**Name: Zameer Ahmed**

**ID: 90641**

**SWE 525 Version Control Git Homework 2 08/23/2016**

**Please complete the following homework and submit it on EMS. Also push your repo to git and put your github repo link in your assignment. Complete this homework before May 3 2016.**

**Part A. You will create a local repository and perform basic operations on it.**

1. Create (somewhere on your computer) a new empty folder named SWE525Hw2. In that folder, create a plain text file with your name and student ID # on a single line. Don't use MS Word for this, use a text editor, for example Notepad or TextEdit (you may need to choose Plain Text under Format or Preferences). Save the file as file1.txt.
2. In your Git shell, navigate to any folder. Use the [**cd** command](http://frost.ics.uci.edu/inf43/cd-info.html) to change your current folder/directory. Note that the Windows Git Bash shell follows Unix/Linux shell conventions so if you're on Windows, you still need to use Linux-style paths with forward slashes (e.g., c:\my\_folder\my\_subfolder would be /c/my\_folder/my\_subfolder). Linux commands like ls, pwd, and grep should all work in the Git Bash shell.
3. Note: At this point you may find that Git wants you to tell it your name and email address. You can do this with two commands like these:
   * **git config --global user.name "Peter Anteater"**
   * **git config --global user.email "panteat@itu.edu"**
   * To look at all your configuration information: **git config --global -1**
4. Create a local Git repo by running the command **git init**.
5. Run **git status**. Note that file1.txt is listed as untracked. We want Git to track it, so run **git add file1.txt**. When you "add" a file you are telling Git to keep track of it. "add" also tells Git to *stage* the file, which means put it in the stage of being ready to be committed.
6. Run **git status** again. Note that file1.txt is now listed as a file to be committed (i.e., it's staged).
7. Let's commit file1.txt to our repository. Run **git commit -m "Committing a new file with my name"**. When you "commit,", you in effect copy all staged files to the repository. The "-m" is a flag (that's what the hyphen indicates) which tells Git that the following string is a message to record with the commit.
8. Run **git log**. This will display the history of changes made to the repository. The one and only entry will be for the commit of file1.txt you just did.
9. Edit file1.txt and change the spelling of your name to something incorrect. Save file1.txt with the error. (This small error stands in for a long complex series of edits that you want to undo.)
10. Run **git reset --hard**. "reset --hard" removes all uncommitted changes, so all files in the repo will return to their contents as of the latest commit. There are many ways to undo changes in git, and "reset --hard" is generally considered to be dangerous. Look at file1.txt and observe the effect of reset --hard.
11. Edit file1.txt to *remove* your student ID# and *include* the name of your major, and save the file.
12. Commit with **git commit -m "Now has my major"**. This doesn't work. Git tells you there are "changes not staged for commit".
13. Try again with **git commit -a -m "Now has my major"**. The power of the "-a" flag is that it tells git to automatically stage all tracked, modified files before the commit.
14. You can also explicitly stage a file. Add the name of your favorite restaurant and favorite movie to to file1.txt, save it, and run **git stage file1.txt**. Now run **git commit -m "Added favorite restaurant"** to commit. "git stage" is really just another name for "git add".
15. You set the commit message to "Added favorite restaurant", but the file also includes your favorite movie, so maybe we should have included that in our commit message. Amend your commit message with **git commit --amend -m "Added favorite restaurant and movie"**.
16. Run **git log** to make sure you have successfully changed history.
17. You removed your student ID# a few steps back. Let that edit stand in for deleting, a few months ago, a block of code that you now want to examine. git will help you go back in time. Note that each commit has long, seemingly random, string of hexadecimal digits associated with it. This is called a "hash" and is a unique identifier for the commit. Find the hash associated with the “Committing a new file with my name“ commit. Run **git checkout xxxx**, replacing **xxxx** with the first four digits from that hash (thankfully typing in the entire hash is not required). You will see a frightening message about a detached HEAD.
18. git can keep track of separate, parallel, streams of edits to a project. Each stream of edits is called a branch, and a branch can have a name. For instance, multiple programmers who are working on and commiting changes to the same file will probably establish different branches. HEAD is git-ese for the current (not necessarily the last) commit in the current branch. Since we've gone back in time and are potentially (but haven't yet) starting a new branch, HEAD is "detached" (from any established, named branch). Ouch!
19. Take a look at file1.txt and note the later-deleted Student ID#. Now to return to the present: **git checkout master**. "master" is the name of the default branch created when the repository was made. Look at file1.txt again. Run **git log** again and you'll see it has the same three commits.
20. Create a new text file called file2.txt that contains your expected graduation year and first job title on a single line.
21. Stage file2.txt, and then commit it with a useful message.
22. Run **git log**. Notice that you see log entries for both commits that you've performed.
23. Run **git log file2.txt**. Notice that you only see the log entry involving file2.txt.
24. Modify file1.txt to have the name of your favorite color on a new line.
25. Delete file2.txt.
26. Run **git status**. Note that file2.txt is listed as deleted. Also note that the status information helpfully says "git add/rm ..." to update what will be committed.
27. You use the **git add** command to stage a new or modified file. However, to stage the deletion of a file, you need to use the **git rm** command. So run the commands **git add file1.txt** and **git rm file2.txt** to set the stage.
28. Commit the changes with the commit message "Deleting file2.txt".
29. Run the command **git log -p -3**. The -p flag will show you the diffs for each change. The -3 will limit what's displayed to the last 3 log entries. Take a few minutes to look carefully at the output log and see if you can figure out how to interpret it.
    * Note: A command such as **git log** sends text output to the shell using a bash command named "less" to display one windowfull of output at a time ("less" is named after a similar, earlier utility named "more", in a classic example of hilarious techie humor). At the **:** prompt, you can press **h** for help, **q** to exit, **Enter** to advance one line, or **Space** to advance one screenfull.
30. Now you decide you actually wanted to keep file2.txt, but you deleted it! Fortunately, you had added it to git, so you can still get it back. There are several ways to do this. The simplest is probably to use the command **git checkout HEAD~1 file2.txt**. What does this do? HEAD represents the most recent commit or snapshot. ~1 tells Git to go back one version from the most recent snapshot (i.e., HEAD). In this older snapshot, file2.txt still existed, and checkout tells Git to retrieve it. If you now look in your folder, you'll see file2.txt. And is file1.txt changed?
31. Run **git add file2.txt** to stage file2.
32. Run **git status**. Note that file2.txt is staged. Commit it with the commit message "Re-adding file2.txt".
33. Run the command **git log -p** to see all of the log entries.
34. Now run the command **git log -p > git\_log\_partB.txt**. (The **>** is a shell command that redirects the output of the program on **>**'s left to the file named on **>**'s right.)
35. Open git\_log\_partB.txt. It should like just like the output you saw for step 33. If you're on Windows and viewing it in Notepad, the spacing will probably look wrong, so try opening it in a different text editor (like Wordpad or Notepad++).
36. Upload git\_log\_partB.txt to the EMS by deadline

