4/9/2022

SCM TASK 1

BY-

ABHISHEK KUMAR

ROLL NO. 2110990065

About Version Control:--

Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later. Version control, also known as source control, is the practice of tracking and managing changes to software code. Version control systems are software tools that help software teams manage changes to source code over time.

benefits of version control systems:-

- **Generate Backups:** Creating a backup of the current version of that repository is perhaps the most important benefit of using a version control system. Having numerous backups on various machines is beneficial because it protects data from being lost in the event of a server failure.
- **Experiment:** When a team works on a software project, they frequently use clones of the main project to build new features, test them, and ensure that they work properly before adding them to the main project. This could save time as different portions of the code can be created simultaneously.
- **Keep History:** Keeping track of the changes in a code file would assist you and new contributors understand how a certain section of the code was created. How did it begin and evolve over time to get at its current state.
- **Collaboration:** One of the most important advantages of version control systems, particularly DVCS, is that it allowed us to participate to projects we enjoyed despite the fact that we were in separate countries

Local Version Control Systems

Many people's version-control method of choice is to copy files into another directory. This approach is very common because it is so simple, but it is also incredibly error prone. It is easy to forget which directory you're in and accidentally write to the wrong file or copy over files you don't mean to.

Centralized Version Control Systems

Centralized version control systems are based on the idea that there is a single “central” copy of your project somewhere (probably on a server), and programmers will “commit” their changes to this central copy. Committing a change simply means recording the change in the central system.

Main benefits (CVCS):

- Centralized systems are typically easier to understand and use
- You can grant access level control on directory level
- performs better with binary files

Distributed Version Control Systems

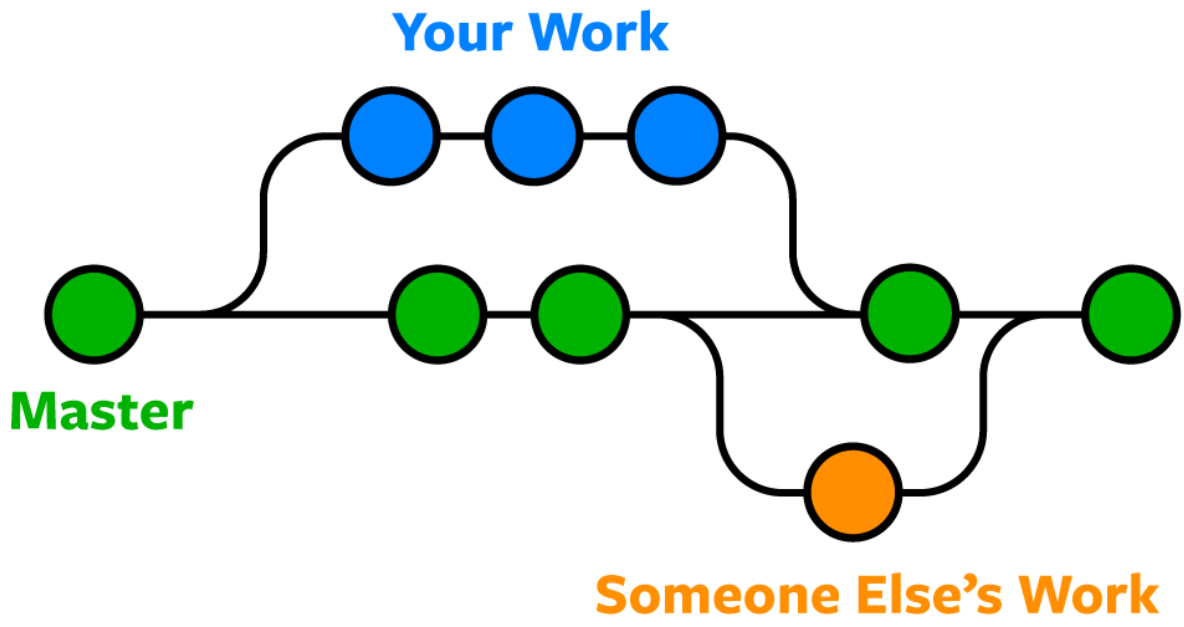
In distributed version control, every developer “clones” a copy of a repository and has the full history of the project on their own hard drive. This copy (or “clone”) has all of the metadata of the original.

Main benefits (DVCS):

- Performance of distributed systems is better
- Branching and merging is much easier
- With a distributed system, you don’t need to be connected to the network all the time (complete code repository is stored locally on PC)

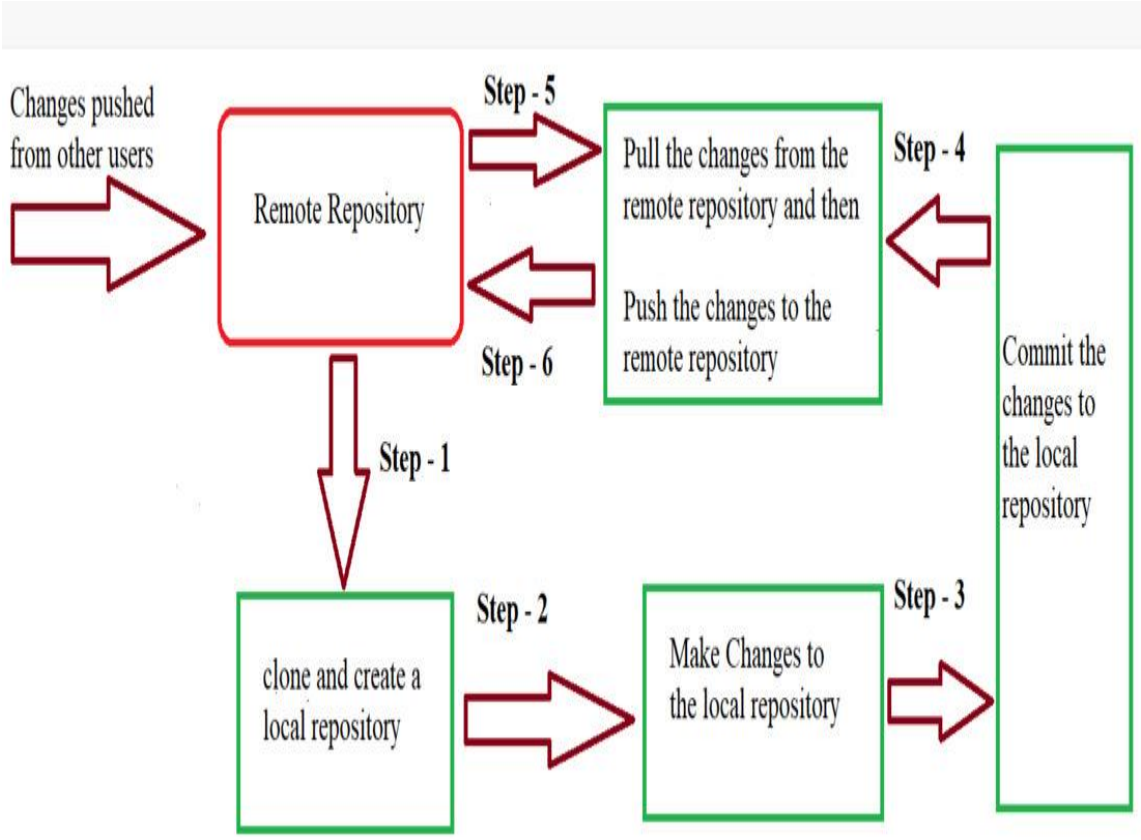
What is Git?

Git is software for tracking changes in any set of files, usually used for coordinating work among programmers collaboratively developing source code during software development.



