CSCI 2275: Data Structures - Project Specifications
MiniGit (A mini version control system)

Due December 1, 11:59 PM MDT

1 Version Control System

A *version control system*, also known as revision control system, is a class of systems responsible for managing changes to a set of documents typically containing computer programs, documents, web sites, or other collections of information. Version control systems track the changes users make to files, so users have a historical record of all of the changes, and they can revert to specific versions should they ever need to. Version control systems also make collaboration easier, allowing changes by multiple people to all be merged into one source. In this project, you are required to implement a toy version-control system that we call `minigit`. This document provides minimum capability your tool should have, but you are encouraged to add more functionality to bring it closer in capability to well-known version control systems such as `git`, `mercurial` or `cvs` (Concurrent Versions System).

2 Phase I: Core Features

2.1 Overview

For the first phase of the project, you will need to create a `miniGit` program with the following core functionality:

1. Initialising a new repo
2. Adding files to the current commit
3. Removing files from the current commit
4. Committing changes
5. Searching commits based on a keyword
6. Checking out a specific version based on a unique commit number

This project is intended to be more open-ended than the preceding course assignments. With that, you are given minimal starter code. You will have to create your own test cases, for which you will need to generate a set of test files. Your `miniGit` repository needs to be able to accept files of .cpp, .hpp, and .txt type (i.e. you do not need to worry about PDF or executable files, etc.)

2.2 Implementation Requirements

2.2.1 User interface

The user shall interact with the program via a list of choices presented in a textual menu. The program should continue running indefinitely until the user chooses to quit. The menu is to be implemented using a switch statement and a while loop (as has been done for the assignments in the class thus far.)

2.2.2 Initializing a new repository

Executing the program will prompt the user with an option to initialize an empty repository in the current directory. The `init` function will take one argument, the size of the hash table. Pass this value from the main function. For submission code, all students are asked to keep their table size fixed at 5.

In order to create or delete directories (folders) from within a C++ program, you may use the filesystem library (https://en.cppreference.com/w/cpp/filesystem) by using the following sequence of code. In order to use this library, you will need to compile your code with the `g++ -std=c++17` command in a standard Mac or Linux terminal:

```
#include <filesystem>
namespace fs = std::filesystem;

 ....

fs::remove_all(".minigit"); // removes a directory and its contents
fs::create_directory(".minigit");  // create a new directory
```

Note that you are not allowed to use the `copy` function from the filesystem library. You must implement your own file-copy procedure.

If the user chooses to initialize, a doubly linked list will be created with a single head node. **Going forward, each doubly linked list (DLL) node will correspond to a single commit.** Each DLL node will contain a member with a unique commit number as well as a head pointer to a singly linked list (SLL). The first node in the DLL should have a commit number of 0. Each DLL node will also store a commit message.

It is suggested that you define the DLL and SLL structs as follows:

```
struct BranchNode {
    int commitID;
    string commitMessage; // no more than three words
```

```
    BranchNode* next; //DLL next node
    BranchNode* previous; //DLL previous node
    FileNode* fileHead;  //SLL head
};

struct FileNode {
    string name; //name of the local file
    int version; //version of the file in .minigit
    FileNode* next; //sll next node
};
```

The SLL will then be used to store a list of files in the current commit. The initialization step will also create a new sub-directory within the current directory called `.minigit`. The user will then be given the following choices: (1). Add file, (2). Remove file, (3). Commit, and (4). Checkout.

2.2.3 Adding A File

If the user chooses to add a file to the repository, the following sequence should occur:

1. Prompt user to enter a file name.
2. Check whether the file with the given name exists in the current directory. If not, keep prompting the user to enter a valid file name.
3. The SLL is checked to see whether the file has already been added. A file by the same name cannot be added twice.
4. A new SLL node gets added containing the name of the input file, name of the repository file, as well as a pointer to the next node. The repository file name should be the combination of the original file name and the version number. For example, if user file `help.txt` is added, the new file to be saved in the `.minigit` repository should be named `help00.txt`, where `00` is the version number. (The initial file version should be `00`.) The above naming system is just a suggestion. You may implement your own naming convention. For instance, you may choose to call the new file in the `.minigit` repositry as `help.txt__k` or `__k__help.txt` where k is the version number.

