Navigation

Project 3: Gitlet, your own version-control system

Introduction

Welcome to Gitlet, the final project of the semester! The following spec is incredibly long and detailed; it's easily the most nuanced project that you'll work on in this class. However, *all* of the information in this spec is here for a reason. Though you may not get through all of this info on the first, second, or even third read-throughs, please keep revisiting the various tabs on the left as you progress through the project. You'll often find the answers to your questions tucked away in some section of this spec, so it behooves you to refer to this document before relying on a Piazza, Gitbug, or office hours response (as generations of students have learned the hard way).

Lastly, while this project may seem quite daunting and intense, it's one of the most rewarding parts of the CS61B experience! Students have said as much for years; the lessons and techniques that you learn from completing this project will help you immensely in future projects, courses, and even jobs. So, without further ado, let's get started!

Note: This spec is now paginated, which means that each piece of the spec has its own dedicated page! We made this change to make the spec easier to navigate; however, if you prefer the old version of the spec with everything on one page, you can find that version **here**.

Acknowledgements

Navigation

Andrew Huang, Yan Zhao, Matthew Chow, especially Alan Yao, Daniel Nguyen, and Armani Ferrante for providing feedback on this project. Thanks to git for being awesome.

This project was largely inspired by this excellent article by Philip Nilsson.

This project was created by Joseph Moghadam. Modifications for Fall 2015, Fall 2017, and Fall 2019 by Paul Hilfinger.

Useful Links

Listed below are many high quality resources compiled across multiple semesters to help you get started/unstuck on Gitlet. These videos and resources will be linked in the relevant portions of the spec, but they are here as well for your convenience. More resources may be created throughout the duration of the project as needed; if so, they will be linked here as well.

- Git Intros: These should mostly be review at this point since you have been using Git throughout the semester, but it is *vital* that you have a strong understanding of Git itself before trying to implement Gitlet. Be sure you understand the contents of these videos thoroughly before proceeding.
  - Part 1
  - Part 2
- Gitlet Intros: The introduction to our mini-version of Git, Gitlet.
  - Part 1
  - Part 2

    - Part 4
    - Slides
- OH Presentations (from Fall 2021)
    - Getting Started: Recording, Slides
    - Testing and Debugging: Recording, Slides
    - Merge: Recording, Slides
- Understanding Branch
- Understanding Merge
- Testing
- Gitlet FAQ/Help Doc
- Gitlet Review Session 1 (4/13) Video
- Gitlet Review Session 1 (4/13) Slides
- Gitlet Presentation 2 (4/20) Video
- Gitlet Presentation 2 (4/20) Slides
- Gitlet Presentation 3 (4/27) Video
- Gitlet Presentation 3 (4/27) Slides
- Staff Gitlet Testing Files

Overview of Gitlet

In this project you'll be implementing a version-control system that mimics some of the basic features of the popular system Git. Ours is smaller and simpler, however, so we have named it Gitlet.

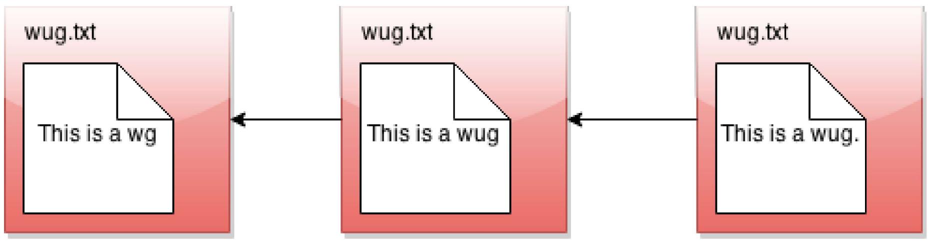
A version-control system is essentially a backup system for related collections of files. The main functionality that Gitlet

1. Saving the contents of entire directories of files. In Gitlet, this is called *committing*, and the saved contents themselves are called *commits*.
2. Restoring a version of one or more files or entire commits. In Gitlet, this is called *checking out* those files or that commit.
3. Viewing the history of your backups. In Gitlet, you view this history in something called the *log*.
4. Maintaining related sequences of commits, called *branches*.
5. Merging changes made in one branch into another.

The point of a version-control system is to help you when creating complicated (or even not-so-complicated) projects, or when collaborating with others on a project. You save versions of the project periodically. If at some later point in time you accidentally mess up your code, then you can restore your source to a previously committed version (without losing any of the changes you made since then). If your collaborators make changes embodied in a commit, you can incorporate (*merge*) these changes into your own version.

In Gitlet, you don't just commit individual files at a time. Instead, you can commit a coherent set of files at the same time. We like to think of each commit as a *snapshot* of your entire project at one point in time. However, for simplicity, many of the examples in the remainder of this document involve changes to just one file at a time. Just keep in mind you could change multiple files in each commit.

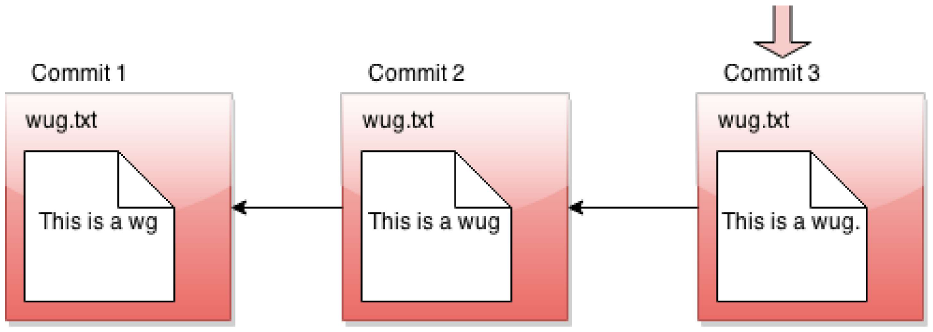
In this project, it will be helpful for us to visualize the commits we make over time. Suppose we have a project consisting just of the file `wug.txt`, we add some text to it, and commit it. Then we modify the file and commit these changes. Then we modify the three total versions of this file, each one later in time than the previous. We can visualize these commits like so:



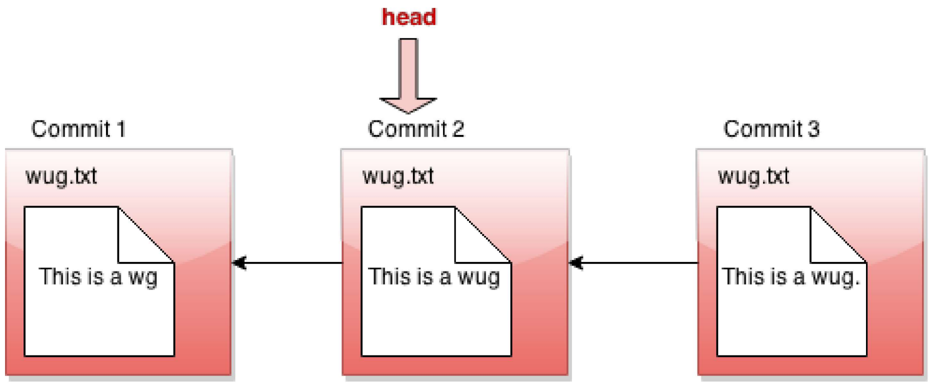
Here we've drawn an arrow indicating that each commit contains some kind of reference to the commit that came before it. We call the commit that came before it the *parent commit*—this will be important later. But for now, does this drawing look familiar? That's right; it's a linked list!

The big idea behind Gitlet is that we can visualize the history of the different versions of our files in a list like this. Then it's easy for us to restore old versions of files. You can imagine making a command like: "Gitlet, please revert to the state of the files at commit #2", and it would go to the second node in the linked list and restore the copies of files found there, while removing any files that are in the first node, but not the second.

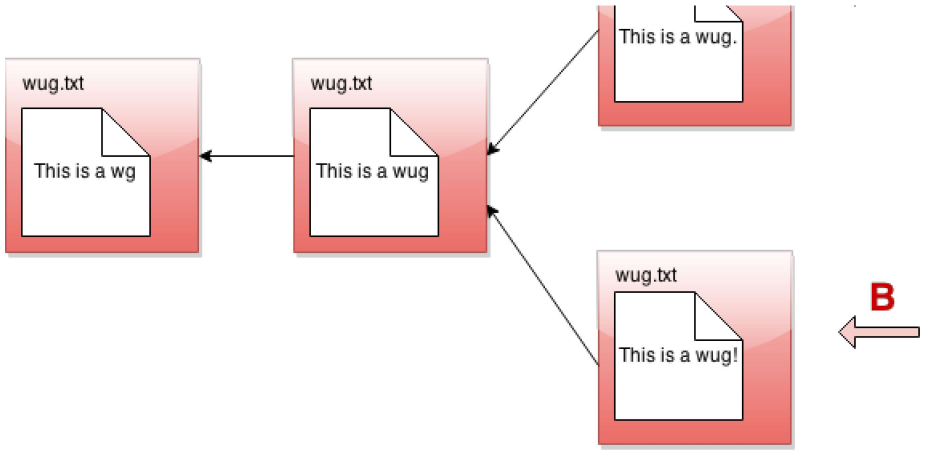
If we tell Gitlet to revert to an old commit, the front of the linked list will no longer reflect the current state of your files, which might be a little misleading. In order to fix this problem, we introduce something called the *head* pointer. The head pointer keeps track of where in the linked list we currently are. Normally, as we make commits, the head pointer will stay at the front of the linked list, indicating that the latest commit reflects the current state of the files:



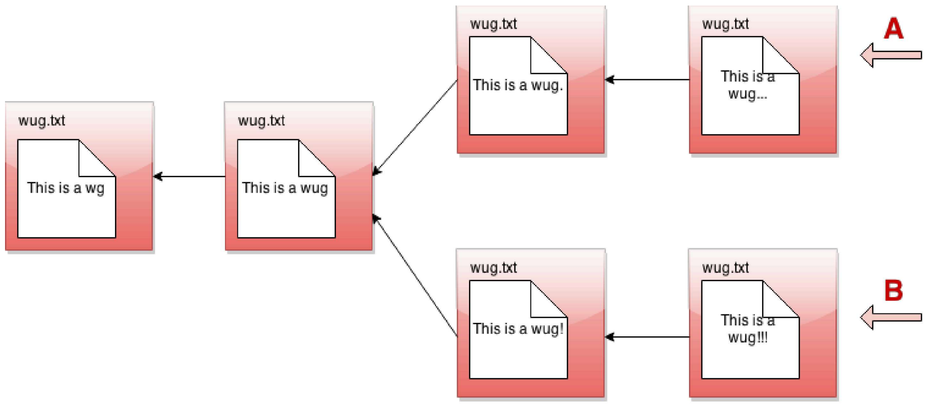
However, let's say we revert to the state of the files at commit #2 (technically, this is the *reset* command, which you'll see later in the spec). We move the head pointer back to show this:



All right, now, if this were all Gitlet could do, it would be a pretty simple system. But Gitlet has one more trick up its sleeve: it doesn't just maintain older and newer versions of files, it can maintain *differing* versions. Imagine you're coding a project, and you have two ideas about how to proceed: let's call one Plan A, and the other Plan B. Gitlet allows you to save both versions, and switch between them at will. Here's what this might look like, in our pictures:



It's not really a linked list anymore. It's more like a tree. We'll call this thing the *commit tree*. Keeping with this metaphor, each of the separate versions is called a *branch* of the tree. You can develop each version separately:



There are two pointers into the tree, representing the furthest point of each branch. At any given time, only one of these is the currently active pointer, and this is what's called the head pointer. The head pointer is the pointer at the front of the current branch.

That's it for our brief overview of the Gitlet system! Don't worry if you don't fully understand it yet; the section above was just to give you a high level picture of what its meant to do. A detailed spec of what you're supposed to do for this project follows this section.

