CSCI 2270: Data Structures - Project Specifications
MiniGit (A mini version control system)

Due Tuesday, April 27, 11:59 PM

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. Adding files to the current commit
2. Removing files from the current commit
3. Committing changes
4. 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 not given any 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. 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 `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.

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

```
struct doublyNode{
    int commitNumber;
    singlyNode * head;
    doublyNode * previous;
    doublyNode * next;
}

struct singlyNode{
    std::string fileName;    // Name of local file
    std::string fileVersion; // Name of file in .minigit folder
    singlyNode * next;
}
```

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. The current SLL should be traversed in its entirety, and for every node

(a) Check whether the corresponding `fileVersion` file exists in `.minigit` directory. If the `fileVersion` file *does not* exist, copy the file from the current directory into the `.minigit` directory. The newly copied file should get the name from 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:

- File is unchanged: do nothing.
- 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.

2. Once all the files have been scanned, the final step of the commit will create a new Doubly Linked List node of the repository. 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 nodes commit number incremented by one.

2.2.6 Checkout

At any point, the user should be able to checkout any previous version of the repository. If the user chooses to checkout 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 loose their local changes if they checkout 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 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).

2.3 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.

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 will 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.

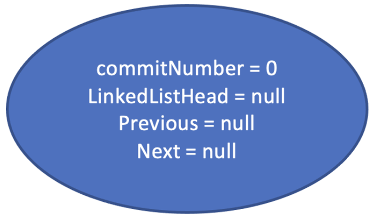

Figure 1: Repository data structure after initialization

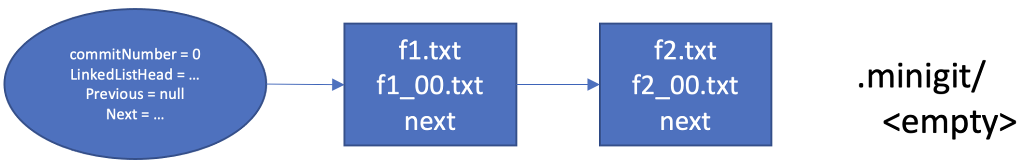

Figure 2: Repository data structure after adding `f1.txt` and `f2.txt`

Next, the user decides to commit their changes. 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. The figure also shows the files that are now present in the `.minigit` sub-directory.

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

- 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. 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