2.2.4 Removing a file

If the user chooses to remove a file from the repository, the following steps should take place:

1. Prompt user to enter a file name.
2. Check the SLL for whether the file exists in the current version of the repository.
3. If found, delete the SLL node.

2.2.5 Committing Changes

Once the user chooses to commit their changes, the following steps need to be taken:

1. Ask user for a commit message. A commit message can have at most three words (space separated). No two DLL node should have exact same commit messages. However there could be overlap in terms of the sub-strings. For example `"computer science"` and `"computer science fun"`. **If the user enters an invalid commit message, prompt user until the system gets a valid commit message. Populate the commit message field of the current DLL node**.

2. The current SLL should be traversed in its entirety, and for every node. Check whether the corresponding `fileVersion` file exists in `.minigit` directory. Then, do one of two things:

   (a) If the `fileVersion` file *does not* exist, copy the file from the current directory into the `.minigit` directory. The newly copied file should get its name from the file name combined with the node's `fileVersion` member. (Note: this will only be the case when a file is added to the repository for the first time.)

   (b) If the `fileVersion` file *does* exist in `.minigit`, check whether the current directory file has been changed (i.e. has it been changed by the user?) with respect to the `fileVersion` file. (To do the comparison, you can read in the file from the current directory into one string and read in the file from the `.minigit` directory into another string, and check for equality.) Based on the comparison result, do the following:

      i. File is unchanged: do nothing.

      ii. File is changed: copy the file from the current directory to the `.minigit` directory, and give it a name with the incremented version number. Also, update the SLL node member `fileVersion` to the incremented name.

3. Parse the commit message by breaking up the string into separate sub-strings. This will be required for the search function.

   (a) Use a single white space as the delimiter.

   (b) For each word (sub-string) in the commit message:

      i. Calculate the hash function for the key.

      ii. Retrieve the chain and scan the chain for the key word.

      iii. If the a node with the key is found, then add the current commit number to the `commitNumber` vector.

      iv. Otherwise, create a new node with the key. Add the current commit number to the `commitNumber`. Add the node to the chain.

4. Once all the files have been scanned, the final step of the commit will create a new Doubly Linked List node. An exact (deep) copy of the SLL from the previous node

shall be copied into the new DLL node. The commit number of the new DLL node will be the previous node's commit number incremented by one. Make the commit message of this newly created DLL empty.

2.2.6 Checkout

At any point, the user should be able to check out any previous version of the repository. If the user chooses to check out a version, they should be prompted to enter a commit number. For a valid commit number, the files in the current directory should be overwritten by the the corresponding files from the `.minigit` directory. (It is a good idea to issue a warning to the user that they will lose their local changes if they check out a different version before making a commit with their current local changes.)

This step will require a search through the DLL for a node with a matching commit number. Also, note that you must disallow add, remove, and commit operations when the current version is different from the most recent commit (the last DLL node).

2.2.7 Searching with a Hash Table

The commit function will ask users to provide a short message (string) with every commit. This string will be used for the implementation of the search function. A search key will be a string of single word. Implement a *minigit search* function that should take a search key and print all commit IDs whose message contains the given search key. Implement this search function efficiently using a hash-table where keys of the hash-table will correspond to the individual words (sub-strings) in the commit message string and the values should be the commit IDs. The collision resolution mechanism should utilize linked list based chaining. The implementation of the Hash Table class will look like this:

```
struct  hashNode{
    std:: string key;
    std:: vector<int> commitNumber;
    struct hashNode* next;
}

class HashTable{
    int tableSize;    // No. of buckets (linked lists)
    hashNode* *table; // Pointer to an array
                      // containing lists of hashNodes
    int hashFunction(string key);
    ...
}

int HashTable::hashFunction(string s)
{
    int sum=0,index=0;
    for(string::size_type i=0; i < s.length(); i++)
```

```
        {
            sum += s[i];
        }
        index = sum % tableSize;
        return index;
    }
```

Entries will be added to the hash-table during each commit.