But a last word here: commit trees are *immutable*: once a commit node has been created, it can **never** be destroyed (or changed at all). We can only add new things to the commit tree, One of Gitlet's goals is to allow us to save things that we worked on in the past so we don't delete them accidentally; this functionality would be jeopardized if we were allowed to edit past commits.

Internal Structures

Real Git distinguishes several different kinds of *objects*. For our purposes, the important ones are

- ***blobs***: Essentially the contents of files.
- ***trees***: Directory structures mapping names to references to blobs and other trees (subdirectories).
- ***commits***: Combinations of log messages, other metadata (commit date, author, etc.), a reference to a tree, and references to parent commits. The repository also maintains a mapping from *branch heads* (in this course, we've used names like `master`, `proj2`, etc.) to references to commits, so that certain important commits have symbolic names.

We will simplify from Git still further by

- Incorporating trees into commits and not dealing with subdirectories (so there will be one "flat" directory of plain files for each repository).
- Limiting ourselves to merges that reference two parents (in real Git, there can be any number of parents.)
- Having our metadata consist only of a timestamp and log message. A commit, therefore, will consist of a log message, timestamp, a mapping of file names to blob references, a parent reference, and (for merges) a second parent reference.

unique integer id that serves as a reference to the object. An interesting feature of Git is that these ids are *universal*: unlike a typical Java implementation, two objects with exactly the same content will have the same id on all systems (i.e. my computer, your computer, and anyone else's computer will compute this same exact id). In the case of blobs, "same content" means the same file contents. In the case of commits, it means the same metadata, the same mapping of names to references, and the same parent reference. The objects in a repository are thus said to be *content addressable*.

Both Git and Gitlet accomplish this the same way: by using a *cryptographic hash function* called SHA-1 (Secure Hash 1), which produces a 160-bit integer hash from any sequence of bytes. Cryptographic hash functions have the property that it is extremely difficult to find two different byte streams with the same hash value (or indeed to find *any* byte stream given just its hash value), so that essentially, we may assume that the probability that any two objects with different contents have the same SHA-1 hash value is 2^{-160} or about 10^{-48}. Basically, we simply ignore the possibility of a hashing collision, so that the system has, in principle, a fundamental bug that in practice never occurs!

Fortunately, there are library classes for computing SHA-1 values, so you won't have to deal with the actual algorithm. All you have to do is to make sure that you correctly label all your objects. In particular, this involves

- Including all metadata and references when hashing a commit.
- Distinguishing somehow between hashes for commits and hashes for blobs. A good way of doing this involves a well-thought out directory structure within the `.gitlet` directory. Another way to do so is to hash in an extra word for each

commits.

By the way, the SHA-1 hash value, rendered as a 40-character hexadecimal string, makes a convenient file name for storing your data in your `.gitlet` directory (more on that below). It also gives you a convenient way to compare two files (blobs) to see if they have the same contents: if their SHA-1s are the same, we simply assume the files are the same.

For remotes (like `origin` and `shared`, which we've been using all semester), we'll simply use other Gitlet repositories. Pushing simply means copying all commits and blobs that the remote repository does not yet have to the remote repository, and resetting a branch reference. Pulling is the same, but in the other direction. Remotes are extra credit in this project and not required for full credit.

Reading and writing your internal objects from and to files is actually pretty easy, thanks to Java's *serialization* facilities. The interface `java.io.Serializable` has no methods, but if a class implements it, then the Java runtime will automatically provide a way to convert to and from a stream of bytes, which you can then write to a file using the I/O class `java.io.ObjectOutputStream` and read back (and deserialize) with `java.io.ObjectInputStream`. The term "serialization" refers to the conversion from some arbitrary structure (array, tree, graph, etc.) to a serial sequence of bytes. You should have seen and gotten practice with serialization in lab 11. You'll be using a very similar approach here, so do use your lab11 as a resource when it comes to persistence and serialization.

Here is a summary example of the structures discussed in this section. As you can see, each commit (rectangle) points to some blobs (circles), which contain file contents. The commits contain the file names and references to these blobs, as well as a parent link. These references, depicted as arrows, are represented in

Navigation

Introduction

Useful Links

Overview of Gitlet

Internal Structures

Detailed Spec of Behavior

The Commands

Miscellaneous Things to Know about the Project

Dealing with Files

Serialization Details

Testing

Understanding Acceptance Tests

Design Document and Checkpoint

Grader Details

Things to Avoid

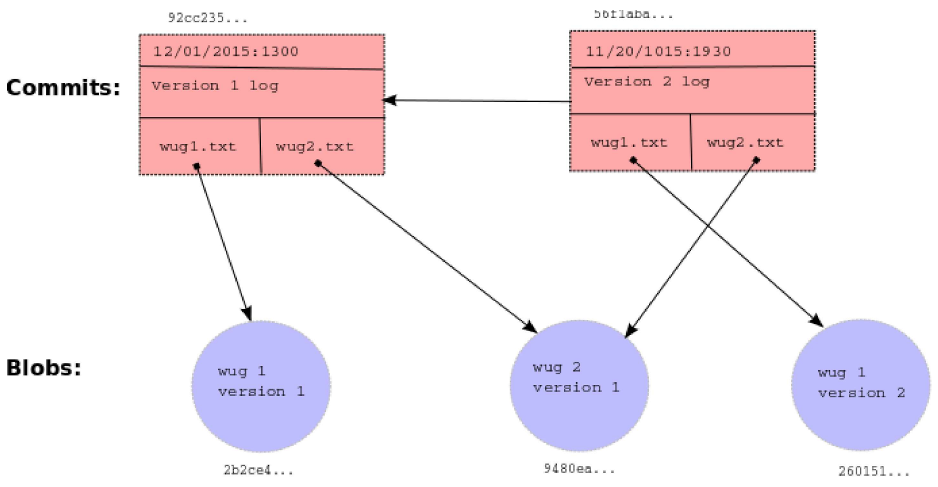
Going Remote (Extra Credit)

The Remote Commands

Diffs (Extra Credit)

Diff Utility

hexadecimal numerals above the commits and below the blobs). The newer commit contains an updated version of `wug1.txt`, but shares the same version of `wug2.txt` as the older commit. Your commit class will somehow store all of the information that this diagram shows: a careful selection of internal data structures will make the implementation easier or harder, so it behooves you to spend time planning and thinking about the best way to store everything.



Detailed Spec of Behavior

The only structure requirement we're giving you is that you have a class named `gitlet.Main` and that it has a main method. Here's your skeleton code for this project (in package Gitlet):

```java
public class Main {
    public static void main(String[] args) {
        // FILL IN
    }
}
```

We are also giving you some utility methods for performing a number of mostly file-system-related tasks, so that you can concentrate on the logic of the project rather than the peculiarities of dealing with the OS.

project—in fact, please do. But don't use any external code (aside from JUnit), and don't use any programming language other than Java. You can use all of the Java Standard Library that you wish, plus utilities we provide.

The majority of this spec will describe how `Main.java`'s main method must react when it receives various Gitlet commands as command-line arguments. But before we break down command-by-command, here are some overall guidelines the whole project should satisfy:

- In order for Gitlet to work, it will need a place to store old copies of files and other metadata. All of this stuff **must** be stored in a directory called `.gitlet`, just as this information is stored in directory `.git` for the real git system (files with a `.` in front are hidden files. You will not be able to see them by default on most operating systems. On Unix, the command `ls -a` will show them.) A Gitlet system is considered "initialized" in a particular location if it has a `.gitlet` directory there. Most Gitlet commands (except for the `init` command) only need to work when used from a directory where a Gitlet system has been initialized—i.e. a directory that has a `.gitlet` directory. The files that *aren't* in your `.gitlet` directory (which are copies of files from the repository that you are using and editing, as well as files you plan to add to the repository) are referred to as the files in your *working directory*.

- Most commands have runtime or memory usage requirements. You must follow these. Some of the runtimes are described as constant "relative to any significant measure". The significant measures are: any measure of number or size of files, any measure of number of commits. You can ignore time required to serialize or deserialize, *with the one caveat that your serialization time cannot* *added, committed, etc* (what is serialization? You'll see later in the spec). You can also assume that getting from a hash table is constant time.

- Some commands have failure cases with a specified error message. The exact formats of these are specified later in the spec. All error message end with a period; since our autograding is literal, be sure to include it. If your program ever encounters one of these failure cases, it must print the error message and not change anything else. *You don't need to handle any other error cases except the ones listed as failure cases*.
- There are some failure cases you need to handle that don't apply to a particular command. Here they are:
    - If a user doesn't input any arguments, print the message `Please enter a command.` and exit.
    - If a user inputs a command that doesn't exist, print the message `No command with that name exists.` and exit.
    - If a user inputs a command with the wrong number or format of operands, print the message `Incorrect operands.` and exit.
    - If a user inputs a command that requires being in an initialized Gitlet working directory (i.e., one containing a `.gitlet` subdirectory), but is not in such a directory, print the message `Not in an initialized Gitlet directory.`
- Some of the commands have their differences from real Git listed. The spec is not exhaustive in listing *all* differences from git, but it does list some of the bigger or potentially confusing and misleading ones.

Some of our autograder tests will break if you print anything more than necessary.

- Always exit with exit code 0, even in the presence of errors. This allows us to use other exit codes as an indication that something blew up.
- The spec classifies some commands as "dangerous". Dangerous commands are ones that potentially overwrite files (that aren't just metadata)—for example, if a user tells Gitlet to restore files to older versions, Gitlet may overwrite the current versions of the files. Just FYI.

The Commands

We now go through each command you must support in detail. Remember that good programmers always care about their data structures: as you read these commands, you should think first about how you should store your data to easily support these commands and second about if there is any opportunity to reuse commands that you've already implemented (hint: there is ample opportunity in this project to reuse code you've already written).

init

- **Usage**: `java gitlet.Main init`
- **Description**: Creates a new Gitlet version-control system in the current directory. This system will automatically start with one commit: a commit that contains no files and has the commit message `initial commit` (just like that, with no punctuation). It will have a single branch: `master`, which initially points to this initial commit, and `master` will be the current branch. The timestamp for this initial commit will be 00:00:00 UTC, Thursday, 1 January 1970 in whatever format you choose for dates (this is called "The (Unix)

initial commit in all repositories created by Gitlet will have exactly the same content, it follows that all repositories will automatically share this commit (they will all have the same UID) and all commits in all repositories will trace back to it.

- **Runtime**: Should be constant relative to any significant measure.
- **Failure cases**: If there is already a Gitlet version-control system in the current directory, it should abort. It should NOT overwrite the existing system with a new one. Should print the error message `A Gitlet version-control system already exists in the current directory.`
- **Dangerous?**: No
- **Our line count**: ~25

add

- **Usage**: `java gitlet.Main add [file name]`
- **Description**: Adds a copy of the file as it currently exists to the *staging area* (see the description of the `commit` command). For this reason, adding a file is also called *staging* the file *for addition*. Staging an already-staged file overwrites the previous entry in the staging area with the new contents. The staging area should be somewhere in `.gitlet`. If the current working version of the file is identical to the version in the current commit, do not stage it to be added, and remove it from the staging area if it is already there (as can happen when a file is changed, added, and then changed back). The file will no longer be staged for removal (see `gitlet rm`), if it was at the time of the command.
- **Runtime**: In the worst case, should run in linear time relative to the size of the file being added and $\lg N$, for N

- **Failure cases**: If the file does not exist, print the error message `File does not exist.` and exit without changing anything.
- **Dangerous?**: No
- **Our line count**: ~20

commit

- **Usage**: `java gitlet.Main commit [message]`
- **Description**: Saves a snapshot of tracked files in the current commit and staging area so they can be restored at a later time, creating a new commit. The commit is said to be *tracking* the saved files. By default, each commit's snapshot of files will be exactly the same as its parent commit's snapshot of files; it will keep versions of files exactly as they are, and not update them. A commit will only update the contents of files it is tracking that have been staged for addition at the time of commit, in which case the commit will now include the version of the file that was staged instead of the version it got from its parent. A commit will save and start tracking any files that were staged for addition but weren't tracked by its parent. Finally, files tracked in the current commit may be untracked in the new commit as a result being *staged for removal* by the `rm` command (below).

