Subject Name: Source Code Management

Subject Code: CS181

Department: CSE

Cluster: Beta

Submitted By:

Ansh Wadhwa

2110990217

G8

Submitted To:

Mr. Monit Kapoor

Index

1. What is GIT and why is it used?

Git is a DevOps tool used for source code management. Git is software for tracking changes in any set of files. It is a free and open-source version control system used to handle small to very large projects efficiently.
Git is used to tracking changes in the source code, enabling multiple developers to work together on non-linear development
Git is an example of a DVCS (hence Distributed Version Control System).

2. What is GITHUB?

GitHub is a code hosting platform for version control and collaboration. It lets you and others work together on projects from anywhere.

3. What is Repository?

A repository contains all of your project's files and each file's revision history. You can discuss and manage your project's work within the repository. A Git repository is the .git/ folder inside a project. This repository tracks all changes made to files in your project, building a history over time. Meaning, if you delete the .git/ folder, then you delete your project's history.

4. What is Version Control System (VCS)?

Version Control Systems are the software tools for tracking/managing all the changes made to the source code during the project development. It keeps a record of every single change made to the code. It also allows us to turn back to the previous version of the code if any mistake is made in the current version. Without a VCS in place, it would not be possible to monitor the development of the project.

5. Types of VCS

- Local Visual Control System
- Centralized Version Control System
- Distributed Version Control System

Local Visual Control System

Local Version Control System is located in your local machine. If the local machine crashes, it would not be possible to retrieve the files, and all the information will be lost. If anything happens to a single version, all the versions made after that will be lost.

Centralized Version Control System

The Centralized Version Control Systems, there will be a single central server that contains all the files related to the project, and many collaborators checkout files from this single server (you will only have a working copy). The problem with the Centralized Version Control Systems is if the central server crashes, almost everything related to the project will be lost.

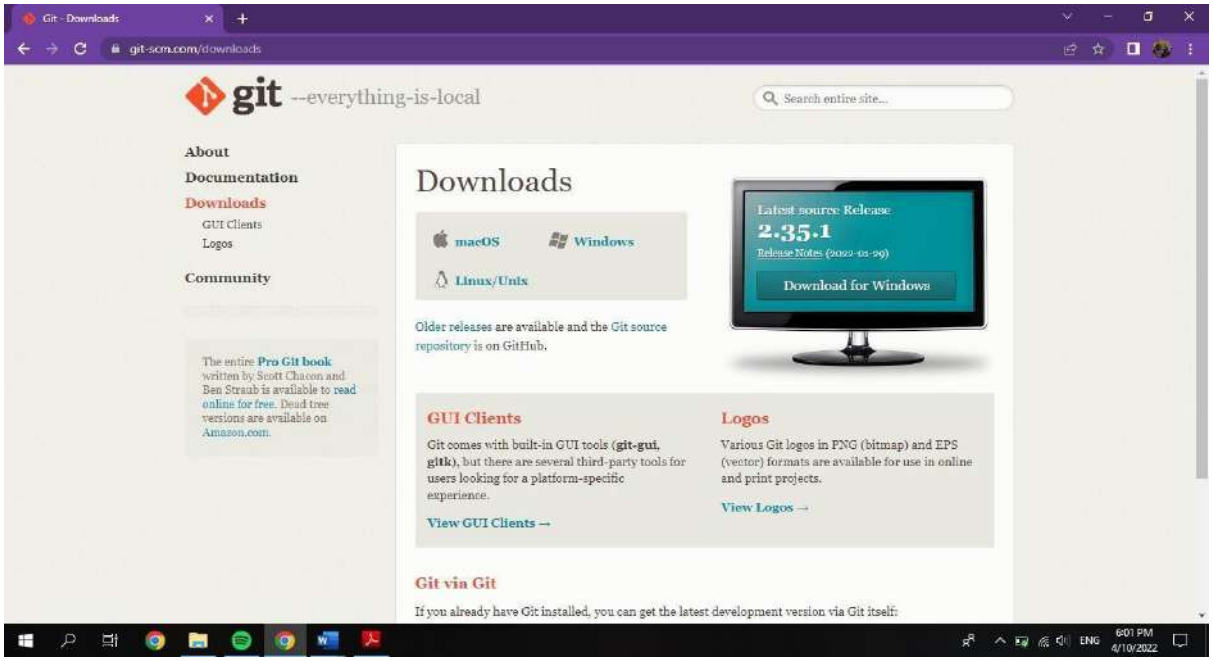
Distributed Version Control System

In a distributed version control system, there will be one or more servers and many collaborators similar to the centralized system. But the difference is, not only do they check out the latest version, but each collaborator will have an exact copy of the main repository on their local machines. Each user has their own repository and a working copy. This is very useful because even if the server crashes we would not lose everything as several copies are residing in several other computers.

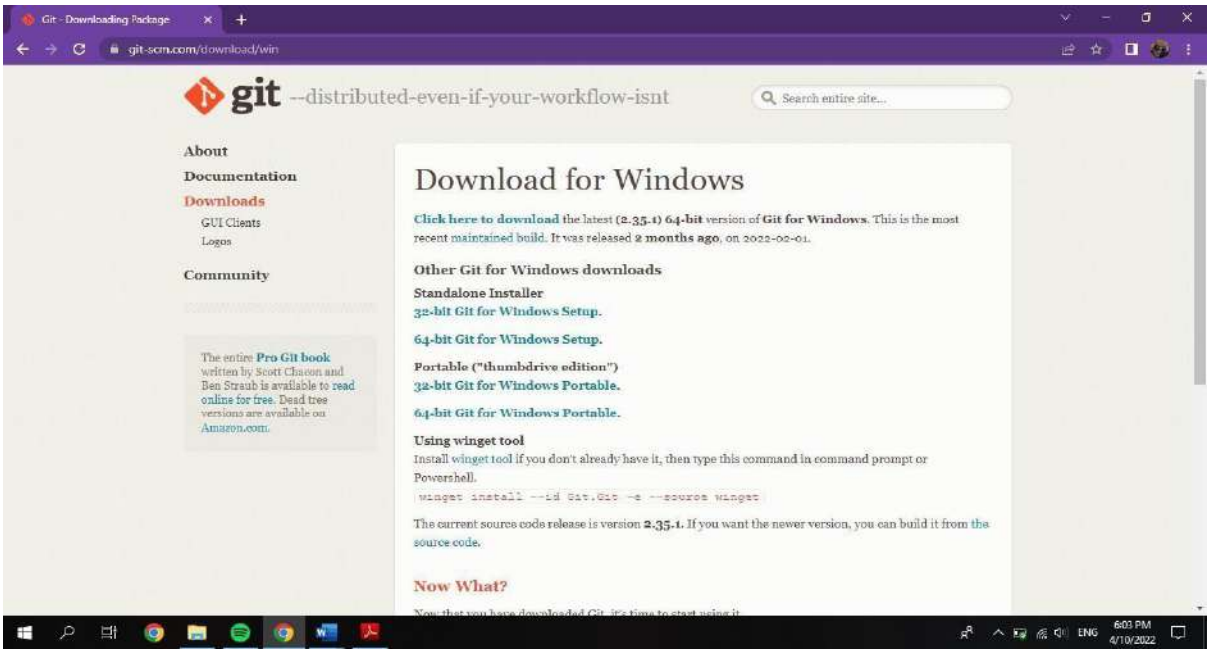
Experiment - 1

Aim: Setting up the git client.

Git Installation: Download the Git installation program (Windows, Mac, or Linux) from Git - Downloads (git-scm.com).



1. After opening this site, you have to select your operating system by clicking on it. Here I will show you the steps for the Windows operating system.

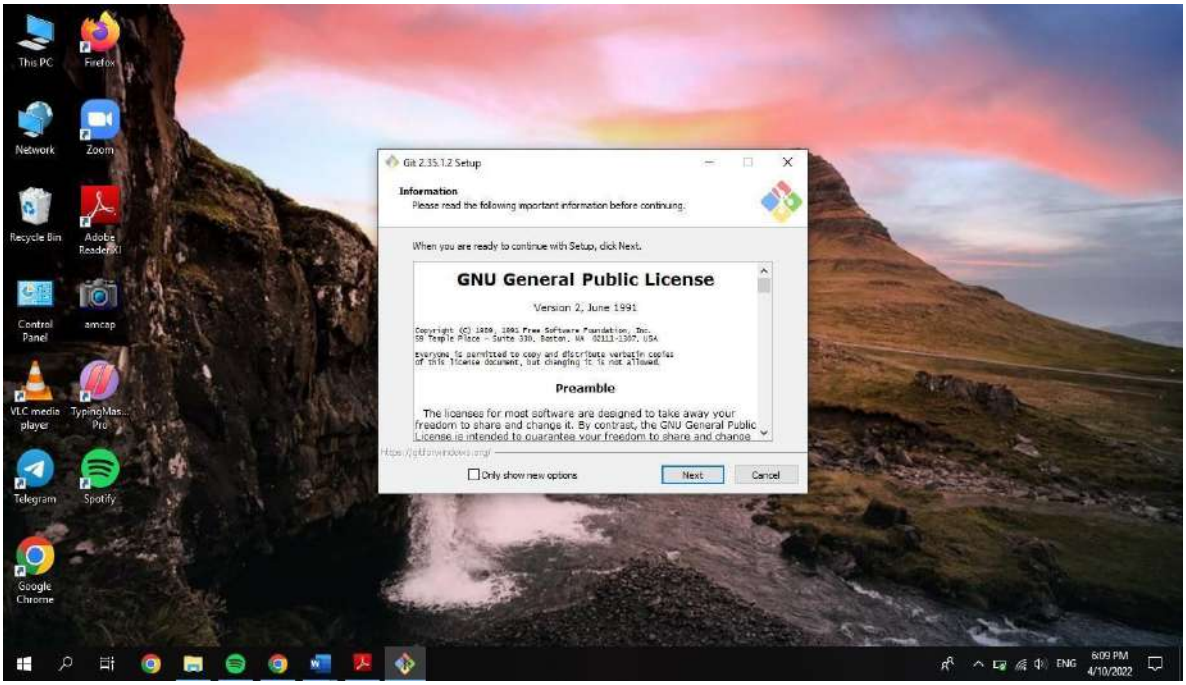


2. Now select the processor of the system you have. (Most of the system are now of 64-bits) After selecting the processor your download will start.

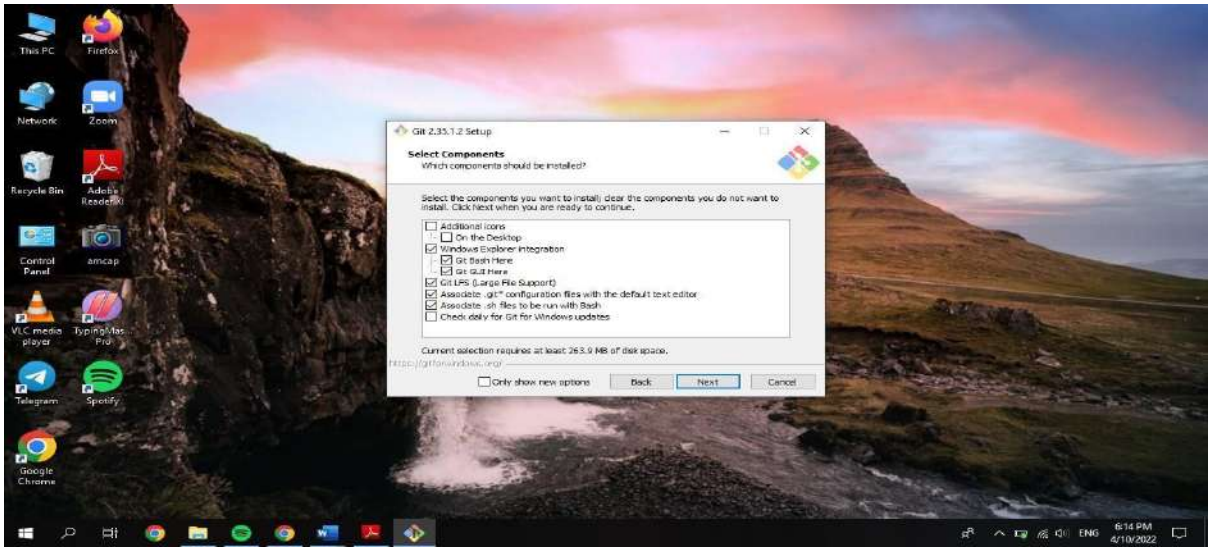
3. Now you have to open this folder.