Git Life Cycle:-

Git is used in our day-to-day work, we use git for keeping a track of our files, working in a collaboration with our team, to go back to our previous code versions if we face some error. Git helps us in many ways. Let us look at the Life Cycle that git has and understand more about its life cycle. Let us see some of the basic steps that we follow while working with Git



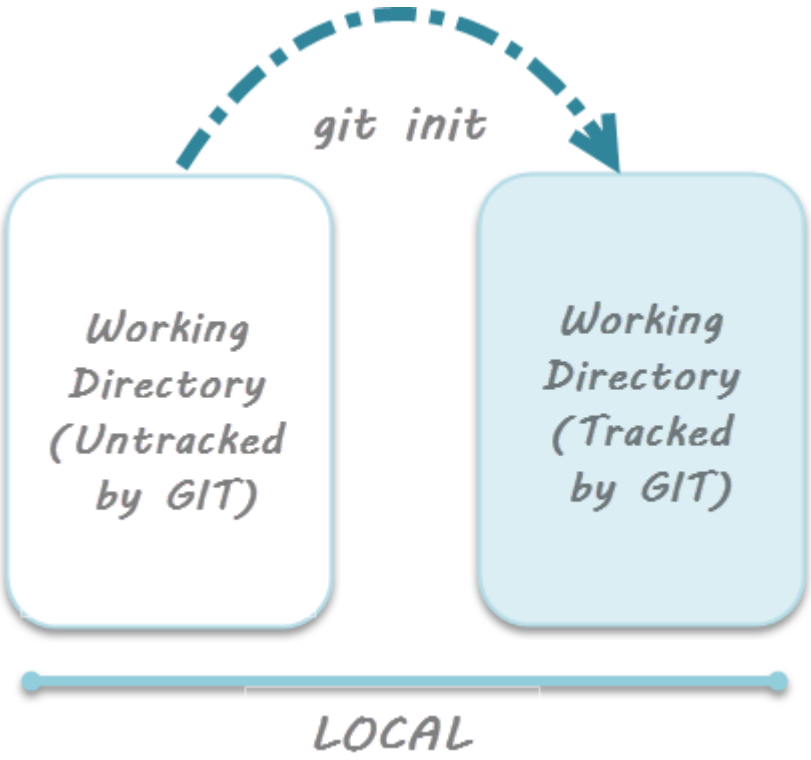
- ***In Step – 1****, We first clone any of the code residing in the remote repository to mak e our won local repository.*
- ***In Step-2*** *we edit the files that we have cloned in our local repository and make th e necessary changes in it.*
- ***In Step-3*** *we commit our changes by first adding them to our staging area and co mmitting them with a commit message.*
- ***In Step – 4 and Step-5*** *we first check whether there are any of the changes done in the remote repository by some other users and we first pull that changes.*
- *If there are no changes we directly proceed with* ***Step – 6*** *in which we push our ch anges to the remote repository and we are done with our work.*

When a directory is made a git repository, there are mainly 3 states which make the essence of Git Version Control System. The three states are-

- Working directory
- Staging area
- Git directory

Working Directory

Whenever we want to initialize our local project directory to make it a git repository, we use the ***git init*** command. After this command, git becomes aware of the files in the project although it doesn't track the files yet. The files are further tracked in the staging area.
git init



Staging Area

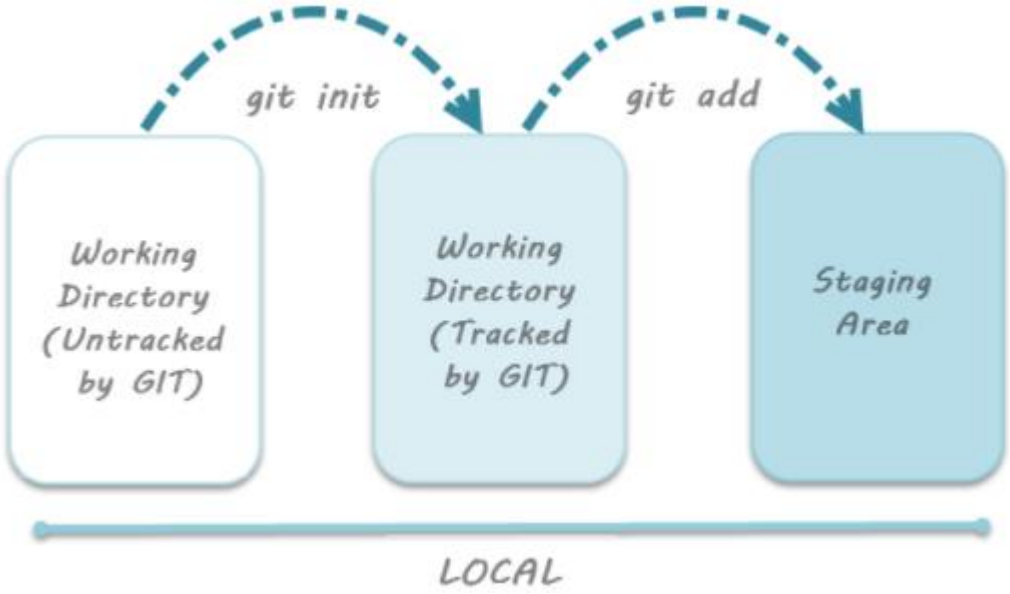
Now, to track the different versions of our files we use the command ***git add***. We can term a staging area as a place where different versions of our files are stored. ***git add*** command copies the version of your file from your working directory to the staging area. We can, however, choose which files we need to add to the staging area because in our working directory there are some files that we don't want to get tracked, examples include node modules, env files, temporary files, etc. Indexing in Git is the one that helps Git in understanding which files need to be added or sent. You can find your staging area in the ***.git*** folder inside the ***index*** file.

// to specify which file to add to the staging area

git add <filename>

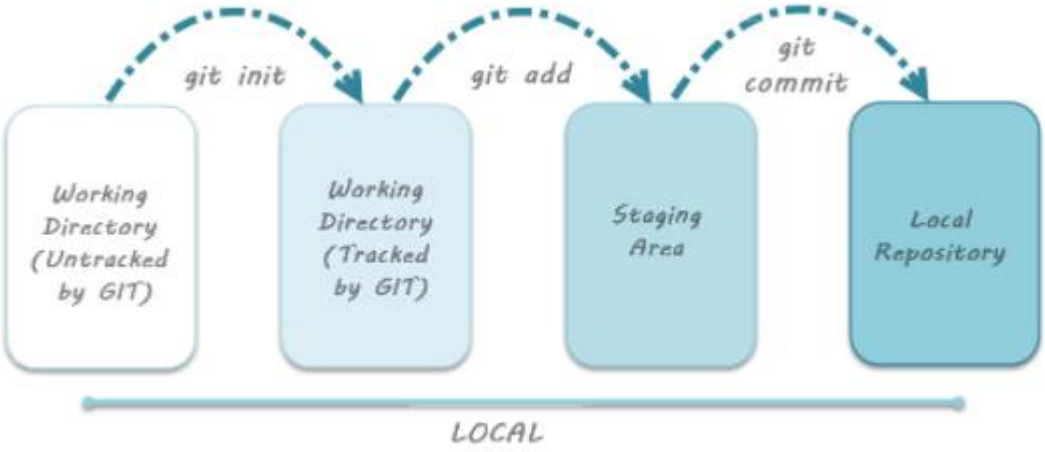
// to add all files of the working directory to the staging area

git add .