  The bottom line: By default a commit is the same as its parent. Files staged for addition and removal are the updates to the commit. Of course, the date (and likely the message) will also be different from the parent.

- **Some additional points about commit**:
  - The staging area is cleared after a commit.

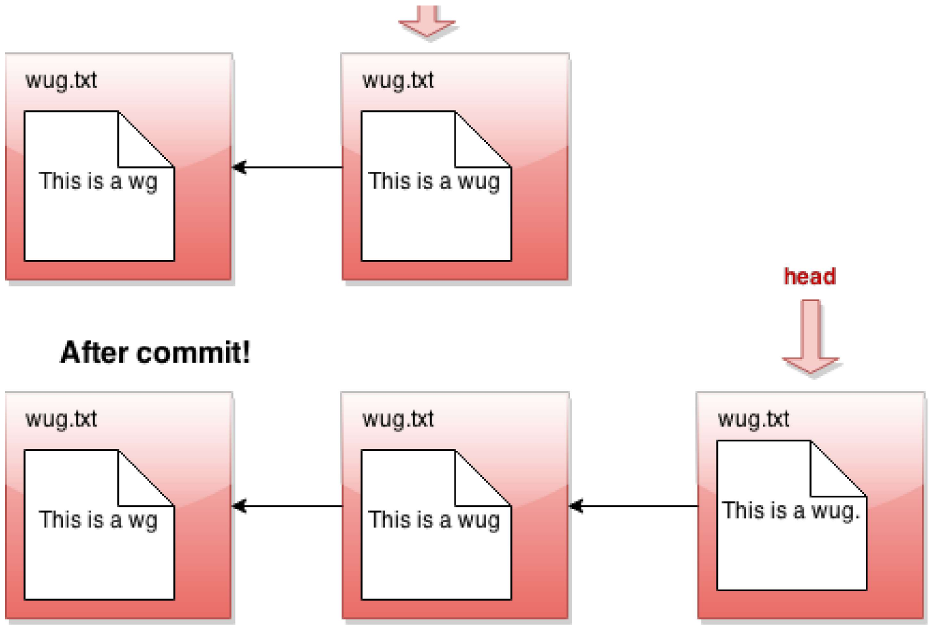
removes files in the working directory (other than those in the `.gitlet` directory). The `rm` command *will* remove such files, as well as staging them for removal, so that they will be untracked after a `commit`.

  - Any changes made to files after staging for addition or removal are ignored by the `commit` command, which *only* modifies the contents of the `.gitlet` directory. For example, if you remove a tracked file using the Unix `rm` command (rather than Gitlet's command of the same name), it has no effect on the next commit, which will still contain the deleted version of the file.
  - After the commit command, the new commit is added as a new node in the commit tree.
  - The commit just made becomes the "current commit", and the head pointer now points to it. The previous head commit is this commit's parent commit.
  - Each commit should contain the date and time it was made.
  - Each commit has a log message associated with it that describes the changes to the files in the commit. This is specified by the user. The entire message should take up only one entry in the array `args` that is passed to `main`. To include multiword messages, you'll have to surround them in quotes.
  - Each commit is identified by its SHA-1 id, which must include the file (blob) references of its files, parent reference, log message, and commit time.
- **Runtime**: Runtime should be constant with respect to any measure of number of commits. Runtime must be no worse than linear with respect to the total size of files the commit

requirement: Committing must increase the size of the `.gitlet` directory by no more than the total size of the files staged for addition at the time of commit, not including additional metadata. This means don't store redundant copies of versions of files that a commit receives from its parent. You *are* allowed to save whole additional copies of files; don't worry about only saving diffs, or anything like that.

- **Failure cases**: If no files have been staged, abort. Print the message `No changes added to the commit.` Every commit must have a non-blank message. If it doesn't, print the error message `Please enter a commit message.` It is *not* a failure for tracked files to be missing from the working directory or changed in the working directory. Just ignore everything outside the `.gitlet` directory entirely.
- **Dangerous?**: No
- **Differences from real git**: In real git, commits may have multiple parents (due to merging) and also have considerably more metadata.
- **Our line count**: ~35

Here's a picture of before-and-after commit:



rm

- **Usage**: `java gitlet.Main rm [file name]`
- **Description**: Unstage the file if it is currently staged for addition. If the file is tracked in the current commit, stage it for removal and remove the file from the working directory if the user has not already done so (do *not* remove it unless it is tracked in the current commit).
- **Runtime**: Should run in constant time relative to any significant measure.
- **Failure cases**: If the file is neither staged nor tracked by the head commit, print the error message `No reason to remove the file.`
- **Dangerous?**: Yes (although if you use our utility methods, you will only hurt your repository files, and not all the other files in your directory.)
- **Our line count**: ~20

log

- **Usage**: `java gitlet.Main log`

information about each commit backwards along the commit tree until the initial commit, following the first parent commit links, ignoring any second parents found in merge commits. (In regular Git, this is what you get with `git log --first-parent`). This set of commit nodes is called the commit's *history*. For every node in this history, the information it should display is the commit id, the time the commit was made, and the commit message. Here is an example of the *exact* format it should follow:

```
===
commit a0da1ea5a15ab613bf9961fd86f010cf74c7ee4
Date: Thu Nov 9 20:00:05 2017 -0800
A commit message.

===
commit 3e8bf1d794ca2e9ef8a4007275acf3751c7170f
Date: Thu Nov 9 17:01:33 2017 -0800
Another commit message.

===
commit e881c9575d180a215d1a636545b8fd9abfb1d2b
Date: Wed Dec 31 16:00:00 1969 -0800
initial commit
```

There is a `===` before each commit and an empty line after it. As in real Git, each entry displays the unique SHA-1 id of the commit object. The timestamps displayed in the commits reflect the current timezone, not UTC; as a result, the timestamp for the initial commit does not read Thursday, January 1st, 1970, 00:00:00, but rather the equivalent Pacific Standard Time. Display commits with the most recent at the top. By the way, you'll find that the Java classes `java.util.Date` and `java.util.Formatter` are useful for getting and formatting times. Look into them instead of trying to construct it manually yourself!

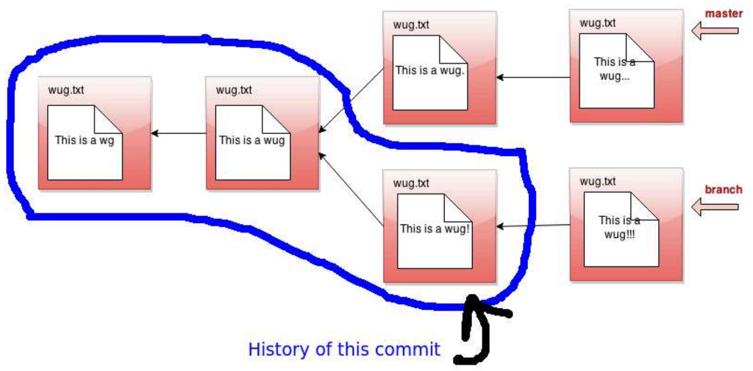
For merge commits (those that have two parent commits), add a line just below the first, as in

Merge: 4975af1 2c1ead1 Date: Sat Nov 11 12:30:00 2017 -0800 Merged development into master.

where the two hexadecimal numerals following "Merge:" consist of the first seven digits of the first and second parents' commit ids, in that order. The first parent is the branch you were on when you did the merge; the second is that of the merged-in branch. This is as in regular Git.

- **Runtime**: Should be linear with respect to the number of nodes in head's history.
- **Failure cases**: None
- **Dangerous?**: No
- **Our line count**: ~20

Here's a picture of the history of a particular commit. If the current branch's head pointer happened to be pointing to that commit, log would print out information about the circled commits:



The history ignores other branches and the future. Now that we have the concept of history, let's refine what we said earlier about the commit tree being immutable. It is immutable precisely in the sense that *the history of a commit with a particular id may never change, ever*. If you think of the commit tree as nothing is that each history is immutable.

global-log

- **Usage**: `java gitlet.Main global-log`
- **Description**: Like log, except displays information about all commits ever made. The order of the commits does not matter. Hint: there is a useful method in `gitlet.Utils` that will help you iterate over files within a directory.
- **Runtime**: Linear with respect to the number of commits ever made.
- **Failure cases**: None
- **Dangerous?**: No
- **Our line count**: ~10

find

- **Usage**: `java gitlet.Main find [commit message]`
- **Description**: Prints out the ids of all commits that have the given commit message, one per line. If there are multiple such commits, it prints the ids out on separate lines. The commit message is a single operand; to indicate a multiword message, put the operand in quotation marks, as for the `commit` command above.
- **Runtime**: Should be linear relative to the number of commits.
- **Failure cases**: If no such commit exists, prints the error message `Found no commit with that message.`
- **Dangerous?**: No
- **Differences from real git**: Doesn't exist in real git. Similar effects can be achieved by grepping the output of log.

status

- **Usage**: `java gitlet.Main status`
- **Description**: Displays what branches currently exist, and marks the current branch with a `*`. Also displays what files have been staged for addition or removal. An example of the *exact* format it should follow is as follows.

```
=== Branches ===
*master
other-branch

=== Staged Files ===
wug.txt
wug2.txt

=== Removed Files ===
goodbye.txt

=== Modifications Not Staged For Commit ===
junk.txt (deleted)
wug3.txt (modified)

=== Untracked Files ===
random.stuff
```

There is an empty line between sections. Entries should be listed in lexicographic order, using the Java string-comparison order (the asterisk doesn't count). A file in the working directory is "modified but not staged" if it is

- Tracked in the current commit, changed in the working directory, but not staged; or
- Staged for addition, but with different contents than in the working directory; or
- Staged for addition, but deleted in the working directory; or
- Not staged for removal, but tracked in the current commit and deleted from the working directory.

the working directory but neither staged for addition nor tracked. This includes files that have been staged for removal, but then re-created without Gitlet's knowledge. Ignore any subdirectories that may have been introduced, since Gitlet does not deal with them.

The last two sections (modifications not staged and untracked files) are extra credit, worth 1 point. Feel free to leave them blank (leaving just the headers).

- **Runtime**: Make sure this depends only on the amount of data in the working directory plus the number of files staged to be added or deleted plus the number of branches.
- **Failure cases**: None
- **Dangerous?**: No
- **Our line count**: ~45

checkout

Checkout is a general command that can do a few different things depending on what its arguments are. There are 3 possible use cases. In each section below, you'll see 3 bullet points. Each corresponds to the respective usage of checkout.

- **Usages**:
  1. `java gitlet.Main checkout -- [file name]`
  2. `java gitlet.Main checkout [commit id] -- [file name]`
  3. `java gitlet.Main checkout [branch name]`
- **Descriptions**:
  1. Takes the version of the file as it exists in the head commit, the front of the current branch, and puts it in the working directory, overwriting the version of the

version of the file is not staged.

2. Takes the version of the file as it exists in the commit with the given id, and puts it in the working directory, overwriting the version of the file that's already there if there is one. The new version of the file is not staged.

3. Takes all files in the commit at the head of the given branch, and puts them in the working directory, overwriting the versions of the files that are already there if they exist. Also, at the end of this command, the given branch will now be considered the current branch (HEAD). Any files that are tracked in the current branch but are not present in the checked-out branch are deleted. The staging area is cleared, unless the checked-out branch is the current branch (see **Failure cases** below).

- **Run times**:

  1. Should be linear relative to the size of the file being checked out.

  2. Should be linear with respect to the total size of the files in the commit's snapshot. Should be constant with respect to any measure involving number of commits. Should be constant with respect to the number of branches.

- **Failure cases**:

  1. If the file does not exist in the previous commit, abort, printing the error message `File does not exist in that commit.`

  2. If no commit with the given id exists, print `No commit with that id exists.` Otherwise, if the file does not

for failure case 1.