4. After opening you will given a notification “Do you want to allow this app to make changes in your PC”

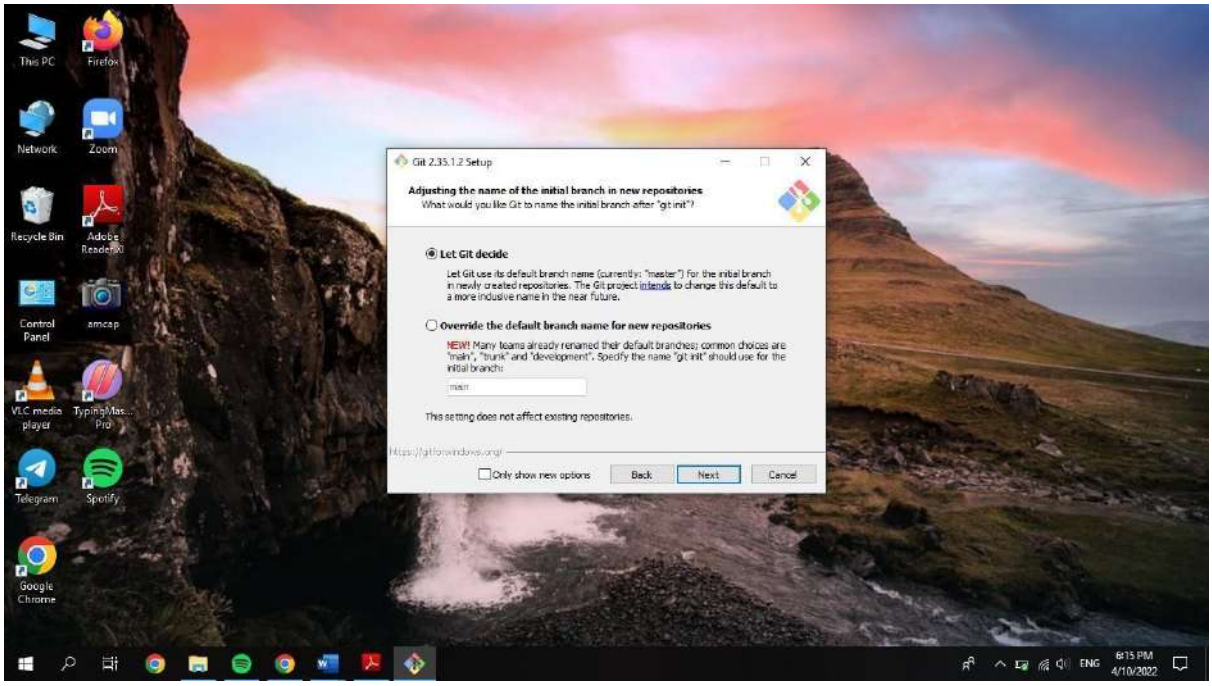
5. Click Next



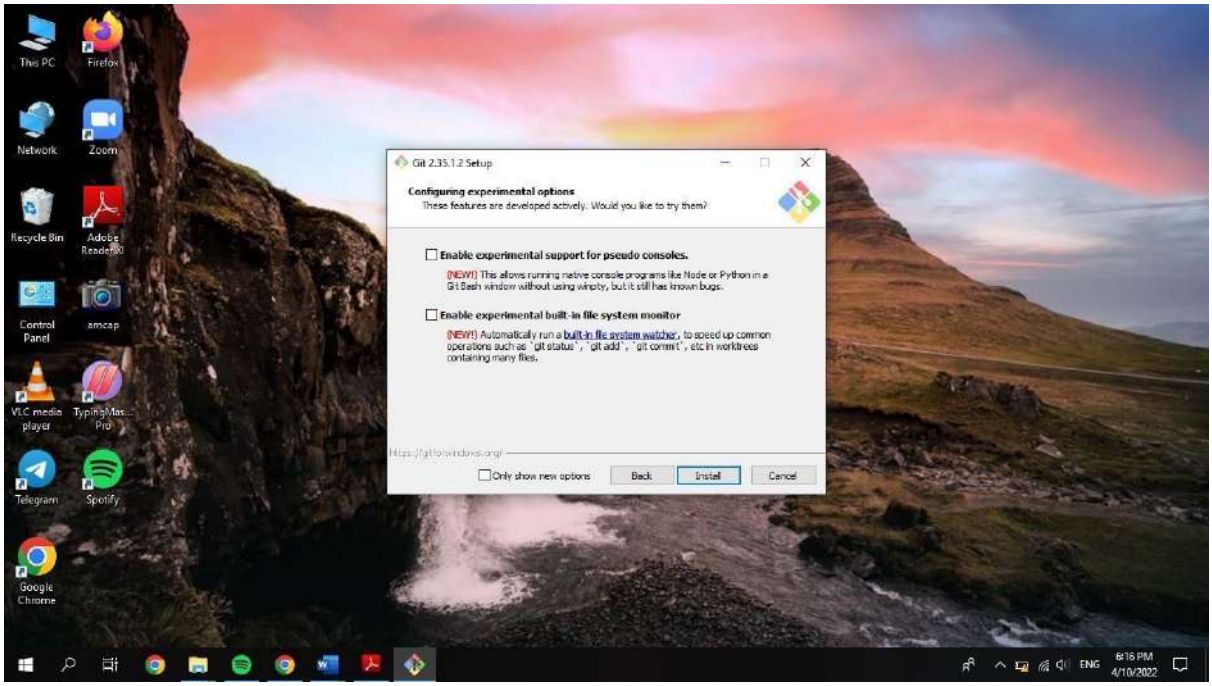
6. Click Next



7. Click Next



8. Click on Install



9. Click on Finish after the installation is finished.

10. The installation of the git is finished and now we have to setup git client and GitHub account.

\

Experiment - 2

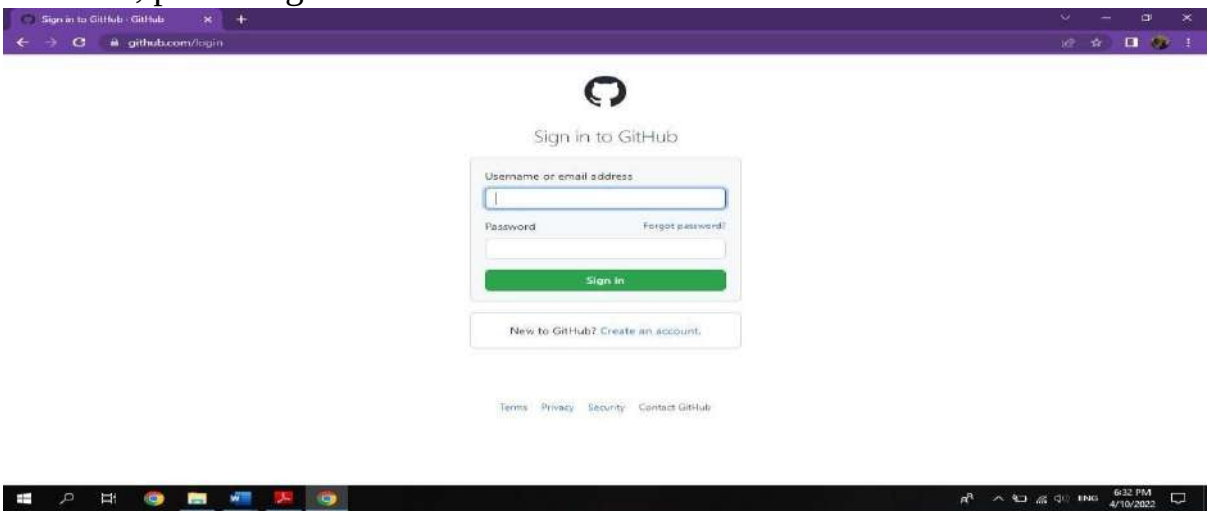
Aim: Setting up GitHub Account

The first steps in starting with GitHub are to create an account, choose a product that fits your needs best, verify your email, set up two-factor authentication, and view your profile.

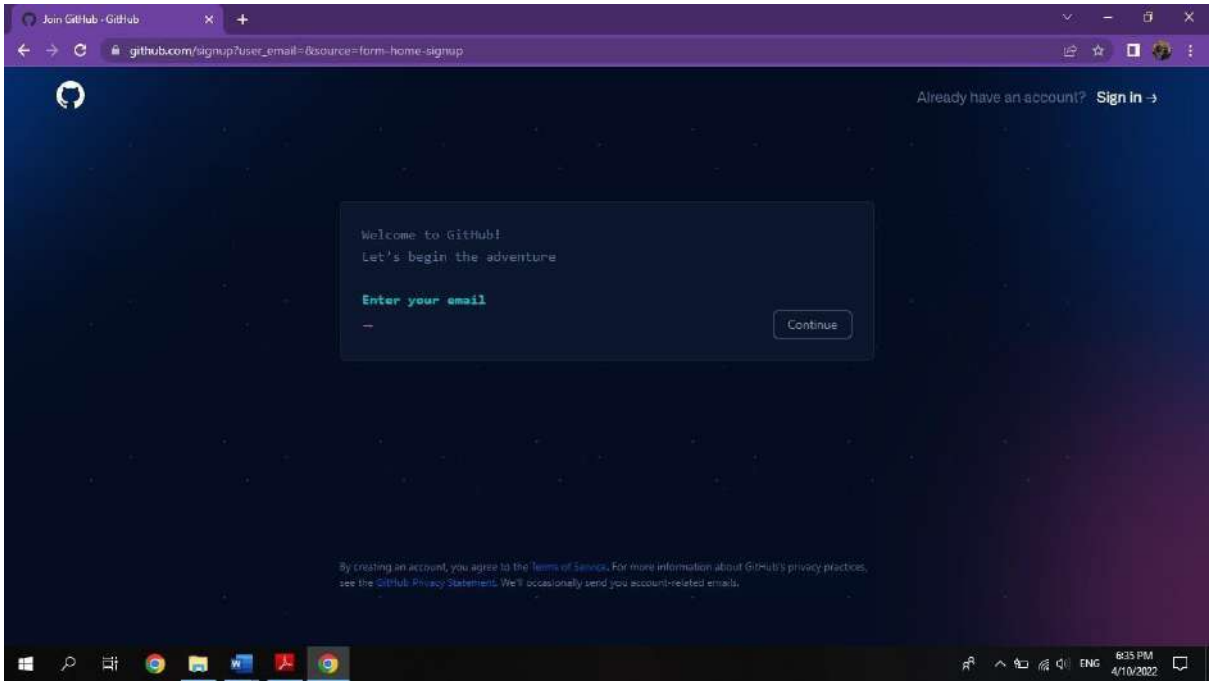
There are several types of accounts on GitHub. Every person who uses GitHub has their own user account, which can be part of multiple organisations and teams. Your user account is your identity on GitHub.com and represents you as an individual.

1. **Creating an account:** To sign up for an account on GitHub.com, navigate to https://github.com/ and follow the prompts.

2. Click on Create an account if you are a new user or if you have already an account, please login.



3. After Clicking on create a new account you will be redirected to a new page where you have to enter your email id which you want to use for your account. Now enter your password you want to create for your GitHub account. After that you will be asked to enter your username.



4. Now Click on Create Account.
5. Verify it from your email and you are all set to go.

Experiment - 3

Aim: Program to generate logs

Git Status:

The git status command **displays the state of the working directory and the staging area**. It lets you see which changes have been staged, which haven't, and which files aren't being tracked by Git. Status output does not show you any information regarding the committed project history.

Git Init:

The git init is **one way to start a new project with Git**. To start a repository, use either git init or git clone - not both. To initialize a repository, Git creates a hidden directory called .git . That directory stores all of the objects and refs that Git uses and creates as a part of your project's history.

Git Add:

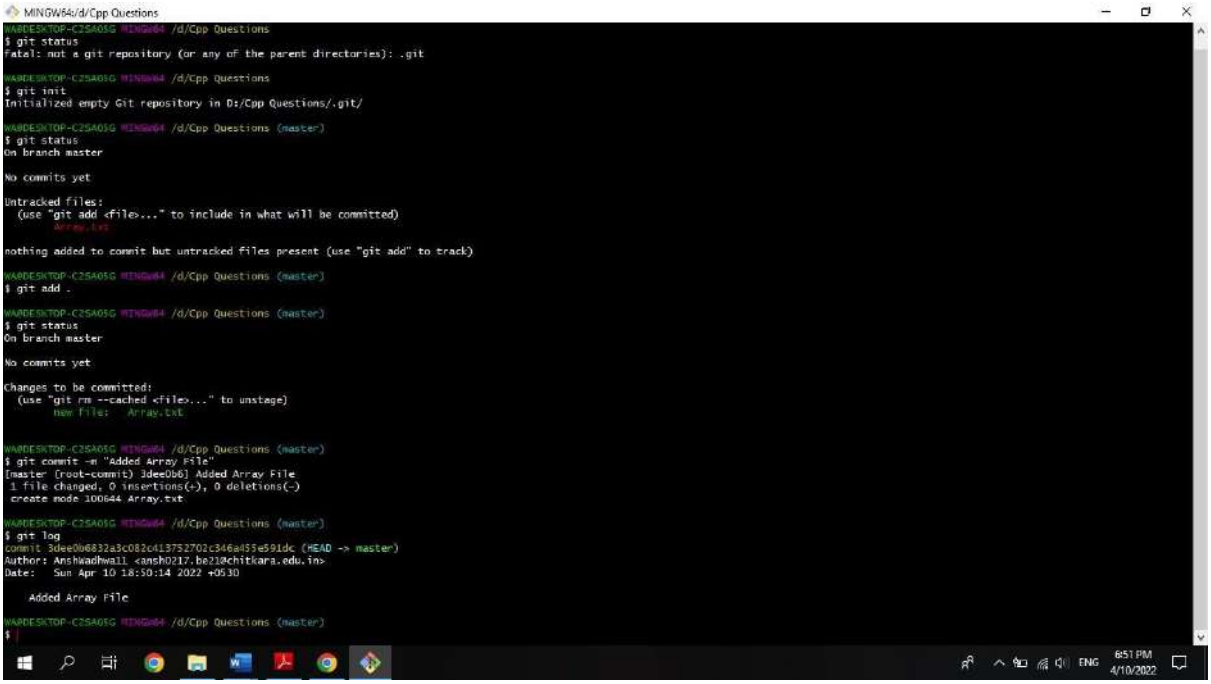
git add [filename] **selects that file, and moves it to the staging area, marking it for inclusion in the next commit**. You can select all files, a directory, specific files, or even specific parts of a file for staging and commit.

Git commit:

The git commit command **captures a snapshot of the project's currently staged changes**. Committed snapshots can be thought of as “safe” versions of a project—Git will never change them unless you explicitly ask it to.

Git Log:

Git log is **a utility tool to review and read a history of everything that happens to a repository**. Multiple options can be used with a git log to make history more specific. Generally, the git log is a record of commits.