My Repo:

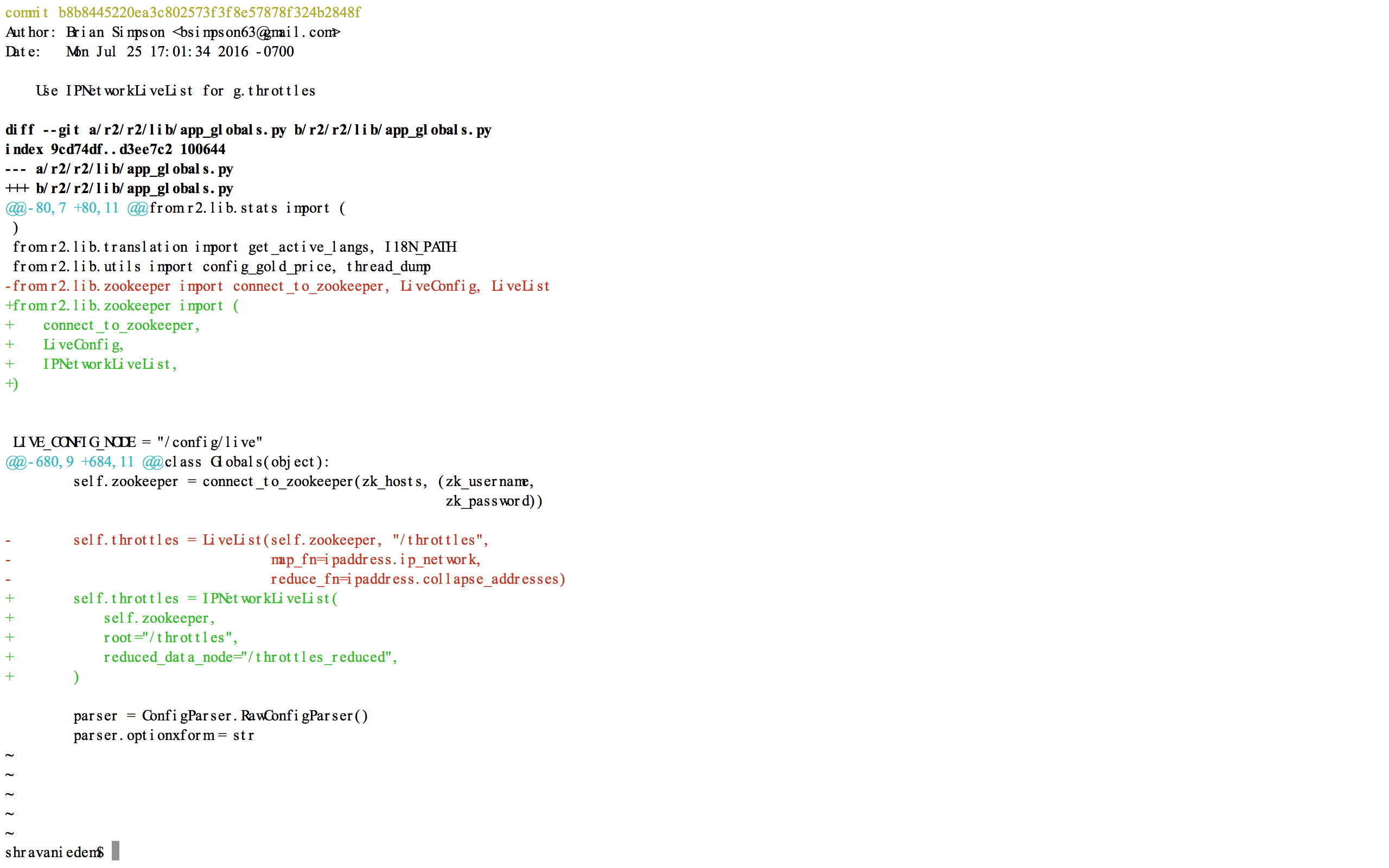
https://github.com/zameerahmed123/GIT\_Assignment\_2.git

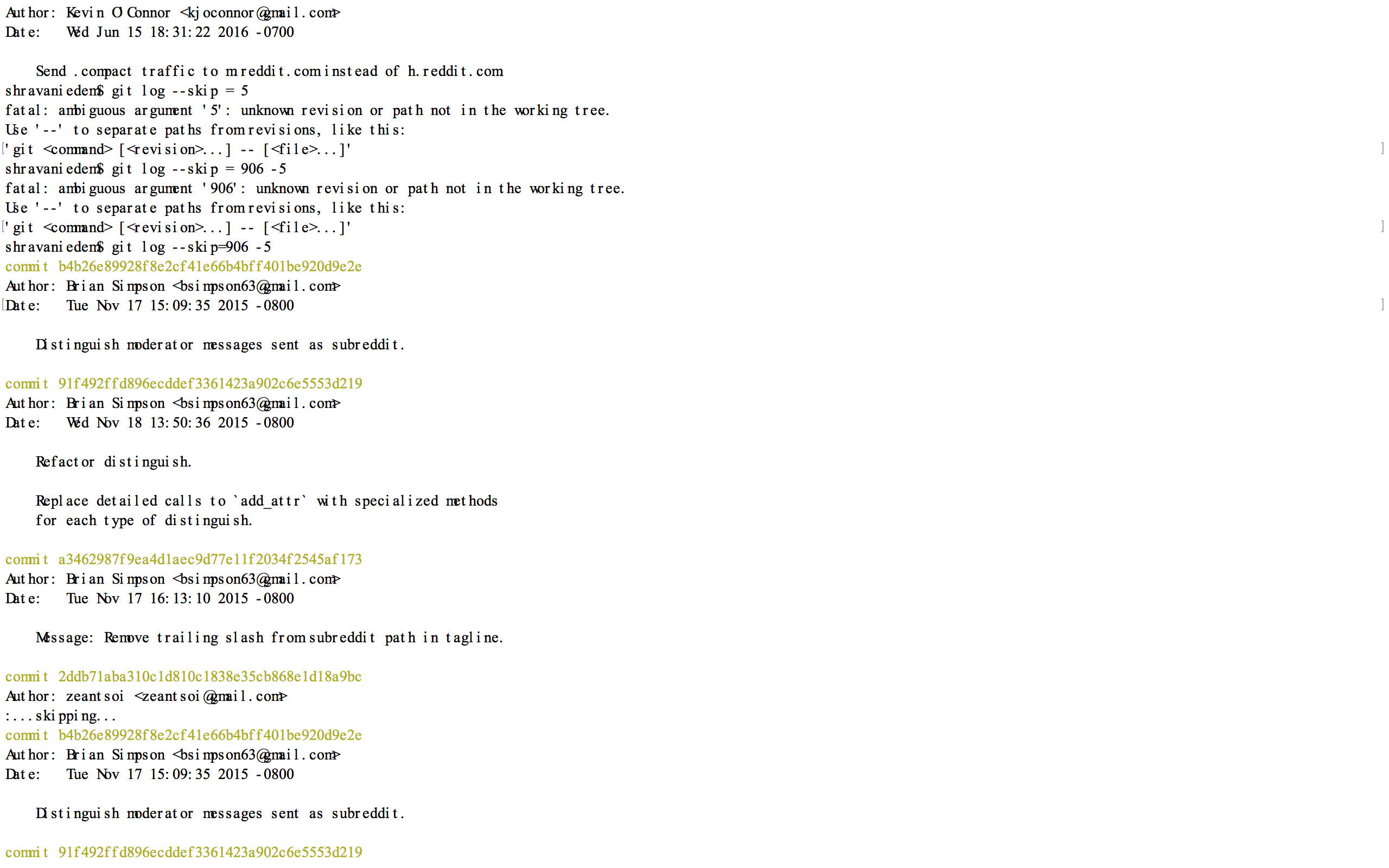
**Part B.** Much of Git's power comes from the interaction between your local repository and remote repositories. Remote Git repositories help you collaborate with others on a software project.

