[CS200]-Software Tools and Technologies II **Fall 2020-21**

Homework: II

Team Name: Peaky Blinders Roll Numbers:11940150, 11940370, 11940380

Solution of problem 1. The answers to all parts are given below with relevant examples if needed:

a Generally in a git repository if you want to work in multiple branches you need to every time checkout that particular branch to make changes to it. Thus at any time you have only one active working directory.

But git allows you to have multiple working branches in your repo at a single time so you don't need to checkout every time. This is done using the **git worktree** command. It allows you to have multiple worktrees in the same repository. It has several commands like add, remove, move, etc that allow you to add, delete and move worktrees around in the repository. At any time you have a main worktree in which you work and other linked worktrees if any. Changes committed in a worktree are local to itself and all this is done without a single checkout.

Given below is a small example demonstrating the same. First create a workspace directory and open a bash session for it. Then type the following commands in it:

```
git init
echo "hi" >> f1 # create dummy file to commit and create master branch
git add
git commit -m "f1 added"
git branch fix  # create new branch named fix
git branch new-feature  # create new branch named new-feature
git worktree add fix1 fix   # new worktree named fix1 under branch fix
git worktree add feature1 new-feature # similar to above.
```

Now open another bash session in one the newly added worktrees say feature1 and type the following commands:

```
echo "Hello" >> f2  # new file in worktree feature1
git add .
git commit -m "hello added" # commit file f2 to current worktree only
```

Now go back to the workspace folder bash session and type the *git graph* command to get the git graph.

The git graph is shown in Figure 2.1.

From the graph it is visible that the our addition of f2 was only done to the new-feature branch under the feature1 folder and while doing this we didn't do a single checkout. We actually worked in two directories simultaneously.

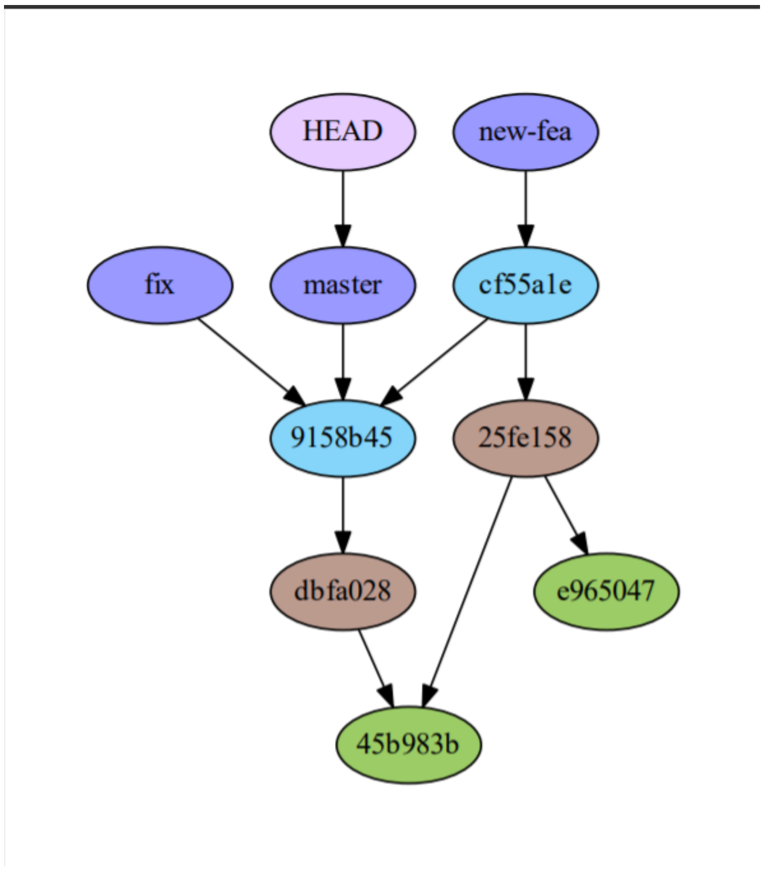


Figure 2.1: Git Graph for the example in part a

b The script below depicts the commands that according to me should lead to a directed edge from stage to working directory. I am assuming that ”directed edge from stage to working directory” means copying files from stage to working directory. First create a workspace folder and initialize git inside it.

```
echo "hello" >> f1
git add f1
cat f1 #Displays hello
git commit -m "c1"
echo "bye" >> f1
cat f1 #Displays hello\nbye (\n stands for newline)
git add f1
echo "bye" >> f1
cat f1 #Displays hello\nbye\nbye (\n stands for newline)
git checkout -- f1
cat f1 #Displays hello\nbye (\n stands for newline)
```