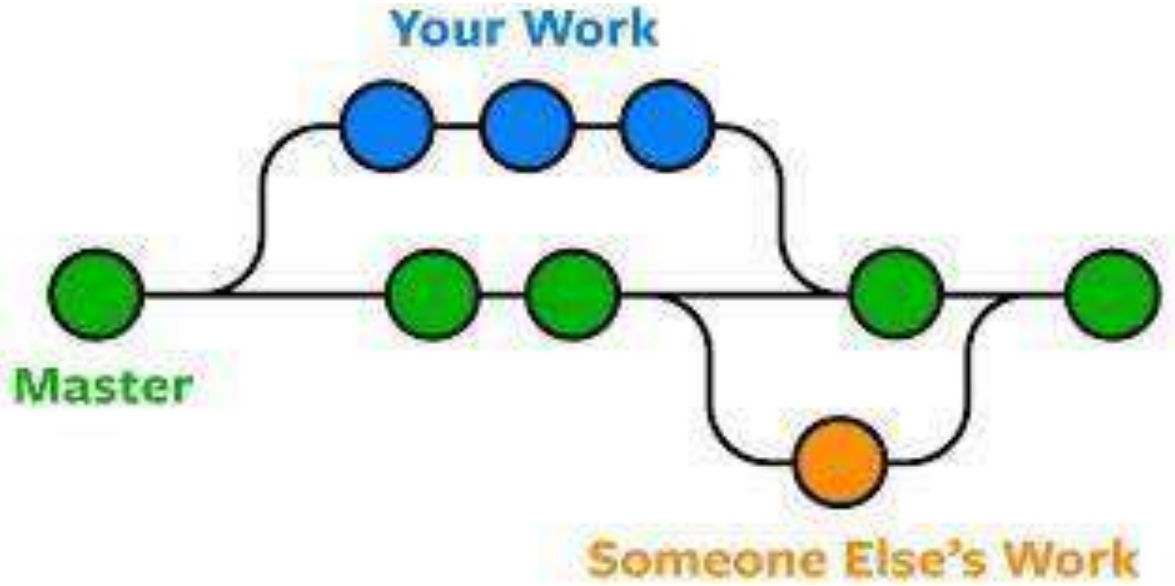

MINGW64:/d/Cpp Questions
$ git status
Fatal: not a git repository (or any of the parent directories): .git
$ git init
Initialized empty Git repository in D:/Cpp Questions/.git/
$ git status
On branch master
No commits yet
Untracked files:
(use "git add <file>..." to include in what will be committed)
Array.txt
nothing added to commit but untracked files present (use "git add" to track)
$ git add .
$ git status
On branch master
No commits yet
Changes to be committed:
(use "git rm --cached <file>..." to unstage)
new file: Array.txt
$ git commit -m "Added Array File"
[master (root-commit) 3dee0b6] Added Array File
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 Array.txt
$ git log
commit 3dee0b6832a3c082c413752702c346a455e591dc (HEAD -> master)
Author: AnshWadhwall <anshD217.be21@chitkara.edu.in>
Date: Sun Apr 10 18:50:14 2022 +0530
Added Array File
ENG
6:51 PM
4/10/2022

Experiment – 4

Aim: Create and visualize branches in Git

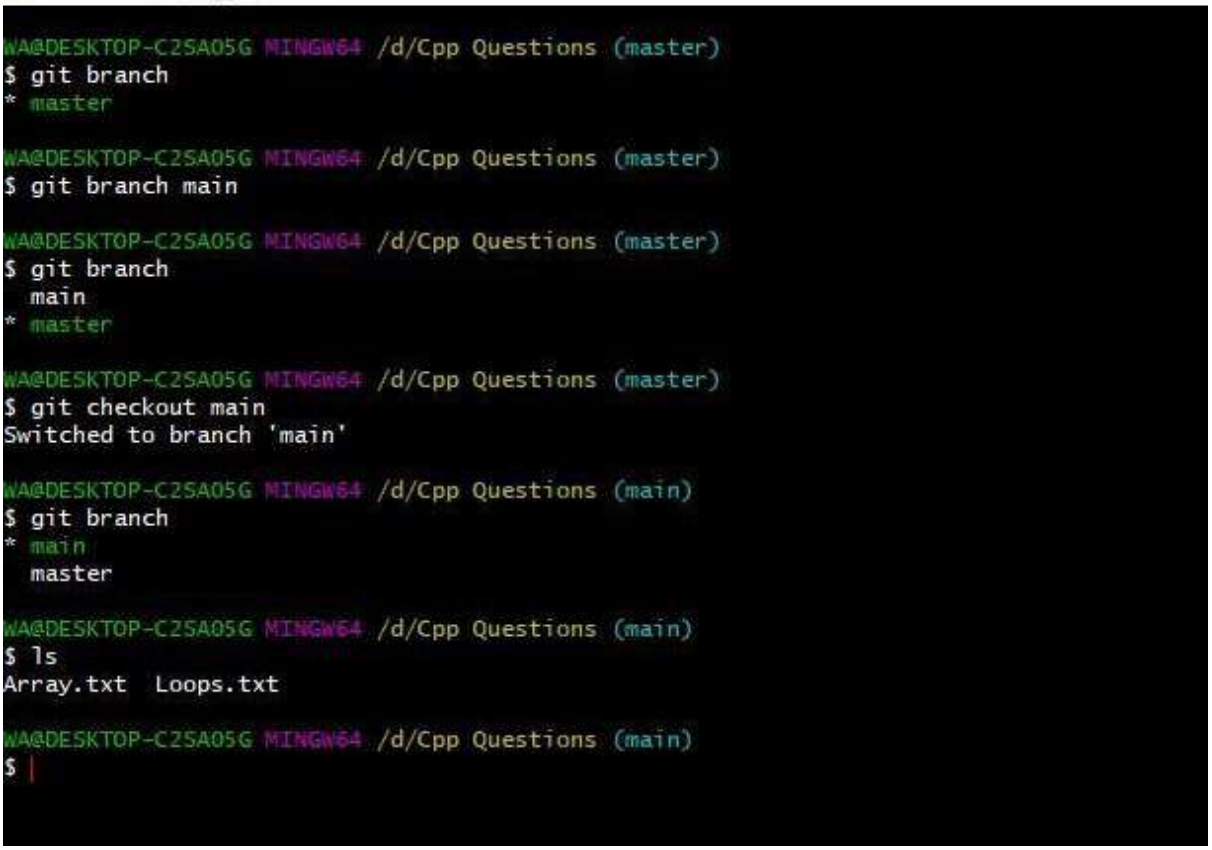
Git Branches:

A branch in Git is an independent line of work(a pointer to a specific commit). It allows users to create a branch from the original code (master branch) and isolate their work. Branches allow you to work on different parts of a project without impacting the main branch.



Let us see the command of it:
Firstly, add a new branch, let us suppose the branch name is activity1.
For this use command → **git branch name [**adding new branch**] git branch [**use to see the branch's names**]**
git checkout branch name [use to switch to the given branch**]**

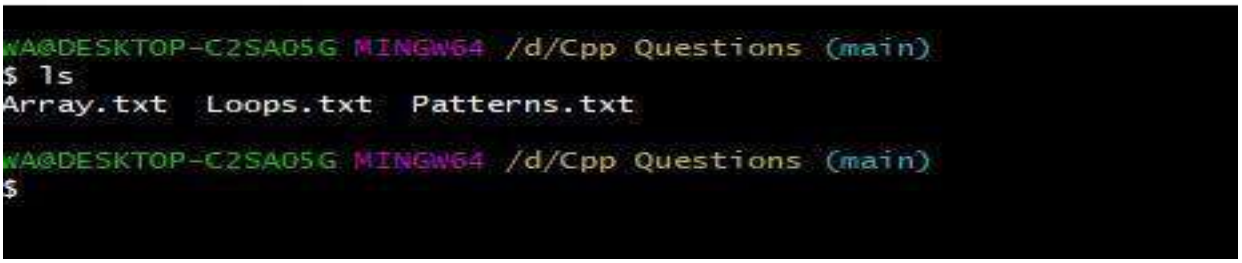
MINGW64:/d/Cpp Questions



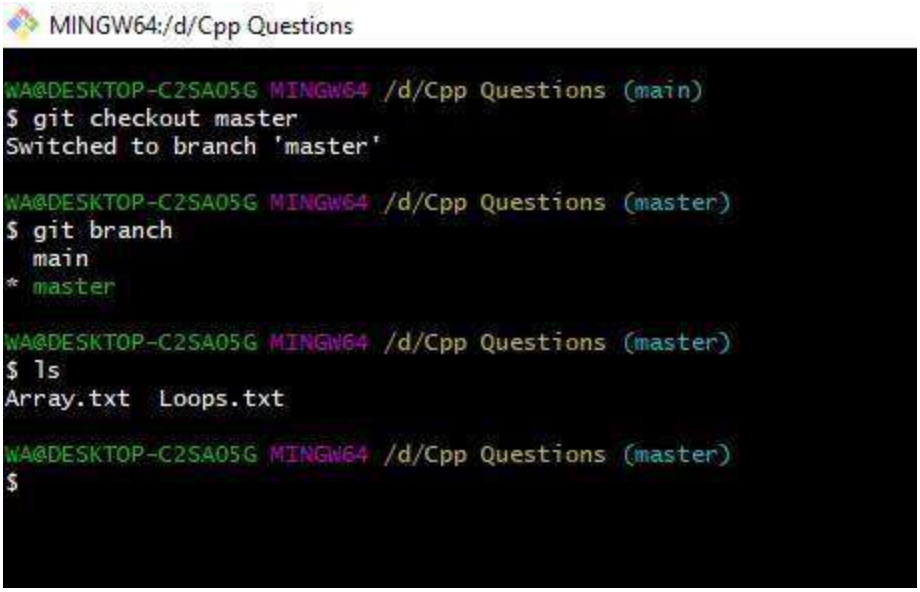
In this you can see that firstly ‘git branch’ shows only one branch in green colour but when we add a new branch using ‘git branch main’, it shows 2 branches but the green colour and star is on master. So, we have to switch to main by using ‘git checkout main’. If we use ‘git branch’, now you can see that the green colour and star is on main. It means you are in main branch and all the data of master branch is also on activity1 branch.

Now add a new file in activity1 branch, do some changes in file and commit the file.

MINGW64:/d/Cpp Questions



If we switched to master branch, ‘Patterns.txt’ file is not there. But the file is in main branch.



To add these files in master branch, we have to do merging. For this firstly switch to master branch and then use command:

Git merge branch name [use to merge branch**]**

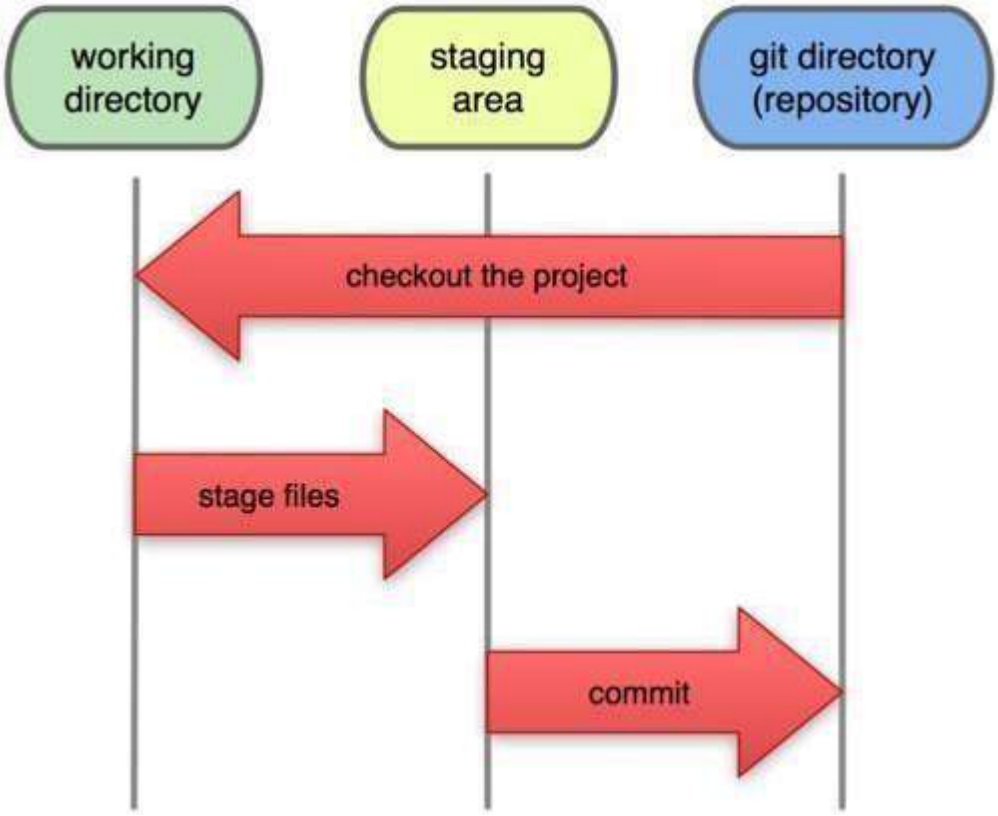
MINGW64:/d/Cpp Questions

```
WA@DESKTOP-C2SA05G MINGW64 /d/Cpp Questions (master)
$ git branch
  main
* master

WA@DESKTOP-C2SA05G MINGW64 /d/Cpp Questions (master)
$ ls
Array.txt  Loops.txt

WA@DESKTOP-C2SA05G MINGW64 /d/Cpp Questions (master)
$ git merge main
Updating 578dd81..4d21dd0
Fast-forward
 Patterns.txt | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 Patterns.txt

WA@DESKTOP-C2SA05G MINGW64 /d/Cpp Questions (master)
$ git branch
  main
* master

WA@DESKTOP-C2SA05G MINGW64 /d/Cpp Questions (master)
$ ls
Array.txt  Loops.txt  Patterns.txt

WA@DESKTOP-C2SA05G MINGW64 /d/Cpp Questions (master)
$
```

Experiment – 5

Aim: Git Life Cycle Description

Now let's understand the three-stage architecture of Git:



Working Directory: This is the directory that we've initialized, and here all the changes are made to commit on GitHub.

Staging Area: This is where we first put out code or files of the working repository. The command that we use to stage code is, "git add --a", "git add File Name" or "git add -A".
In simple terms, staging means telling Git what files we want to commit (new untracked files, modified files, or deleted files).

Git directory(repository): This is where all the commits are stored whenever we make a commit. We can revert to an older version of or project using the "git checkout" command from this directory.

Some Important Commands

- **Git operations and commands:**

First of all, Create a local repository using Git. For this, you have to make a folder in your device, right click and select "**Git Bash Here**". This opens the Git terminal. To create a new local repository, use the command "**git init**" and it creates a folder **.git**.

- When we use GIT for the first time, we have to give the user name and email so that if I am going to change in project, it will be visible to all.

  For this, we use command →

  "git config --global user.name *Name*"
  "git config --global user.email *email*"

  For verifying the user's name and email, we use →

  "git config --global user.name"
  "git config --global user.email"

- **ls** → It gives the file names in the folder.
- **ls -lart** → Gives the hidden files also.
- **git status** → Displays the state of the working directory and the staged snapshot.
- **touch filename** → This command creates a new file in the repository.
- **Clear** → It clears the terminal.
- **rm -rf .git** → It removes the repository.
- **git log** → displays all of the commits in a repository's history
- **git diff** → It compares my working tree to staging area

Experiment – 6

Aim: Add Collaborators on Github Repo.