Git Directory

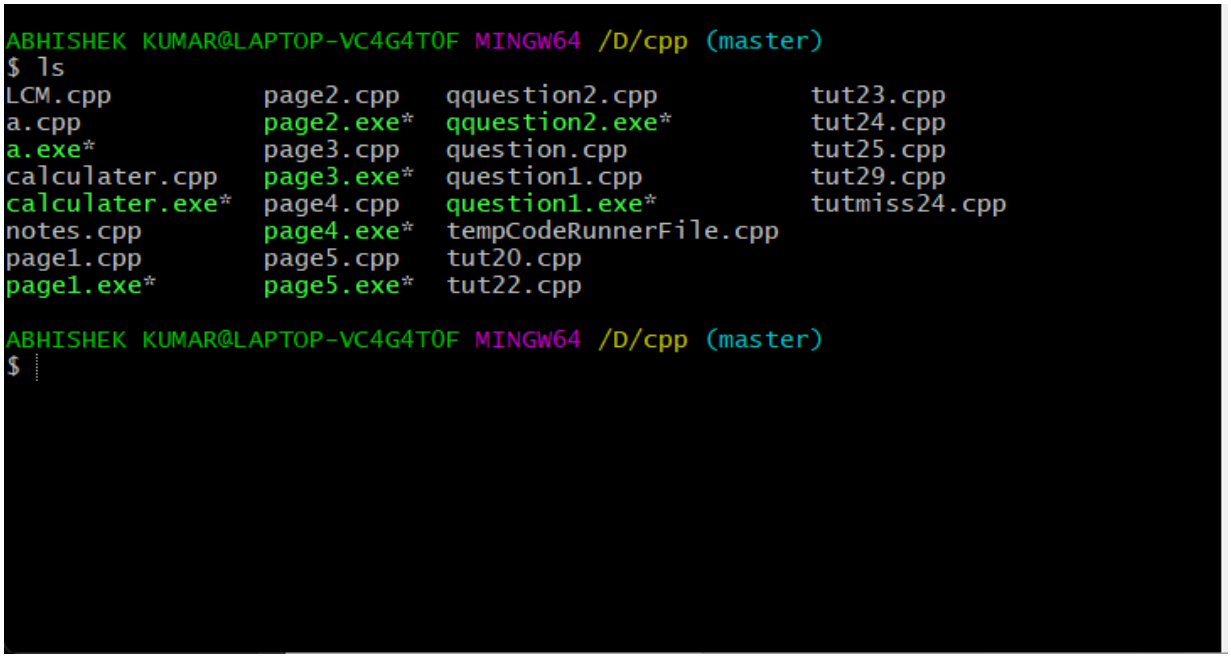
Now since we have all the files that are to be tracked and are ready in the staging area, we are ready to commit our files using the ***git commit*** command. Commit helps us in keeping the track of the metadata of the files in our staging area. We specify every commit with a message which tells what the commit is about. Git preserves the information or the metadata of the files that were committed in a Git Directory which helps Git in tracking files and basically it preserves the photocopy of the committed files. Commit also stores the name of the author who did the commit, files that are committed, and the date at which they are committed along with the commit message.
git commit -m <Commit Message>



COMMAND

ls

The ls command lists the current directory contents and by default will not show hidden files. If you pass it the -a flag, it will display hidden files. You can navigate into the . git directory like any other normal directory.



pwd

The Bash command pwd is used to print the 'present working directory'

```
ABHISHEK KUMAR@LAPTOP-VC4G4T0F MINGW64 /D/cpp (master)
$ pwd
/D/cpp

ABHISHEK KUMAR@LAPTOP-VC4G4T0F MINGW64 /D/cpp (master)
$
```

```
git init
```

The `git init` command is used to generate a new, empty Git repository or to reinitialize an existing one. With the help of this command, a .git subdirectory is created, which includes the metadata, like subdirectories for objects and template files, needed for generating a new Git repository.

```
ABHISHEK KUMAR@LAPTOP-VC4G4T0F MINGW64 /D/cpp (master)
$ git init
Reinitialized existing Git repository in D:/cpp/.git/

ABHISHEK KUMAR@LAPTOP-VC4G4T0F MINGW64 /D/cpp (master)
$
```

```
git clone <url>
```

The git clone command is used to target an existing repository and create a clone, or copy of the target repository.

```
ABHISHEK KUMAR@LAPTOP-VC4G4TOF MINGW64 /D/cpp (master)
$ git clone <url>
bash: syntax error near unexpected token `newline'

ABHISHEK KUMAR@LAPTOP-VC4G4TOF MINGW64 /D/cpp (master)
$ git clone https://github.com/abhi-srk/c-code.git
Cloning into 'c-code'...
remote: Repository not found.
fatal: repository 'https://github.com/abhi-srk/c-code.git/' not found

ABHISHEK KUMAR@LAPTOP-VC4G4TOF MINGW64 /D/cpp (master)
$
```

git status

The git status command displays the state of the working directory and the staging area

```
ABHISHEK KUMAR@LAPTOP-VC4G4TOF MINGW64 /D/cpp (master)
$ git status
On branch master
Your branch is up to date with 'origin/master'.

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

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

ABHISHEK KUMAR@LAPTOP-VC4G4TOF MINGW64 /D/cpp (master)
$
```

git add --a

The git add --a command is used to add file contents to the Index (Staging Area). This command updates the current content of the working tree to the staging area. It also prepares the staged content for the next commit.

```
ABHISHEK KUMAR@LAPTOP-VC4G4TOF MINGW64 /D/cpp (master)
$ git add --a

ABHISHEK KUMAR@LAPTOP-VC4G4TOF MINGW64 /D/cpp (master)
$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   LCM.cpp


ABHISHEK KUMAR@LAPTOP-VC4G4TOF MINGW64 /D/cpp (master)
$
```

```
git commit -m "message"
```

The git commit -m command captures a snapshot of the project's currently staged changes. Committed snapshots can be thought of as "safe" versions of a project

```
ABHISHEK KUMAR@LAPTOP-VC4G4TOF MINGW64 /D/cpp (master)
$ git commit -m "message"
[master 936538d] message
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 LCM.cpp

ABHISHEK KUMAR@LAPTOP-VC4G4TOF MINGW64 /D/cpp (master)
$
```

```
git branch
```

The git branch command is used to List all of the branches in your repository.

```
ABHISHEK KUMAR@LAPTOP-VC4G4TOF MINGW64 /D/cpp (master)
$ git branch
* master
```

```
git branch <branch>
```

The git branch <branch> command is used to Create a new branch called ＜branch＞.

```
ABHISHEK KUMAR@LAPTOP-VC4G4TOF MINGW64 /D/cpp (master)
$ git branch new

ABHISHEK KUMAR@LAPTOP-VC4G4TOF MINGW64 /D/cpp (master)
$ git branch
* master
  new

ABHISHEK KUMAR@LAPTOP-VC4G4TOF MINGW64 /D/cpp (master)
$
```

```
git checkout <branch name>
```

The git checkout command is used to switch the currently active branch.

```
ABHISHEK KUMAR@LAPTOP-VC4G4TOF MINGW64 /D/cpp (master)
$ git checkout new
Switched to branch 'new'

ABHISHEK KUMAR@LAPTOP-VC4G4TOF MINGW64 /D/cpp (new)
$
```

```
git merge <branch name>
```

The git merge command is used to merge the branches.

```
ABHISHEK KUMAR@LAPTOP-VC4G4TOF MINGW64 /D/cpp (new)
$ git merge new
Already up to date.
```

```
git log
```

The git log command shows a list of all the commits made to a repository.

```
ABHISHEK KUMAR@LAPTOP-VC4G4T0F MINGW64 /D/cpp (new)
$ git log
commit 936538d117f29a6475731f20634f940e1405a6e4 (HEAD -> new, master)
Author: unknown <abhishekumar9896e@gmail.com>
Date:   Sat Apr 9 11:18:30 2022 +0530

    message

commit c236bb80d15a00d37c312e606c7faee965ba9f6c (origin/master)
Author: unknown <abhishekumar9896e@gmail.com>
Date:   Tue Mar 29 21:39:17 2022 +0530

    first commit

ABHISHEK KUMAR@LAPTOP-VC4G4T0F MINGW64 /D/cpp (new)
$
```

Source Code Management (CS181)

CSE Batch 2021

Submitted by:

Student Name: Abhishek kumar.