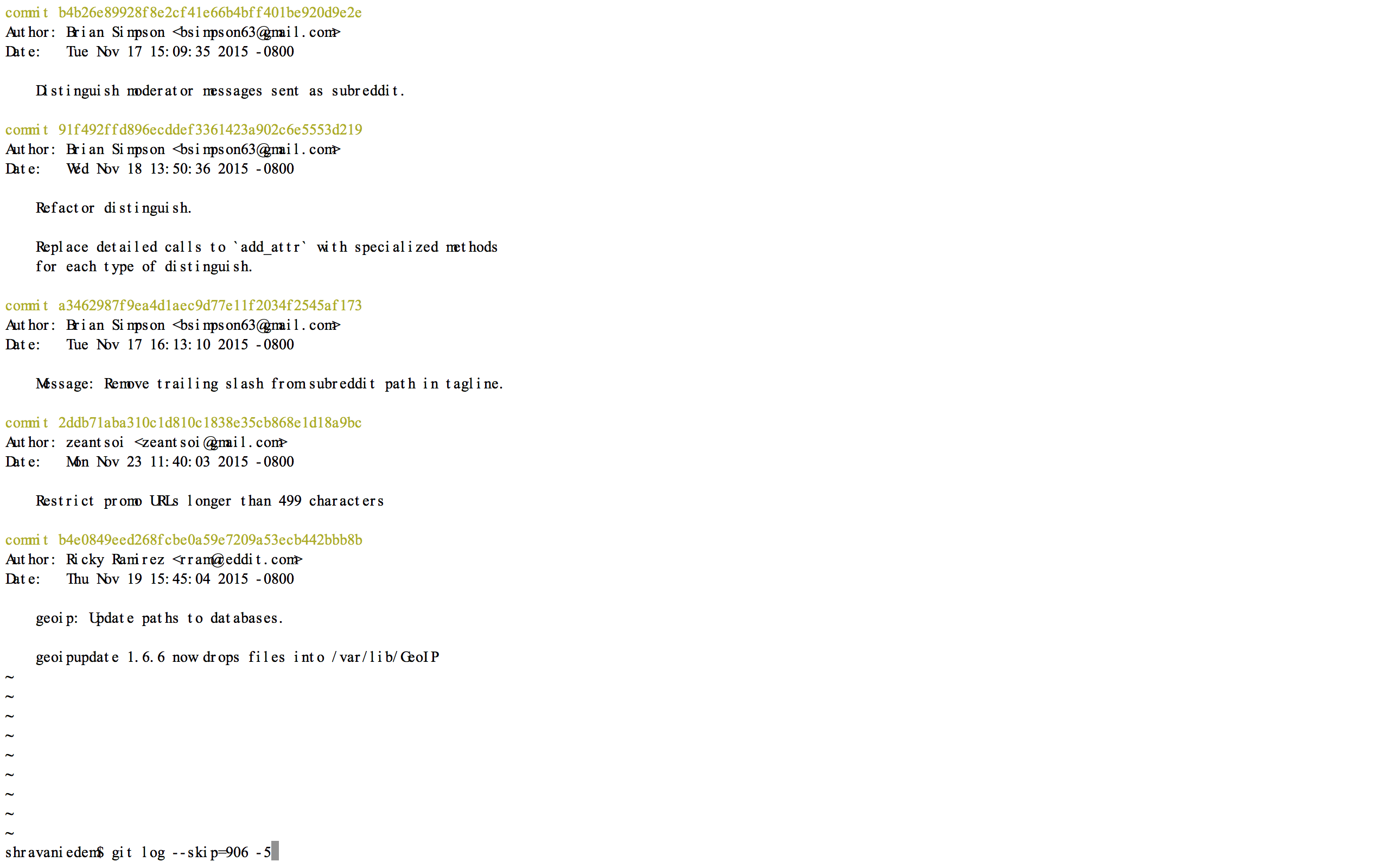
There are several services out there that will host a remote git repository for you for free, the most popular of which is GitHub.com. GitHub.com is particularly popular among the open source community, and there are pieces of software you probably use every day that are hosted on GitHub.com. [Reddit](http://reddit.com/) is one such software (web application) that is maintained on Github