Theory:

In GitHub, we can invite other GitHub users to become collaborators to our private repositories (which expires after 7 days if not accepted, restoring any unclaimed licenses). Being a collaborator, of a personal repository you can pull (read) the contents of the repository and push (write) changes to the repository. You can add unlimited collaborators on public and private repositories (with some per day limit restrictions). But, in a private repository, the owner of the repo can only grant write-access to the collaborators, and they can't have the read-only access.

GitHub also restricts the number of collaborators we can invite within a period of 24 hours. If we exceed the limit, then either we have to wait for 24hours or we can also create an organization to collaborate with more people.

Actions that can be Performed by Collaborators

Collaborators can perform a number of actions into someone else's personal repositories, they have gained access to Some of them are,

- Create, merge, and close pull requests in the repository
- Publish, view, install the packages
- Fork the repositories
- Make the changes on the repositories as suggested by the Pull requests.
- Mark issues or pull requests as duplicate
- Create, edit, and delete any comments on commits, pull requests, and issues in the repository
- Removing themselves as collaborators on the repositories.
- Manage releases in the repositories.

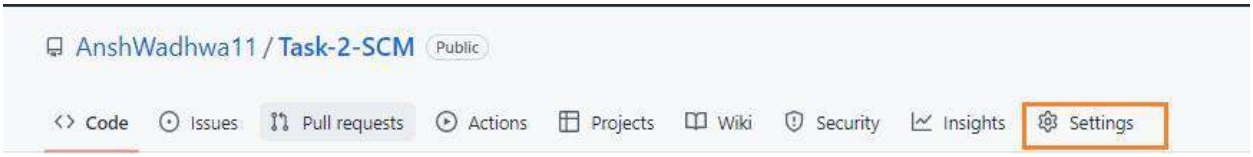
Now, let's see how can we invite collaborators to our repositories.
Inviting Collaborators to your personal repositories

Procedure:

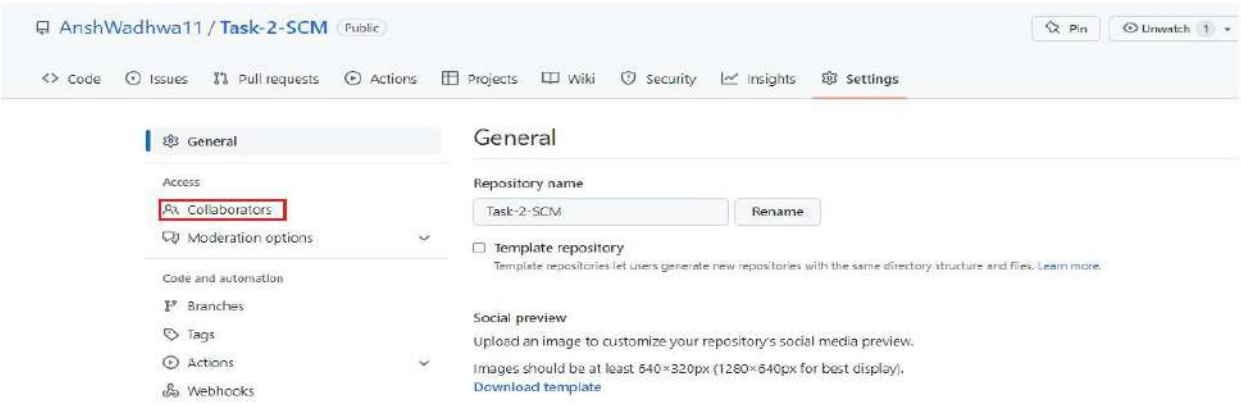
1. Get the usernames of the GitHub users you will be adding as collaborators. In case, they are not on GitHub, ask them to sign in to GitHub.

2. Go to your repository(intended to add collaborators).

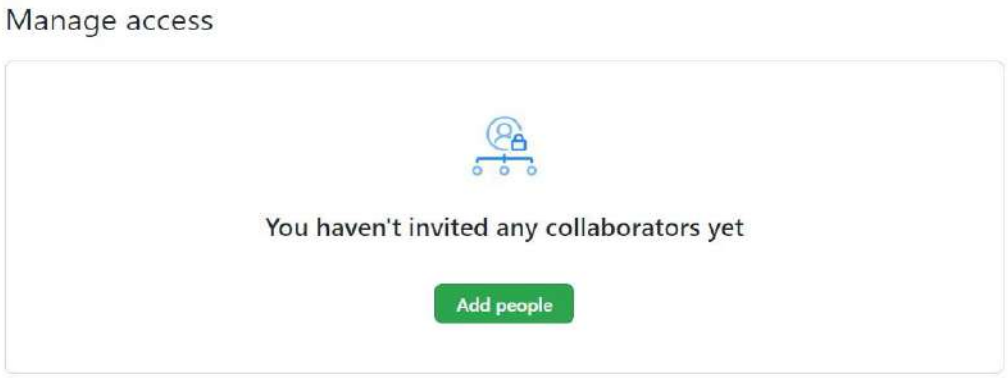
3. Click into the Settings.



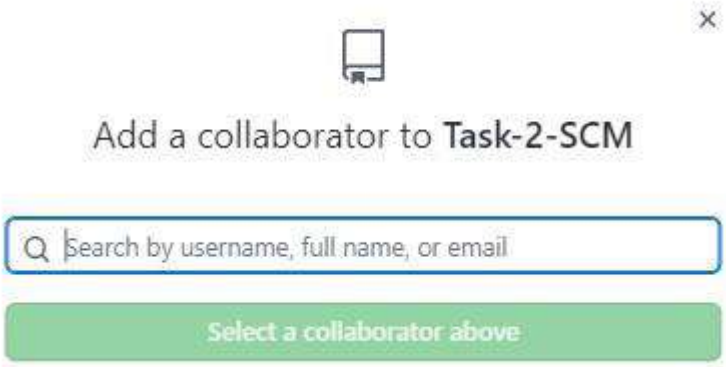
4. A settings page will appear. Here, into the left-sidebar click into the Collaborators.



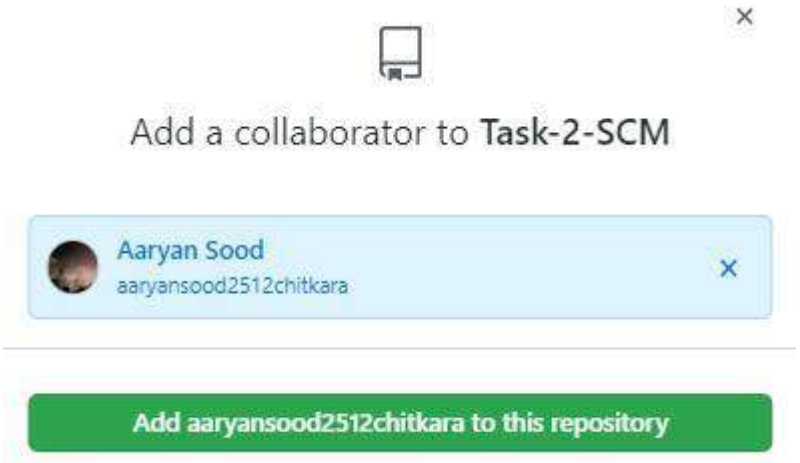
5. Then a confirm password page may appear, enter your password for the confirmation.
6. Next, click into Add People.



7. Then a search field will appear, where you can enter the username of the ones you want to add as collaborator.

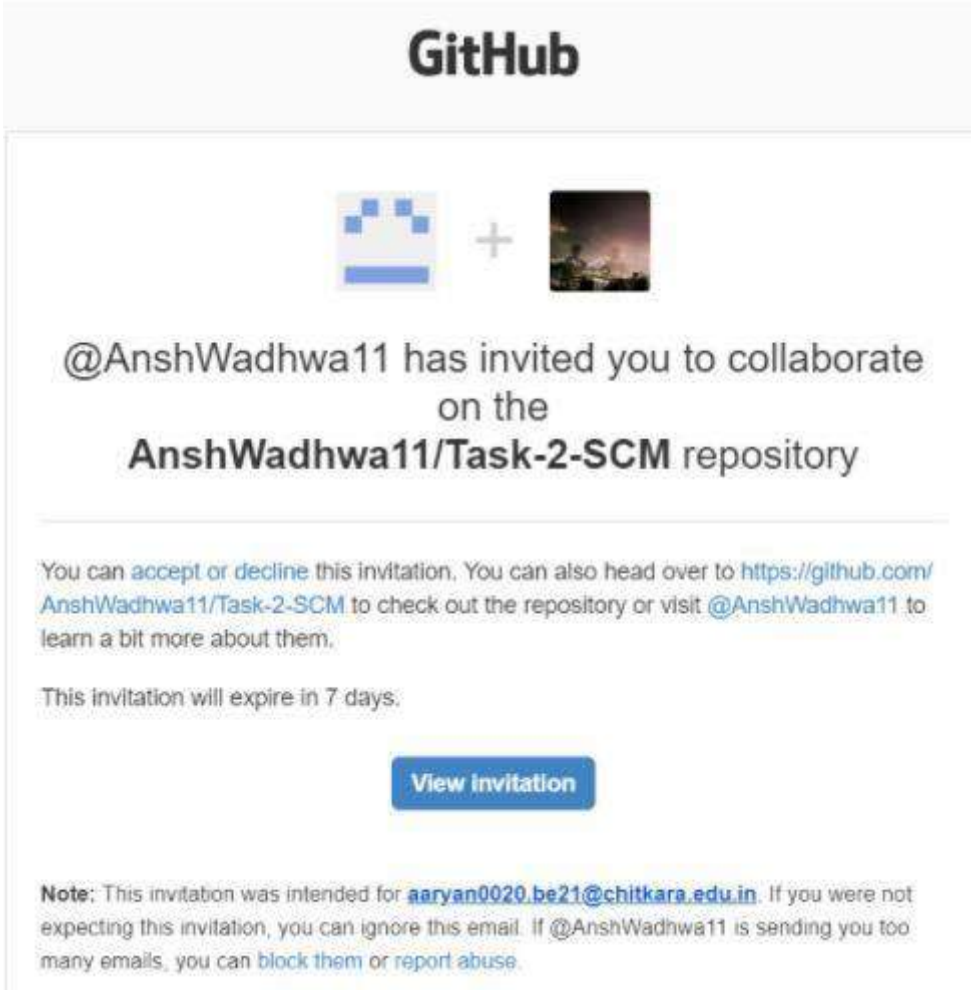


8. After selecting the people, add them as collaborator

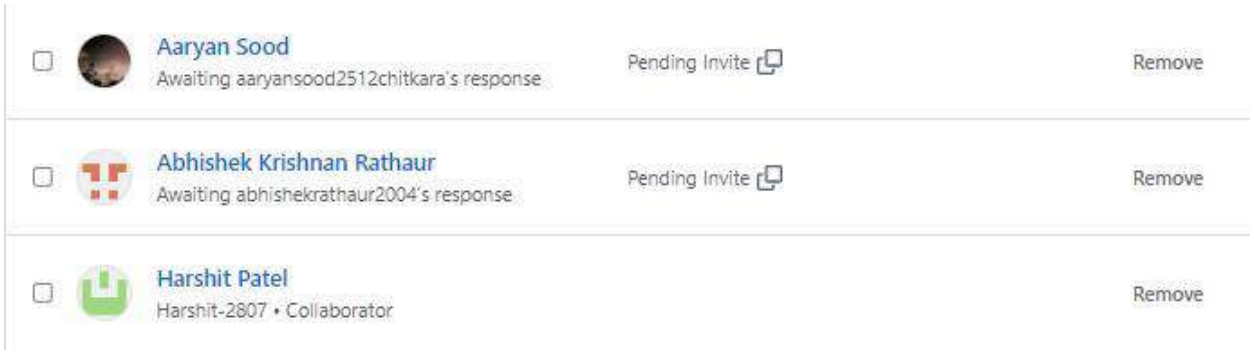


9. After sending the request for collaboration to that person whom we wanted as our collaborator for our project will get a email from the

Team leader (by GitHub). He/she has to open its Email to view the invitation.



10. We are done adding a multiple collaborator. Now, they will get a mail regarding invitation to your repository. Once they accept, they will have collaborator access to your repository. Till then it will be in pending invitation state. You can also add more collaborator and delete the existing one as depicted below.


Aaryan Sood
Awaiting aaryansood2512chitkara's response
Pending Invite
Remove
Abhishek Krishnan Rathaur
Awaiting abhishekrathaur2004's response
Pending Invite
Remove
Harshit Patel
Harshit-2807 • Collaborator
Remove

Experiment – 7

Aim: Fork and Commit

Theory: A fork is a copy of a repository that we manage.
Forks let us make changes to a project without affecting the original repository. We can fetch updates from or submit changes to the original repository with pull requests.

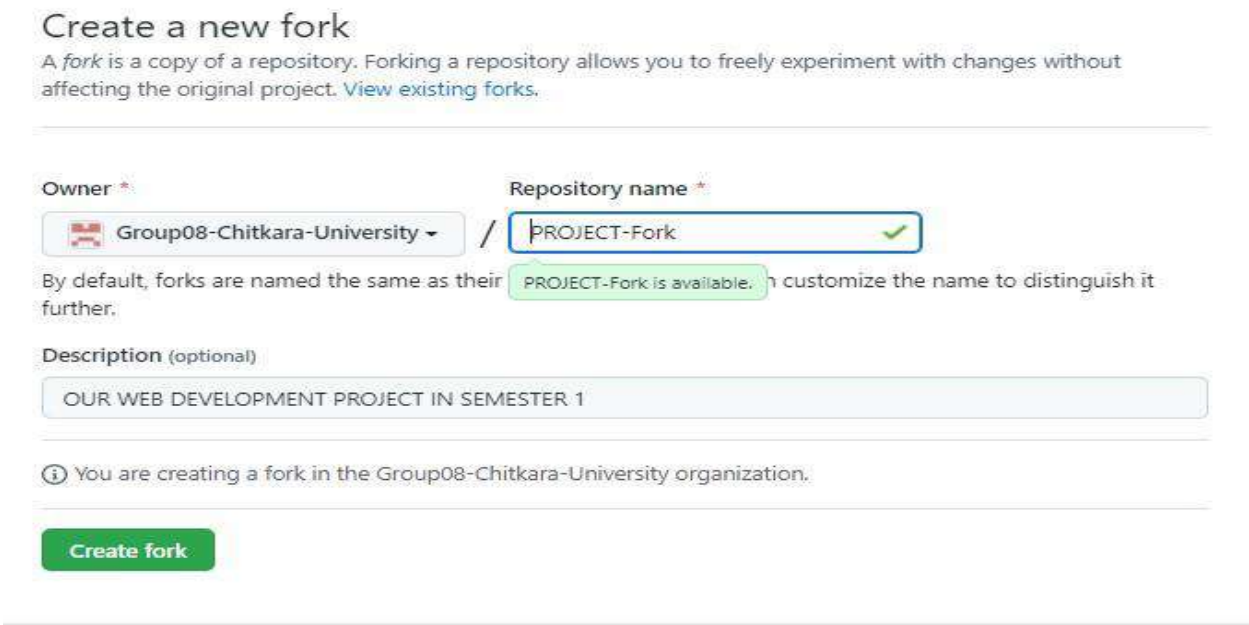
If we need to fork a GitHub or GitLab repo, it's as simple as navigating to the landing page of the repository in your web browser and clicking on the *Fork* button on the repository's home page. A forked copy of that Git repository will be added to your personal GitHub or GitLab repo. That's it. That's all we have to do to fork a Git repo.

