**SWE 525 Version Control Git Homework 2 12/02/2016**

**Name: Vignesh Ravichandran Student ID: 90452**

**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.**

1. **How do we go about performing the following techniques in Git**
2. ***Finding which commit caused a particular bug***

**git bisect**

1. ***Listing commits with different formatting***

**You can use ‘git log’ with different flags like --oneline, --decorate, --graph, --pretty to achieve different formatting**

1. ***Listing assumed-unchanged files***

**git ls-files –v | grep ‘^[[:lower:]]’**

1. ***Deleting the current local branch after merging***

**git branch –d branchName**

1. ***Rewriting the entire history of a branch***

**There are various commands that you can use such as git rebase, git rebase –i, git commit --amed, git reflog that can be used to rewrite history of a branch. It depends on the scenario.**

1. ***Deleting a remote branch***

**git push origin --delete branchName**

1. **After a lengthy session you’ve made a bunch of commits. But you’re not quite happy with the way they’re organized, and some of those commit messages could use rewording.**

**Interactive rebase “git rebase –i” and ‘git commit --amend” can be used to achieve the desired result**

1. **We intend to release a project, but it involves a file that should be kept private for some reason. Perhaps I left my credit card number in a text file and accidentally added it to the project. Deleting the file is insufficient, for the file can be accessed from older commits. We must remove the file from all commits.**

**You could try the following command:**

**git filter-branch --tree-filter ‘rm <file>’ HEAD**

1. **You’ve just discovered a broken feature in your program which you know for sure was working a few months ago. Argh! Where did this bug come from? If only you had been testing the feature as you developed. It’s too late for that now. However, provided you’ve been committing often, Git can pinpoint the problem. How?**

**git bisect (selects a commit somewhere in the middle of the history and checks it out)**

**Compile and test out the checkout version if it fails then mark it as bad using ‘git bisect bad’ else if good mark it as ‘git bisect good’. Once there is atleast one good and one bad commit then git will try to narrow down on the commits between the good and bad commits following the same procedure. Once the root bad commit has been found you can fix it and then do a ‘git bisect reset’ to reset to original HEAD.**

1. **The HEAD tag is like a cursor that normally points at the latest commit, advancing with each new commit. Some Git commands let you move it. Which command lets you move the Head**

**git reset, git revert, git checkout**

1. **You have committed both first.txt and second.txt as one commit. However, you made a mistake. You intended to commit first.txt in the first commit and the second.txt in the second one.**

**Execute the following command**

**git rebase –i <commit>^**

**Now mark the commit with the action edit**

**To edit the commit use git reset HEAD^ and then add the changes that you want to the first commit, in this scenario it is first.txt. then commit the index.**

**Repeat the last step for the second commit. Then do a git rebase --continue**

1. **While you were working you noticed a typographic error in file.txt - you wrote wordl instead of world. Unfortunately, you have made another commit on top of the typo so simple git commit --amend is not enough.Fix the typographic error by amending commit in history.**

**Perform the following command**

**git rebase –i ‘<commitId>^’**

**Once done, do the necessary changes to the file and perform the below command to commit with the same message you had for the commit**

**git commit --all --amend --no-edit**

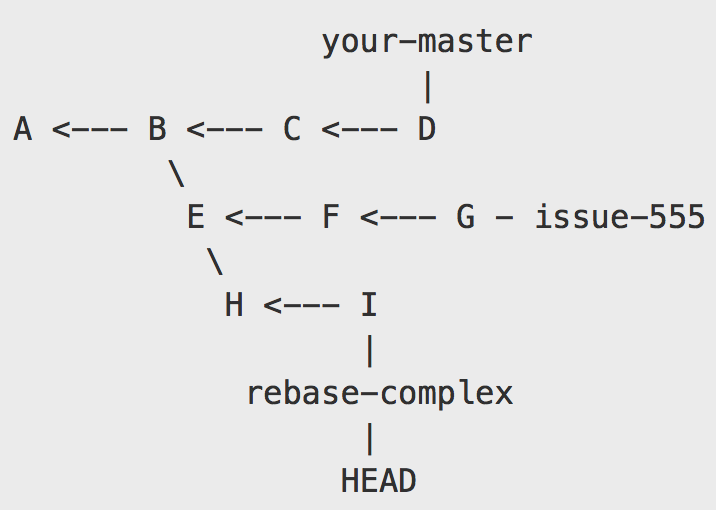
**Now perform the following command to go back to the previous head commit**

**git rebase --continue**

1. **You just did a git reset --hard HEAD^ and threw out your last commit. Well, it turns out you really did need those changes. You’ll never be able to implement that algorithm that perfectly twice, so you need it back. How will you do that?**

**git merge <SHA1 of the lost commit> should help get back the lost changes**

1. **You were working on an issue-555 topic branch. You noticed a bug, which you fixed in rebase-complex topic branch. Then, you finished issue-555. However, you need to have bug fixed in your-master branch without any work done on issue-555. Situation is as follows:**

****

Only H and I commits should be rebased onto D.

