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

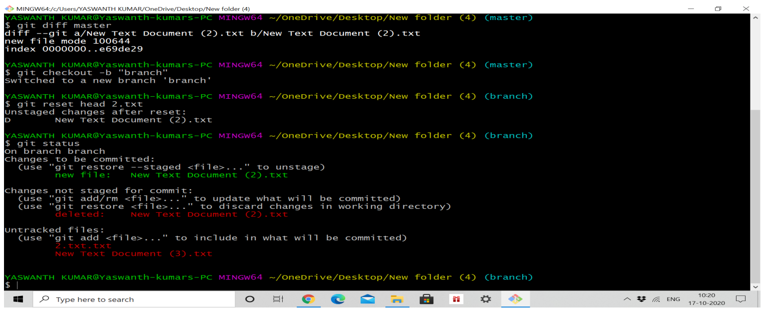
Homework: 02

Group Name:**Virtual-Reality**

Name-id no:-email-id
Konatham Yaswanthkumar - 11940580- konathamy@iitbhilai.ac.in
Vempala Suryateja- 11941330 - vempalas@iitbhilai.ac.in
Bommireddy Shiva sai reddy-11940280-bommir@iiitbhilai.ac.in

Solution of problem 1. (a)
A git repository can support multiple working trees, allowing you to check out more than one branch at a time. With git worktree add, a new working tree is associated with the repository.This new working tree is called a ”linked working tree” as opposed to the ”main working tree” prepared by ”git init” or ”git clone”. A repository has one main working tree (if it’s not a bare repository) and zero or more linked working trees.The mechanism which git uses to have multiple working directories is by us ing Git work-tree which manages multiple working trees attached to the same repository.If a working tree is deleted without using git worktree remove, then its associated administrative files, which reside in the repository, will eventually be removed automatically we can run git worktree prune in the main or any linked working tree to clean up any stale administrative files.

(b)
1)git status
2)git diff
3)git checkout
4)git reset head file name
1.git status : The git status command displays the state of the working directory and the staging area.
2.git diff : git diff is a multi-use Git command that when executed runs a diff function on Git data sources. These data sources can be commits, branches, files
3.git checkout : The git checkout command lets you navigate between the branches created by git branch
4.git reset head file name : which converts file in staging area to working directory



```
YASWANTH KUMAR@Yaswanth-kumars-PC MINGW64 ~/OneDrive/Desktop/New folder (4) (master)
$ git diff master
diff --git a/New Text Document (2).txt b/New Text Document (2).txt
new file mode 100644
index 0000000..e69de29

YASWANTH KUMAR@Yaswanth-kumars-PC MINGW64 ~/OneDrive/Desktop/New folder (4) (master)
$ git checkout -b "branch"
Switched to a new branch 'branch'

YASWANTH KUMAR@Yaswanth-kumars-PC MINGW64 ~/OneDrive/Desktop/New folder (4) (branch)
$ git reset head 2.txt
Unstaged changes after reset:
D       New Text Document (2).txt

YASWANTH KUMAR@Yaswanth-kumars-PC MINGW64 ~/OneDrive/Desktop/New folder (4) (branch)
$ git status
On branch branch
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   New Text Document (2).txt

Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        deleted:    New Text Document (2).txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        2.txt.txt
        New Text Document (3).txt


YASWANTH KUMAR@Yaswanth-kumars-PC MINGW64 ~/OneDrive/Desktop/New folder (4) (branch)
$ |
```

MINGW64:/c/Users/YASWANTH KUMAR/OneDrive/Desktop/New folder (4)
Type here to search
ENG
10:20
17-10-2020

Figure 1.1: work flow for above command

(c)
git add -p:
In order to commit the changes, we have to first stage the changes which is done using the git-add command. And if you want to commit only parts of a file, you can use the interactive mode, which is turned on by the -p option. It continuously shows small portions of the changed files and asks you what to do. In each step, you can mark hunks, which is a nearby set of changes, for staging or to be ignored for now.
Most of the commands in this mode are self-explanatory. When lost, you only have to type the ? symbol, which lists the commands and explains what they do. Let's focus on the following two commands:
s - split the current hunk into smaller hunks
e - manually edit the current hunk
Git Cola command:
it has ability to stage separate lines
When you select at least one symbol (or multiple lines) it stages the lines involved and if you don't select anything, it stages the hunk you have the cursor on.

(d)
The easiest way to turn multiple commits in a feature branch into a single commit is to reset the feature branch changes in the master and commit everything again.
Note that this is not touching the feature branch at all. If you would merge the feature branch into the master again at a later stage all of its commits would reappear in the log.

You may also do it the other way round (merging master into the branch and resetting to the master state) but this will destroy your commits in the feature branch, meaning you can not push it to origin.
This method is harder than using the get reset method above. Also it doesn't work well if you merged the master into the feature branch previously (you'll need to resolve all conflicts again).
What we are describing here will destroy commit history and can go wrong. For this reason, do the squashing on a separate branch:
This means, you take the first commit, and squash the following onto it. If you remove a line, the corresponding commit is actually really lost. Don't bother changing the commit messages because they are ignored. After saving the squash settings, your editor will open once more to ask for a commit message for the squashed commit.

You can now merge your feature as a single commit into the master:
code for the following

```
git checkout -b squashed _ feature
git rebase -i master
pick fda59df commit 1
pick x536897 commit 2
pick c01a668 commit 3
pick fda59df commit 1
squash x536897 commit 2
squash c01a668 commit 3
git checkout master
git merge squashed _ feature
```

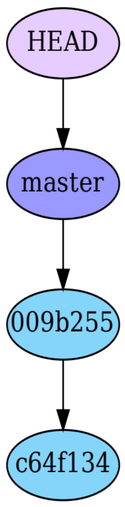


Figure 1.2: before commiting into a single commit

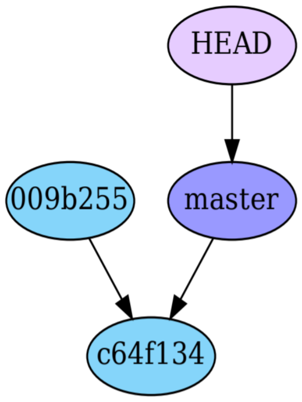


Figure 1.3: before commiting into a single commit