2.3 Example Flow

Once the user starts the `minigit` program, they are prompted to initialize a new repository. If they choose to do so, a new DLL is created, with a single node. Also, a new `.minigit` sub-directory is created in the current directory. Figure 1 shows a diagram of what the data structure should look like after the initialization.

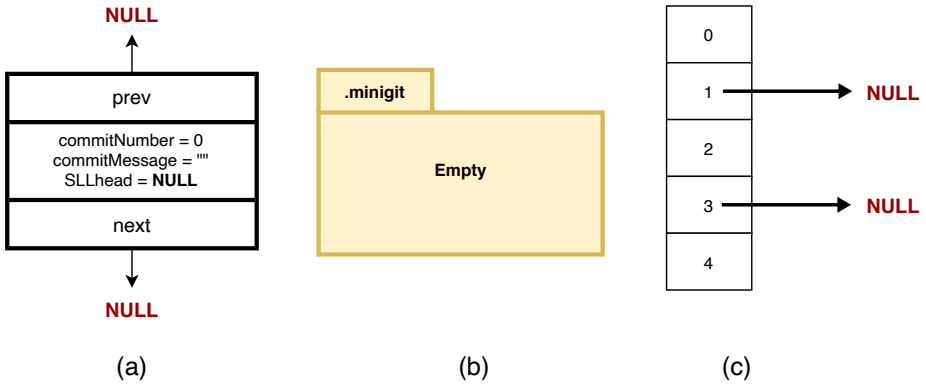

Figure 1: Repository data structure after initialization. a) The initial DLL state b) The state of minigit folder c) the hash table is also empty at this point

The user then gets the option to add files to the repository. For example, they choose to add files `f1.txt` and `f2.txt`. These actions result in two new SLL nodes being created. The diagram in Figure 2 illustrates the state of the data structure after the two adds. The figure also shows that there are currently no files in the `.minigit` directory.

Next, the user decides to commit their changes. The user is prompted for a valid commit message. For example, the commit message is `"computer science"`.

After the commit a new DLL node is created, with the SLL copied from the previous DLL node. The state of the data structure right after the commit is visualized in the Figure 3 diagram `a`. The sub-figure `b` also shows the files that are now present in the `.minigit` sub-directory. Assume that the word `"science"` got a hash value of 1 and `"computer"` got 3. Figure 3 (c) shows the hash-table.

Let us say that the user then decides to do the following:

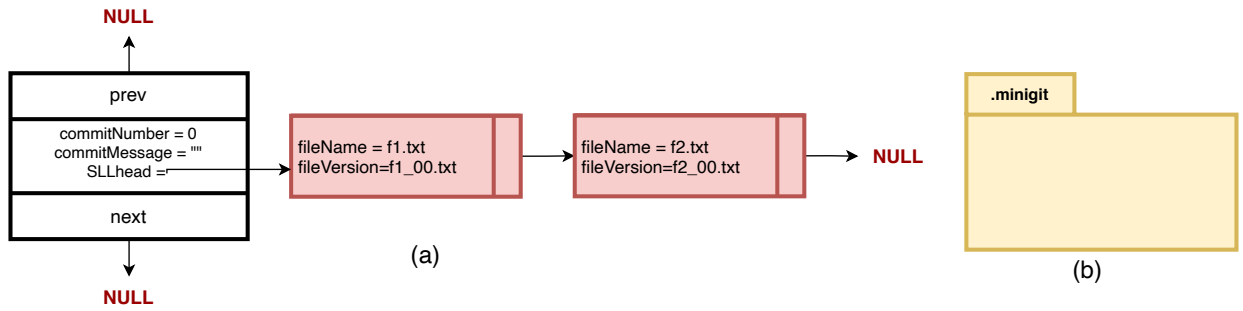

Figure 2: Repository data structure after adding `f1.txt` and `f2.txt`. a) The current state of DLL b) The state of minigit folder. It will contain copies of files with version number appended to their names