As you can see the

```
git checkout -- f1
```

copies the last staged version of f1 and overwrites the f1 in working directory with this version. Same can be achieved with

```
git restore f1
```

. Also we can use the fact that when we add a file to the stage then even if we have not committed it yet a blob gets created for its contents(if they are unique). Now keeping this in mind we can use the commands:

```
git cat-file -p <hash value of blob of staged version> > f1
or
git show <hash value of blob of staged version> > f1
```

This will overwrite the content of f1 in working directory with content of the staged version’s blob. Thus these four commands should serve the purpose according to me.

c To commit a file in parts one can use the commmand **git add –patch <filename>**. After this git divides the file with changes into hunks. You can decide changes in which hunks to commit and which not to commit. You can also divide a hunk into smaller hunks to pinpoint to a change. Also you can use **git add –interactive** to do the same. A small example is given below: Create a folder then add a file in it and put some content in it. Now open a bash session here and do the following commands:

```
git init
git add .
git commit -m "sample 1 added"
```

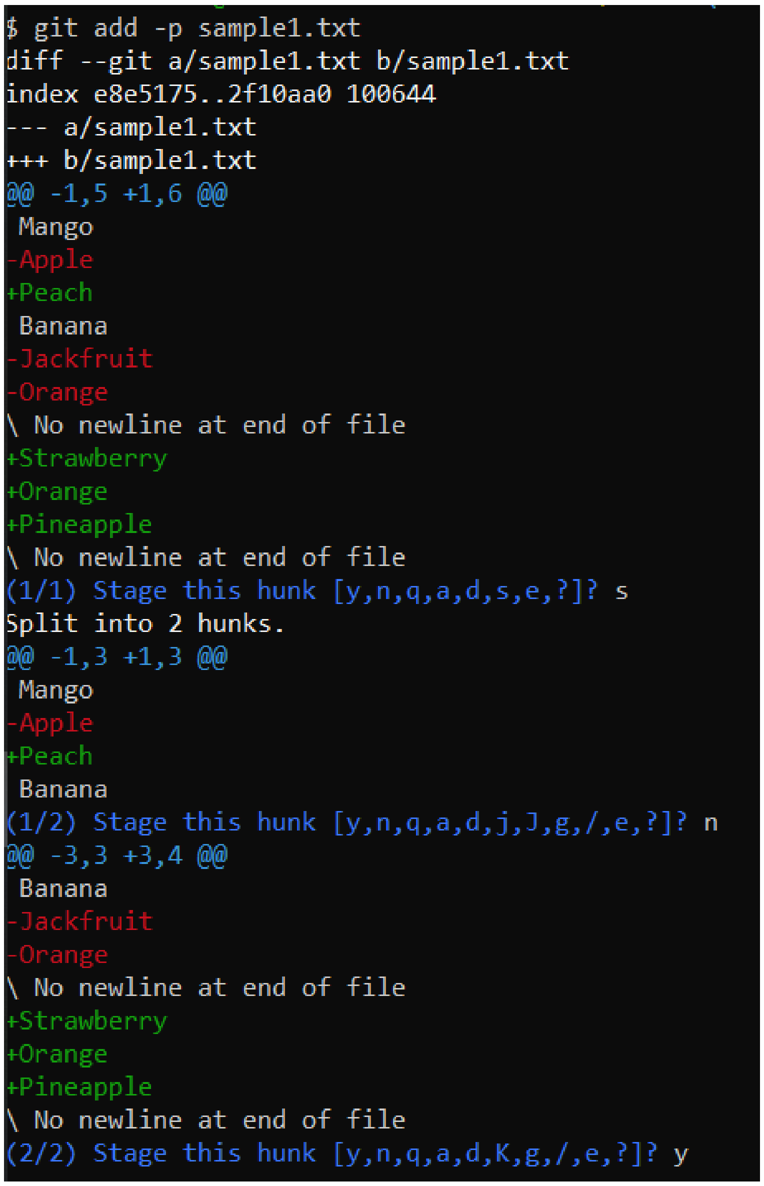
Now open the file and make a few changes in it and now we only a commit a few changes made in it. Do the following:

```
     git add --patch sample1.txt
```

