std::string> all_paths;
        for (auto& [p, _] : files1) all_paths.insert(p);
        for (auto& [p, _] : files2) all_paths.insert(p);
        for (auto& path : all_paths) {
            std::string h1 = files1.count(path) ? files1[path] : "";
            std::string h2 = files2.count(path) ? files2[path] : "";
            if (h1 == h2) continue;
            std::vector<std::string> lines1, lines2;
             // read file
            if (!h1.empty()) {
                std::string content = read_file(objects_dir + "/" + h1);
                std::stringstream ss(content);
```

```
                std::string line;
                while (std::getline(ss, line)) lines1.push_back(line);
            }
            if (!h2.empty()) {
                std::string content = c2.empty() ? read_file(path) :
read_file(objects_dir + "/" + h2);
                std::stringstream ss(content);
                std::string line;
                while (std::getline(ss, line)) lines2.push_back(line);
            }
            std::cout << "--- " << path << "\n+++ " << path << "\n";
            for (size_t i = 0, j = 0; i < lines1.size() || j <
lines2.size();) {
                if (i < lines1.size() && j < lines2.size() && lines1[i] ==
lines2[j]) {
                    std::cout << " " << lines1[i] << "\n";
                    ++i; ++j;
                } else if (j < lines2.size() && (i >= lines1.size() ||
lines1[i] != lines2[j])) {
                    std::cout << "+" << lines2[j] << "\n";
                    ++j;
                } else {
                    std::cout << "-" << lines1[i] << "\n";
                    ++i;
                }
            }
        }
    }
};
// parses command line arguments and implement the commands
int main(int argc, char* argv[]) {
    if (argc < 2) {
        std::cout << "Usage: minigit <command> [args...]\n";
        return 1;
    }
    MiniGit mg;
    std::string cmd = argv[1];
    if (cmd == "init") {
        mg.init();
```

```
    } else if (cmd == "add" && argc == 3) {
        mg.add(argv[2]);
    } else if (cmd == "commit" && argc == 4 && std::string(argv[2]) ==
"-m") {
        mg.commit(argv[3]);
    } else if (cmd == "log") {
        mg.log();
    } else if (cmd == "branch" && argc == 3) {
        mg.branch(argv[2]);
    } else if (cmd == "checkout" && argc == 3) {
        mg.checkout(argv[2]);
    } else if (cmd == "merge" && argc == 3) {
        mg.merge(argv[2]);
    } else if (cmd == "diff" && (argc == 3 || argc == 4)) {
        if (argc == 3) mg.diff(argv[2]);
        else mg.diff(argv[2], argv[3]);
    } else {
        std::cout << "Invalid command or arguments\n";
        return 1;
    }
    return 0;
}


            }
        }
        collect_files(tree1, "", files1);
        if (c2.empty()) {
            for (auto& [path, hash] : staging_area) files2[path] = hash;
        } else {
            if (!fs::exists(objects_dir + "/" + commit2)) {
                std::cout << "Error: Commit not found\n";
                return;
            }
            std::string tree2;
            content = read_file(objects_dir + "/" + commit2);
            ss.clear();
            ss.str(content);
            while (std::getline(ss, line)) {
                if (line.find("tree: ") == 0) {
```

```
                    tree2 = line.substr(6);
                    break;
                }
            }
            collect_files(tree2, "", files2);
        }
    // collect all unique paths of file
        std::unordered_set<std::string> all_paths;
        for (auto& [p, _] : files1) all_paths.insert(p);
        for (auto& [p, _] : files2) all_paths.insert(p);
        for (auto& path : all_paths) {
            std::string h1 = files1.count(path) ? files1[path] : "";
            std::string h2 = files2.count(path) ? files2[path] : "";
            if (h1 == h2) continue;
            std::vector<std::string> lines1, lines2;
        // read file
            if (!h1.empty()) {
                std::string content = read_file(objects_dir + "/" + h1);
                std::stringstream ss(content);
                std::string line;
                while (std::getline(ss, line)) lines1.push_back(line);
            }
            if (!h2.empty()) {
                std::string content = c2.empty() ? read_file(path) :
read_file(objects_dir + "/" + h2);
                std::stringstream ss(content);
                std::string line;
                while (std::getline(ss, line)) lines2.push_back(line);
            }
            std::cout << "--- " << path << "\n+++ " << path << "\n";
            for (size_t i = 0, j = 0; i < lines1.size() || j <
lines2.size();) {
                if (i < lines1.size() && j < lines2.size() && lines1[i] ==
lines2[j]) {
                    std::cout << " " << lines1[i] << "\n";
                    ++i; ++j;
                } else if (j < lines2.size() && (i >= lines1.size() || li
nes1[i] != lines2[j])) {
                    std::cout << "+" << lines2[j] << "\n";
```

```
                    ++j;
                } else {
                    std::cout << "-" << lines1[i] << "\n";
                    ++i;
                }
            }
        }
    }
};
// parses command line arguments and implement the commands
int main(int argc, char* argv[]) {
    if (argc < 2) {
        std::cout << "Usage: minigit <command> [args...]\n";
        return 1;
    }
    MiniGit mg;
    std::string cmd = argv[1];
    if (cmd == "init") {
        mg.init();
    } else if (cmd == "add" && argc == 3) {
        mg.add(argv[2]);
    } else if (cmd == "commit" && argc == 4 && std::string(argv[2]) ==
"-m") {
        mg.commit(argv[3]);
    } else if (cmd == "log") {
        mg.log();
    } else if (cmd == "branch" && argc == 3) {
        mg.branch(argv[2]);
    } else if (cmd == "checkout" && argc == 3) {
        mg.checkout(argv[2]);
    } else if (cmd == "merge" && argc == 3) {
        mg.merge(argv[2]);
    } else if (cmd == "diff" && (argc == 3 || argc == 4)) {
        if (argc == 3) mg.diff(argv[2]);
        else mg.diff(argv[2], argv[3]);
    } else {
        std::cout << "Invalid command or arguments\n";
        return 1;
    }
```

```
    return 0;
}
```

In this project, we developed MiniGit, a lightweight version control system that captures the essence of Git's local file tracking, commits, branching, and merging using fundamental data structures like DAGs and hashing. While it lacks remote capabilities and advanced Git features, MiniGit serves as a practical tool to understand the underlying mechanics of version control systems and strengthen programming skills in file I/O, data structures, and software design.