Roll no: 2110990065

Group No.: G01

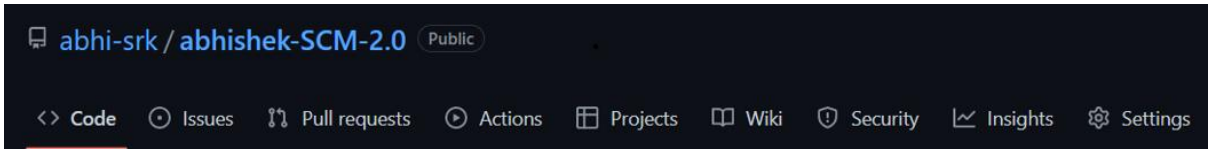
Add collaborators on GitHub Repo

What is GitHub Collaboration?
GitHub collaboration is a space where you can invite another developer to your repository and work together at an organization level, splitting the task and 2nd person will have the rights to add or merge files to the main repository without further permission.

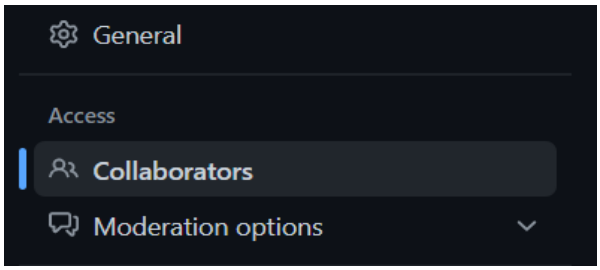
Let's invite a Collaborator
In this post, we will create a new repository and invite a collaborator to our repository by sending an invitation. A detailed procedure on how to invite a collaborator to your GitHub repository is mentioned below.

Step 1: On GitHub.com, navigate to the main page of the repository.

Step 2: Under your repository name, click **Settings**.

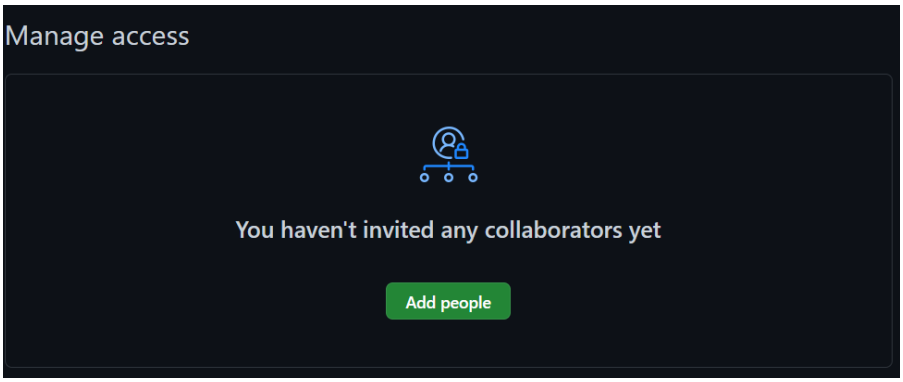


Step 3: In the "Access" section of the sidebar, click **Collaborators & teams**.



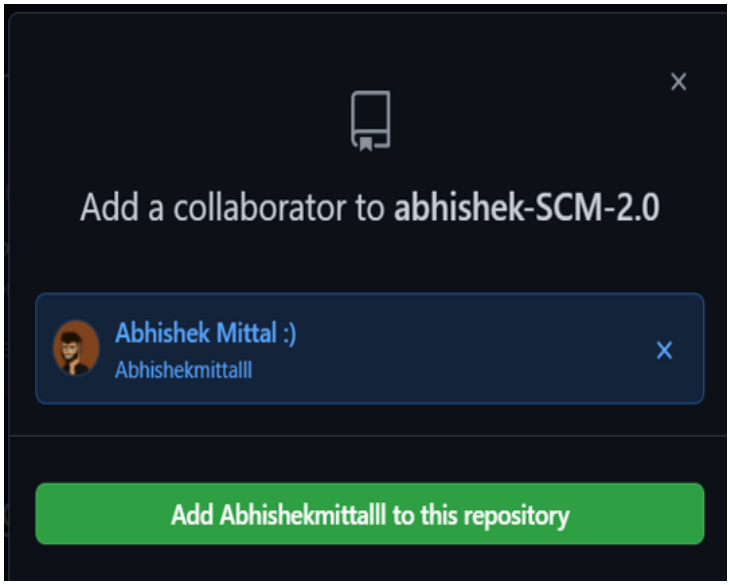
Step 4: Enter your GitHub account password for confirmation.

Step 5: Click on **Add people**.

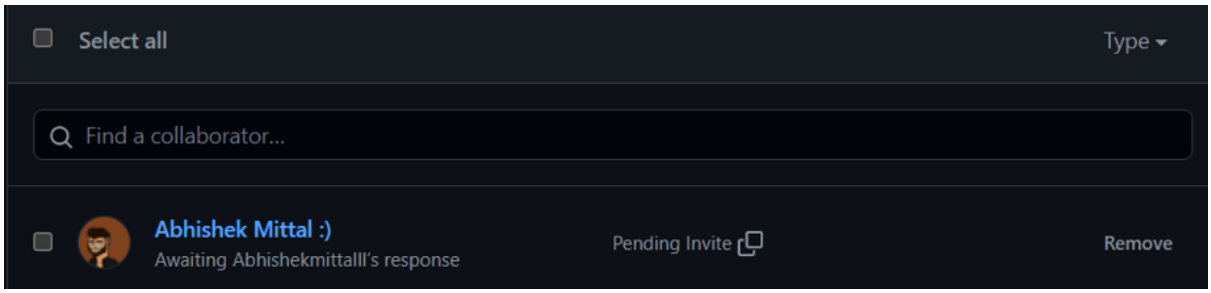


Step 6: In the search field, start typing the name of person you want to invite, then click a name in the list of matches.

Step 7: Click **Add NAME to REPOSITORY**.



Step 8: The user will receive an email inviting them to the repository. Once they accept your invitation, they will have collaborator access to your repository.



How to delete a collaborator from GitHub Repository?

Go to the Repository, Click on Manage access under settings tab, you can see a list of collaborators under Mange access, on right side of each listing there is a delete icon

Fork and Commit

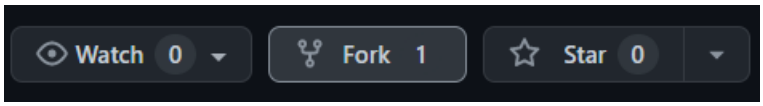
What is Forking in GitHub?

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.

Forking a repository

Step 1: On GitHub.com, navigate to the repository you wish to fork.

Step 2: In the top-right corner of the page, click **Fork**.

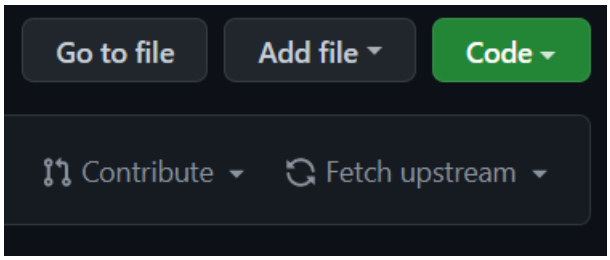


Step 3: Now go to your profile and see the latest forked repository in your repository tab.