3. If no branch with that name exists, print `No such branch exists.` If that branch is the current branch, print `No need to checkout the current branch.` If a working file is untracked in the current branch and would be overwritten by the checkout, print `There is an untracked file in the way; delete it, or add and commit it first.` and exit; perform this check before doing anything else.

- **Differences from real git**: Real git does not clear the staging area and stages the file that is checked out. Also, it won't do a checkout that would overwrite or undo changes (additions or removals) that you have staged.

A [commit id] is, as described earlier, a hexadecimal numeral. A convenient feature of real Git is that one can abbreviate commits with a unique prefix. For example, one might abbreviate

```
a0da1ea5a15ab613bf9961fd86f010cf74c7ee48
```

as

```
a0da1e
```

in the (likely) event that no other object exists with a SHA-1 identifier that starts with the same six digits. You should arrange for the same thing to happen for commit ids that contain fewer than 40 characters. Unfortunately, using shortened ids might slow down the finding of objects if implemented naively (making the time to find a file linear in the number of objects), so we won't worry about timing for commands that use shortened ids. We suggest, however, that you poke around in a `.git` directory (specifically, `.git/objects`) and see how it manages to speed up its search. You will perhaps recognize a familiar data structure implemented with the file system rather than pointers.

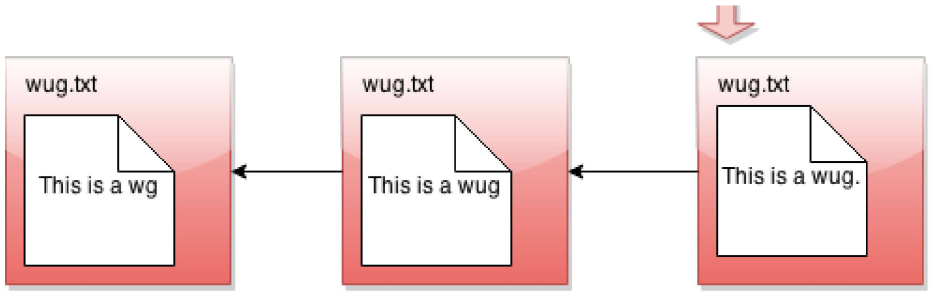
area: otherwise files scheduled for addition or removal remain so.

- **Dangerous?**: Yes!
- **Our line counts**:
    - ~15
    - ~5
    - ~15

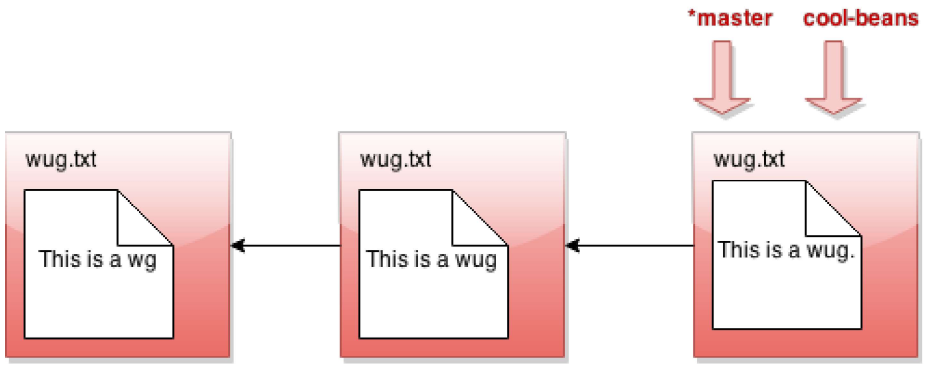
branch

- **Usage**: `java gitlet.Main branch [branch name]`
- **Description**: Creates a new branch with the given name, and points it at the current head node. A branch is nothing more than a name for a reference (a SHA-1 identifier) to a commit node. This command does NOT immediately switch to the newly created branch (just as in real Git). Before you ever call branch, your code should be running with a default branch called "master".
- **Runtime**: Should be constant relative to any significant measure.
- **Failure cases**: If a branch with the given name already exists, print the error message `A branch with that name already exists.`
- **Dangerous?**: No
- **Our line count**: ~10

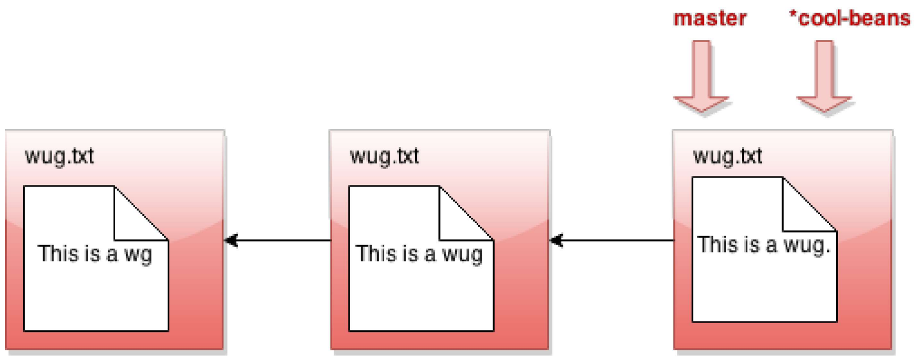
All right, let's see what branch does in detail. Suppose our state looks like this:



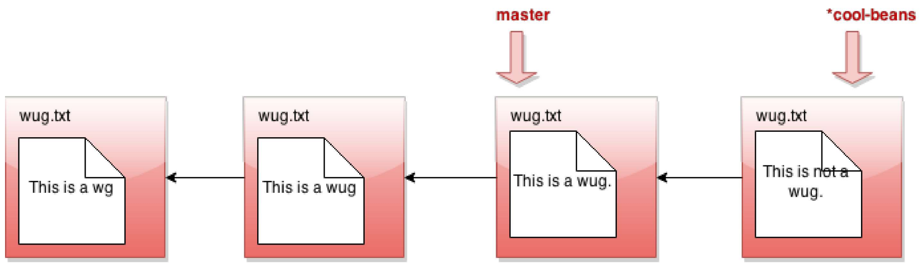
Now we call `java gitlet.Main branch cool-beans`. Then we get this:



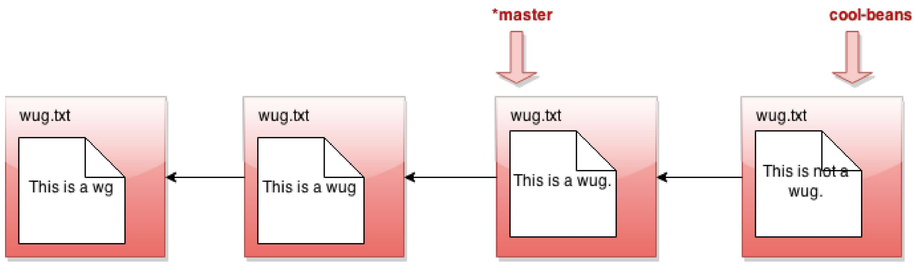
Hmm... nothing much happened. Let's switch to the branch with `java gitlet.Main checkout cool-beans`:



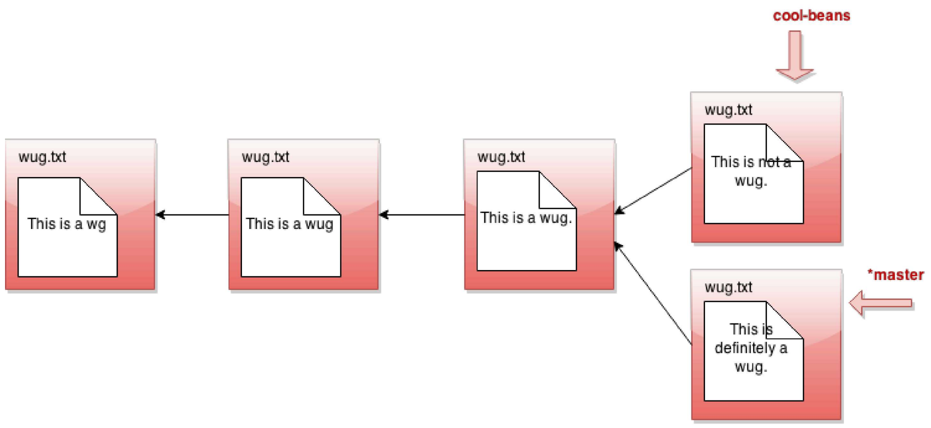
Nothing much happened again?! Okay, say we make a commit now. Modify some files, then `java gitlet.Main add...` then `java gitlet.Main commit...`



line. What's going on? Maybe I should go back to my other branch with `java gitlet.Main checkout master`:



Now I make a commit...



Phew! So that's the whole idea of branching. Did you catch what's going on? All that creating a branch does is to give us a new pointer. At any given time, one of these pointers is considered the currently active pointer, also called the HEAD pointer (indicated by *). We can switch the currently active head pointer with `checkout [branch name]`. Whenever we commit, it means we add a child commit to the currently active HEAD commit, even if a child commit is already there. This naturally creates branching behavior, since one parent commit can have multiple children commits.

Make *sure* that the behavior of your `branch`, `checkout`, and `commit` match what we've described above. This is pretty core functionality of Gitlet that many other commands will depend upon. If any of this core functionality is broken, very many of our autograder tests won't work!

- **Usage**: `java gitlet.Main rm-branch [branch name]`
- **Description**: Deletes the branch with the given name. This only means to delete the pointer associated with the branch; it does not mean to delete all commits that were created under the branch, or anything like that.
- **Runtime**: Should be constant relative to any significant measure.
- **Failure cases**: If a branch with the given name does not exist, aborts. Print the error message `A branch with that name does not exist.` If you try to remove the branch you're currently on, aborts, printing the error message `Cannot remove the current branch.`
- **Dangerous?**: No
- **Our line count**: ~15

reset

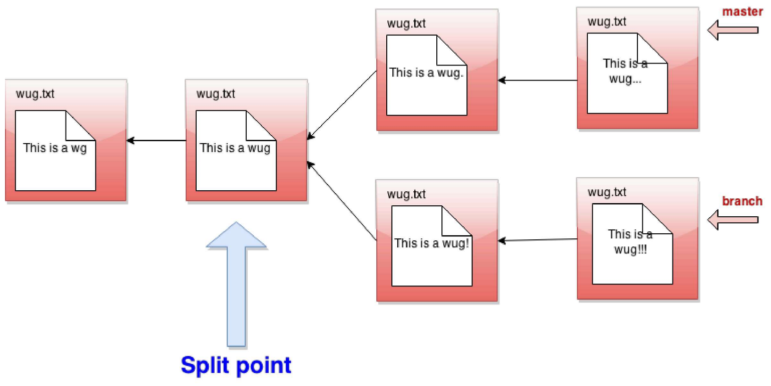
- **Usage**: `java gitlet.Main reset [commit id]`
- **Description**: Checks out all the files tracked by the given commit. Removes tracked files that are not present in that commit. Also moves the current branch's head to that commit node. See the intro for an example of what happens to the head pointer after using reset. The `[commit id]` may be abbreviated as for `checkout`. The staging area is cleared. The command is essentially `checkout` of an arbitrary commit that also changes the current branch head.
- **Runtime**: Should be linear with respect to the total size of files tracked by the given commit's snapshot. Should be constant with respect to any measure involving number of commits.

`commit with that id exists.` If a working file is untracked in the current branch and would be overwritten by the reset, print `There is an untracked file in the way; delete it, or add and commit it first.` and exit; perform this check before doing anything else.