Try to do this with a single git command.

**git rebase --onto your-master issue-555 rebase-complex**

1. **Explain what the following commands do?**
2. **git checkout -f COMMIT**

**Force checkout a commit**

1. **git clean -f -d**

**Removes untracked files and directories, also the subdirectories with .git folder**

1. **git checkout master~5**

**checkout master and revert to 5 commits back**

1. **git daemon --detach**

**detaches the git daemon from shell**

1. **git rebase -i HEAD~10**

**picks the commit that is behind HEAD by 10 commits**

1. **git bisect reset**

**cleans up the bisection state and returns to the original HEAD**

|  |
| --- |
| 1. **git fsck | awk '{print $3}' | xargs -I '{}' git show {} >> blob\_contents**   **First prints out the connectivity and validity of the databse.**  **prints out the third column from the git fsck output. Then git show {} gets applied to each**  **column and the output is saved in file named blob\_contents**  **h, git fetch origin && git remote show origin | grep tracked | grep -v master**  **| awk '{print $1}' | xargs -I '{}' bash -c 'if [ ! "$(git cherry origin/master origin/{})" ];**  **then git push origin :{}; fi**  **Fetches remote origin branch, then grep all the tracked files and then grep the tracked files from master.**  **Print the first column from the master and if in that printed output if there is no**  **“git cherry origin/master origin” then perform a “git push origin”.** |

1. **git push --force origin feature**

**Forces pushes the current branch to the remote branch “origin/feature”**

1. **Please read the following case study and explain your understanding.**

**Cherry-Picking specific commits from another branch**

I’m often asked how to merge only specific commits from another branch into the current one. The reason you’d want to do this is to merge specific changes you need now, leaving other code changes you’re not interested in right now behind.

First of all, use git log or the awesome [GitX](http://gitx.frim.nl/) tool to see exactly which commit you want to pick. An example:

dd2e86 - 946992 - 9143a9 - a6fd86 - 5a6057 [master]

\

76cada - 62ecb3 - b886a0 [feature]

Let’s say you’ve written some code in commit 62ecb3 of the feature branch that is very important right now. It may contain a bug fix or code that other people need to have access to now. Whatever the reason, you want to have commit 62ecb3 in the master branch right now, but not the other code you’ve written in the feature branch. ~ Here comes git cherry-pick. In this case, 62ecb3 is the cherry and you want to pick it!

git checkout master

git cherry-pick 62ecb3

That’s all. 62ecb3 is now applied to the master branch and commited (as a new commit) in master. cherry-pick behaves just like merge. If git can’t apply the changes (e.g. you get merge conflicts), git leaves you to resolve the conflicts manually and make the commit yourself.

Cherry picking a range of commits

In some cases picking one single commit is not enough. You need, let’s say three consecutive commits. cherry-pick is not the right tool for this. rebase is. From the previous example, you’d want commit 76cada and 62ecb3 in master.

The flow is to first create a new branch from feature at the last commit you want, in this case 62ecb3.

git checkout -b newbranch 62ecb3

Next up, you rebase the newbranch commit --onto master. The 76cada^ indicates that you want to start from that specific commit.

git rebase --onto master 76cada^

The result is that commits 76cada through 62ecb3 are applied to master.

Problem:

Merging specific commits from another branch to current branch

Subject Matter:

There are times when you would want to merge specific changes (commits) from another branch to your current branch. For example, you might have a big fix in a different branch that you would want to merge to your current branch right now.

Solution:

The solution would be to use ‘git cherry-pick’ if you want to apply just one commit from a different branch to the current branch or if you would like to apply a range of commits to the current branch, rebase could be used.

How it is implemented:

As given in the case study, if you would like to apply the commit ‘62ecb3’ from feature branch to the master branch then the following commands would help achieve it

git branch master

git cherry-pick 62ecb3

If you would like to apply a sequence of commits, for example, from 76cada to 62ecb3 to master then the following commands would help achieve it