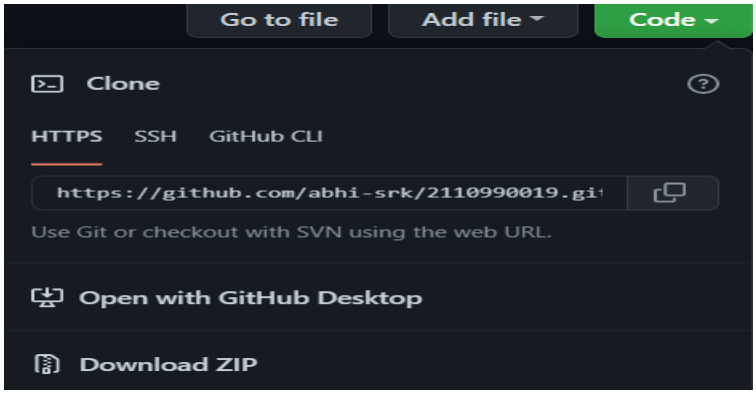
Cloning forked repository

Step 1: On GitHub.com, navigate to forked repository.

Step 2: Above the list of files, click **Code**.



Step 3: To clone the repository using HTTPS, under "Clone with HTTPS", copy the link.



Step 4: Open Git Bash.

Step 5: Change the working directory to the location where you want to clone the repo.

Step 6: Type **git clone**, and then paste the URL you copied earlier. It will look like this, with your GitHub username instead of YOUR-USERNAME:

$ git clone https://github.com/abhi-srk/2110990019.git

Step 7: Press Enter. Your local clone will be created.

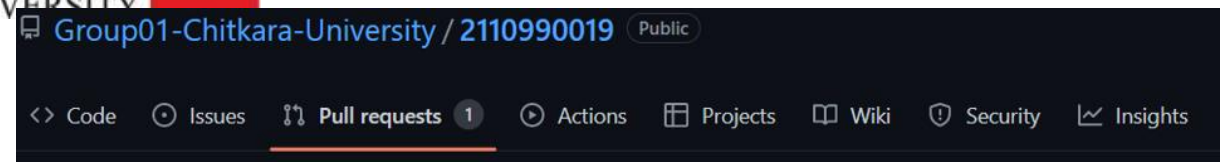
```
ABHISHEK KUMAR@LAPTOP-VC4G4T0F MINGW64 ~/OneDrive/Desktop
$ git clone https://github.com/abhi-srk/2110990019.git
Cloning into '2110990019'...
remote: Enumerating objects: 15, done.
remote: Counting objects: 100% (15/15), done.
remote: Compressing objects: 100% (8/8), done.
remote: Total 15 (delta 6), reused 12 (delta 6), pack-reused 0
Receiving objects: 100% (15/15), 1.45 MiB | 99.00 KiB/s, done.
Resolving deltas: 100% (6/6), done.
```

Creating a pull request

Step 1: Make changes to the file and it's time to stage and commit the file and push.

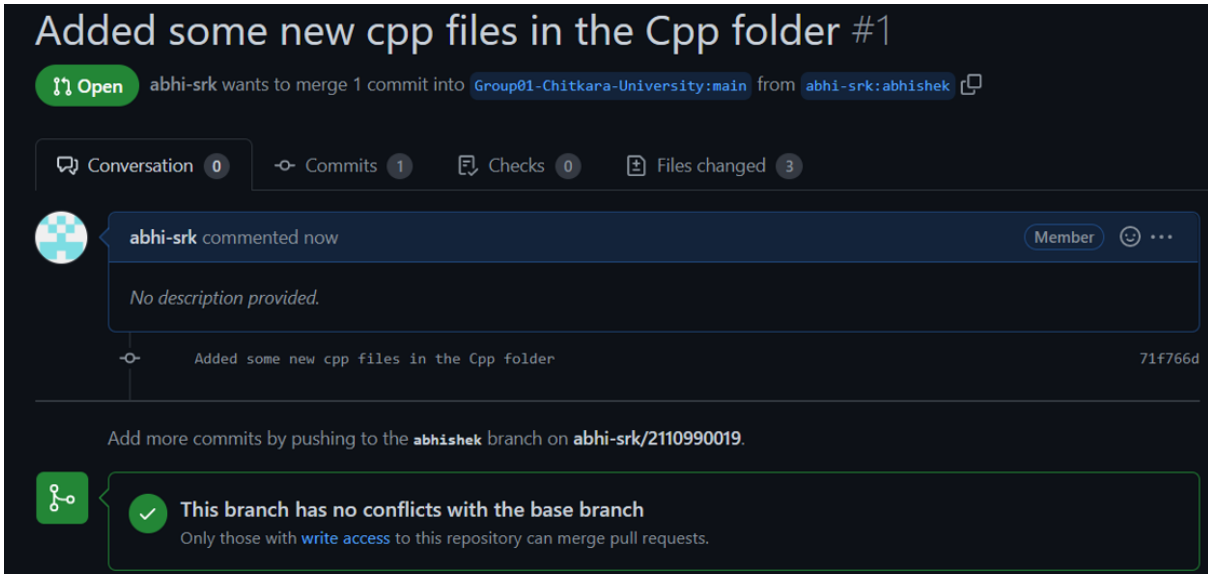
```
ABHISHEK KUMAR@LAPTOP-VC4G4T0F MINGW64 ~/OneDrive/Desktop/2110990019 (abhishek)
$ git status
On branch abhishek
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        CPP-Prog/BookAllocation.cpp
        CPP-Prog/FindPeak.cpp
        CPP-Prog/bubblesort.cpp

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

ABHISHEK KUMAR@LAPTOP-VC4G4T0F MINGW64 ~/OneDrive/Desktop/2110990019 (abhishek)
$ git add .

ABHISHEK KUMAR@LAPTOP-VC4G4T0F MINGW64 ~/OneDrive/Desktop/2110990019 (abhishek)
$ git commit -m "Added some new cpp files in the Cpp folder"
[abhishek 71f766d] Added some new cpp files in the Cpp folder
 3 files changed, 118 insertions(+)
 create mode 100644 CPP-Prog/BookAllocation.cpp
 create mode 100644 CPP-Prog/FindPeak.cpp
 create mode 100644 CPP-Prog/bubblesort.cpp

ABHISHEK KUMAR@LAPTOP-VC4G4T0F MINGW64 ~/OneDrive/Desktop/2110990019 (abhishek)
$ git push origin abhishek
Enumerating objects: 8, done.
Counting objects: 100% (8/8), done.
Delta compression using up to 8 threads
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 1.50 KiB | 1.50 MiB/s, done.
Total 6 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
remote:
remote: Create a pull request for 'abhishek' on GitHub by visiting:
remote:      https://github.com/abhi-srk/2110990019/pull/new/abhishek
remote:
To https://github.com/abhi-srk/2110990019.git
 * [new branch]      abhishek -> abhishek
```

Step 2: Now on the main branch, go to **Pull requests** section and click on **New pull request** button.



Step 3: Make sure to add a good comment in the merge and explain what's your commit is all about.

Step 4: In few Organisations, there will be Github actions scheduled for auto-check and any one of the code reviewers will review the file and approve the changes or suggest any issue with your pull request.



Merge and Resolve conflicts due to own activity and collaborators activity

Understanding GitHub Conflicts

Let's imagine a case where two developers working on one repository by creating 2 different branch and same code line and both of them pulled the file to main repository, Now the GitHub is good understanding the code and merge the conflicts automatically but it fails understanding

if developer has put any comments. Now Let's look into How to resolve merge conflicts in GitHub.

Let's resolve merge conflict:

Step 1: Create a branch from the master/base branch and checkout that branch using **git checkout <branch name>** command.

Step 2: Modify the files you wished to, then stage that file and then commit.

Step 3: Now, checkout to base branch, make changes to files if you wish else merge the branch to base branch using "**git merge <branch to be merged>**" command.

```
ABHISHEK KUMAR@LAPTOP-VC4G4T0F MINGW64 ~/OneDrive/Desktop/2110990065/CPP Programs (master)
$ git merge abhishek
Auto-merging CPP Programs/bubblesort.cpp
CONFLICT (content): Merge conflict in CPP Programs/bubblesort.cpp
Automatic merge failed; fix conflicts and then commit the result.
```

Git hasn't automatically created a new merge commit. It has paused the process while you resolve the conflict.