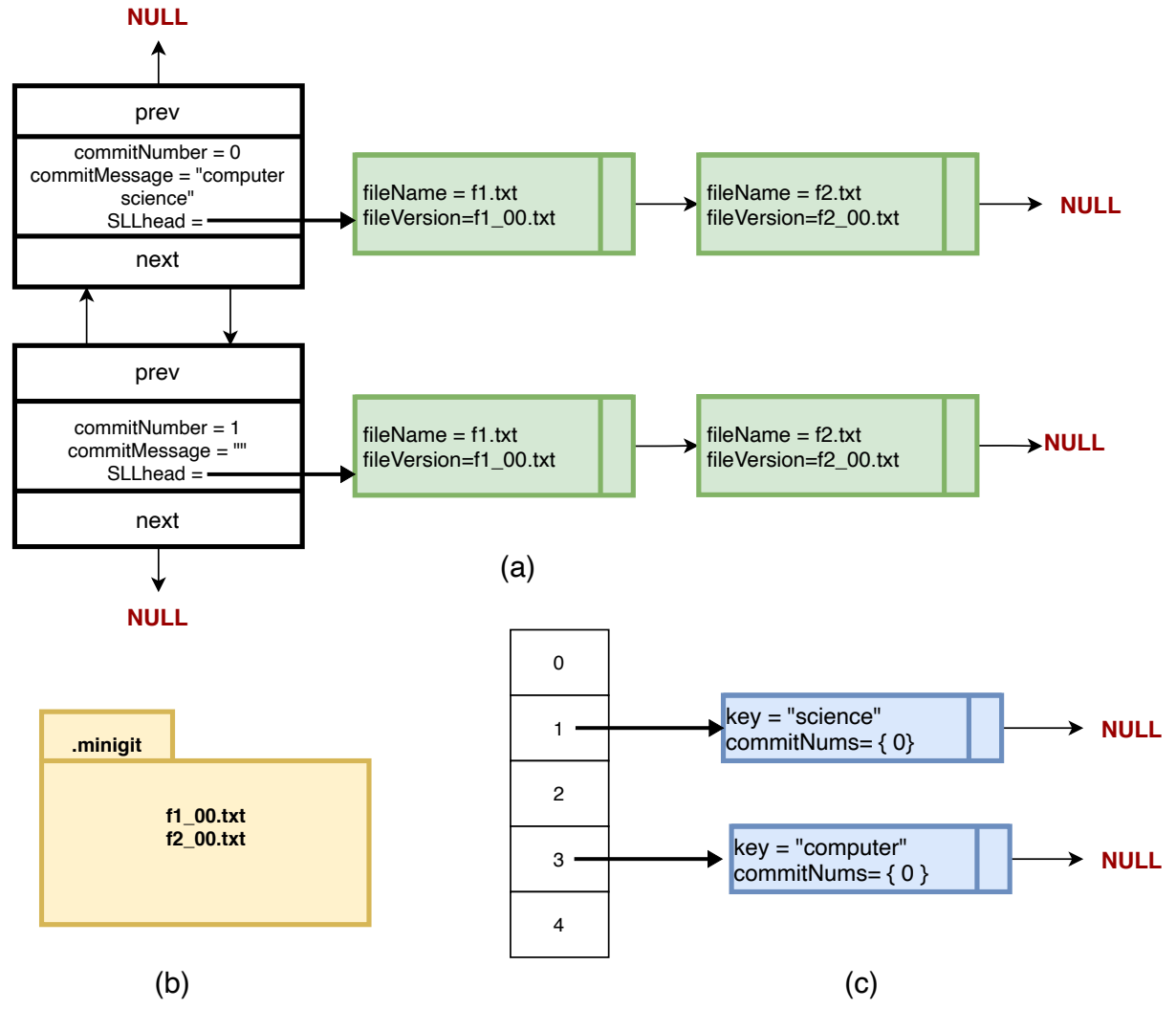

Figure 3: Repository data structure after the first commit. a) The current state of DLL b) The state of minigit folder.c) the hash table