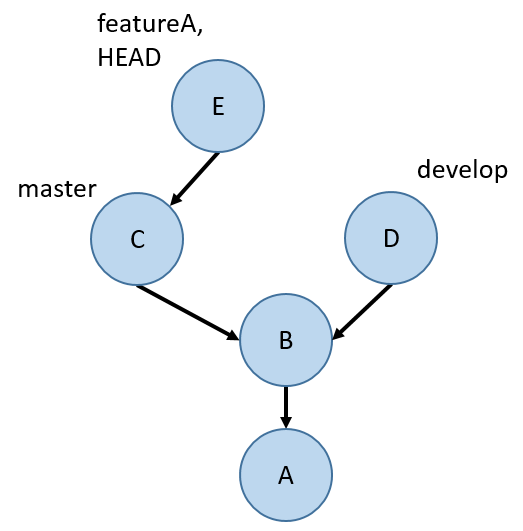
git checkout -b newbranch 62ecb3

git rebase --onto master 76cada^

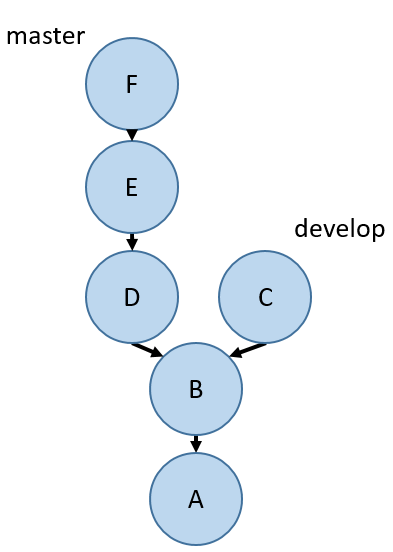
1. Why don't you think git deletes branches you rebase from? (To clarify, I mean if you do git rebase develop while on featureA, git won't delete the commits where featureA used to be.)

git rebase does not delete the branch because it copies all the commits in the branch you rebase from to the branch you are rebasing to.

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

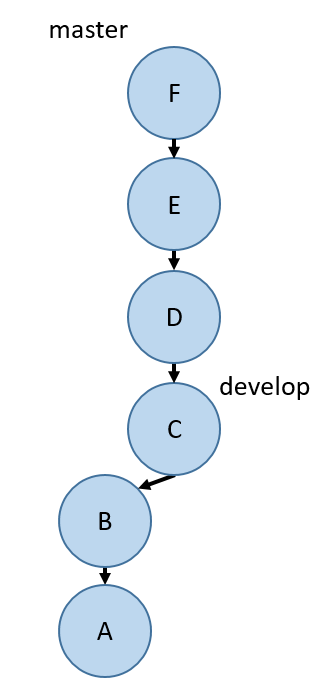


If you do a git fetch and checkout the githw7-rebase-master branch, you'll notice some new commits. (You should also checkout the githw7-rebase-develop branch so that you can access it later.) The new commits resemble the following git graph (replace master with githw7-rebase-master and develop with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



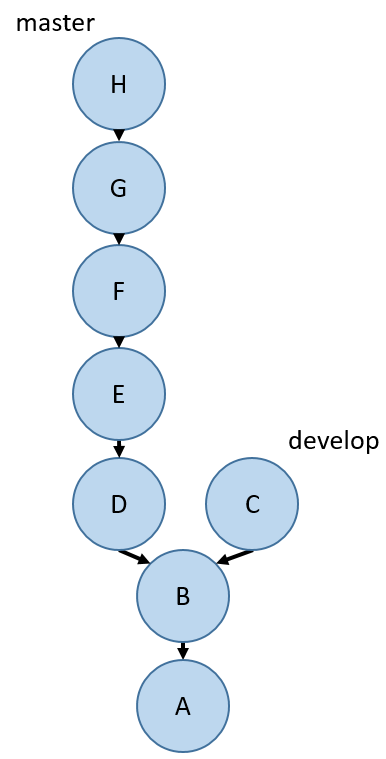
Do one rebase to make it look like the following graph (inaccessible commits not shown) (replace master with githw7-rebase-master and develop with githw7-rebase-develop):

git rebase develop

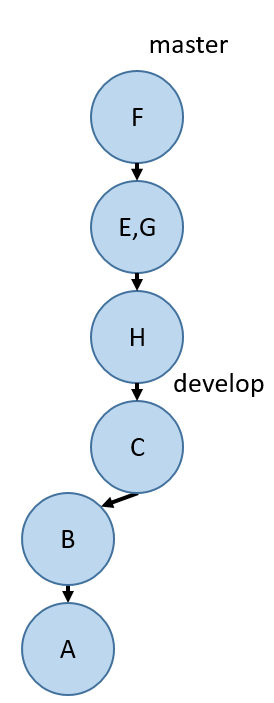


Afterwards, push githw7-rebase-master. You will need to do git push -f githw7-rebase-master in order to push (just this once! don't do this in general!).  You may run into a conflict. Fix the conflict however you like - but it still needs to match the git graph above. Git should walk you through how 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!).

* git rebase -i HEAD~5
* Delete commit D in the file shown by rebase
* Reorder the commits in the order H<-E<-G<-F
* Replace pick with squash for G
* Save the file and then do:

git rebase --continue

git branch develop

git rebase develop

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

Merging:

* Merging creates a merge commit that ties the history of two branches involved in the merge
* It is a non destructive operation
* It pollutes the branch history by creating merge commits
* History of the project becomes difficult to read due to these merge commits

Rebasing:

* Rebasing re writes history by creating brand new commits for each commit in the original branch
* It creates a linear history
* The history of the project is cleaner
* Re writing can be catastrophic
* Rebasing loses the context provided by merge commit

I think I would still choose rebasing since it creates a linear history and the potential pitfalls of using rebase could be avoided by having educated users.