Step 4: If you want to use a graphical tool to resolve these issues, you can run **git mergetool**, which fires up an appropriate visual merge tool and walks you through the conflicts

Step 5: After exiting merge tool, if you're happy with that, and you verify that everything that had conflicts has been staged, you can type **git commit** to finalize the merge commit.

```
ABHISHEK KUMAR@LAPTOP-VC4G4TOF MINGW64 ~/OneDrive/Desktop/2110990065/CPP Programs (master|MERGING)
$ git commit -m "Conflict resolved."
[master f2a30fa] Conflict resolved.

ABHISHEK KUMAR@LAPTOP-VC4G4TOF MINGW64 ~/OneDrive/Desktop/2110990065/CPP Programs (master)
$ git log --oneline
f2a30fa (HEAD -> master) Conflict resolved.
3a8fd91 Added comment in master branch.
e4386f0 (abhishek) Added comment in abhishek branch.
ffb5b9d (origin/master, origin/HEAD) Added cpp programs
1e1b3c3 first tast

ABHISHEK KUMAR@LAPTOP-VC4G4TOF MINGW64 ~/OneDrive/Desktop/2110990065/CPP Programs (master)
$
```

You can modify commit message with details about how you resolved the merge and explain why you did the changes you made if these are not obvious.

Reset and Revert

How to reset a Git commit

Let's start with the Git command reset. Practically, you can think of it as a "rollback"—it points your local environment back to a previous commit. By "local environment," we mean your local repository, staging area, and working directory.

Take a look at Figure 1. Here we have a representation of a series of commits in Git. A branch in Git is simply a named, movable pointer to a specific commit. In this case, our branch *master* is a pointer to the latest commit in the chain.

```
$ git log --oneline
b764644 File with three lines
7c709f0 File with two lines
9ef9173 File with one line
```

What happens if we want to roll back to a previous commit. Simple—we can just move the branch pointer. Git supplies the reset command to do this for us. For example, if we want to reset master to point to the commit two back from the current commit, we could use either of the following methods:

```
$ git reset 9ef9173 (using an absolute commit SHA1 value 9ef9173)
or
$ git reset current~2 (using a relative value -2 before the "current" tag)
```

Figure 2 shows the results of this operation. After this, if we execute a git log command on the current branch (master), we'll see just the one commit.

```
$ git log –oneline
9ef9173 File with one line
```

The git reset command also includes options to update the other parts of your local environment with the contents of the commit where you end up. These options include: hard to reset the commit being pointed to in the repository, populate the working directory with the contents of the commit, and reset the staging area; soft to only reset the pointer in the repository; and mixed (the default) to reset the pointer and the staging area.

Using these options can be useful in targeted circumstances such as git reset -- hard . This overwrites any local changes you haven't committed. In effect, it resets (clears out) the staging area and overwrites content in the working directory with the content from the commit you reset to. Before you use the hard option, be sure that's what you really want to do, since the command overwrites any uncommitted changes.

How to revert a Git commit

The net effect of the git revert command is similar to reset, but its approach is different. Where the reset command moves the branch pointer back in the chain (typically) to "undo" changes, the revert command adds a new commit at the end of the chain to "cancel" changes. The effect is most easily seen by looking at Figure 1 again. If we add a line to a file in each commit in the chain, one way to get back to the version with only two lines is to reset to that commit, i.e., *git reset HEAD~1*

Another way to end up with the two-line version is to add a new commit that has the third line removed—effectively cancelling out that change. This can be done with a git revert command, such as:

$ git revert HEAD

Because this adds a new commit, Git will prompt for the commit message:

```
Revert "File with three lines"

This reverts commit
b764644bad524b804577684bf74e7bca3117f554.

# 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:   file1.txt
#
```

Figure 3 (below) shows the result after the revert operation is completed.

If we do a git log now, we'll see a new commit that reflects the contents before the previous commit

```
$ git log --oneline
11b7712 Revert "File with three lines"
b764644 File with three lines
7c709f0 File with two lines
9ef9173 File with one line
```

Here are the current contents of the file in the working directory:

```
$ cat <filename>
Line 1
Line 2
```

Source Code Management

(CS181)

Submitted by

Team Member 1. Abhishek Kumar 2110990065.

Team Member 2. Abhishek Mittal 2110990067.

Team Member 3. Aaryan Singh 2110990019.

Department of Computer Science & Engineering

Chitkara University Institute of Engineering and Technology, Punjab

Jan- June

(2021-22)

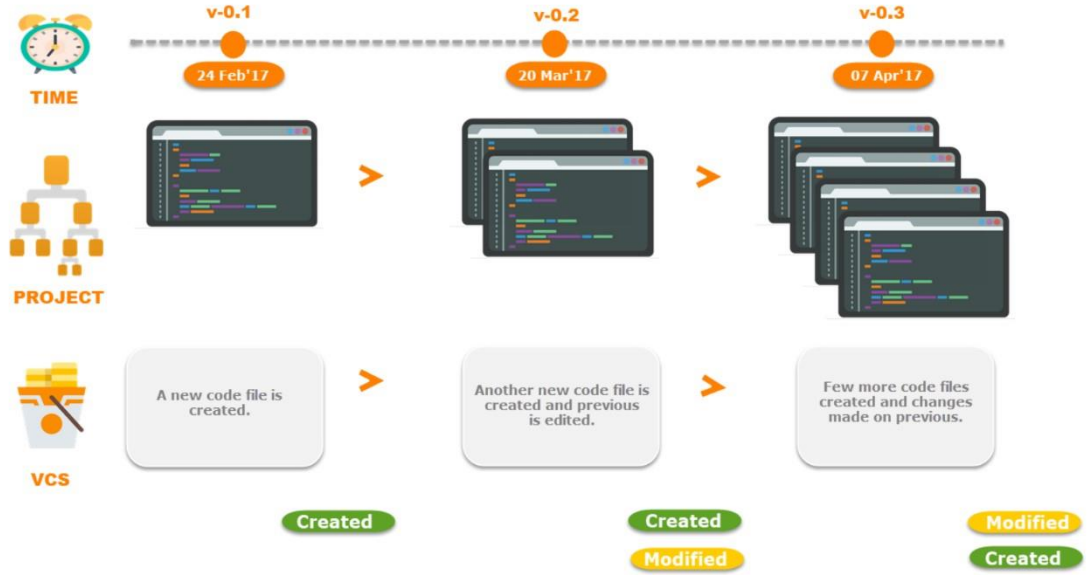
Institute/School Name	**Chitkara University Institute of Engineering and Technology**		
Department Name	**Department of Computer Science & Engineering**		
Programme Name	**Bachelor of Engineering (B.E.), Computer Science & Engineering**		
Course Name	**Source Code Management**	Session	**2021-22**
Course Code	**CS181**	Semester/Batch	**2nd/2021**
Vertical Name	**Beta**	Group No	G-01
Course Coordinator	**Dr. Monit Kapoor.**		

CHITKARA UNIVERSITY

Table of Content

1. Version control with Git

What is "version control", and why should you care? Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later. For the examples in this book, you will use software source code as the files being version controlled, though in reality you can do this with nearly any type of file on a computer. If you are a graphic or web designer and want to keep every version of an image or layout (which you would most certainly want to), a Version Control System (VCS) is a very wise thing to use. It allows you to revert selected files back to a previous state, revert the entire project back to a previous state, compare changes over time, see who last modified something that might be causing a problem, who introduced an issue and when, and more. Using a VCS also generally means that if you screw things up or lose files, you can easily recover. In addition, you get all this for very little overhead.