Procedure:

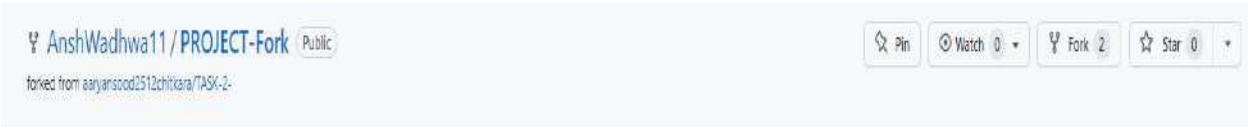
1. Go to the repo you want to fork.

2. Find the Fork button on the top right corner.

3. Click on **Fork.**



After this, we will see how to work in the forked repositories which were forked by the collaborators. If the collaborator tries to change in his forked repository, it will not affect the main repository



What is COMMIT in GitHub?

Commit is like a snapshot of your repository. These commits are snapshots of your entire repository at specific times. You should make new commits often, based around logical units of change.

Over time, commits should tell a story of the history of your repository and how it came to be the way that it currently is.

Commits include lots of metadata in addition to the contents and message, like the author, timestamp and more.

It is similar to saving a file that's been edited, a commit records changes to one or more files in your branch. Git assigns each commit a unique ID, called a SHA or hash, that identifies:

- The Specific Changes
- When the Changes were made

- Who created the changes

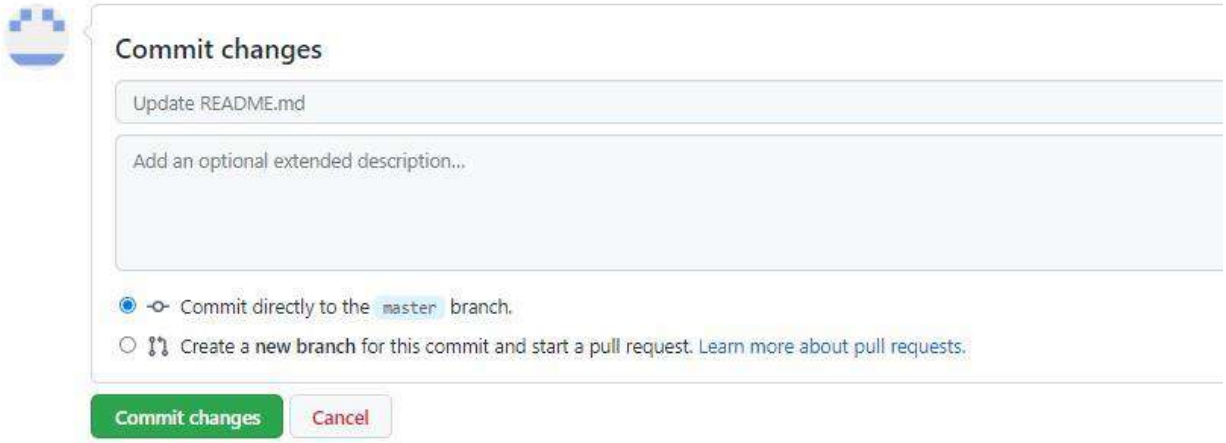
When you make a commit, you must include a commit message that briefly describes the changes, you can also add a co-author on any commits you collaborate on.

Now, we will see the main repository content of the main Project-SCM Repository. We can see that there is a **README.md** file

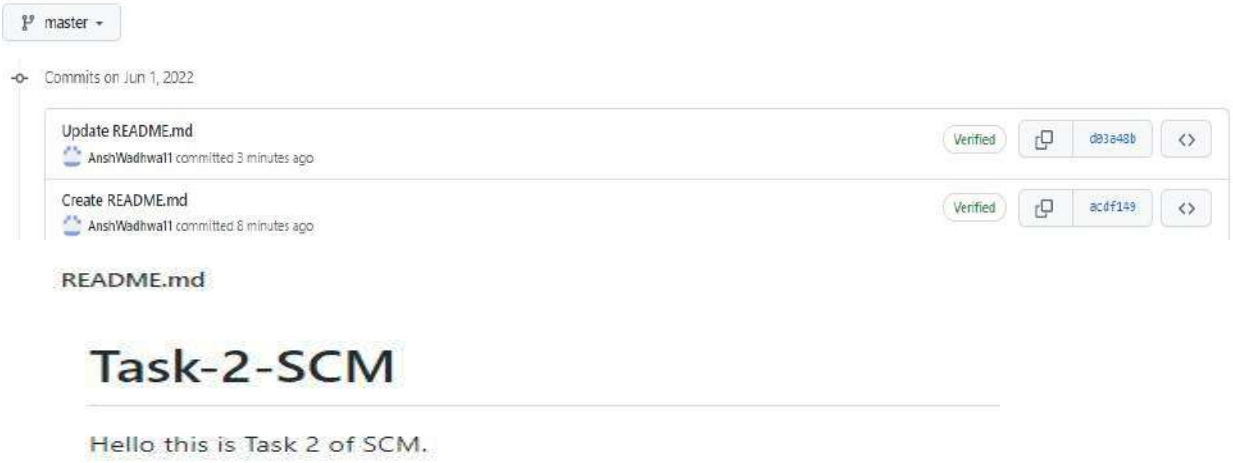
The content in the **README.md** file in the main **Mascots** repository, is given as below:

- Now, when **AnshWadhwa11** (collaborator) tries to change the content of the **README.md** file in his forked repository i.e.,

- We can see that the **README.md** file has been edited and now it will be committed in this forked repository



- We can see the Commit history of the AnshWadhwa11/Task-2-SCM Repository.



Experiment – 8

Aim: Merge and Resolve conflicts created due to own activity and collaborators activity.

Theory:

- Do changes in new branch and commit those change. And checkout to master branch and again do changes and commit it. Now in master branch merge the another branch.

```
WA@AnshWadhwa MINGW64 /d/ansh (master)
$ git checkout feature
Switched to branch 'feature'

WA@AnshWadhwa MINGW64 /d/ansh (feature)
$ git add .

WA@AnshWadhwa MINGW64 /d/ansh (feature)
$ git commit -m "Modified File"
[feature a3d3953] Modified File
 1 file changed, 1 insertion(+)
```

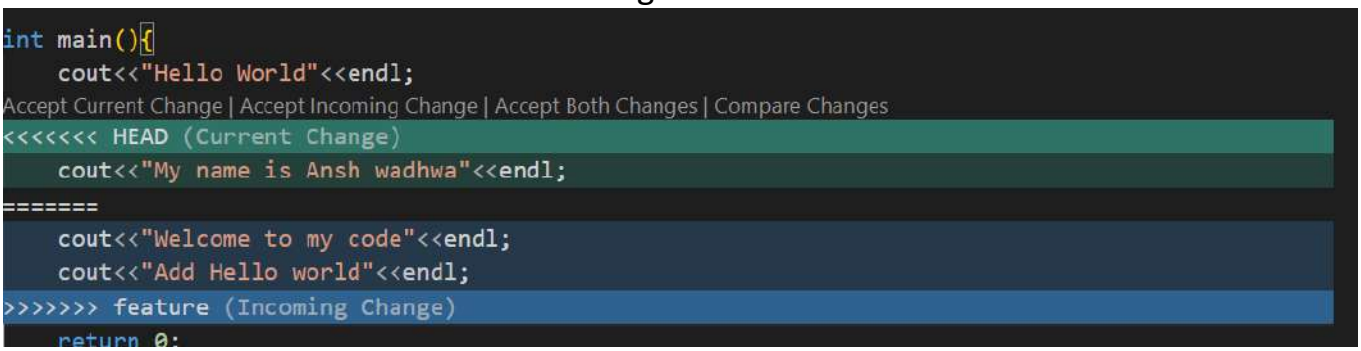
- Now Commit on Master Branch.

```
WA@AnshWadhwa MINGW64 /d/ansh (feature)
$ git checkout master
Switched to branch 'master'

WA@AnshWadhwa MINGW64 /d/ansh (master)
$ git add .

WA@AnshWadhwa MINGW64 /d/ansh (master)
$ git commit -m "Add Myself"
[master 19efa4c] Add Myself
 1 file changed, 1 insertion(+)
```

- Now try to merge it will give Conflicts Error.

```
WA@AnshWadhwa MINGW64 /d/ansh (master)
$ git merge feature
Auto-merging hello.cpp
CONFLICT (content): Merge conflict in hello.cpp
Automatic merge failed; fix conflicts and then commit the result.
```

- Use Command “git mergetool” to solve the conflict. **git -mergetool** – Run merge conflict resolution tools to resolve merge conflicts



- Press “I” to insert, after insertion . Press “:wq”. The merge conflict is solvedand our Activity branch is merged to master branch.

Experiment – 9

Aim: Reset and Revert

Theory: While Working with Git in certain situations we want to undo changes in the working area or index area, sometimes remove commits locally or remotely and we need to reverse those changes. There are 3 different ways in which we can undo the changes in our repository, these are ***git reset***, ***git checkout***, and ***git revert***. git checkout and git reset in fact can be used to manipulate commits or individual files. These commands can be confusing so it's important to find out the difference between them and to know which command should be used at a particular point of time.

Let's make a sample git repository with a file ***constructor.cpp*** and Code written inside it.

```
WA@AnshWadhwa MINGW64 /d/practice (master)
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        constructor.cpp

nothing added to commit but untracked files present (use "git add" to track)

WA@AnshWadhwa MINGW64 /d/practice (master)
$ git add constructor.cpp

WA@AnshWadhwa MINGW64 /d/practice (master)
$ git commit -m "Bank Constructor code"
[master (root-commit) 3130b11] Bank Constructor code
 1 file changed, 67 insertions(+)
 create mode 100644 constructor.cpp

WA@AnshWadhwa MINGW64 /d/practice (master)
$ git log
commit 3130b11788e35c35b2583ee4108119c7ffaf4d4f (HEAD -> master)
Author: AnshWadhwa11 <ansh0217.be21@chitkara.edu.in>
Date:   Wed Jun 1 22:54:23 2022 +0530

    Bank Constructor code
```

We can see that we have a single commit is done on the code file that has been committed with added new files in it. Now let's add some more comment or code to our code file. Let's add another line ***Hello World.*** Doing this change, our code file now needs to be added to the staging area for getting the commit done. This updates are currently in the working area and to see them we will see those using ***git status***.

```
WA@AnshWadhwa MINGW64 /d/practice (master)
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        constructor.cpp

nothing added to commit but untracked files present (use "git add" to track)
```

Now we have a change ***Hello World*** which is untracked in our working repository and we need to discard this change. So, the command that we should use here is –

1. Git checkout

Git checkout is used to discard the changes in the working repository.

git checkout <filename>

When we write git checkout command and see the status of our git repository and also the code file we can see that our changes are being discarded from the working directory and we are again back to the code file that we had before. Now, what if we want to unstage a file. We stage our files before committing them and at a certain point, we might want to unstage a file. Let's add ***Hello World*** again to our text document and stage them using the ***git add*** command.

```
WA@AnshWadhwa MINGW64 /d/practice (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   constructor.cpp

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   constructor.cpp


WA@AnshWadhwa MINGW64 /d/practice (master)
$ git checkout -- constructor.cpp
```

2. Git Reset

Git reset is used when we want to unstage a file and bring our changes back to the working directory. Git reset can also be used to remove commits from the local repository.

git reset HEAD <filename>

Whenever we unstage a file, all the changes are kept in the working area.

We are back to the working directory, where our changes are present but the file is now unstaged. Now there are also some commits that we don't want to get committed and we want to remove them from our local repository. To see how to remove the commit from our local repository let's stage and commit the changes that we just did and then remove that commit.

We have 2 commits now, with the latest being the Added Hello World commit which we are going to remove. The command that we would be using now is – git reset Head~1

Point to be noted :-

- HEAD~1 here means that we are going to remove the topmost commit or the latest commit that we have done.
- We cannot remove a specific commit with the help of git reset , for ex : we cannot say that we want to remove the second commit or the third commit , we can only remove latest commit or latest 2 commits . latest

C
S

N commits.(HEAD~n) [n here means n recent commits that needs to be deleted].

After using the above command we can see that our commit is being deleted and also our file is again unstaged and is back to the working directory. There are different ways in which git reset can actually keep your changes.

- ***git reset --soft HEAD~1*** - This command will remove the commit but would not unstage a file. Our changes still would be in the staging area.
- ***git reset --mixed HEAD~1*** or ***git reset HEAD~1*** **-** This is the default command that we have used in the above example which removes the commit as well as unstage the file and our changes are stored in the working directory.
- ***git reset --hard HEAD~1*** **-** This command removes the commit as well as the changes from your working directory. This command can also be called destructive command as we would not be able to get back the changes so be careful while using this command.

Points to keep in mind while using git reset command -

- If our commits are not published to remote repository, then we can use git reset.
- Use git reset only for removing commits that are present in our local directory and not in remote directory.
- We cannot remove a specific commit with the help of git reset, for ex: we cannot say that we want to remove the second commit or the third commit, we can only remove latest commit or latest 2 commits ... latest N commits. (HEAD~n) [n here means n recent commits that needs to be deleted].

We just discussed above that the git reset command cannot be used to delete commits from the remote repository, then how do we remove the unwanted commits from the remote repository.