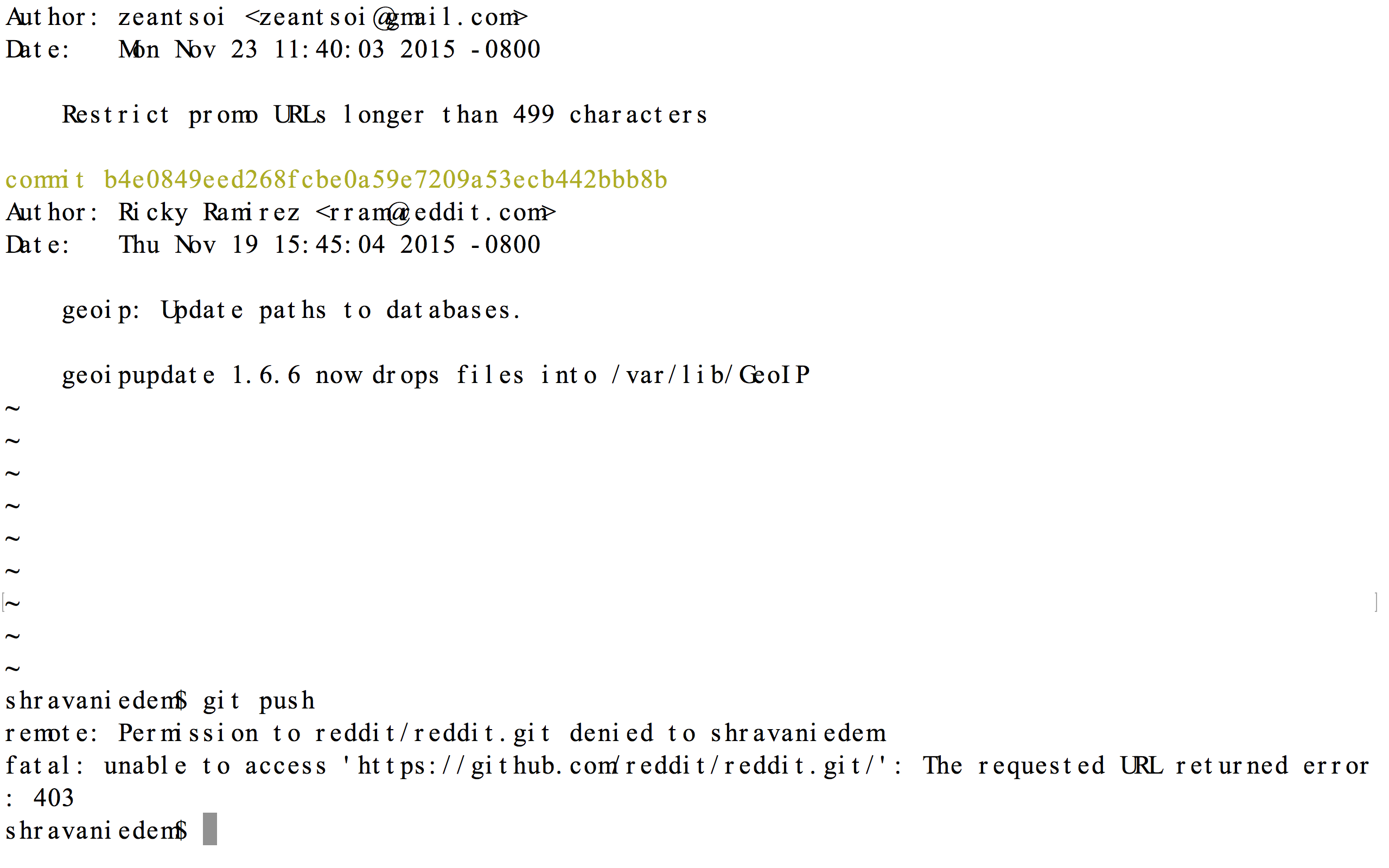
1. Let's use git to view the Reddit source code repository hosted on GitHub. Start git, use cd to navigate to an empty folder, and run the command **git clone https://github.com/reddit/reddit.git**.
2. You now have a local copy (on your computer) of the remote repository. It's important to understand that this is not only a copy of the source code, but also a copy of the history of changes stored by git. And since it's a local repository, all the commands you used in the previous part of the assignment will work. To try this out, run the command **cd reddit** to go inside the reddit project folder, then run **git log**. This shows you the most recent log messages for changes made to the Reddit source code.
3. Here's another variation on the git log command to try: **git log -1 -p --before='2014-03-31 11:52:45'**. This is nothing new except for the --before='2014-03-31 11:52:45' part. That tells git you only want to see log entries for changes made before March 31, 2014 at 11:52:45am. As you saw in part B, the -1 means you only want to see one entry, and the -p means you want to see a diff of the changes. As you can see, a contributor named "Andre D" changed the mouseover property to "cancelTimeout" from "queueShow" in the file r2/r2/public/static/js/saved.js.
4. Since we have a complete copy of the Reddit project's repository, we also have a copy of every snapshot going all the way back to the beginning of the project. To see the log entries for the earliest commits, run the command **git log --reverse**.
5. Recall that each commit is given a unique hash (aka SHA1). Many commands in Git can take a hash as input. For example, try **git log -1 4778b17e939e119417cc5ec25b82c4e9a65621b2** and **git show 4778b17e939e119417cc5ec25b82c4e9a65621b2**. (Don't forget that you can use only the first four digits of the hash. If git complains that the short SHA1 is ambiguous (because more than one commit has the same first four digits), try adding a few more digits from the long hash.)
6. One more git log option to know about is **--skip=*N***, where ***N*** is a non-negative integer. This means to skip ***N*** commits before starting to show the commit output. Try **git log --skip=100 -5**.
7. Run a log showing five commits to Reddit, skipping the first *N* commits, where *N* is the first three digits of your ITU student id number. For instance, if your id is 12345678, then *N* is 123. Leading zeros are no problem. Redirect the log output to a text file and upload it to EMS by deadline.
8. If you wanted (and had the permissions), you could modify part of the Reddit source, commit your changes (on your local repo), and then use **git push** to copy your local branch back to the GitHub repository. That's beyond the scope of Informatics 43, but it's good to know that git is a sophisticated tool that facilitates multiple people updating shared files, logging their updates, and assisting with the resolution of updates that conflict.



****







**Part C.**

You should write your homework in a file called hw2.txt. Submit your homework via git as described in the last question. Put your git link in the homework.

Create a new git repository somewhere in your filesystem, and configure a remote (called whatever you want,

My Repo:

https://github.com/zameerahmed123/GIT\_Assignment\_2.git

Sol:

New Repo created,

What command did you use to configure the remote?