- **Dangerous?**: Yes!
- **Differences from real git**: This command is closest to using the `--hard` option, as in `git reset --hard [commit hash]`.
- **Our line count**: ~10

merge

- **Usage**: `java gitlet.Main merge [branch name]`
- **Description**: Merges files from the given branch into the current branch. This method is a bit complicated, so here's a more detailed description:
  - First consider what might be called the *split point* of the current branch and the given branch. For example, if `master` is the current branch and `branch` is the given branch:



    The split point is a *latest common ancestor* of the current and given branch heads:

    - A *common ancestor* is a commit to which there is a path (of 0 or more parent pointers) from

    - A *latest* common ancestor is a common ancestor that is not an ancestor of any other common ancestor. For example, although the leftmost commit in the diagram above is a common ancestor of `master` and `branch`, it is also an ancestor of the commit immediately to its right, so it is not a latest common ancestor. If the split point *is* the same commit as the given branch, then we do nothing; the merge is complete, and the operation ends with the message `Given branch is an ancestor of the current branch.` If the split point is the current branch, then the effect is to check out the given branch, and the operation ends after printing the message `Current branch fast-forwarded.` Otherwise, we continue with the steps below.

- Any files that have been *modified* in the given branch since the split point, but not modified in the current branch since the split point should be changed to their versions in the given branch (checked out from the commit at the front of the given branch). These files should then all be automatically staged. To clarify, if a file is "modified in the given branch since the split point" this means the version of the file as it exists in the commit at the front of the given branch has different content from the version of the file at the split point.

- Any files that have been modified in the current branch but not in the given branch since the split point should stay as they are.

and given branch in the same way (i.e., both files now have the same content or were both removed) are left unchanged by the merge. If a file was removed from both the current and given branch, but a file of the same name is present in the working directory, it is left alone and continues to be absent (not tracked nor staged) in the merge.

- Any files that were not present at the split point and are present only in the current branch should remain as they are.
- Any files that were not present at the split point and are present only in the given branch should be checked out and staged.
- Any files present at the split point, unmodified in the current branch, and absent in the given branch should be removed (and untracked).
- Any files present at the split point, unmodified in the given branch, and absent in the current branch should remain absent.
- Any files modified in different ways in the current and given branches are *in conflict*. "Modified in different ways" can mean that the contents of both are changed and different from other, or the contents of one are changed and the other file is deleted, or the file was absent at the split point and has different contents in the given and current branches. In this case, replace the contents of the conflicted file with

```
<<<<<<< HEAD
contents of file in current branch
=======
contents of file in given branch
>>>>>>>
```

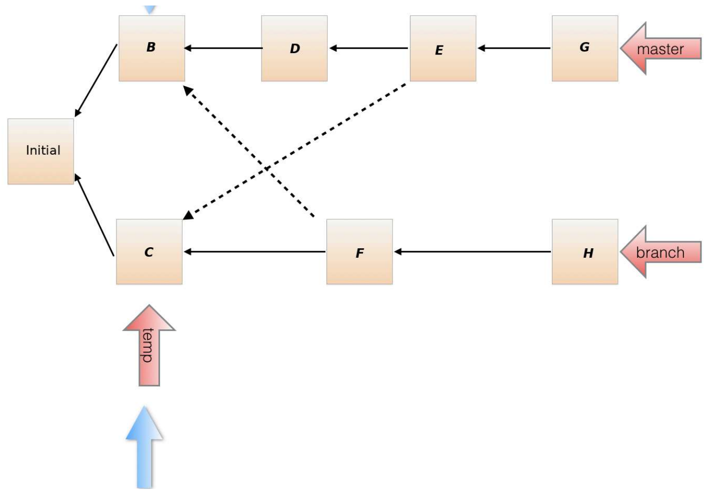
contents) and stage the result. Treat a deleted file in a branch as an empty file. Use straight concatenation here. In the case of a file with no newline at the end, you might well end up with something like this:

```
<<<<<<< HEAD
contents of file in current branch=======
contents of file in given branch>>>>>>>
```

This is fine; people who produce non-standard, pathological files because they don't know the difference between a line terminator and a line separator deserve what they get.

- Once files have been updated according to the above, and the split point was not the current branch or the given branch, merge automatically commits with the log message `Merged [given branch name] into [current branch name].` Then, if the merge encountered a conflict, print the message `Encountered a merge conflict.` on the terminal (not the log). Merge commits differ from other commits: they record as parents both the head of the current branch (called the *first parent*) and the head of the branch given on the command line to be merged in.

- There is one complication in the definition of the split point. You may have noticed that we referred to "a", rather than "the" latest common ancestor. This is because there can be more than one in the case of "criss-cross merges", such as this:



Here, the solid lines are first parents and the dashed lines are the merged-in parents. Both the commits pointed by blue arrows above are latest common ancestors. Here's how it was created:

```
java gitlet.Main init
java gitlet.Main branch branch
[various edits...]
java gitlet.Main commit "B"
java gitlet.Main checkout branch
[various edits...]
java gitlet.Main commit "C"
java gitlet.Main branch temp
java gitlet.Main merge master  # Create comm
[various edits...]
java gitlet.Main commit "H"
java gitlet.Main checkout master
[various edits...]
java gitlet.Main commit "D"
java gitlet.Main merge temp    # Create comm
[various edits...]
java gitlet.Main commit "G"
```

Now if we want to merge branch into master, we have two possible split points: the commits marked by the two blue arrows. You might want to think about why it can make a difference which gets used as the split point. We'll use the following rule to choose which of multiple possible split points to use:

to the head of the current branch (that is, is reachable by following the fewest parent pointers along some path).

- If multiple candidates are at the same closest distance, choose any one of them as the split point. (We will make sure that this only happens in our test cases when the resulting merge commit is the same with any of the closest choices.)

So in this example, we would choose commit C as the split point when merging `branch` into `master`, since there is a shorter path from G to C than from G to B. If instead we were currently on `branch` and merging in branch `master`, we could use either commit B or C, since both are the same distance from commit H.

By the way, we hope you've noticed that the set of commits has progressed from a simple sequence to a tree and now, finally, to a full directed acyclic graph.

- **Runtime**: $O(N \lg N + D)$, where N is the total number of ancestor commits for the two branches and D is the total amount of data in all the files under these commits.
- **Failure cases**: If there are staged additions or removals present, print the error message `You have uncommitted changes.` and exit. If a branch with the given name does not exist, print the error message `A branch with that name does not exist.` If attempting to merge a branch with itself, print the error message `Cannot merge a branch with itself.` If merge would generate an error because the commit that it does has no changes in it, just let the normal commit error message for this go through.

overwritten or deleted by the merge, print `There is an untracked file in the way; delete it, or add and commit it first.` and exit; perform this check before doing anything else.

- **Dangerous?**: Yes!
- **Differences from real git**: Real Git does a more subtle job of merging files, displaying conflicts only in places where both files have changed since the split point.

  Real Git has a different way to decide which of multiple possible split points to use.

  Real Git will force the user to resolve the merge conflicts before committing to complete the merge. Gitlet just commits the merge, conflicts and all, so that you must use a separate commit to resolve problems.

  Real Git will complain if there are unstaged changes to a file that would be changed by a merge. You may do so as well if you want, but we will not test that case.
- **Our line count**: ~70
- **Conceptual Check**

Miscellaneous Things to Know about the Project

Phew! That was a lot of commands to go over just now. But don't worry, not all commands are created equal. You can see for each command the approximate number of lines we took to do each part (that this only counts code specific to that command -- it doesn't double-count code reused in multiple commands). You shouldn't worry about matching our solution exactly, but hopefully it gives you an idea about the relative time consumed by each don't leave it for the last minute!

This is an ambitious project, and it would not be surprising for you to feel lost as to where to begin. Therefore, feel free to collaborate with others a little more closely than usual, with the following caveats:

- Acknowledge all collaborators in comments near the beginning of your `gitlet/Main.java` file.
- Don't share specific code; all collaborators must produce their own versions of the algorithms they come up with, so that we can see they differ.

The Piazza megathreads typically get very long for Gitlet, but they are full of very good conversation and discussion on the approach for particular commits. In this project more than any you should take advantage of the size of the class and see if you can find someone with a similar question to you on the megathread. It's very unlikely that your question is so unique to you that nobody else has had it (unless it is a bug that relates to your design, in which case you should submit a Gitbug).

By now this spec has given you enough information to get working on the project. But to help you out some more, there are a couple of things you should be aware of:

Dealing with Files

This project requires reading and writing of files. In order to do these operations, you might find the classes `java.io.File` and `java.nio.file.Files` helpful. Actually, you may find various things in the `java.io` and `java.nio` packages helpful. Be sure to read the `gitlet.Utils` package for other things we've written for you. If you do a little digging through all of these, you might find a couple of methods that will make the I/O portion of this project writers, scanners, or streams, you're making things more complicated than need be.

Serialization Details

If you think about Gitlet, you'll notice that you can only run one command every time you run the program. In order to successfully complete your version-control system, you'll need to remember the commit tree across commands. This means you'll have to design not just a set of classes to represent internal Gitlet structures during execution, but you'll need an analogous representation as files within your `.gitlet` directories, which will carry across multiple runs of your program.

As indicated earlier, the convenient way to do this is to serialize the runtime objects that you will need to store permanently in files. In Java, this simply involves implementing the `java.io.Serializable` interface:

```
import java.io.Serializable;

class MyObject implements Serializable {
    ...
}
```

This interface has no methods; it simply marks its subtypes for the benefit of some special Java classes for performing I/O on objects. For example,

```
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;
...
    MyObject obj = ....;
    File outFile = new File(someFileName);
    try {
        ObjectOutputStream out =
            new ObjectOutputStream(new FileOutputStrea
        out.writeObject(obj);
        ...
    `
```

Navigation

will convert `obj` to a stream of bytes and store it in the file whose name is stored in `someFileName`. The object may then be reconstructed with a code sequence such as

```
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
...
    MyObject obj;
    File inFile = new File(someFileName);
    try {
        ObjectInputStream inp =
            new ObjectInputStream(new FileInputStream(
        obj = (MyObject) inp.readObject();
        inp.close();
    } catch (IOException | ClassNotFoundException excp
        ...
        obj = null;
    }
```

The Java runtime does all the work of figuring out what fields need to be converted to bytes and how to do so.

There is, however, one annoying subtlety to watch out for: Java serialization follows pointers. That is, not only is the object you pass into `writeObject` serialized and written, but any object it points to as well. If your internal representation of commits, for example, represents the parent commits as pointers to other commit objects, then writing the head of a branch will write all the commits (and blobs) in the entire subgraph of commits into one file, which is generally not what you want. To avoid this, don't use Java pointers to refer to commits and blobs in your runtime objects, but instead use SHA-1 hash strings. Maintain a runtime map between these strings and the runtime objects they refer to. You create and fill in this map while Gitlet is running, but never read or write it to a file.

commits as well as SHA-1 strings to avoid the bother and execution time required to look them up each time. You can store such pointers in your objects while still avoiding having them written out by declaring them "transient", as in

```
    private transient MyCommitType parent1;
```

Such fields will not be serialized, and when back in and deserialized, will be set to their default values (null for reference types). You must be careful when reading the objects that contain transient fields back in to set the transient fields to appropriate values.

Unfortunately, looking at the serialized files your program has produced with a text editor (for debugging purposes) would be rather unrevealing; the contents are encoded in Java's private serialization encoding. We have therefore provided a simple debugging utility program you might find useful: `gitlet.DumpObj`. See the Javadoc comment on `gitlet/DumpObj.java` for details.

Testing

NOTE (4/17/22): We have pushed a change to the testing Makefile so that the command `make check` will run your created tests in `student_tests` as well as the ones in `samples`. Please run `git fetch shared` and `git merge shared/proj3 -m "some message here"` to get these new changes locally.

As usual, testing is part of the project. Be sure to provide your own acceptance tests for each of the commands, covering all the specified functionality. Also, feel free to add unit tests to `UnitTest.java` or other testing classes it invokes in its `main` method. We don't provide any unit tests for Gitlet since unit tests are very dependent on your implementation.

to write acceptance tests: `testing/tester.py`. As with Project #2, this interprets testing files with an `.in` extension. As with projects 0-2, the following commands will run your unit tests, acceptance tests, or the entire testing suite respectively:

```
make unit
make acceptance
make check
```

Furthermore, we've added an additional Makefile target:

```
make doc
```

This command will generate a group of files based on your current project code that represents the "design" of your code. For instance, if you have some class `A.java` in your project, one of the files that `make doc` generates will be a file that clearly names all of the methods, variables, and constants that you have so far in `A.java`. This file will be called `allclasses-index.html`, so try opening it in a web browser to see what was generated!