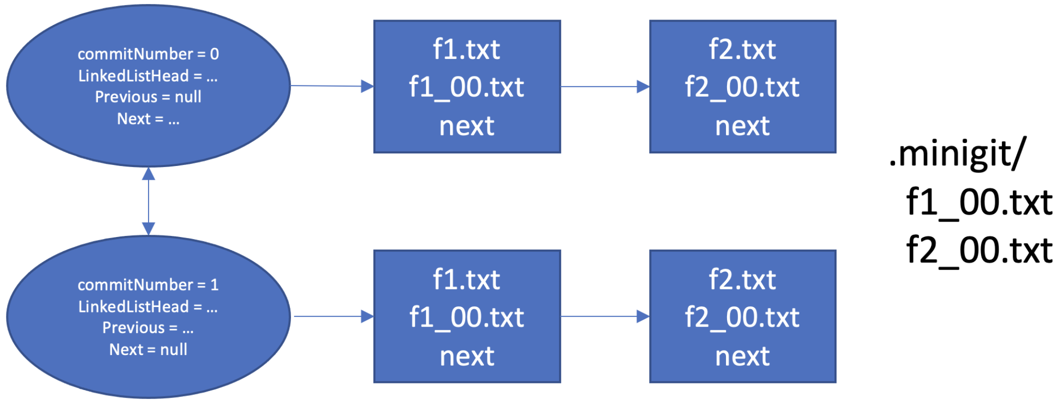

Figure 3: Repository data structure after the first commit

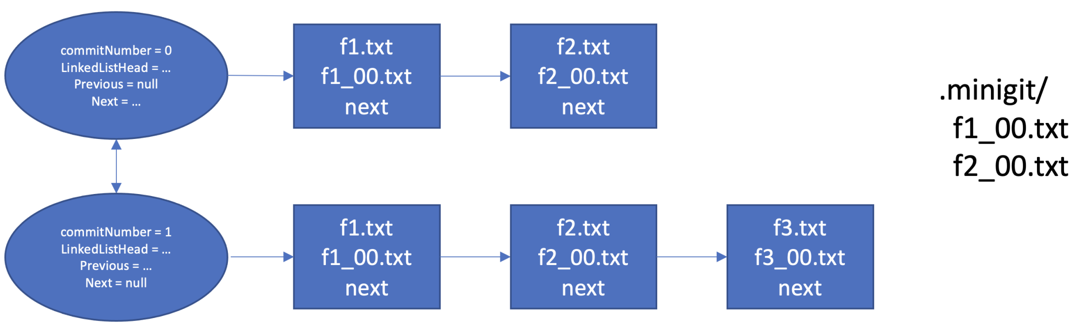

Figure 4: Repository data structure after user actions. *Note: string member of the SLL node corresponding to `f1_00.txt` remains unchanged until commit.*

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.

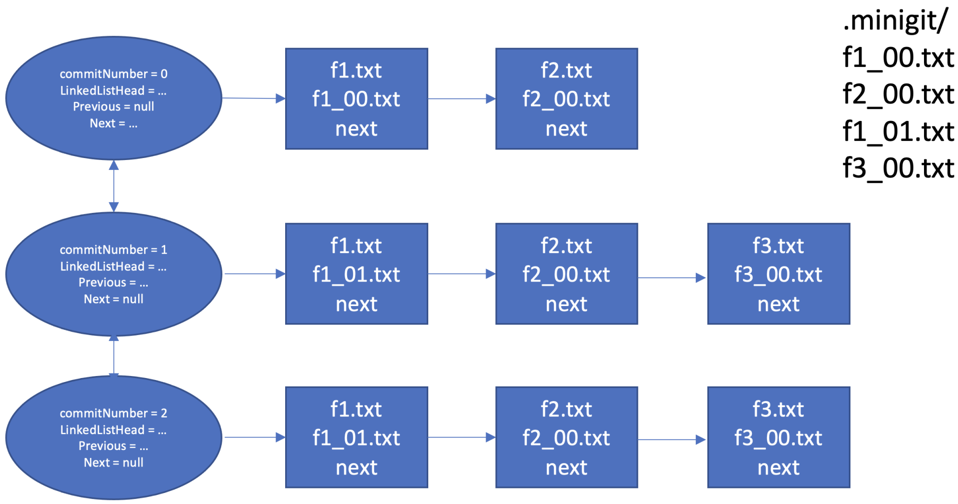

Figure 5: Repository data structure after the second commit

3 Phase II: Additional Features for Extra Credits

We suggest the following additional features that you may choose to implement for extra credits (Max of 5% and 1% per completed extension.). In order to qualify for extra credit, each extension must be fully completed, documented (in the readme file), and demo'ed to TA furing the interview grading. TAs will have discretion whether to assign points or not. (e.g. if a library function is used to make the extension trivial, no points will be awarded).

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. **Commit with a Message and Efficient Search**: Extend your commit function by allowing users to provide a short message (string) with every commit, and implement a `minigit search` function that should take a string and print all commit IDs whose message contains the given search string. Implement this search function efficiently using a hash-table where keys of the hashtable will correspond to the individual words in the commit message string and values should be the commit IDs.

3. **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`).

4. **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.

5. **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 three 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. Your files should be named in the following manner:

- `miniGit.hpp` - header file
- `miniGit.cpp` - implementation file
- `driver.cpp` - driver file containing the user interface
- `readme.txt` - description of program functionality and special features implemented in the project

You should first focus on completing the core functionality of Phase I. Whether or not you choose to pursue any of the additional features from Phase II, you should have one submission that **only contains the core functionality implementations from Phase I**. Name this file *<lastname>*`_Phase1.zip`, where *lastname* is the last name of one of the team members.

Any additional features from Phase II need to be submitted as a second zip file, named *<lastname>*`_Phase2.zip`. This should contain all of the source code files as the phase 1 version, such that it can be compiled and run independently. **In order for your TA to consider these features, an updated `readme` needs to describe them in detail.**

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

4.2 Interview grading

Mandatory interview grading will be conducted for this project. Details on scheduling the interviews will be forthcoming.