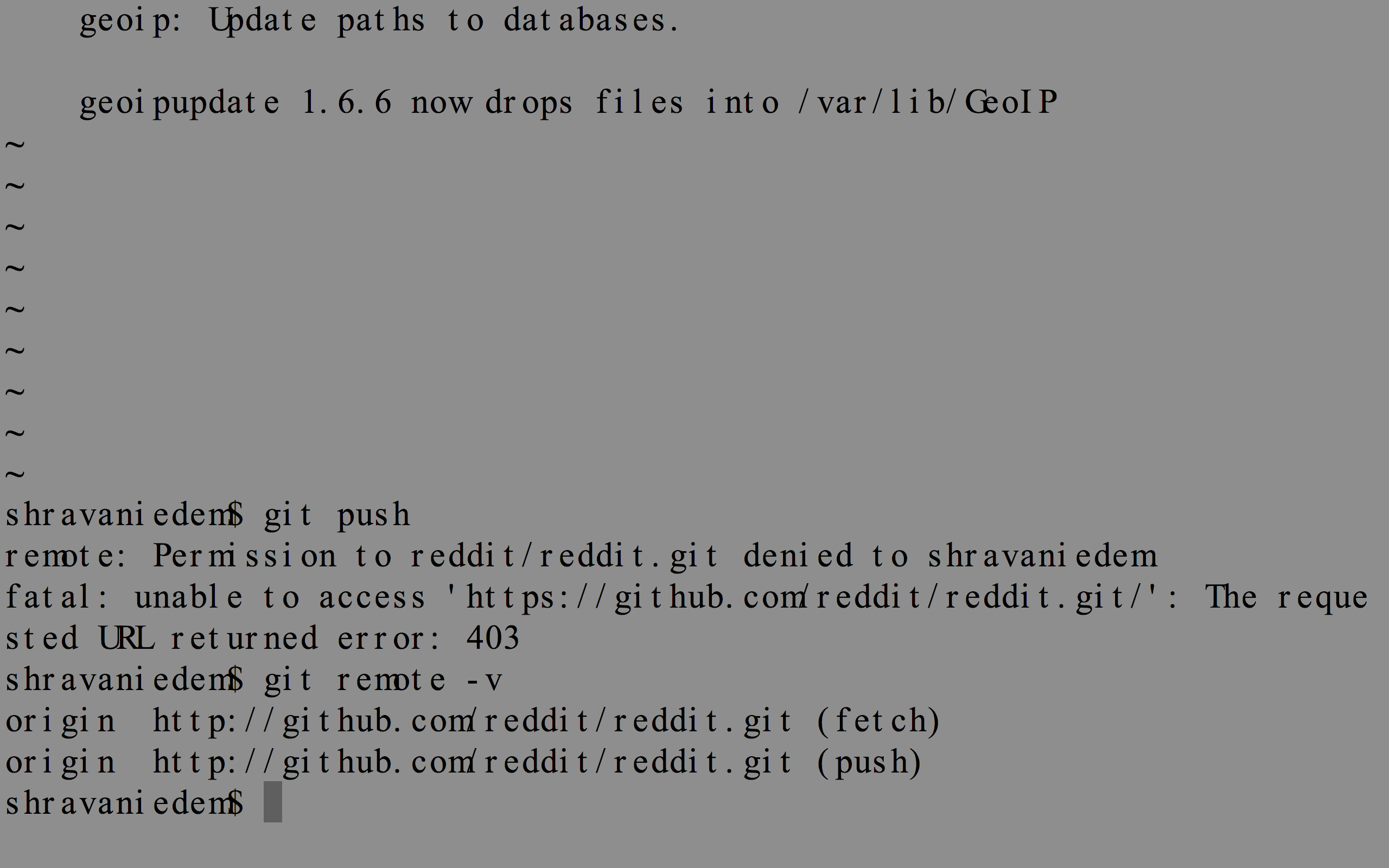
Commands: Git remote add origin

<https://github.com/zameerahmed123/GIT_Assignment_2.git>

Git push origin master

Instead of creating a new git repository and configuring it with this remote, I could have asked you to git clone the repo from afs. If you had done so, what would the output of git remote -v be?

Run git fetch remotename (substituting your remote name for remotename).



Did anything appear in your working directory? Why?

* Nothing appeared.

Update your local master branch with the remote one, and give the command you used to do so.

* $ git checkout <local branch name>

$ git merge <remote branch name>

Your working directory should have been updated. You should check that. That means the master branch was updated on the remote. But it's possible that when we ran git fetch, we fetched other branches as well! How could we (i.e. what command could we use) find all the branches we may have gotten from the remote?

* $ git fetch branch name

Spoiler: there's another branch. Check it out, find the file on that branch, and give the first five letters of that file as well as the command you used to checkout the branch. NOTE: It's very bad practice to directly git checkout the remote-tracking branch. Instead, try to checkout the branch so that it creates a local branch based on the remote-tracking branch. If you're not sure how (it's pretty easy to guess, though), you can look it up.

You're going to be submitting your homework on the master branch, so you may want to git checkout back there.

Command to check out remote branch

* $ git checkout --track -b <local branch> <remote>/<tracked branch>

What is a bare repository? What makes it different from normal repositories? In what situations is it used?

* If the user does not need the history of work then he creates bare repository.
* Generally, we use bare repository for team projects.
* One creates bare repository and shares with other team member.
* Every team member share his or her work using this repository.
* All repositories contains .git folder and every user’s commit history but bare repository does not contain .git folder.

In lecture, I was careful to explain the two-step procedure of using git fetch and git merge remotename/branchname to get commits from a remote. When would this be a good idea as opposed to simply doing git pull remotename branchname? Briefly discuss remote-tracking branches and their relevance in your answer.

**Sol:**

* Git fetch command fetches all updates from remote repository.
* The user can use “git fetch command” if he or she wants only update or if the user wants to check status of local repository by comparing remote repository.
* Git pull command is a combination of two commands such as git fetch and git merge.
* If we need all remote repository updates in local repository we needs to use git pull command.
* We use git pull command to fetch remote repository updates and to merge remote repo update with local repo update.
* Remote Tracking Branches get all data from central repository by using git pull and git fetch commands (git merge command only modifies tracking branches) User cannot modify remote tracking branches.
* Remote Tracking Branches get data from shared remote repository.

|  |  |
| --- | --- |
| **Branch Type** | **Access remote repository** |
| Remote tracking branch | Git fetch, git pull |
| Local branch | Git merge, git pull |

If you want to see, your remote tracking branches then use git branch –r command. This command will show list of remote tracking branches.

I did not ask you to do this, but suppose I asked you to make a commit on the 'other' branch. If I wanted you to push this branch, what command would you use?

* git push origin <branch name>

Submit this homework by typing your answers to these questions in a file called hw5.txt. You should commit this on the master branch of your repository. Most importantly, make sure to push the master branch!

I did not ask you to do this, but suppose I asked you to make a commit on the 'other' branch. If I wanted you to push this branch, what command would you use?

Submit this homework by typing your answers to these questions in a file called hw2.txt. You should commit this on the master branch of your repository. Most importantly, make sure to push the master branch!

**Part D.**

Why do we call the place we put stashes a "stack"? What git stash operation can we do that is usually impossible with stacks?

**Sol:**

* Git stash command put changes in stack so you can safely change your branch without committing last changes.
* While working on a project if there is something wrong and we want to switch branches for some work but do not want to commit the work and we just want to switch your branch.
* At this instance, we can put our changes in stack using git stash command.

Create a repository and create branch named githw2. Do the following on this branch (remember to use good commit message!)