If you'd like to run a single test, within the `testing` subdirectory, running the commands (after first running `make`)

```
cd testing
python3 tester.py --verbose FILE.in ...
```

where `FILE.in ...` is a list of specific `.in` files you want to check, will provide additional information such as what your program is outputting. The command

```
python3 tester.py --verbose --keep FILE.in
```

will, in addition, keep around the directory that `tester.py` produces so that you can examine its files at the point the tester script detected an error.

We've provided some examples in the directory `testing/samples`. Don't put your own tests in that subdirectory;

our tests vs your tests (which may be buggy!). Put all your `.in` files in another folder called `student_tests` within the `testing` directory.

In effect, the tester implements a very simple *domain-specific language (DSL)* that contains commands to

- Set up or remove files from a testing directory;
- Run `java gitlet.Main`;
- Check the output of Gitlet against a specific output or a regular expression describing possible outputs;
- Check the presence, absence, and contents of files.

Running the command

```
python3 tester.py
```

(with no operands, as shown) in the testing directory will provide a message documenting this language.

As usual, we will test your code *on the the instructional machines*, so do be sure it works there!

We've added a few things to the Makefile to adjust for differences in people's setups. If you are on Windows, you can still use our makefile unchanged by using

```
make PYTHON=python check
```

You can pass additional flags to `tester.py` with, for example,

```
make TESTER_FLAGS="--show=all --keep"
```

Lastly, we also have a way of using the IntelliJ debugger to debug Gitlet acceptance tests. This may seem impossible, since we run everything from the command line; however, IntelliJ provides a feature called "Remote JVM Debugging" that will

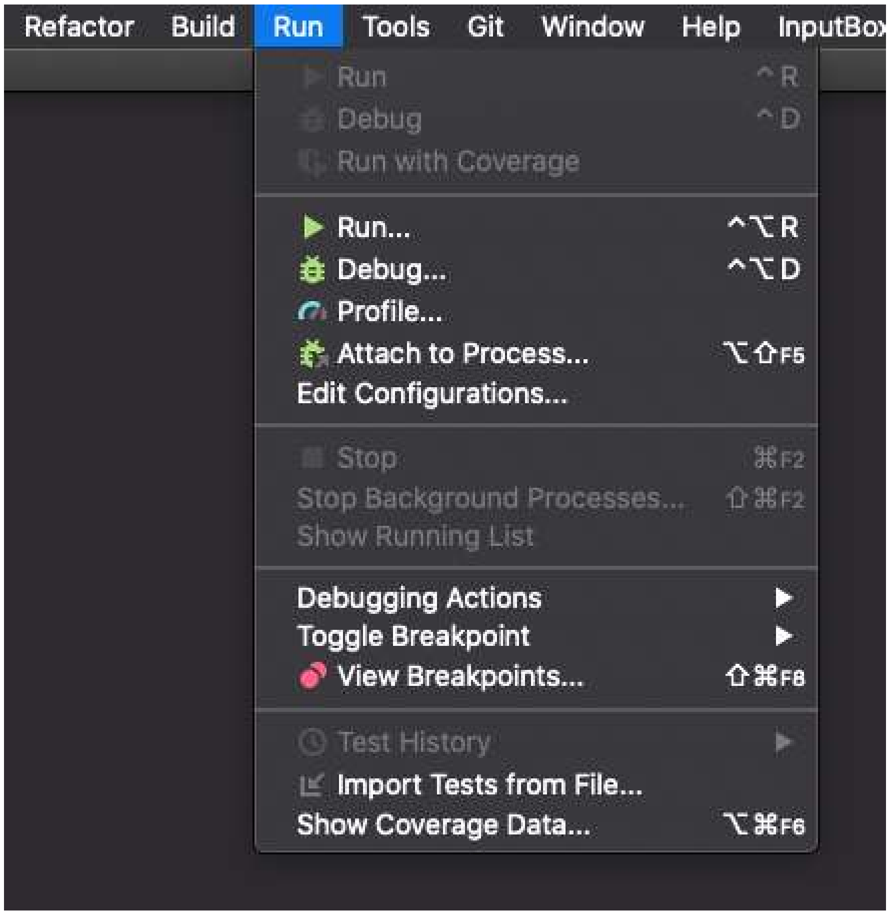
Navigation

tests.

A walk-through of the rest of these testing details can be found here. This video goes over all the steps listed here in the spec but for the Capers lab, so if you find yourself confused on the directions then check it out. The Capers lab was lab 6 in Spring 2021 when this video was made, but for us it is lab 12.

Without JUnit tests, you may be wondering how to debug your code. We'll walk you through how you will do that in Gitlet.

To debug an acceptance test, we first need to let IntelliJ know that we want to debug remotely. Navigate to your IntelliJ and open your `proj3` project if you don't have it open already. At the top, go to "Run" -> "Run":

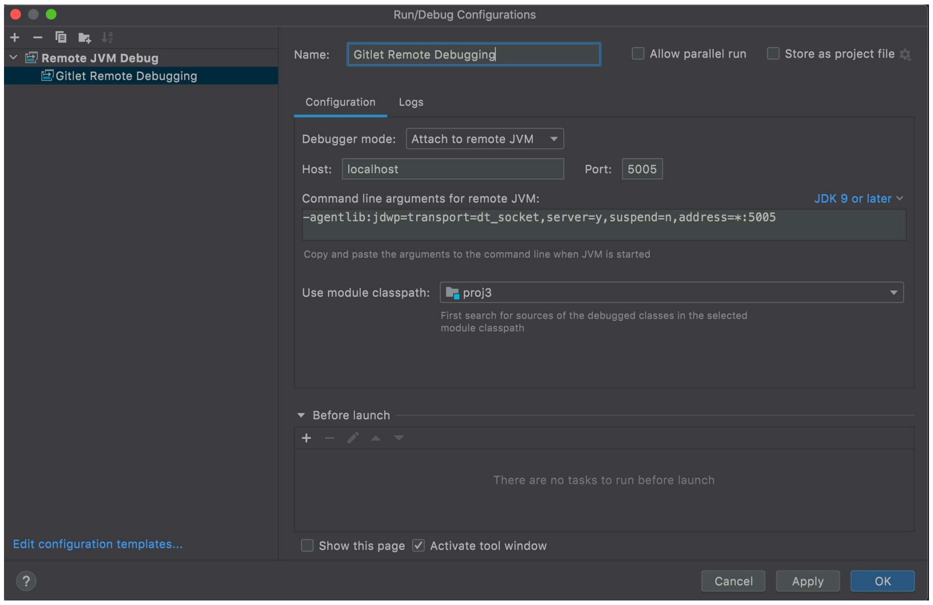


You'll get a box asking you to Edit Configurations that will look like the below:

Navigation

Yours might have more or less of those boxes with other names if you tried running a class within IntelliJ already. If that's the case, just click the one that says "Edit Configurations"

In this box, you'll want to hit the "+" button in the top left corner and select "Remote JVM Debug." It should now look like this:



Run/Debug Configurations

Remote JVM Debug
Gitlet Remote Debugging

Name: Gitlet Remote Debugging
Allow parallel run
Store as project file

Configuration Logs

Debugger mode: Attach to remote JVM
Host: localhost
Port: 5005

Command line arguments for remote JVM: JDK 9 or later
```
-agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=*:5005
```
Copy and paste the arguments to the command line when JVM is started

Use module classpath: proj3
First search for sources of the debugged classes in the selected module classpath

Before launch
There are no tasks to run before launch

Edit configuration templates...
Show this page ✓ Activate tool window

Cancel Apply OK

We just need the default settings. You should add a descriptive name in the top box, perhaps "Gitlet Remote Debug". After you add a name, go ahead and hit "Apply" and then exit from this screen. **Before we leave IntelliJ**, place a breakpoint in the `main` method of the `Main` class, so we can actually debug. Make sure this breakpoint will actually be reached; just put it on the first line of the `main` method.

Now you'll navigate to the `testing` directory within your terminal. The script that will connect to the IntelliJ JVM is `tester.py` with the `--debug` flag: use the following command to launch the testing script:

```
python3 tester.py --debug samples/test01-init.in
```

.in file. If you'd like the .gitlet folder to stay after the test is completed to investigate its contents, then use the `--keep` flag:

```
python3 tester.py --keep --debug samples/test01-init.i
```

For our example it doesn't matter what you do; we've just included it in case you'd like to take a look around. By default, the `.gitlet` that is generated is deleted.

If you see an error message, then it means you are probably not in the `testing` directory. Check those two things, and if you're still confused then ask a TA.

Otherwise, you should be ready to debug! You'll see something like this:

```
test01-init: You are in debug mode.
    In this mode, you will be shown each command from
    If you would like to step into and debug the comma
      Once you have done so, go back to IntelliJ and c
    If you would like to move on to the next command,
[line 3]: gitlet init
>>>
```

The text above contains helpful tips. What we see next is the name of the `.in` file we're debugging, then a series of lines that begin with `[line #]` and `>`.

Lines that begin with `[line #]` are the gitlet commands that will be run on your `Main` class, i.e. a specific execution of your program. These correspond to the commands we saw in the `.in` file on the right side of the `>`.

Lines that begin with `>>>` are for you to enter debug commands on. The 2 commands are listed above.

Remember that each input file will list multiple commands and therefore multiple executions of our program. We need to first figure out what command is the culprit.

command without debugging it. You can think of it as bringing you to the next command.

One of these will error: either your code will produce a runtime error, or your output wasn't the same. For example:

```
> python3 testing/tester.py --debug --keep testing/sam

test01-init: You are in debug mode.
    In this mode, you will be shown each command from
    If you would like to step into and debug the comma
    Once you have done so, go back to IntelliJ and cli
    If you would like to move on to the next command,
[line 3]: gitlet init
>>> n
ERROR (file or directory .gitlet not present)

Ran 1 tests. 0 passed.
```

For us, it was our first command. Notice that we had the `--keep` flag enabled, so we could now investigate the saved directory `test01- init_0` to see what happened. If we debugged again with the `--keep` flag on the same test, we'll get a new directory `test01-init_1` and so on.

Once you've found the command that errors, do it all again except now you can hit `s` (short for "step") to "step into" that command, so to speak. Really what happens is the IntelliJ JVM waits for our script to start and then attaches itself to that execution. So after you press `s`, you should hit the "Debug" button in IntelliJ. Make sure in the top right the configuration is set to the name of the remote JVM config you added earlier (this is why it is helpful to give it a good name).

This will stop your program at wherever your breakpoint was as it's trying to run that command you hit `s` on. Now you can use your normal debugging techniques to step around and see if you're improperly reading/writing some data or some other mistake.

did everything it was supposed to: in these cases, it means you had a bug on a previous command with persistence. For example: let's say your second invocation looks like it is doing everything correctly, except when it tries to read the initial commit (that should have been persistently stored in a file) it receives a blank file (or maybe the file isn't even there). Then, even though the second execution of the program has output that doesn't match the expected, it was really the previous (first) execution that has the bug since it didn't properly persist the data.

These are very common since persistence is a new and initially tricky concept, so when debugging, your first priority is to find the execution that produced the bug. If you didn't, then you would be debugging the second (non-buggy) execution for hours to no avail, since the bug already happened.

Understanding Acceptance Tests

The first thing we'll ask for in Gitbugs and when you come to receive help in Office Hours is a test that you're failing, so it's paramount that you learn to write tests in this project. We've done a lot of work to make this as painless as possible, so please take the time to read through this section so you can understand the provided tests and write good tests yourself.