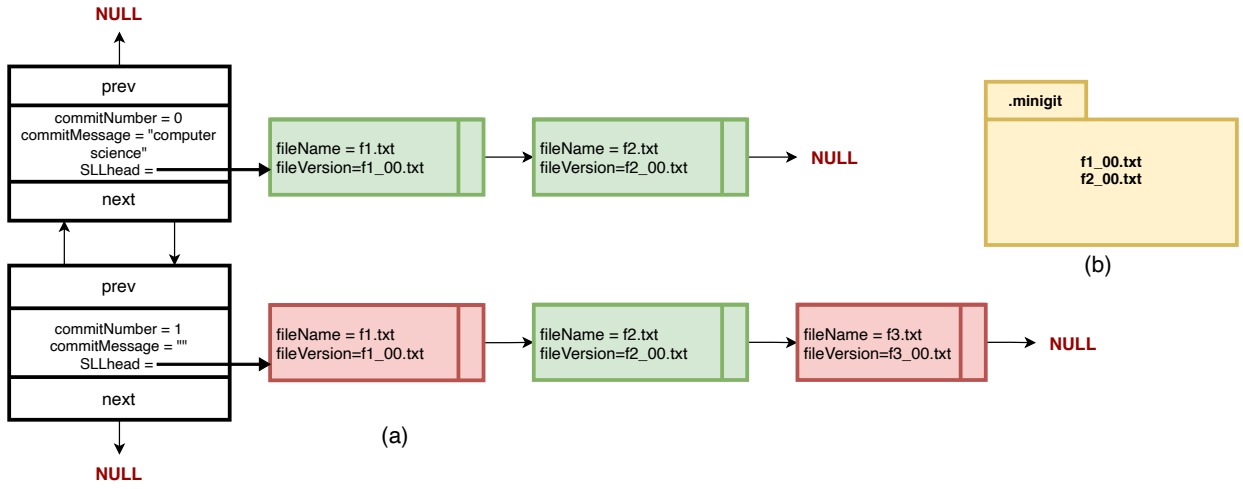

Figure 4: Repository data structure after user actions. *Note: string member of the SLL node corresponding to* `f1_00.txt` *remains unchanged until commit.*

- Make some changes to `f1.txt`
- Do nothing to `f2.txt`
- Add a new file (`f3.txt`).

The SLL that is pointed to by the second DLL node should be used to keep track of these changes. Note that a new node will be created as soon as the user issues an add with `f3.txt`. However, the change to `f1.txt` will not be recorded until the user makes a new commit. The diagram in Figure 4 shows the state of the data structure after the aforementioned user actions.

The user issues a second commit with commit message `"science fun"`. Recall that the commit will traverse the entire corresponding SLL, checking each file against the most recent repository version. Because a change is detected in `f1.txt`, a copy of the file gets saved to the `.minigit` directory. The name of the `fileVersion` member also gets changed to `f1_01.txt`, to reflect the incremented file version. `f2.txt` is unchanged, so the node stays unaltered. `f3.txt` has been newly added, so the initial version of the file gets copied to the `.minigit` directory. Figure 5 shows the post-commit resulting diagram. Assume the hash function for key `fun` is 3. Figure 5 (c) shows the hash-table after this commit.