Create files A.txt, B.txt, and C.txt and commit them all.

Make a new branch githw2-stashing, but do not checkout it.

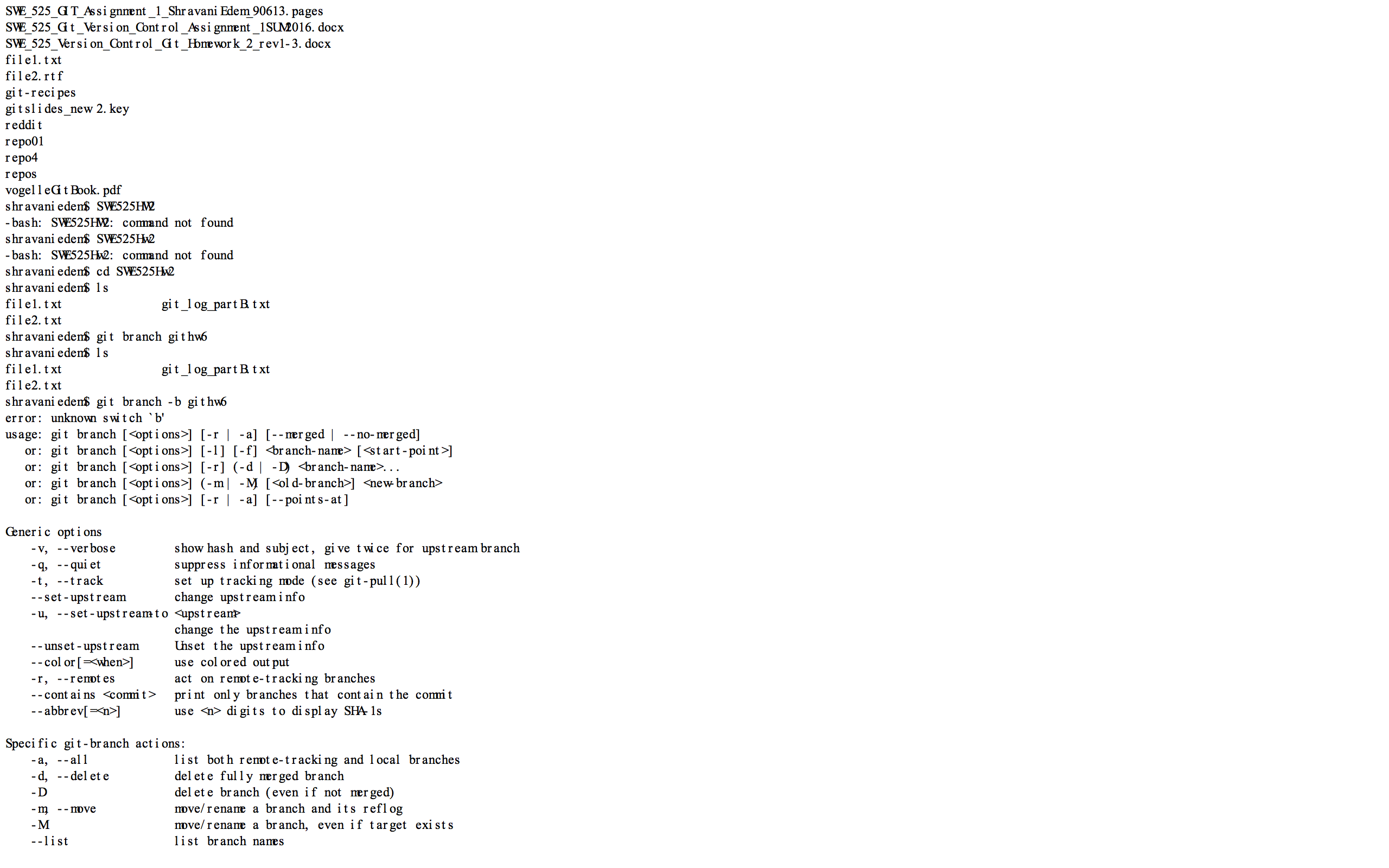
Make some modification to B.txt, delete C.txt, and commit these changes.

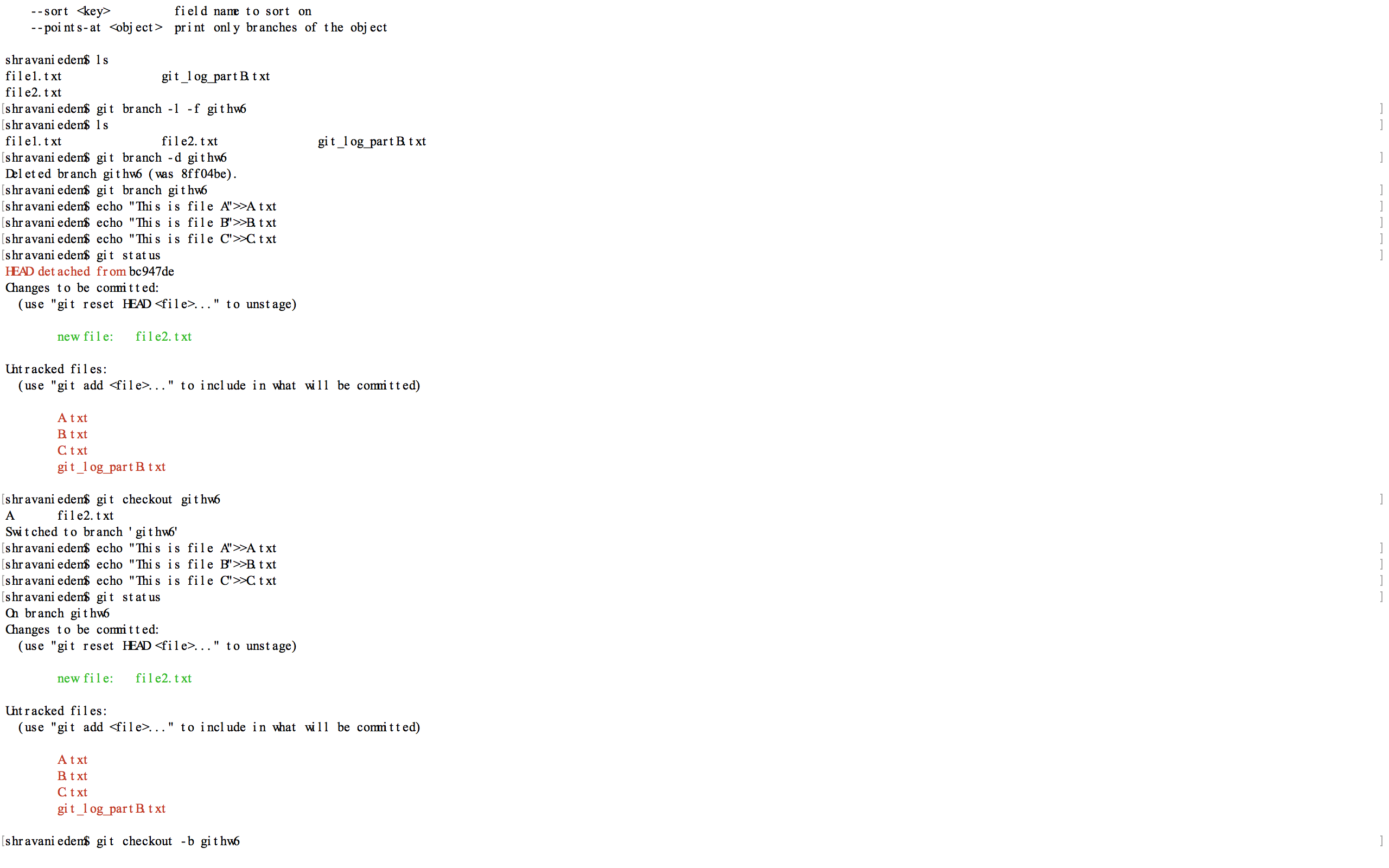
Create a file D.txt, rename A.txt to A2.txt, and commit these changes.