The provided tests are hardly comprehensive, and you'll definitely need to write your own tests to get a full score on the project. To write a test, let's first understand how this all works.

Here is the structure of the `testing` directory:

```
.
├── Makefile
├── student_tests                          <==== Your .in fi
```

```
│   ├── test02-basic-checkout.in
│   ├── test03-basic-log.in
│   ├── test04-prev-checkout.in
│   └── definitions.inc
├── src                       <==== Contains fi
│   ├── notwug.txt
│   └── wug.txt
├── runner.py                 <==== Script to h
```

Just like Capers, these tests work by creating a temporary directory within the `testing` directory and running the commands specified by a `.in` file. If you use the `--keep` flag, this temporary directory will remain after the test finishes so you can inspect it.

Unlike Capers, we'll need to deal with the *contents* of files in our working directory. So in this `testing` folder, we have an additional folder called `src`. This directory stores many pre-filled `.txt` files that have particular contents we need. We'll come back to this later, but for now just know that `src` stores actual file contents. `samples` has the `.in` files of the sample tests (which are the checkpoint tests). When you create your own tests, you should add them to the `student_tests` folder which is initially empty in the skeleton.

The `.in` files have more functions in Gitlet. Here is the explanation straight from the `tester.py` file:

```
# ...  A comment, producing no effect.
I FILE Include.  Replace this statement with the conte
      interpreted relative to the directory containing
C DIR  Create, if necessary, and switch to a subdirect
      the main directory for this test.  If DIR is mis
      back to the default directory.  This command is
      intended to let you set up remote repositories.
T N    Set the timeout for gitlet commands in the rest
      seconds.
+ NAME F
      Copy the contents of src/F into a file named NAM
- NAME
      Delete the file named NAME.
> COMMAND OPERANDS
LINE1
LINE2
```

Navigation

```
        Run gitlet.Main with COMMAND ARGUMENTS as its pa
        its output with LINE1, LINE2, etc., reporting an
        "sufficient" discrepancy.  The <<< delimiter may
        an asterisk (*), in which case, the preceding li
        Python regular expressions and matched according
        or JAR file containing the gitlet.Main program i
        in directory DIR specified by --progdir (default
= NAME F
        Check that the file named NAME is identical to s
        error if not.
* NAME
        Check that the file NAME does not exist, and rep
        does.
E NAME
        Check that file or directory NAME exists, and re
        does not.
D VAR "VALUE"
        Defines the variable VAR to have the literal val
        taken to be a raw Python string (as in r"VALUE")
```

Don't worry about the Python regular expressions thing mentioned in the above description: we'll show you that it's fairly straightforward and even go through an example of how to use it.

Let's walk through a test to see what happens from start to finish. Let's examine `test02-basic-checkout.in`.

Sample test

When we first run this test, a temporary directory gets created that is initially empty. Our directory structure is now:

```
.
├── Makefile
├── student_tests
├── samples
│   ├── test01-init.in
│   ├── test02-basic-checkout.in
│   ├── test03-basic-log.in
│   ├── test04-prev-checkout.in
│   └── definitions.inc
├── src
│   ├── notwug.txt
│   └── wug.txt
├── test02-basic-checkout_0          <==== Just create
```

This temporary directory is the Gitlet repository that will be used for this execution of the test, so we will add things there and run all of our Gitlet commands there as well. If you ran the test a second time without deleting the directory, it'll create a new directory called `test02-basic-checkout_1`, and so on. Each execution of a test uses it's own directory, so don't worry about tests interfering with each other as that cannot happen.

The first line of the test is a comment, so we ignore it.

The next section is:

```
> init
<<<
```

This shouldn't have any output as we can tell by this section not having any text between the first line with `>` and the line with `<<<`. But, as we know, this should create a `.gitlet` folder. So our directory structure is now:

```
.
├── Makefile
├── student_tests
├── samples
│   ├── test01-init.in
│   ├── test02-basic-checkout.in
│   ├── test03-basic-log.in
│   ├── test04-prev-checkout.in
│   └── definitions.inc
├── src
│   ├── notwug.txt
│   └── wug.txt
├── test02-basic-checkout_0
│   └── .gitlet                        <==== Just created
├── runner.py
└── tester.py
```

The next section is:

```
+ wug.txt wug.txt
```

hand side from the src directory and copy its contents to the file on the left-hand side in the temporary directory (creating it if it doesn't exist). They happen to have the same name, but that doesn't matter since they're in different directories. After this command, our directory structure is now:

```
.
├── Makefile
├── student_tests
├── samples
│   ├── test01-init.in
│   ├── test02-basic-checkout.in
│   ├── test03-basic-log.in
│   ├── test04-prev-checkout.in
│   └── definitions.inc
├── src
│   ├── notwug.txt
│   └── wug.txt
├── test02-basic-checkout_0
│   ├── .gitlet
│   └── wug.txt                   <==== Just created
├── runner.py
└── tester.py
```

Now we see what the `src` directory is used for: it contains file contents that the tests can use to set up the Gitlet repository however you wants. If you want to add special contents to a file, you should add those contents to an appropriately named file in `src` and then use the same `+` command as we have here. It's easy to get confused with the order of arguments, so make sure the right-hand side is referencing the file in the `src` directory, and the left-hand side is referencing the file in the temporary directory.

The next section is:

```
> add wug.txt
<<<
```

As you can see, there should be no output. The `wug.txt` file is now staged for addition in the temporary directory. At this point, basic-checkout_0/.gitlet directory since you'll need to somehow persist the fact that wug.txt is staged for addition.

The next section is:

```
> commit "added wug"
<<<
```

And, again, there is no output, and, again, your directory structure within .gitlet might change.

The next section is:

```
+ wug.txt notwug.txt
```

Since wug.txt already exists in our temporary directory, its contents changes to be whatever was in src/notwug.txt.

The next section is

```
> checkout -- wug.txt
<<<
```

Which, again, has no output. However, it should change the contents of wug.txt in our temporary directory back to its original contents which is exactly the contents of src/wug.txt. The next command is what asserts that:

```
= wug.txt wug.txt
```

This is an assertion: if the file on the left-hand side (again, this is in the temporary directory) doesn't have the exact contents of the file on the right-hand side (from the src directory), the testing script will error and say your file contents are not correct.

There are two other assertion commands available to you:

```
E NAME
```

Will assert that there exists a file/folder named NAME in the temporary directory. It doesn't check the contents, only that it

```
* NAME
```

Will assert that there does NOT exist a file/folder named `NAME` in the temporary directory. If there does exist a file/folder with that name, the test will fail.

That happened to be the last line of the test, so the test finishes. If the `--keep` flag was provided, the temporary directory will remain, otherwise it will be deleted. You might want to keep it if you suspect your `.gitlet` directory is not being properly setup or there is some issue with persistence.

Setup for a test

As you'll soon discover, there can be a lot of repeated setup to test a particular command: for example, if you're testing the `checkout` command you need to:

1. Initialize a Gitlet Repository
2. Create a commit with a file in some version (v1)
3. Create another commit with that file in some other version (v2)
4. Checkout that file to v1

And perhaps even more if you want to test with files that were untracked in the second commit but tracked in the first.

So the way you can save yourself time is by adding all that setup in a file and using the `I` command. Say we do that here:

```
# Initialize, add, and commit a file.
> init
<<<
+ a.txt wug.txt
> add a.txt
<<<
> commit "a is a wug"
<<<
```

directory, but with a file extension `.inc`, so maybe we name it `samples/commit_setup.inc`. If we gave it the file extension `.in`, our testing script will mistake it for a test and try to run it individually. Now, in our actual test, we simply use the command:

```
I commit_setup.inc
```

This will have the testing script run all of the commands in that file and keep the temporary directory it creates. This keeps your tests relatively short and thus easier to read.

We've included one `.inc` file called `definitions.inc` that will set up patterns for your convenience. Let's understand what patterns are.

Pattern matching output

The most confusing part of testing is the output for something like `log`. There are a few reasons why:

1. The commit SHA will change as you modify your code and hash more things, so you would have to continually modify your test to keep up with the changes to the SHA.
2. Your date will change every time since time only moves forwards.
3. It makes the tests very long.

We also don't really care the exact text: just that there is some SHA there and something with the right date format. For this reason, our tests use pattern matching.

This is not a concept you will need to understand, but at a high level we define a pattern for some text (i.e. a commit SHA) and then just check that the output has that pattern (without caring about the actual letters and numbers).

Here is how you'd do that for the output of `log` and check that it matches the pattern:

Navigation

```
# You would add your lines here that create commits wi
# specified messages. We'll omit this for this example
> log
===
${COMMIT_HEAD}
${DATE}
added wug

===
${COMMIT_HEAD}
${DATE}
initial commit

<<<*
```

The section we see is the same as a normal Gitlet command, except it ends in `<<<*` which tells the testing script to use patterns. The patterns are enclosed in `${PATTERN_NAME}`.

All the patterns are defined in `samples/definitions.inc`. You don't need to understand the actual pattern, just the thing it matches. For example, `HEADER` matches the header of a commit which should look something like:

```
commit fc26c386f550fc17a0d4d359d70bae33c47c54b9
```

That's just some random commit SHA.

So when we create the expected output for this test, we'll need to know how many entries are in this log and what the commit messages are.

You can do similar things for the `status` command:

```
I definitions.inc
# Add commands here to setup the status. We'll omit th
> status
=== Branches ===
\*master

=== Staged Files ===
g.txt

=== Removed Files ===
=== Untracked Files ===
${ARBLINES}
```

The pattern we used here is `ARBLINES` which is arbitrary lines. If you actually care what is untracked, then you can add that here without the pattern, but perhaps we're more interested in seeing `g.txt` staged for addition.

Notice the `\*` on the branch `master`. Recall that in the `status` command, you should prefix the HEAD branch with a `*`. If you use a pattern, you'll need to replace this `*` with a `\*` in the expected output. The reason is out of the scope of the class, but it is called "escaping" the asterisk. If you don't use a pattern (i.e. your command ends in `<<<` not `<<<*`, then you can use the `*` without the `\`).

The final thing you can do with these patterns is "save" a matched portion. **Warning**: this seems like magic and we don't care at all if you understand how this works, just know that it does and it is available to you. You can copy and paste the relevant part from our provided tests so you don't need to worry too much about making these from scratch. With that out of the way, let's see what this is.

If you're doing a `checkout` command, you need to use the SHA identifier to specify which commit to checkout to/from. But remember we used patterns, so we don't actually know the SHA identifier at the time of creating the test. That is problematic. We'll use `test04-prev-checkout.in` to see how you can "capture" or "save" the SHA:

```
I definitions.inc
# Each ${COMMIT_HEAD} captures its commit UID.
> log
===
${COMMIT_HEAD}
${DATE}
```

```
===
${COMMIT_HEAD}
${DATE}
version 1 of wug.txt

===
${COMMIT_HEAD}
${DATE}
initial commit

<<<*
```

This will set up the UID (SHA) to be captured after the `log` command. So right after this command runs, we can use the `D` command to define the UIDs to variables:

```
# UID of second version
D UID2 "${1}"
# UID of first version
D UID1 "${2}"
```

Notice how the numbering is backwards: the numbering begins at 1 and starts at the top of the log. That is why the current version (i.e. second version) is defined as `"${1}"`. We don't care about the initial commit, so we don't bother capturing it's UID.

Now we can use that definition to checkout to that captured SHA:

```
> checkout ${UID1} -- wug.txt
<<<
```

And now you can make your assertions to ensure the checkout was successful.

Testing conclusion

There are many more complex things you can do with our testing script, but this is enough to write very good tests. You should use our provided tests as an example to get started, and also feel free to discuss on Piazza high level ideas of how to test things. You may also share your `.in` files, but please make sure they're students and staff can see what is going on.

Design Document and Checkpoint

Since you are not working from a substantial skeleton this time, we are asking that everybody submit a design document describing their implementation strategy. It is not graded, but we will insist on having it before helping you with bugs in your program (in Office Hours or via Gitbugs) . See Lab 13 for details on writing a high quality design document.