3. Git Revert

Git revert is used to remove the commits from the remote repository. Since now our changes are in the working directory, let's add those changes to the staging area and commit them.

git revert <commit id of the commit that needs to be removed>

```
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   constructor.cpp


WA@AnshWadhwa MINGW64 /d/practice (master)
$ git add constructor.cpp

WA@AnshWadhwa MINGW64 /d/practice (master)
$ git commit -m "Add Commits in code"
[master 5a94157] Add Commits in code
 1 file changed, 2 insertions(+)

WA@AnshWadhwa MINGW64 /d/practice (master)
$ git log
commit 5a94157937002e45c872fd36b230ec354eaab982 (HEAD -> master)
Author: AnshWadhwa11 <ansh0217.be21@chitkara.edu.in>
Date:   Wed Jun 1 23:52:59 2022 +0530

    Add Commits in code

commit 3130b11788e35c35b2583ee4108119c7ffaf4d4f
Author: AnshWadhwa11 <ansh0217.be21@chitkara.edu.in>
Date:   Wed Jun 1 22:54:23 2022 +0530

    Bank Constructor code

WA@AnshWadhwa MINGW64 /d/practice (master)
$ git revert 5a94157937002e45c872fd36b230ec354eaab982
[master 90f4a61] Revert "Add Commits in code"
 1 file changed, 2 deletions(-)
```

```
Revert "Add Commits in code"

This reverts commit 5a94157937002e45c872fd36b230ec354eaab982.

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch master
# Changes to be committed:
#       modified:   constructor.cpp
#
```

Now we want to delete the commit that we just added to the remote repository. We could have used the git reset command but that would have deleted the commit just from the local repository and not the remote repository. If we do this then we would get conflict that the remote commit is not present locally. So, we do not use git reset here. The best we can use here is git revert.

git revert <commit id of the commit that needs to be removed> Points to keep in mind -

- Using git revert we can undo any commit , not like git reset where we could just remove "n" recent commits.

Now let's first understand what git revert does, git revert removes the commit that we have done but adds one more commit which tells us that the revert has been done.

We can see that the new commit is being added. However since this commit is in local repository so we need to do ***git push*** so that our remote repository also notices that the change has been done.

And as we can see we have a new commit in our remote repository and ***Hello World*** which we added in our 2nd commit is being removed from the local as well as the remote repository.

Difference Table

git checkout	**git reset**	**git revert**
Discards the changes in the working repository.	Unstages a file and bring our changes back to the working directory	Removes the commits from the remote repository.
Used in the local repository.	Used in local repository	Used in the remote repository

Does not make any changes to the commit history.	Alters the existing commit history	Adds a new commit to the existing commit history.
Moves HEAD pointer to a specific commit.	Discards the uncommitted changes.	Rollbacks the changes which we have committed.
Can be used to manipulate commits or files.	Can be used to manipulate commits or files.	Does not manipulate your commits or files.

Project

Problem Statement

There is a problem as we have to code a C++ project but we have to make our part and send it to other participants again and again and then at end we have to combine all the part and attach those snaps which is a tough process. So we want an alternate solution for this to show changes conveniently and accurately. Also we want that if we make the changes it should reflect to all the other participants. Also the other participants can revert the changes and commit the changes.

Objective

The objectives of the project are:

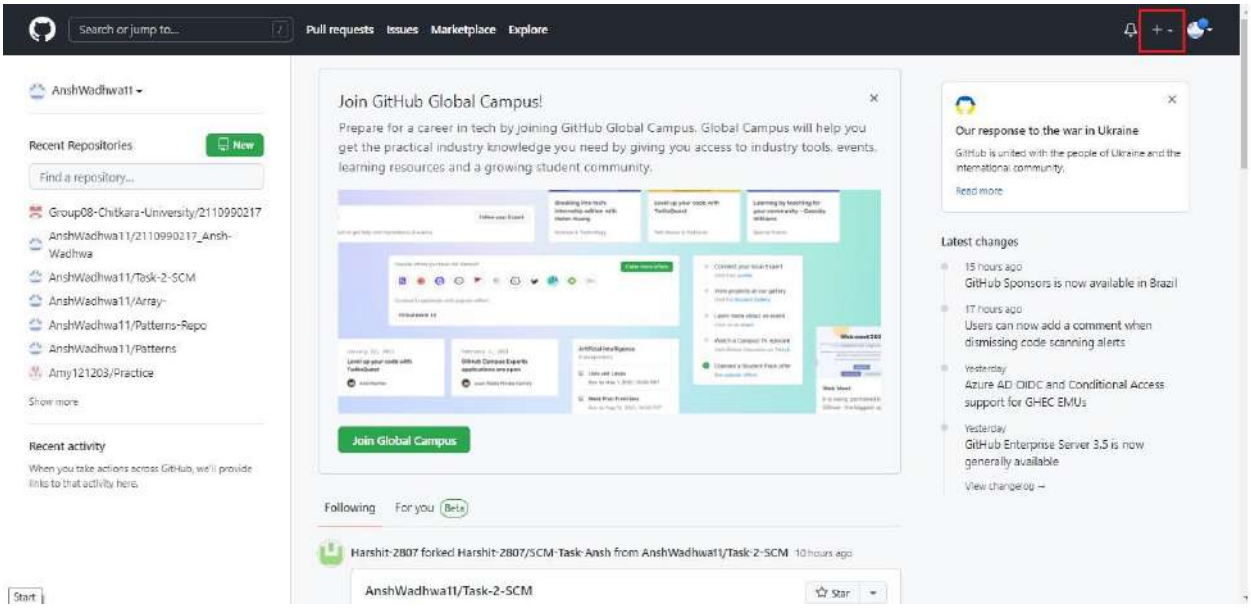
- It is easy to contribute to opensource projects via GitHub.
- It helps to create an excellent document.
- You can attract the recruiter by showing off your work.
- If you have a profile on GitHub, you will have a higher chance of being recruited.
- It allows your work to get out there in front of the public.

Experiment No. 10

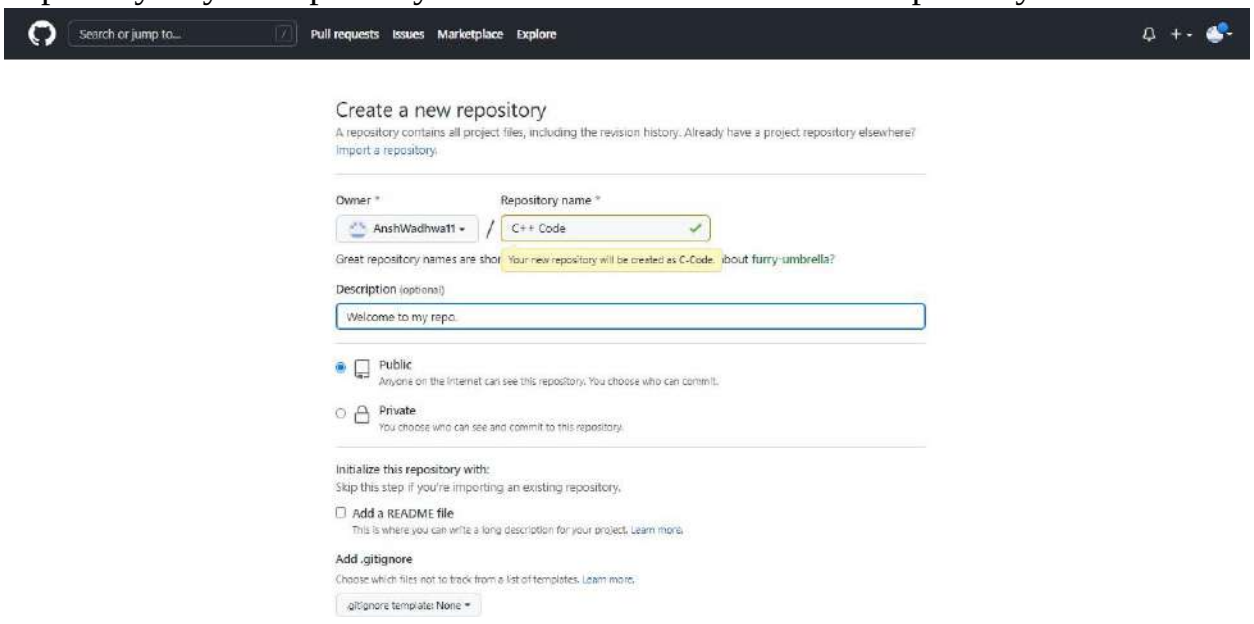
Aim: Create a distributed Repository and add members in project team.

Procedure:

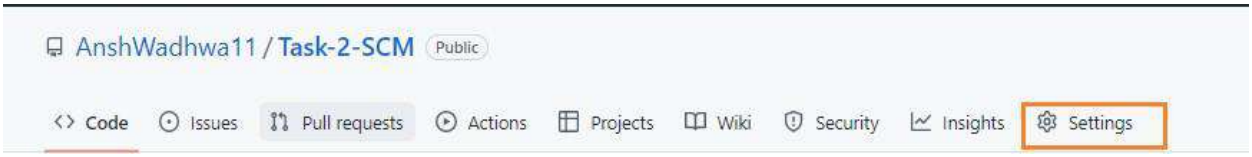
- Open GitHub home page, in the right corner you can find your profile logo beside it is + icon which is used to create repositories. Create Repository from there.



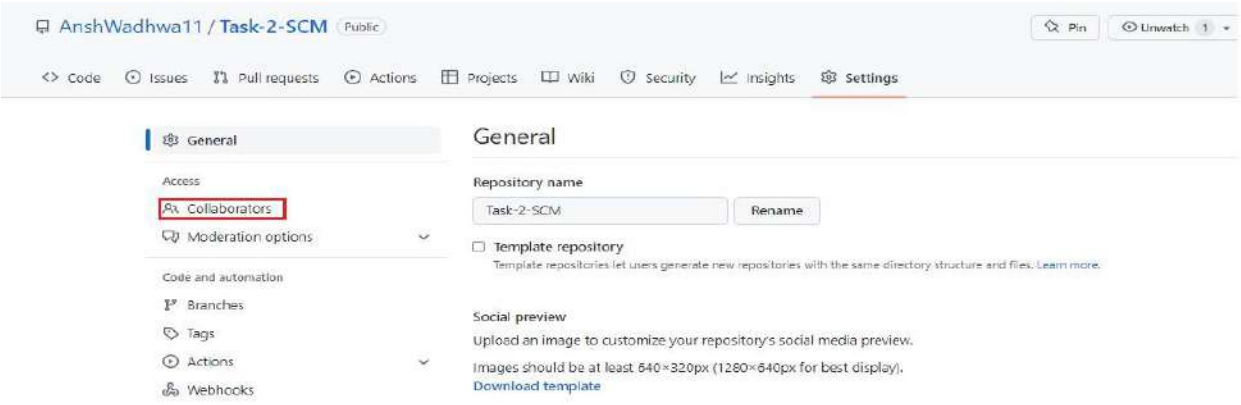
- In creating repository add name and description of your choice and choose privacy of your repository and at bottom click on create repository.



- Adding Members- To add members in your repo go to settings in your repo navigation bar.

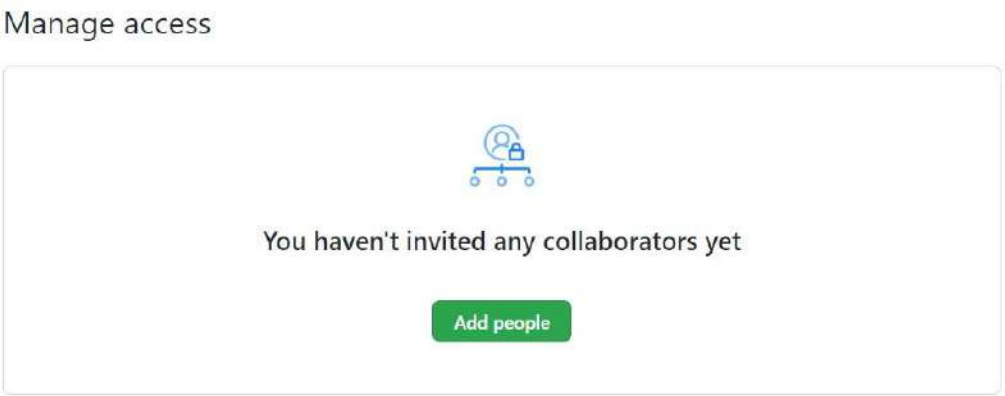


- A settings page will appear. Here, into the left-sidebar click into the Collaborators.

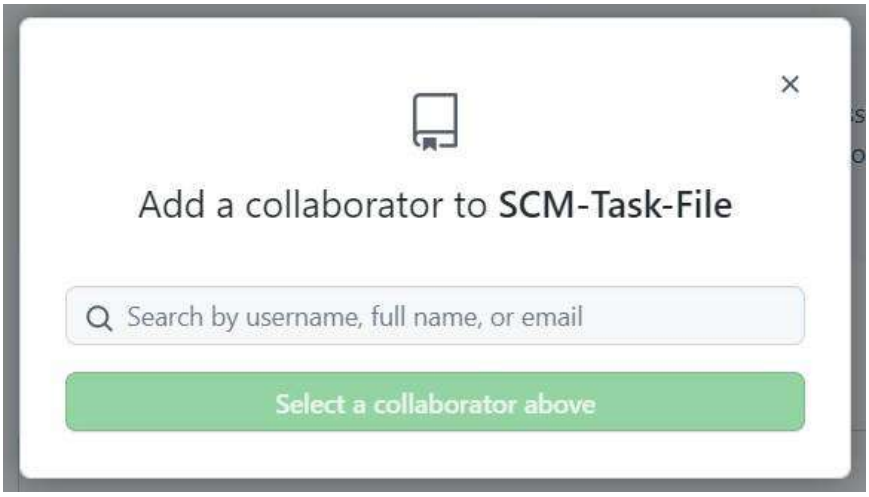


- Then a confirm password page may appear, enter your password for the confirmation.