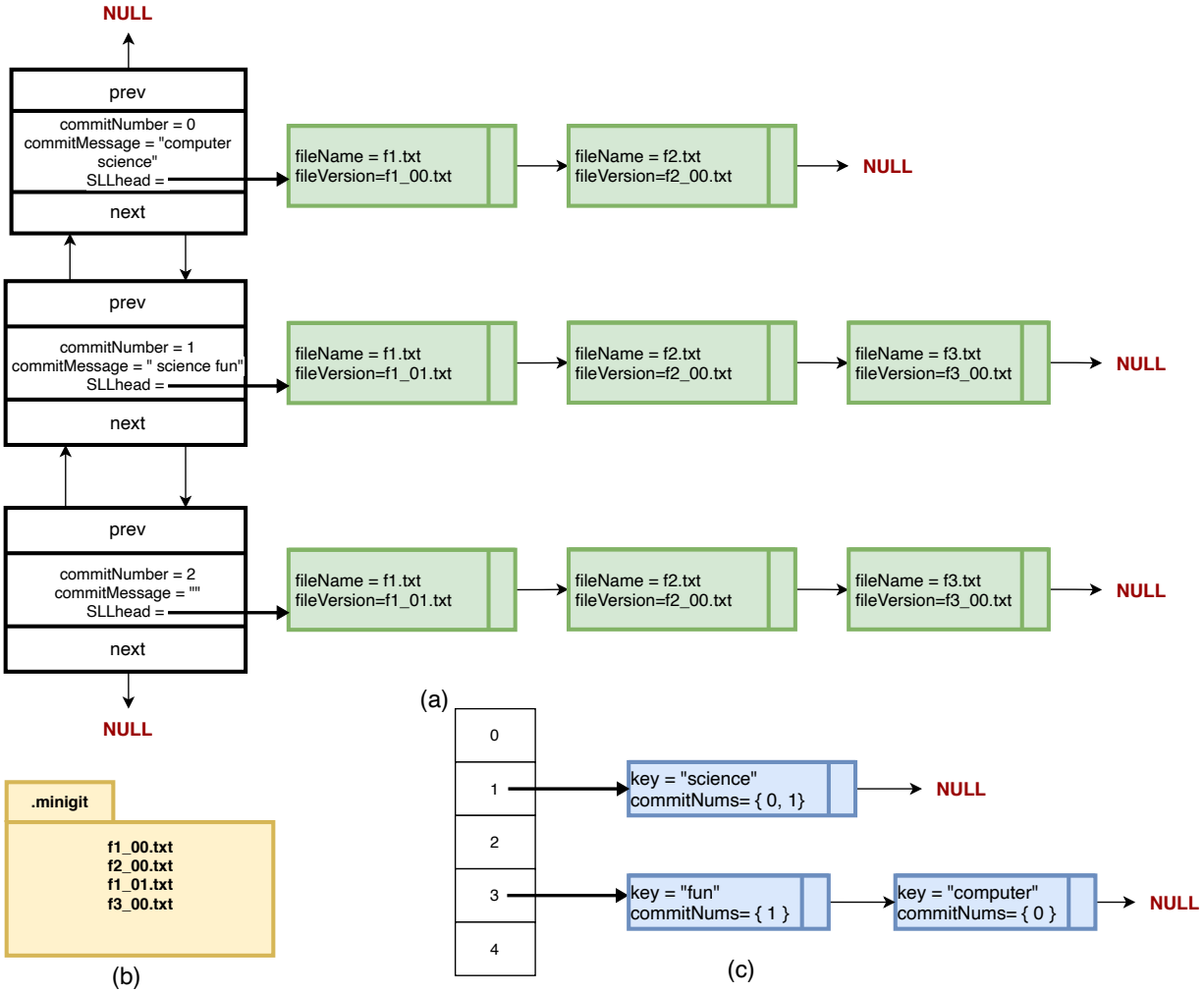

Figure 5: Repository data structure after the second commit

3 Phase II: Additional Features for Extra Credits

We suggest the following additional features that you may choose to implement for your own experience.

1. **Branching**: Implement a version of `git branch` where users can manage separate branches of their files in your `minigit` repository. You are not required to implement merge for those branches. So far you have implemented only a single branch in your minigit implementation, often called the `master` branch. You can implement the branch feature by simply keeping a list of various branches (instead of implementing a tree-like structure). Each branch should be uniquely identified by a name. The commit IDs should be unique within a branch. Hence your checkout should specify both the branch name and the commit ID.