Now git automatically divides the file into hunks and shows the content of the first hunk. You have many actions possible of which some are : to commit changes in that hunk, not commit changes in that hunk, divide the hunk into smaller hunks, skip the current hunk, etc. Like this it gives you the same choices for all the hunks. Once you are done with all the hunks commit the changes you selected. The above process done for the example is in Figure 2.2: Example result given below:

```
    Intial:
            Mango
            Apple
            Banana
            Jackfruit
            Orange
        Changes made:
            Mango
            Peach
            Banana
            Strawberry
            Orange
            Pineapple
        After commiting only selected changes commited:
            Mango
            Apple
            Banana
            Strawberry
            Orange
            Pineapple
```

Thus you can see the change from *Apple* to *Peach* was not committed to the repo while other changes were committed.



```
$ git add -p sample1.txt
diff --git a/sample1.txt b/sample1.txt
index e8e5175..2f10aa0 100644
--- a/sample1.txt
+++ b/sample1.txt
@@ -1,5 +1,6 @@
 Mango
-Apple
+Peach
 Banana
-Jackfruit
-Orange
\ No newline at end of file
+Strawberry
+Orange
+Pineapple
\ No newline at end of file
(1/1) Stage this hunk [y,n,q,a,d,s,e,?]? s
Split into 2 hunks.
@@ -1,3 +1,3 @@
 Mango
-Apple
+Peach
 Banana
(1/2) Stage this hunk [y,n,q,a,d,j,J,g,/,e,?]? n
@@ -3,3 +3,4 @@
 Banana
-Jackfruit
-Orange
\ No newline at end of file
+Strawberry
+Orange
+Pineapple
\ No newline at end of file
(2/2) Stage this hunk [y,n,q,a,d,K,g,/,e,?]? y
```

Figure 2.2: Commit a file in parts example

d Two commits can be combined into a single commit using the **git rebase -i HEAD∼N** where N is number of commits you want to combine. An interactive window opens in the command prompt itself. Here you can decide which commits you want to combine together and also the commit into which everything will be combined. The example is given below:

```
mkdir test3
cd test3
git init
echo "commit1" >> f1
git add .
git commit -m "commit 1"
echo "commit2" >> f1
git add f1
git commit -m "commit 2"
echo "commit3" >> f1
git add f1
git commit -m "commit 3"
git graph
git rebase -i HEAD˜2
git graph
```

In the interactive window during rebase type the following:

```
pick 3932cf1 commit 2
    s f6917c5 commit 3
```

Here commit 3 is squashed into commit 2. Now save the changes and provide a message and it will display rebase successful if everything is done properly.

In every commit we are making changes to a single file(Here adding new content everytime). Now using rebase we are combining the changes made in commit 2 and 3 into a single commit and this new single commit will be put after the first commit in the graph(git flow). The before rebase and after rebase graphs are shown in Figure 2.3. As can be seen from the figure after rebase our original commits remain untouched, only a new commit having the combined changes of the two original commits(2 and 3) is added after the first commit and master now points to this new commit.

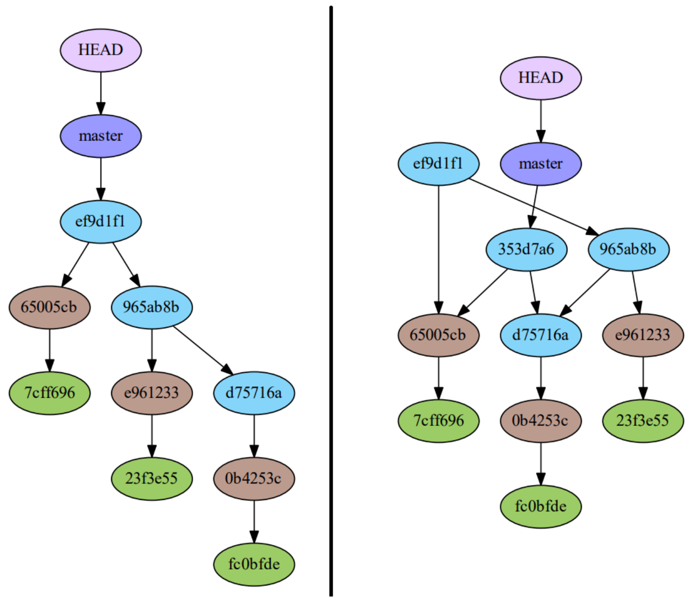


Figure 2.3: Left: Before rebase, Right: After rebase