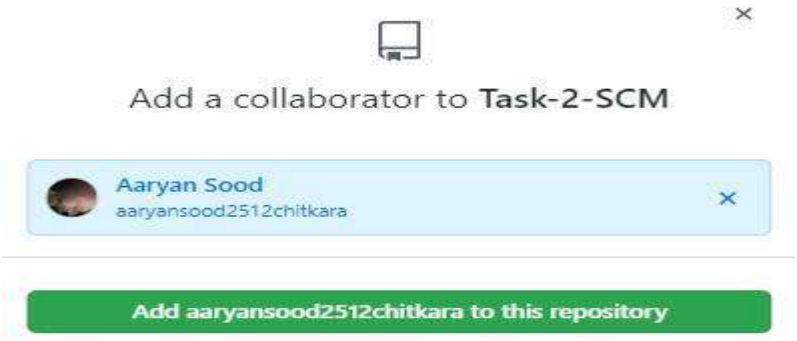
- Next Click into add button.



- Then a search field will appear, where you can enter the username of the ones you want to add as collaborator.



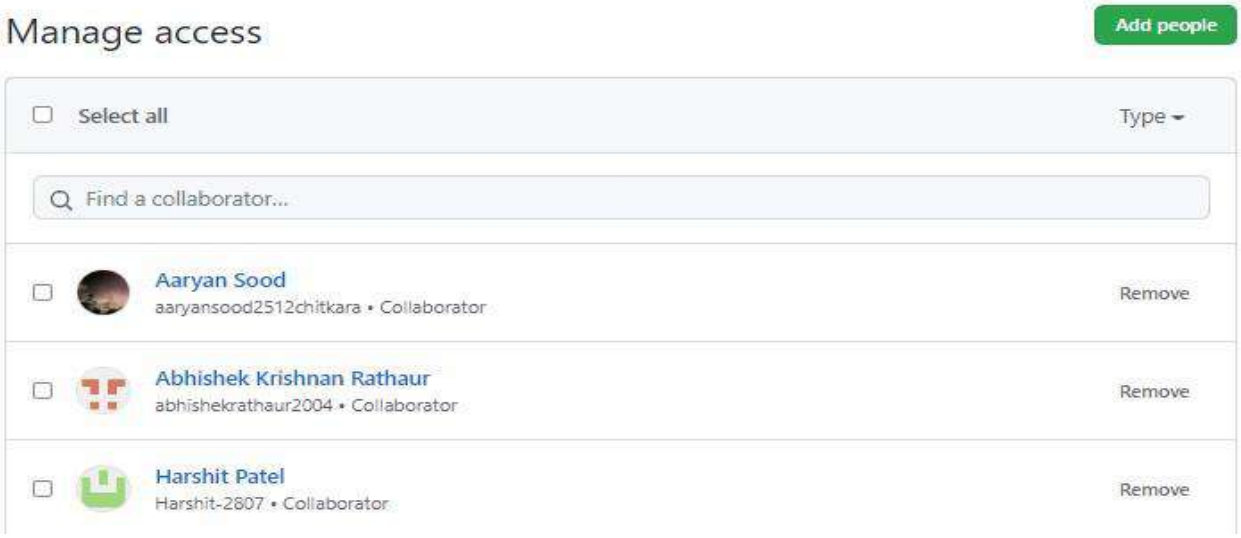
- After selecting the people, add them as collaborator.



- After sending the request for collaboration to that person whom we wanted as our collaborator for our project will get a email from the Team leader (by GitHub). He/she has to open its Email to view the invitation.

- After Clicking the View invitation, He/she will be redirected to the GitHub Page for accepting the invitation sent by the team leader.

- We are done adding a single collaborator. Now, they will get a mail regarding invitation to your repository. Once they accept, they will have collaborator access to your repository. Till then it will be in pending invitation state. You can also add more collaborator and delete the existing one as depicted below.



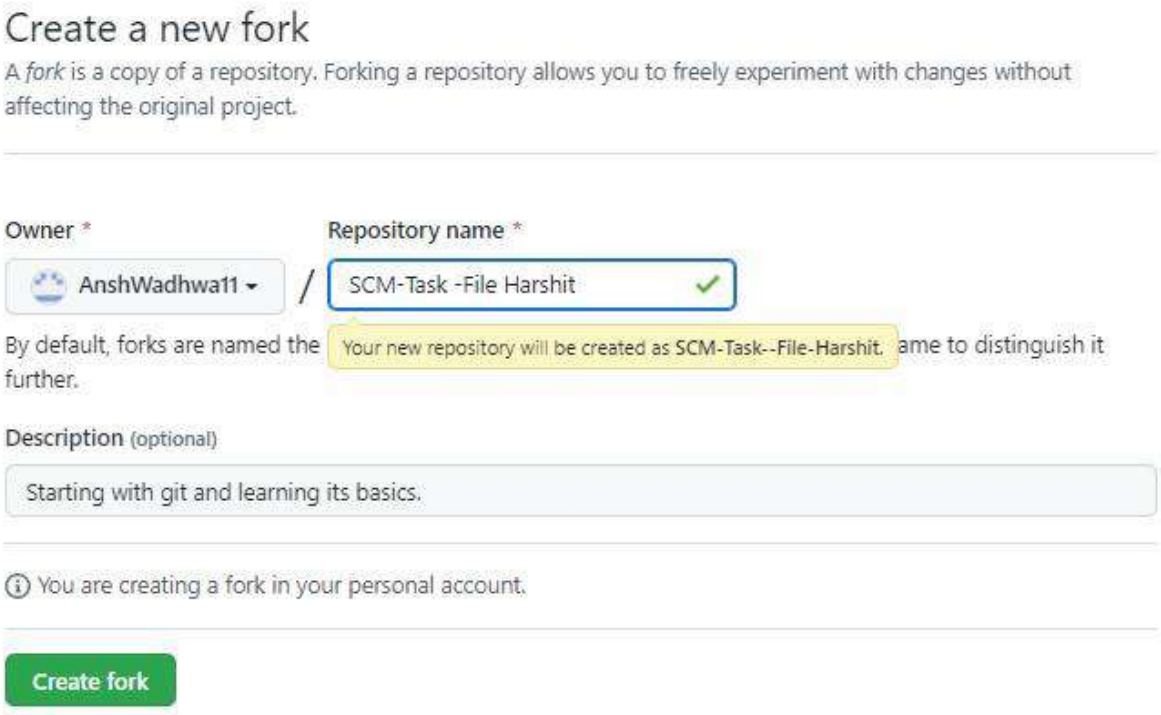
Experiment No.11

Aim: Open and Close a Pull Request

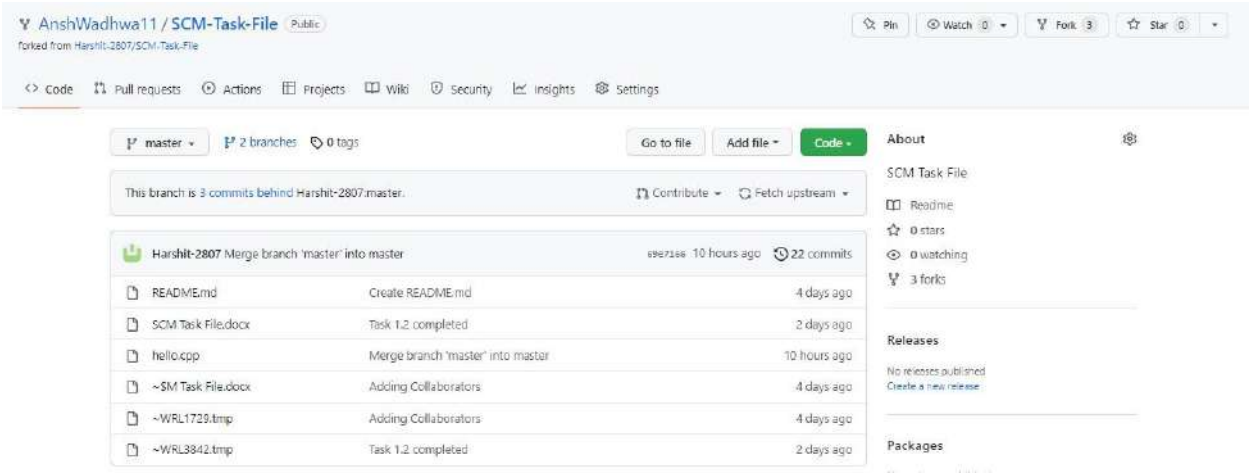
Forking Repositories:

- Open the repo which you want to fork then find fork button in right side of the navigation panel of repo.

- Click on fork.
- Then change the name if you wish to else you can continue with previous name and then click create fork.

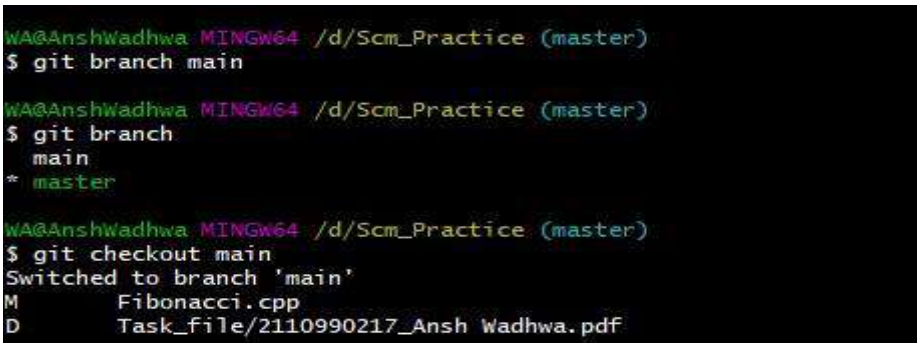


- That's it, copy of the repository is created as your repository now make changes and commit them in your local copy of repository.



Opening Pull Request:

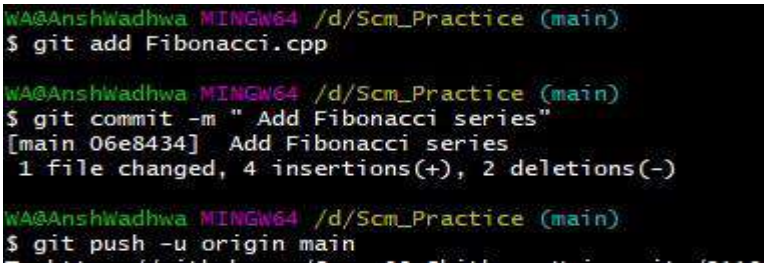
To open a pull request we first have to make a new branch, by using git branch *branch name.*



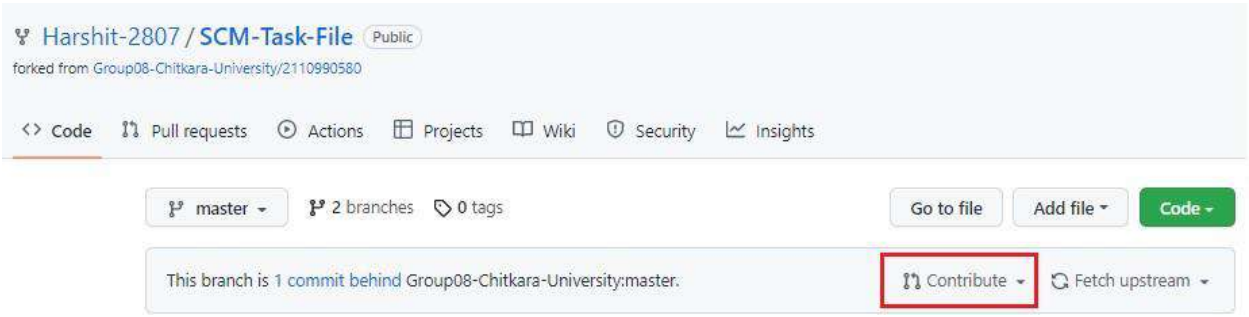
- After making new branch we add a file to the branch or make changes in the existing file.

- Add and commit the changes to the local repository.

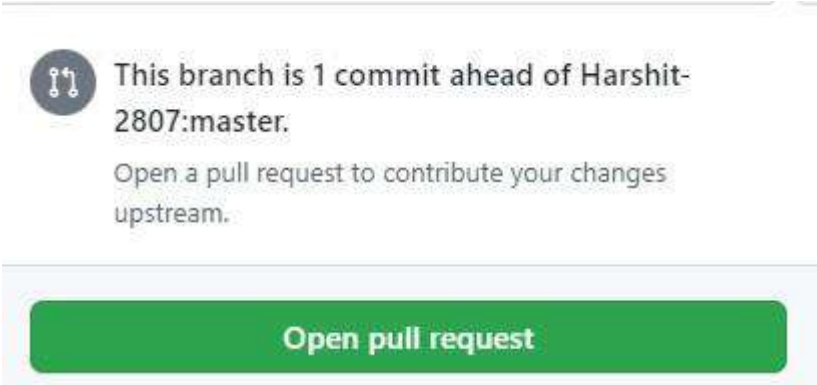
- Use git push origin *branch name* option to push the new branch to the main repository.