Local Version Control Systems

Many people's version-control method of choice is to copy files into another directory (perhaps a time-stamped directory, if they're clever). This approach is very common because it is so simple, but it is also incredibly error prone. It is easy to forget which directory you're in and accidentally write to the wrong file or copy over files you don't mean to. To deal with this issue, programmers long ago developed local VCSs that had a simple database that kept all the changes to files under revision control.

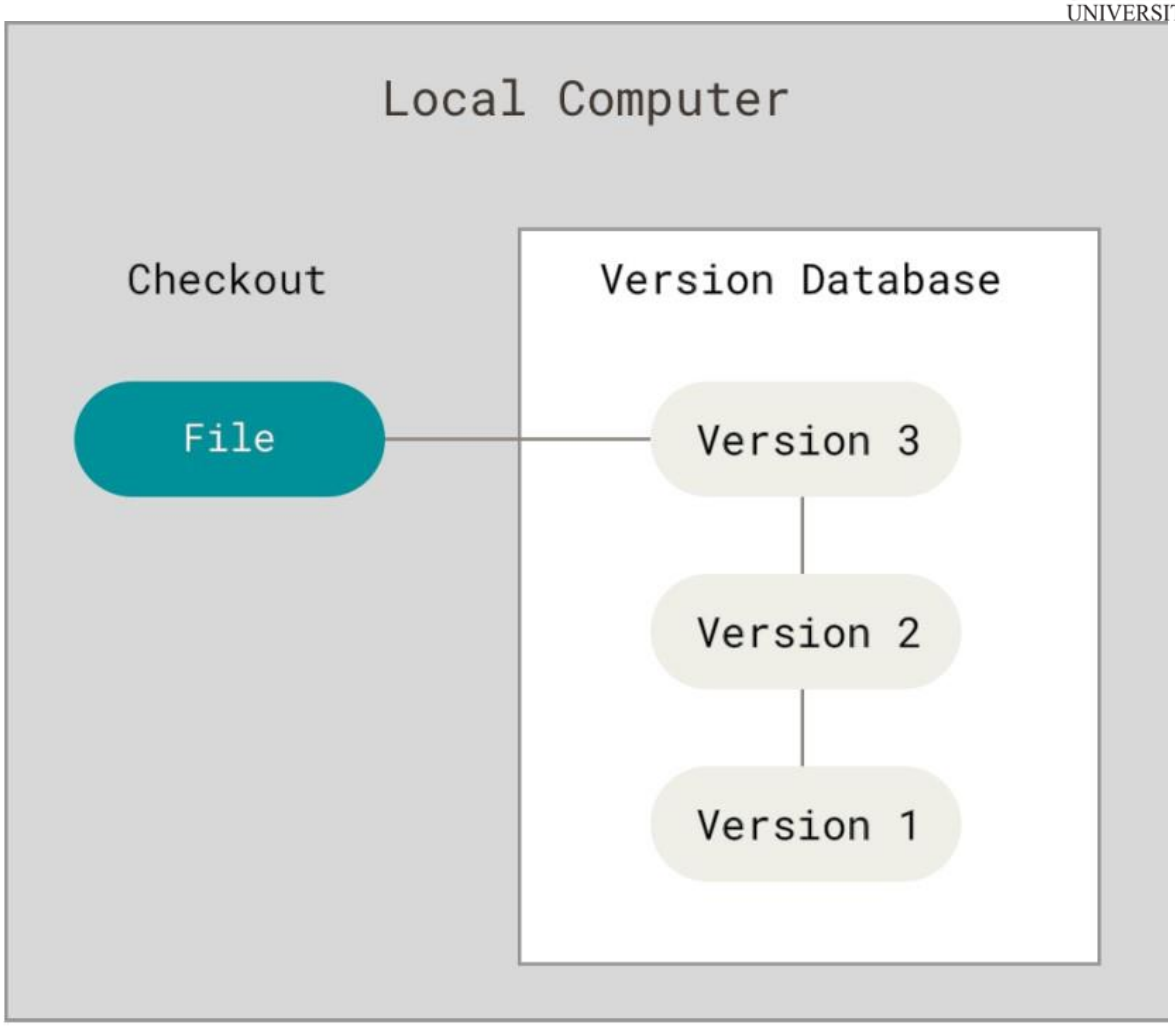


Figure 1. Local version control

One of the most popular VCS tools was a system called RCS, which is still distributed with many computers today. RCS works by keeping patch sets (that is, the differences between files) in a special format on disk; it can then re-create what any file looked like at any point in time by adding up all the patches.

Centralized Version Control Systems

The next major issue that people encounter is that they need to collaborate with developers on other systems. To deal with this problem, Centralized Version Control Systems (CVCSs) were developed. These systems (such as CVS, Subversion, and Perforce) have a single server that contains all the versioned files, and a number of clients that check out files from that central place. For many years, this has been the standard for version control.

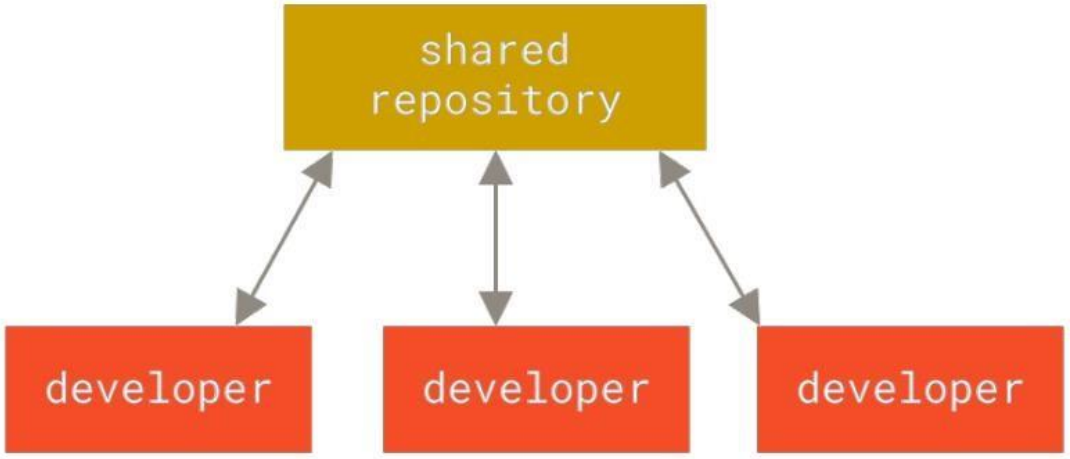


Figure 2. Centralized version control

This setup offers many advantages, especially over local VCSs. For example, everyone knows to a certain degree what everyone else on the project is doing. Administrators have fine-grained control over who can do what, and it's far easier to administer a CVCS than it is to deal with local databases on every client. However, this setup also has some serious downsides. The most obvious is the single point of failure that the centralized server represents. If that server goes down for an hour, then during that hour nobody can collaborate at all or save versioned changes to anything they're working on. If the hard disk the central database is on becomes corrupted, and proper backups haven't been kept, you lose absolutely everything — the entire history of the project except whatever single snapshots people happen to have on their local machines. Local VCSs suffer from this same problem — whenever you have the entire history of the project in a single place, you risk losing everything.

Distributed Version Control Systems

This is where Distributed Version Control Systems (DVCSs) step in. In a DVCS (such as Git, Mercurial, Bazaar or Darcs), clients don't just check out the latest snapshot of the files; rather, they fully mirror the repository, including its full history. Thus, if any server dies, and these systems were collaborating via that server, any of the client repositories can be copied back up to the server to restore it. Every clone is really a full backup of all the data.