There will be an initial **required** checkpoint for the project, due **Friday 4/22 at 11:59PM.** The checkpoint is worth **4 points**. It consists of a programming portion, as well as a conceptual quiz on Gradescope.

You can complete the conceptual quiz on Gradescope by clicking on the assignment titled Projet 3: Gitlet Checkpoint Quiz. The quiz is out of 1 point, and tests your understanding of the Gitlet commands. You have unlimited tries to complete the quiz before the deadline, and should see feedback on the answers you choose. **The project lateness policy will not apply to the checkpoint quiz, so any late quizzes will receive a score of 0.**

For the remaining 3 points from the checkpoint, you can submit the programming portion using the tag `proj3a-` *n* (where, as usual, *n* is simply an integer.) The checkpoint autograder will check that

- Your program compiles.
- You pass the sample tests from the skeleton: `testing/samples/*.in`. These require you to implement

[commit id] -- [file name], and log.

In addition, it will comment on (but not score):

- Whether you pass style checks (it will ignore FIXME comments for now; we won't in the final submission.)
- Whether there are compiler warning messages.
- Whether you pass your own unit and acceptance tests (as run by `make check`).

We ***will*** score these in your final submission.

For the checkpoint grader, when it is released on Tuesday 4/12, you will be able to submit once every 6 hours with full grader outputs. Starting Monday 4/18, you will be able to submit once every 3 hours with full grader outputs. On the due date, Friday 4/22, there will be no restrictions on the grader.

NOTE: The checkpoint grader restrictions are much more lenient than the main project 3 autograder.

Grader Details

The due date for Project 3 is **Friday 4/29 at 11:59 PM**. We will be grading on style, our acceptance tests, and your tests for a total of 24 points.

On release, you will be able to submit once a day without any grader outputs. Starting Wednesday 4/20, you will be able to submit once a day with grader outputs. On Thursday 4/28, you will be able to submit once every six hours, with grader outputs. On Friday 4/29, you will be able to submit once every three hours with grader outputs. On Friday 4/29 at 11pm, you will be able to submit once every 15 minutes, and there are no restrictions to the grader after the deadline.

updated `Makefile` and `tester.py`. These have been changed to align with the autograder on Gradescope. When running `make check`, each test will be run 5 times. To do this while running tests individually with `tester.py`, please add the flags `--reps=5` to set how many times the tests are run. This is done to help avoid any non-determinism that may come up when running your tests. When running your tests multiple times, every run of that test must pass in order to get credit for that test.

Things to Avoid

There are few practices that experience has shown will cause you endless grief in the form of programs that don't work and bugs that are very hard to find and sometimes not repeatable ("Heisenbugs").

1. Since you are likely to keep various information in files (such as commits), you might be tempted to use apparently convenient file-system operations (such as listing a directory) to sequence through all of them. Be careful. Methods such as `File.list` and `File.listFiles` produce file names in an undefined order. If you use them to implement the `log` command, in particular, you can get random results.

2. Windows users especially should beware that the file separator character is `/` on Unix (or MacOS) and `\` on Windows. So if you form file names in your program by concatenating some directory names and a file name together with explicit `/`s or `\`s, you can be sure that it won't work on one system or the other. Java provides a system-dependent file separator character `File.separator`, as in `".gitlet" + File.separator + "something"`, or the multi-argument constructors to `File`, as in \ `new`

place of ".gitlet/something").

3. **Be careful using a `HashMap` when serializing!** The order of things within the `HashMap` is non-deterministic. The solution is to use a `TreeMap` which will always have the same order. More details here. Specifically, iterating through a `HashMap` can lead to non-deterministic behavior, which is avoided with a TreeMap.

Going Remote (Extra Credit)

This is the first of two possible extra-credit features. To save you from yourselves, we will only give credit for one of them, and will ignore the other. Of course, you can do both nevertheless (out of the sheer joy of programming), but don't expect any extra extra credit as a result.

This project is all about mimicking git's local features. These are useful because they allow you to backup your own files and maintain multiple versions of them. However, git's true power is really in its *remote* features, allowing collaboration with other people over the internet. The point is that both you and your friend could be collaborating on a single code base. If you make changes to the files, you can send them to your friend, and vice versa. And you'll both have access to a shared history of all the changes either of you have made.

To get extra credit, implement some basic remote commands: namely `add-remote`, `rm-remote`, `push`, `fetch`, and `pull` You will get 3 extra-credit points for completing them. Don't attempt or plan for extra credit until you have completed the rest of the project.

Depending on how flexibly you have designed the rest of the project, 3 extra-credit points may not be worth the amount of everyone to do it. Our priority will be in helping students complete the main project; if you're doing the extra credit, we expect you to be able to stand on your own a little bit more than most students.

The Remote Commands

A few notes about the remote commands:

- Execution time will not be graded. For your own edification, please don't do anything ridiculous, though.
- All the commands are significantly simplified from their git equivalents, so specific differences from git are usually not notated. Be aware they are there, however.

So now let's go over the commands:

add-remote

- **Usage**: `java gitlet.Main add-remote [remote name] [name of remote directory]/.gitlet
- **Description**: Saves the given login information under the given remote name. Attempts to push or pull from the given remote name will then attempt to use this `.gitlet` directory. By writing, e.g., java gitlet.Main add-remote other ../testing/otherdir/.gitlet you can provide tests of remotes that will work from all locations (on your home machine or within the grading program's software). Always use forward slashes in these commands. Have your program convert all the forward slashes into the path separator character (forward slash on Unix and backslash on Windows). Java helpfully defines the class variable `java.io.File.separator` as this character.

exists, print the error message: `A remote with that name already exists.` You don't have to check if the user name and server information are legit.

- **Dangerous?**: No.

rm-remote

- **Usage**: `java gitlet.Main rm-remote [remote name]`
- **Description**: Remove information associated with the given remote name. The idea here is that if you ever wanted to change a remote that you added, you would have to first remove it and then re-add it.
- **Failure cases**: If a remote with the given name does not exist, print the error message: `A remote with that name does not exist.`
- **Dangerous?**: No.

push

- **Usage**: `java gitlet.Main push [remote name] [remote branch name]`
- **Description**: Attempts to append the current branch's commits to the end of the given branch at the given remote. Details:

  This command only works if the remote branch's head is in the history of the current local head, which means that the local branch contains some commits in the future of the remote branch. In this case, append the future commits to the remote branch. Then, the remote should reset to the front of the appended commits (so its head will be the same as the local head). This is called fast-forwarding.

  If the Gitlet system on the remote machine exists but does not have the input branch, then simply add the branch to

- **Failure cases**: If the remote branch's head is not in the history of the current local head, print the error message `Please pull down remote changes before pushing.` If the remote `.gitlet` directory does not exist, print `Remote directory not found.`
- **Dangerous?**: No.

fetch

- **Usage**: `java gitlet.Main fetch [remote name] [remote branch name]`
- **Description**: Brings down commits from the remote Gitlet repository into the local Gitlet repository. Basically, this copies all commits and blobs from the given branch in the remote repository (that are not already in the current repository) into a branch named `[remote name]/[remote branch name]` in the local `.gitlet` (just as in real Git), changing `[remote name]/[remote branch name]` to point to the head commit (thus copying the contents of the branch from the remote repository to the current one). This branch is created in the local repository if it did not previously exist.
- **Failure cases**: If the remote Gitlet repository does not have the given branch name, print the error message `That remote does not have that branch.` If the remote `.gitlet` directory does not exist, print `Remote directory not found.`
- **Dangerous?** No

pull

- **Usage**: `java gitlet.Main pull [remote name] [remote branch name]`

Navigation

branch name] as for the fetch command, and then merges that fetch into the current branch.

- **Failure cases**: Just the failure cases of `fetch` and `merge` together.
- **Dangerous?** Yes!

Diffs (Extra Credit)

This is the second of two possible extra-credit features. Again, we will only count one of them.

Usage: `java gitlet.Diff [ branch-name [ branch-name ] ]`

Description: The `git diff` command compares the contents of a commit with a working directory or compares two commits, and presents any differences as a *unified diff*. Suppose that the latest commit on the current branch contains the following three files:

```
    f.txt                g.txt                h.txt
    ---------------------------------------------------
    Line 1.              This is a wug.       This is not
    Line 2.
    Line 3.
    Line 4.
    Line 5.
    Line 6.
    Line 7.
    Line 8.
    Line 9.
    Line 10.
    Line 11.
    Line 12.
    Line 13.
    Line 14.
    Line 15.
    Line 16.
    Line 17.
```

and suppose that the working directory contains the following two files:

```
Line 0.            This is a wug.
Line 0.1.
Line 1.
Line 3.
Line 4.
Line 7.
Line 8.
Line 9.
Line 9.1.
Line 9.2.
Line 10.
Line 11.
Line 11.1.
Line 12.
Line 13.1
Line 14.
Line 15.
Line 16.1
Line 17.1
Line 18.
```

Performing the command `java gitlet.Main diff` should produce the following output:

```
diff --git a/f.txt b/f.txt
--- a/f.txt
+++ b/f.txt
@@ -0,0 +1,2 @@
+Line 0.
+Line 0.1.
@@ -2 +3,0 @@
-Line 2.
@@ -5,2 +5,0 @@
-Line 5.
-Line 6.
@@ -9,0 +9,2 @@
+Line 9.1.
+Line 9.2.
@@ -11,0 +13 @@
+Line 11.1.
@@ -13 +15 @@
-Line 13.
+Line 13.1
@@ -16,2 +18,3 @@
-Line 16.
-Line 17.
+Line 16.1
+Line 17.1
+Line 18.
diff --git a/h.txt /dev/null
```

```
    @@ -1 +0,0 @@
    -This is not a wug.
```

The two lines starting `diff --git` indicate the start of the differences for one of the files in the two versions (the head of the current branch and the current (uncommitted) contents of the working directory. These `diff` lines are followed by a `---` and `+++` line, also giving the file name. Next come a sequence of *edits*, each starting with a line beginning and ending with `@@`. The entry

```
    @@ -L1,N1 +L2,N2
```

says that this entry indicates that to get the second version of the file from the first, one removes N1 lines starting at line L1 of the first version, and inserts N2 lines from the second version starting with line L2 of the second version. When N1 is 0, L1 is the number of lines before the lines to be deleted. Likewise, when N2 is 0, L2 is the number of lines in the second version that correspond to the lines preceding line L1 in the first version. When N1 or N2 is 1, it (and the comma in front of it) is not printed (e.g., `@@ -13 +15 @@` in the example.) When one of the versions of the file is missing (as for `h.txt` here), its name is printed as `/dev/null` (the standard Unix empty file), and it is in fact treated as an empty file when comparing. If both versions of a file are identical, no diff is given for that file.

With no arguments, we compare the commit at the head of the current branch with the files in the working directory. Any files in the working directory and not in the current branch are ignored.

With one argument, as in `git diff branch1`, the contents of the commit at the head of the branch named `branch1` are compared to the versions in the working directory.

With two arguments, as in `git diff branch1 branch2`, the contents of the commit at the head of the branch named `branch1` way, that the first (a/) file can be /dev/null.

Failure cases: If the branch in the one-argument case does not exist, print the message `A branch with that name does not exist.` If one or both of the branches in the two-argument case does not exist, print the message `At least one branch does not exist.`

Dangerous?: No.

Diff Utility

We have provided a file `gitlet/Diff.java` in the skeleton, which will compare two sequences of lines read from files, and compute the edits needed to convert one to the other. You will have to read its comments and figure out how to use it to produce the desired output format.

This utility works by finding a *longest common subsequence* of lines in the two files—a subsequence that appears in both files, but not necesaarily in the same place. The lines of the subsequence may be scattered in different ways in the two files (they are not necessarily adjacent to each other), but the lines appear in the same order in each. From such a subsequence, the utility is able to figure out how to change one file into the other by modifying a minimal number of lines.