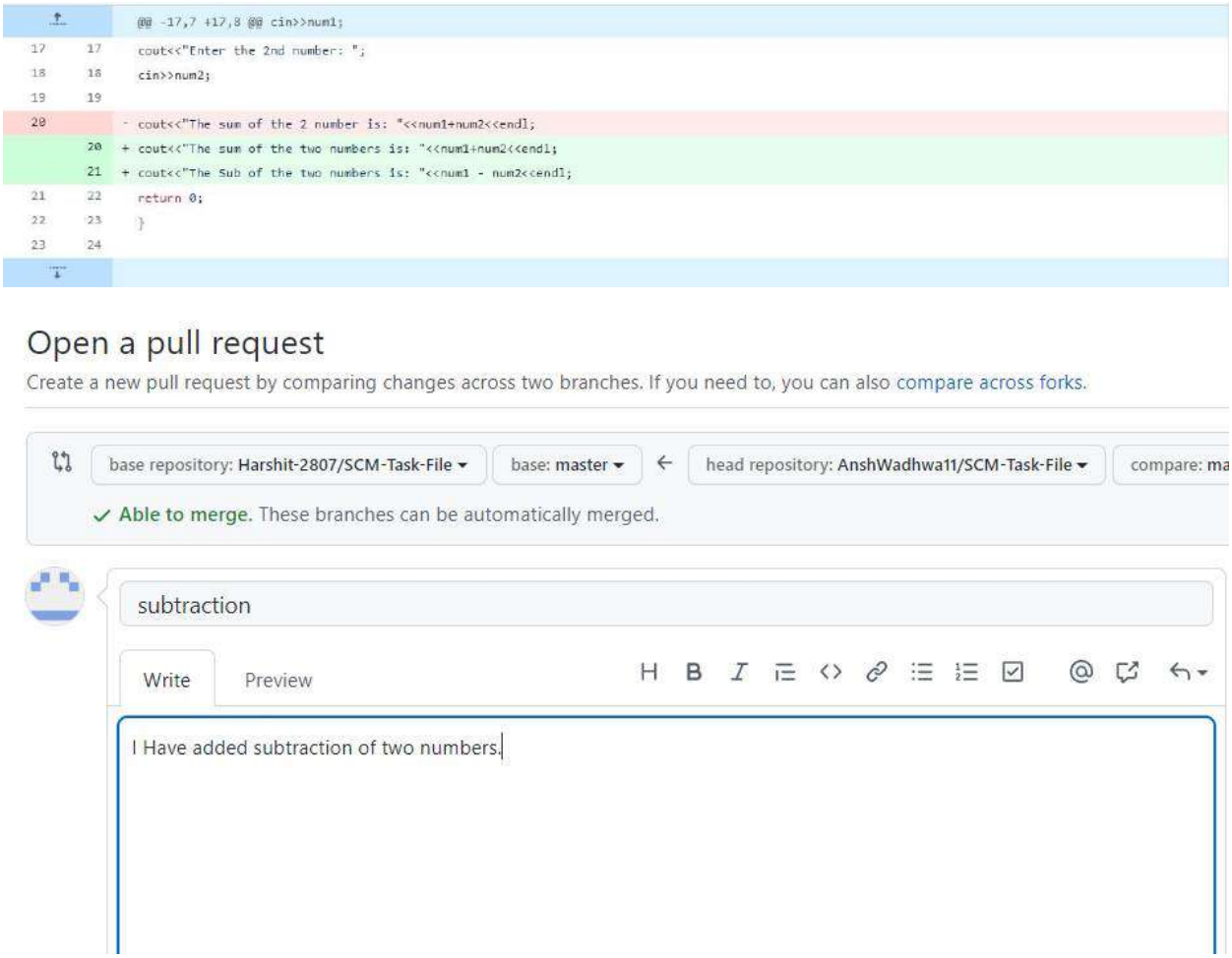
- After you have committed the changes in your forked repository, now you are ready to open a pull request to send changes in the original base file.
- You will be able to find a dialogue box above the list of files committed in your github.com page.



- After clicking on contribute dialogue box showing open pull request button will appear, click on open pull request.



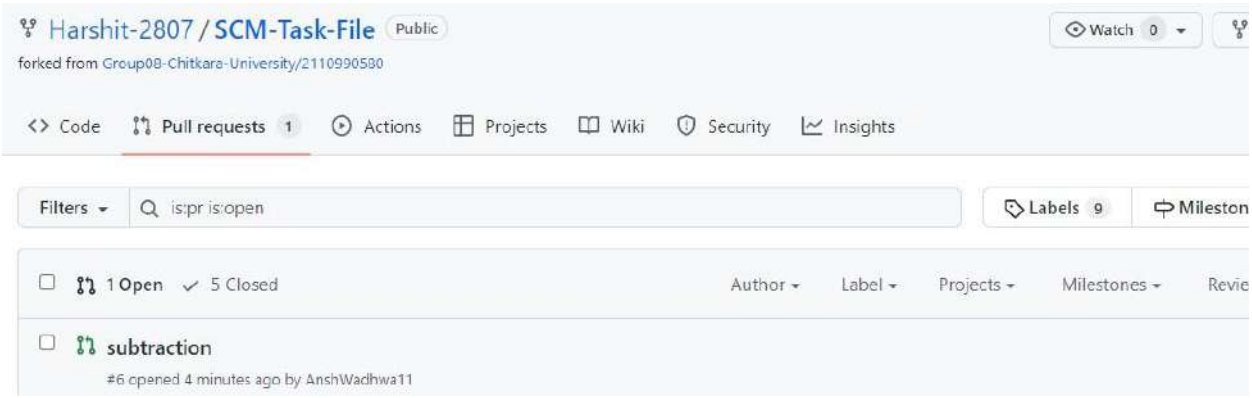
- Click again on create pull request, drop down will appear asking for description of changes you made, type description there and click pull request at bottom.



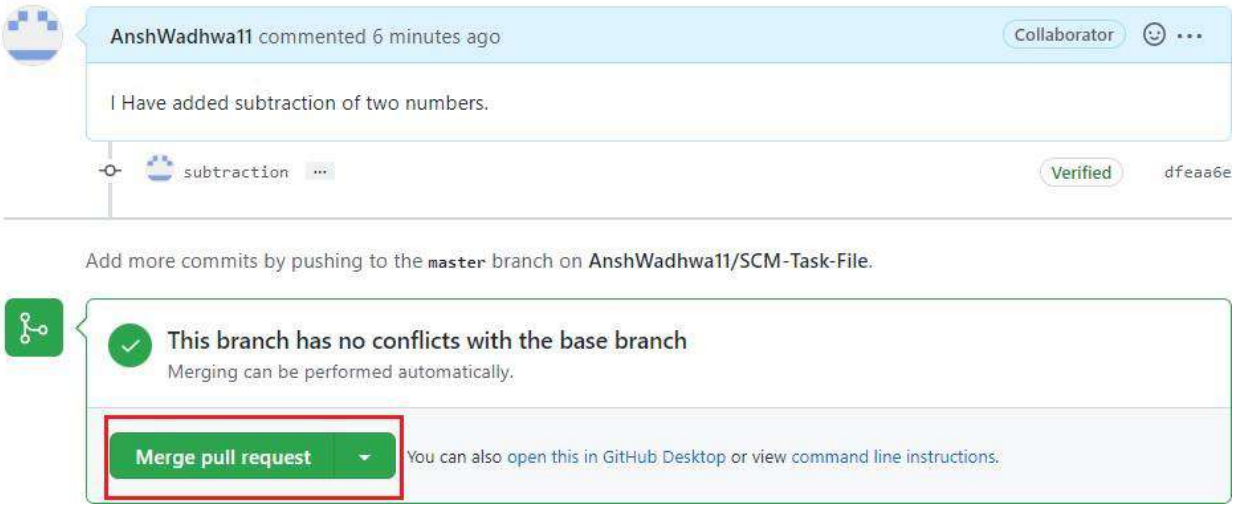
- Pull request is opened successfully, now the owner chooses either to merge or delete the request.

Closing Pull Request:

- Open your repository and click on “Pull Requests” on the repo navigation bar.



- Click on the name of the Pull Requests to either merge or delete the request. There you can find Merge Pull Request button with drop down button beside it, using it you can if you wish to delete the request.



- After clicking on Merge pull request button confirmation will be asked there you just need to confirm by clicking “Confirm Merge Button”.

Experiment No. 12

Aim: Create a pull request on a team member's repo and close pull requests generated by team members on own Repo as a maintainer.

Do the required changes in the repository, add and commit these changes in the local repository in a new branch.

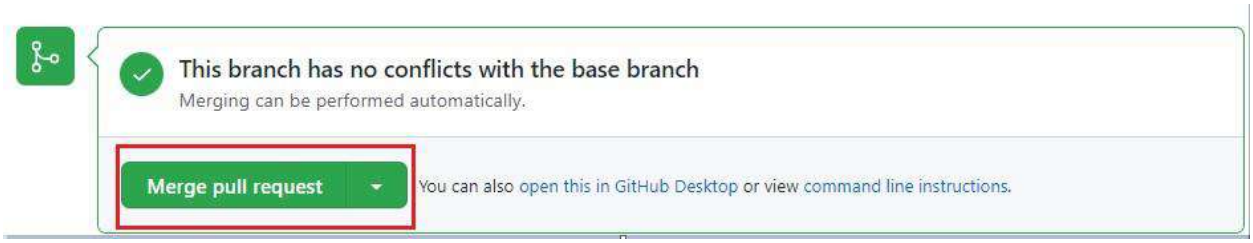
- Push the modified branch using git push origin *branchname*.
- Open a pull request by following the procedure from the above experiment.
- The pull request will be created and will be visible to all the team members.
- Ask your team member to login to his/her Github account.
- They will notice a new notification in the pull request menu.

To create a pull request on a team member's repository and close requests by any other team members as a maintainer follow the procedure given below:-

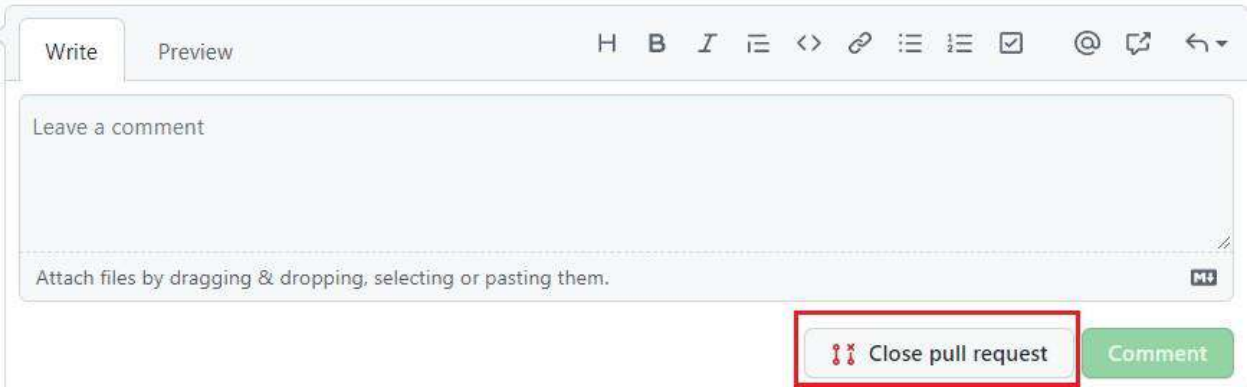
- Click on it. The pull request generated by you will be visible to them.

- Click on the pull request. Two option will be available, either to close the pull request or Merge the request with the main branch.

- By selecting the merge branch option the main branch will get updated for all the team members.

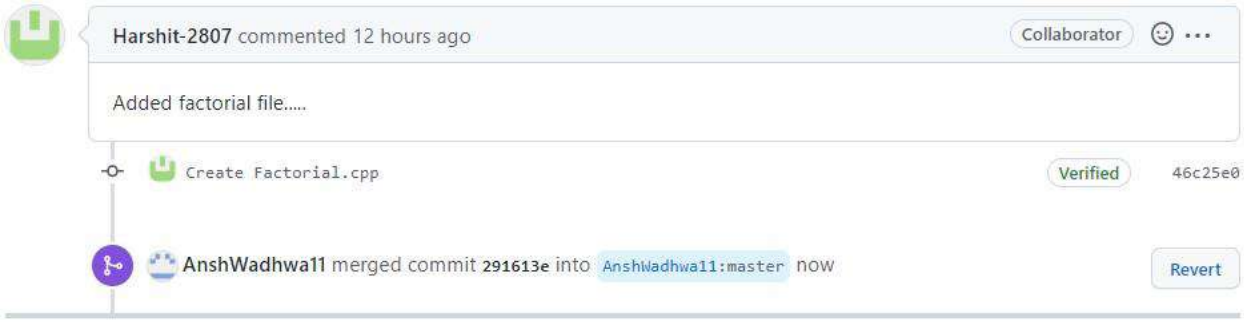


- By selecting close the pull request the pull request is not accepted and not merged with main branch.



- The process is similar to closing and merging the pull request by you. It simply includes an external party to execute.

- The result of merging the pull request is shown below



- Thus, we conclude opening and closing of pull request. We also conclude merging of the pull request to the main branch.

Experiment No. 13

Aim: Publish and print network graphs.

The network graph is one of the useful features for developers on GitHub. It is used to display the branch history of the entire repository network, including branches of the root repository and branches of forks that contain commits unique to the network.

A repository's graphs give you information on traffic, projects that depend on the repository, contributors and commits to the repository, and a repository's forks and network. If you maintain a repository, you can use this data to get a better understanding of who's using your repository and why they're using it.

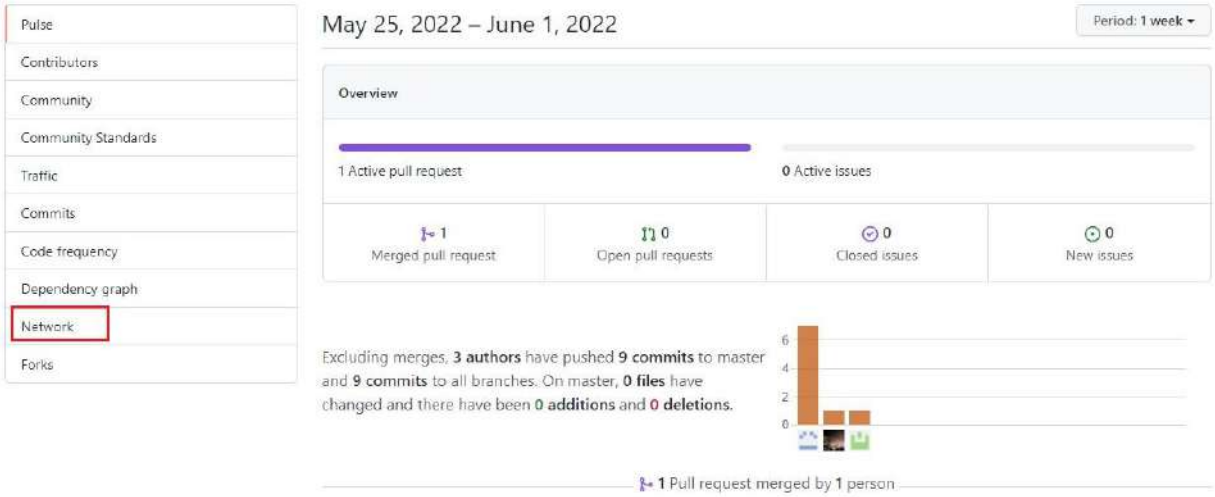
Some repository graphs are available only in public repositories with GitHub Free:

- Pulse
- Contributors
- Traffic
- Commits
- Code frequency
- Network

Steps to access network graphs of respective repository:

1. On GitHub.com, navigate to the main page of the repository.
2. Under your repository name, click **Insights**.

3. At the left sidebar, click on **Network.**



You will get the network graph of your repository which displays the branch history of the entire repository network, including branches of the root repository and branches of forks that contain commits unique to the network.