2. **Friendly Checkouts.** Implement a special checkout `git checkout HEAD` to help users to move back to the most recent commit version. Moreover, allow your users to revert back to the previous version of the code (more like an undo feature) by implementing `checkout` with special commit id, e.g. `git checkout -` (checkout followed by a dash). Finally, allow your users to go to the commit i-steps back in the past (`git checkout HEAD~3`).

3. **Diff and Status**. Implement the `minigit diff` feature that takes a file and prints the first difference (print the whole line) with respect to the most recent commit. Also, implement `minigit status` feature that lists all of the files that have been changed since the last commit.

4. **Serialization and Deserialization.** Your current minigit implementation is not state-ful, i.e. it is not required to keep information between different invocations from within a same directory. In this feature you are required to make your implementation state-ful by serializing and de-serializing your data-structure.

   Serialization is the process of translating a data structure or object state into a format that can be stored (e.g., in a file) and reconstructed later. The process of storing a data-strucutre to a file is called serialization, and the process of re-constructing the data-structure from a file is called deserialization. For this feature, you are required to implemented both serialization and deserialization of your minigit repository so that you can use the minigit using the commandline.

   One way to serialize the repository shown in figure 5 is the following xml-like encoding.

```
<branch dir=".minigit">
    <commit id = 0>
        <files>
            <file src="f1.txt"> f1.txt_00 </file>
            <file src="f2.txt"> f2.txt_00 </file>
        </files>
    </commit>
```

```
    <commit id = 1>
        <files>
            <file src="f1.txt"> f1.txt_01 </file>
            <file src="f2.txt"> f2.txt_00 </file>
            <file src="f3.txt"> f3.txt_00 </file>
        </files>
    </commit>
    <commit id = 2>
        <files>
            <file src="f1.txt"> f1.txt_01 </file>
            <file src="f2.txt"> f2.txt_00 </file>
            <file src="f3.txt"> f3.txt_00 </file>
        </files>
    </commit>
<\branch>
```

This can be accomplished by a traversal to the Doubly-linked list (of commit nodes) and for every commit node another traversal to the singly-linked lists or the files. This xml file can be stored in the .minigit folder with the name (minigit.xml) when program exits. Similarly, such a file can be read the minigit data-structure can be recreated when the program initializes again. Be careful not to delete the contents of the minigit folder between different invocations.

4 Project Submission and Grading

4.1 Deliverables

In order for your project to be graded, it must written in C++ and compile with the `g++ -std=c++17` command in a standard Mac or Linux terminal. **You can work individually or in a group of up to two students.** If you work in a group, a single submission is to be made. The final submission should contain your source code files with the functioning class-based implementation of the `miniGit` program. **The last commit you make to your GitHub repository prior to the deadline will be used for grading.** Your files should be named in the following manner:

- `miniGit.hpp` - header file
- `miniGit.cpp` - implementation file
- `hash.hpp` - header file for hash implementation
- `hash.cpp` - hash table implementation file
- `main_1.cpp` - driver file containing the user interface

- `readme.md` - description of program functionality and special features implemented in the project; **names of team members**

Along with these you have to create test-cases in `test.cpp` for your hash table implementation utilizing the Google Test suite. Specifically, you should have thorough test cases for `insert` and `search` functions of `hash.cpp`. Please refer to the examples given in lecture as well as any of the assignments to see example test-cases. (Note that many of the assignments have additional helper code in separate files other than `test.cpp`. You are not required to do this, and can write all of your tests within `test.cpp`)

You should first focus on completing the core functionality of Phase I. Please only keep code required for Phase I implementations in your master branch of the GitHub repository. Any additional Phase II functionality you choose to add should be in a separate branch. If you are unsure how to create and work on a branch, please ask your TA or Professor.

You may choose to include 3-4 of your test files in the submission, although this is not required.