Push this branch to the afs remote. Do not push githw2-stashing yet.

**Sol:**

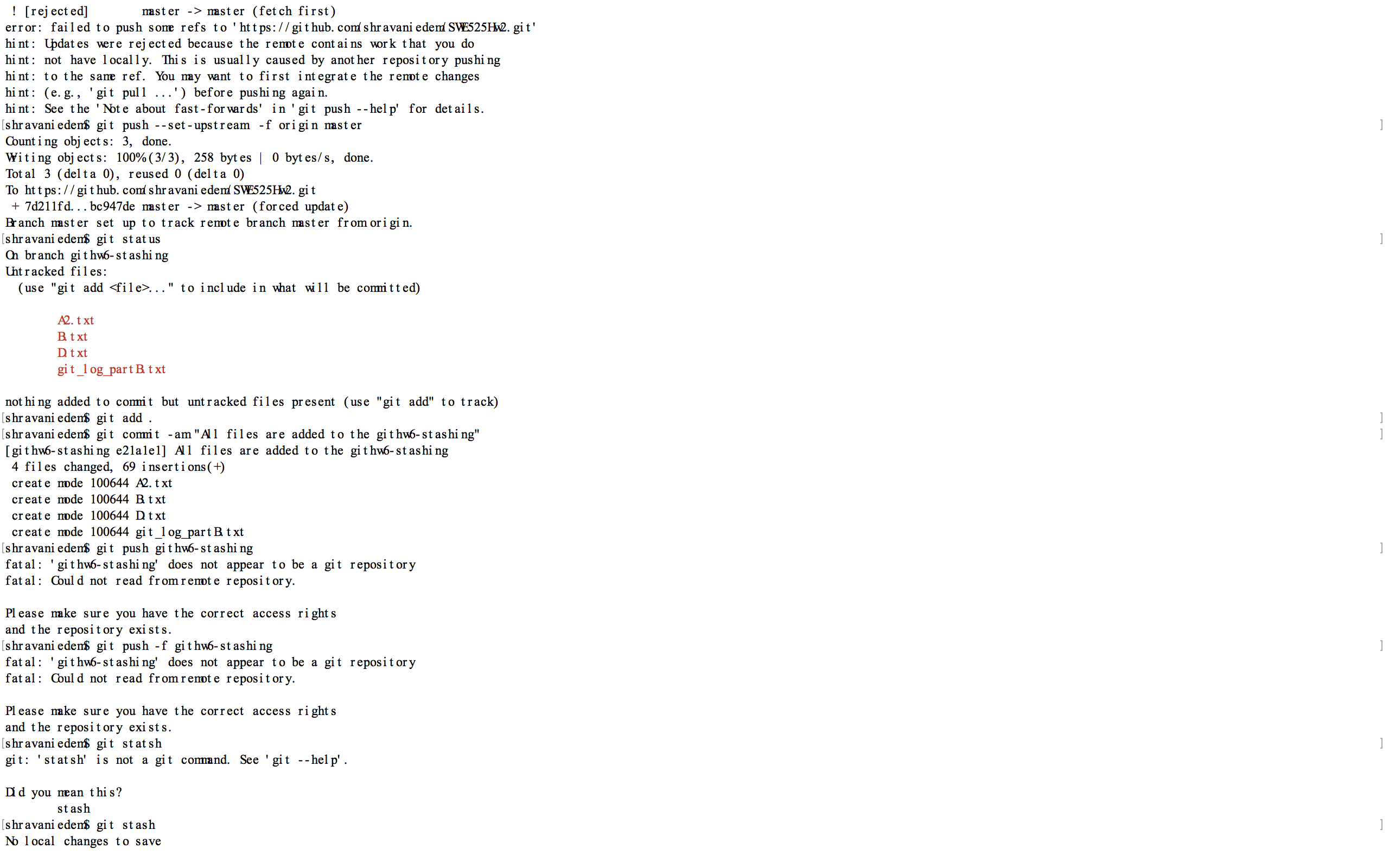
* Modified B.txt file on githw6-stashing branch
* In addition, without committing changes checkout branch
* Error “Please, commit your changes or stash them before you can switch branches”. Error because without committing changes we cannot switch branches.
* Firstly, stashing is necessary if we do some changes on a branch and without committing want switch the branch.
* Put the changes in stash.
* After running git stash command switched the branch.











Checkout the githw6-stashing branch. Modify B.txt, but do not add it to the staging area. Now try to git checkout githw2. What happens? Why? Explain how you could use stashing to solve this problem if all you wanted to do was to checkout githw2, examine B.txt, then return to githw2-stashing and restore the initial state. Then do so.

Push the githw2-stashing branch. Then merge githw2-stashing into githw2, and push githw2.

**Part E.** Write the commands that generated the following git log. You may assume no conflicts occurred during reverts. You may abbreviate making a commit to git commit -m "A". Additionally, what will be left of the repository at HEAD?

**Sol:**

File A, B,C and D created in git repo

$ echo "File A">A.txt

$ echo "File B">B.txt

$ echo "File C">C.txt

$echo "File D">D.txt

$ git add .

$ git commit "Four files added"

$ echo "New line">>A.txt

$ git add A.txt

$ git commit -m "A"

$ echo "New line">>B.txt

$ git add B.txt

$ git commit -m "B"

$ echo " New line ">>C.txt

$ git add C.txt

$ git commit -m "C"

$ git revert 38ea439 (Hash tag For committing C.txt file)

$ git revert f151194f (Hash tag For committing B.txt file)

$ echo " New line ">>D.txt

$ git add D.txt

$ git commit -m "D"

$ git revert 06c1266 (Hash tag of "Revert "Revert "C""")

$ git revert d01f866 (Hash tag For committing A.txt file)

$ git revert d83df79 (Hash tag For committing D.txt file)

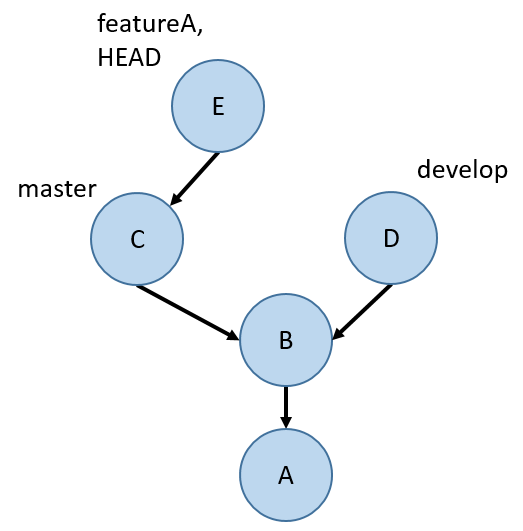
$git lg (search hash tag for "Revert "Revert "C""")

$ git revert a739fa8 (Hash tag of "Revert "Revert "C""")

If you find the following confusing please solve option 2:

**Part E:** Why don't you think git deletes branches you rebase from? (To clarify, I mean if you do git rebase develop while on featureA, git won't delete the commits where featureA used to be.)

To guide your thinking, you may want to consider the following git graph:



**Sol:**

* Transfer data from one branch to another using git rebase without deleting branch.
* Rebase command keeps branch as it is and merge other branch on the top of current branch.
* Git rebase command moves the branch HEAD.

**Ex:**

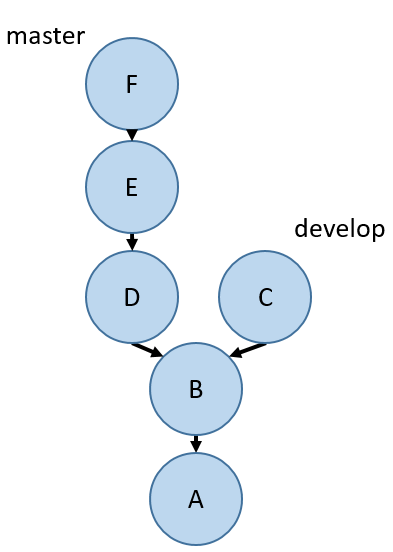
There are three branches featureA, featureB and featureC.

If we are on FeatureA branch and give git rebase develop then it merges the featureC branch with featureA branch. It changes the pointer of featureA branch.

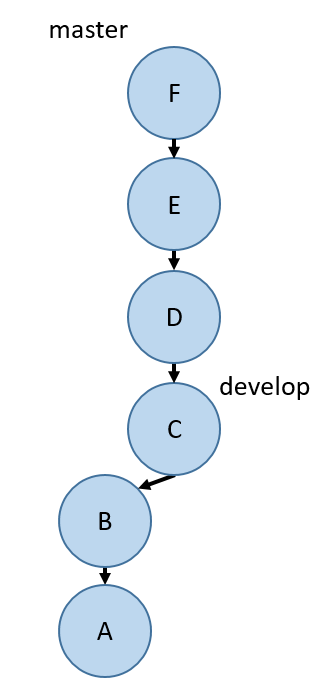
Git will not delete commit history which is on featureA branch and it will not delete the data from develop branch.

We need to delete branch using git branch -d <branch name>

If you do a git fetch and checkout the githw7-rebase-master branch, you'll notice some new commits. (You should also checkout the githw7-rebase-develop branch so that you can access it later.) The new commits resemble the following git graph (replace master with githw7-rebase-master and develop with githw7-rebase-develop):

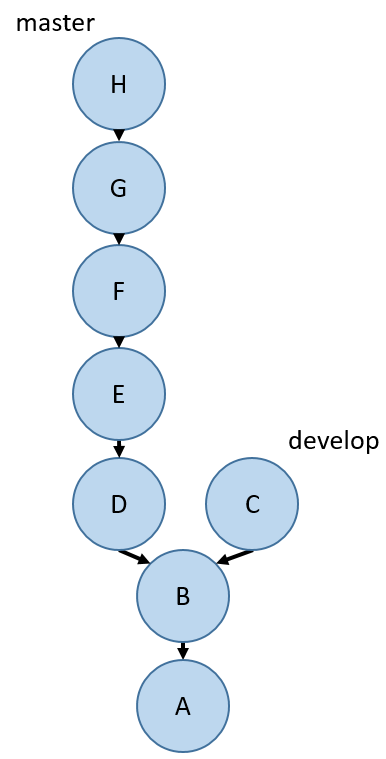


Do one rebase to make it look like the following graph (inaccessible commits not shown) (replace master with githw7-rebase-master and develop with githw7-rebase-develop):



Afterwards, push githw7-rebase-master. You will need to do git push -f githw7-rebase-master in order to push (just this once! don't do this in general!).  You may run into a conflict. Fix the conflict however you like - but it still needs to match the git graph above. Git should walk you through how to deal with this conflict (though of course git status is a good place to start to figure out how to resolve the conflict).

While fetching for the last question, you also got the githw7-rebase-i-master branch. (Make sure to checkout both this branch and githw7-rebase-i-develop so that you can access them locally!) The new commits on this branch should resemble the following git graph (replace master with githw7-rebase-i-master and develop with githw7-rebase-i-develop):



Do one interactive rebase to make it look like the following graph (inaccessible commits not shown; E,G means a single commit containing the changes from both E and G) (replace master with githw7-rebase-i-master and develop with githw7-rebase-i-develop):





Afterwards, push githw7-rebase-i-master. You will need to do git push -f githw7-rebase-i-master in order to push (just this once! don't do this in general!).

Do you prefer merging or rebasing? Why? There isn't a correct answer for this question.

Merging and rebasing both are different concepts. I will prefer anyone of them depending on requirement. Git merge command merges one branch with other branch. After merging branch will create non linear history. If I want linear history then I will prefer rebasing. Git rebase transfer data from one branch to another. Rebase command is used for transferring data not for deleting branch. Rebase command keeps branch as it is and merge other branch on the top of current branch. Git rebase moves branch HEAD.

**Sol:**

<https://github.com/zameerahmed123/GIT_Assignment_2.git>

Pushed the screenshots to repo.