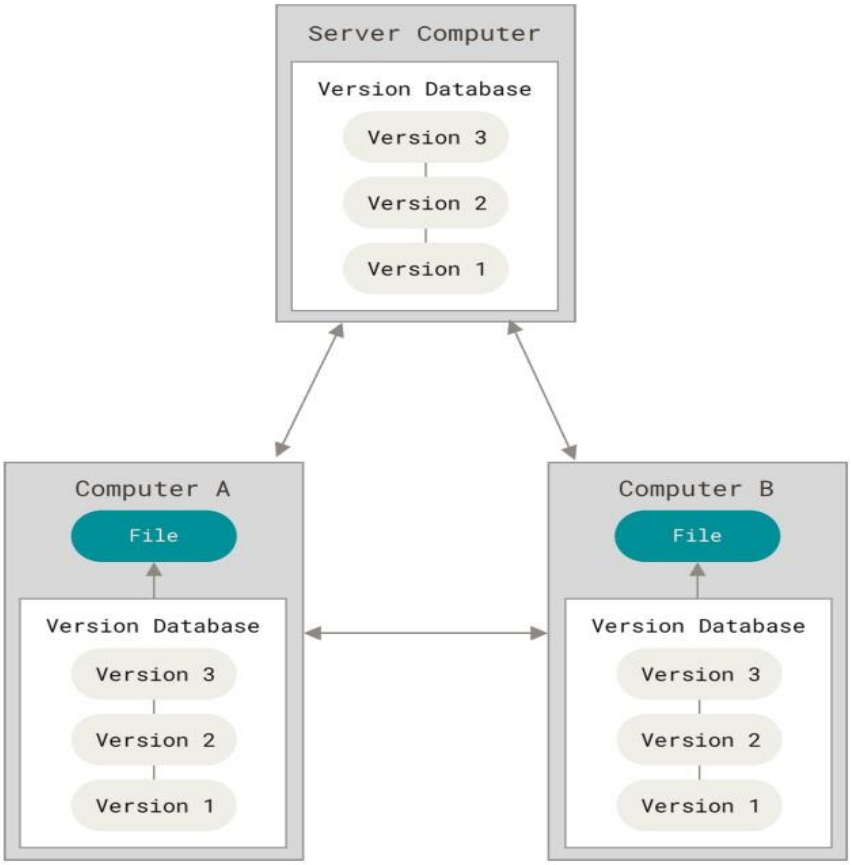


Figure 3. Distributed version control

Furthermore, many of these systems deal pretty well with having several remote repositories they can work with, so you can collaborate with different groups of people in different ways simultaneously within the same project. This allows you to set up several types of workflows that aren't possible in centralized systems, such as hierarchical models.

Why Do We Need It?

Complex code development is impossible without a Version Control System (VCS). Version Control is used to track and control changes to source code. It is an essential tool to ensure the integrity of the codebase.

Why is Version Control Essential?

- Easy Modification of Codebase
- Reverting Error
- Improves visibility.
- Accelerates product delivery.

Version control system keeps track on changes made on a particular software and take a snapshot of every modification .Let's suppose if a team of developer add some new functionalities in a application and the updated version is not working properly so as version control system keeps track of our work so with the help of version control system we can omit the new changes and continue with the previous version .

Benefits of the version control system:

•Enhances the project development speed by providing efficient collaboration,

•Leverages the productivity, expedites product delivery, and skills of the employees through better communication and assistance,

•Reduce possibilities of errors and conflicts meanwhile project development through traceability to every small change,

•Employees or contributors of the project can contribute from anywhere irrespective of the different geographical locations through this VCS,

•For each different contributor to the project, a different working copy is maintained and not merged to the main file unless the working copy is validated. The most popular example is Git, Helix core, Microsoft TFS,

•Helps in recovery in case of any disaster or contingent situation,

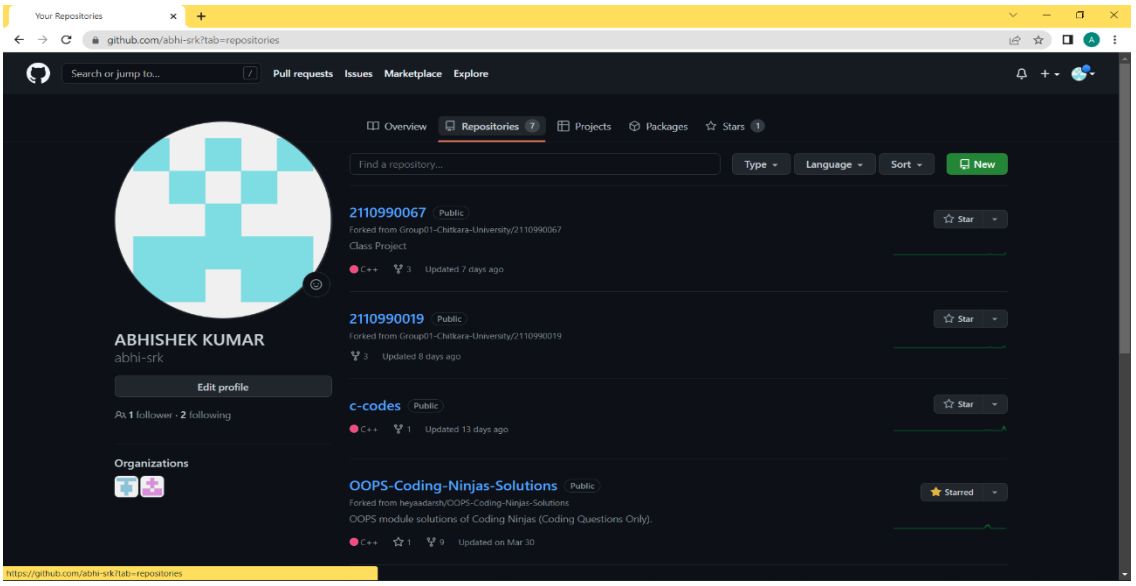
•Informs us about Who, What, When, Why changes have been made.

4. Concepts and Commands

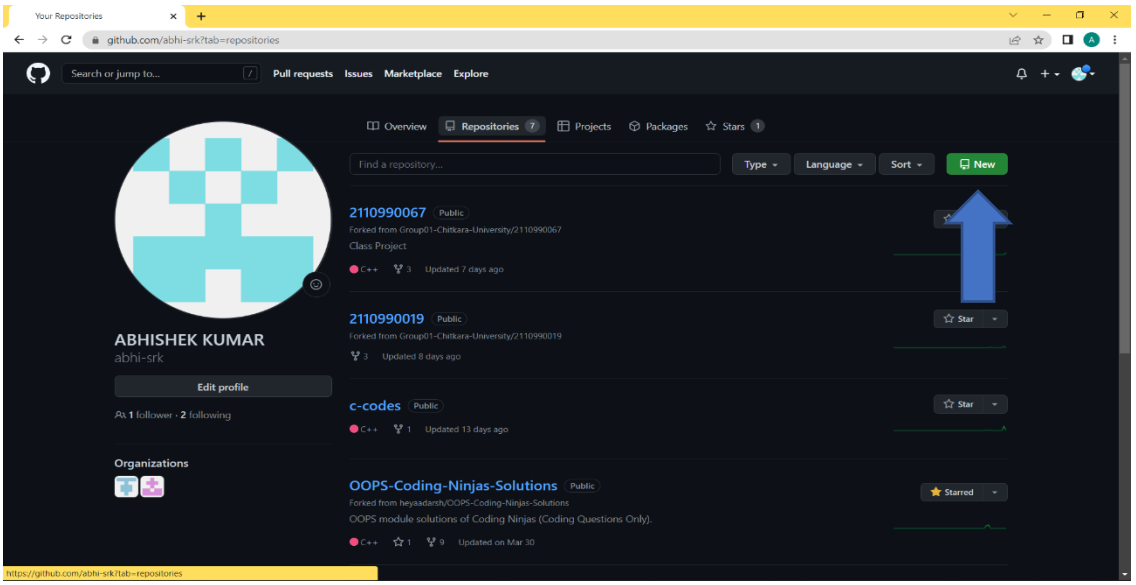
5.Workflow and Discussion.

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

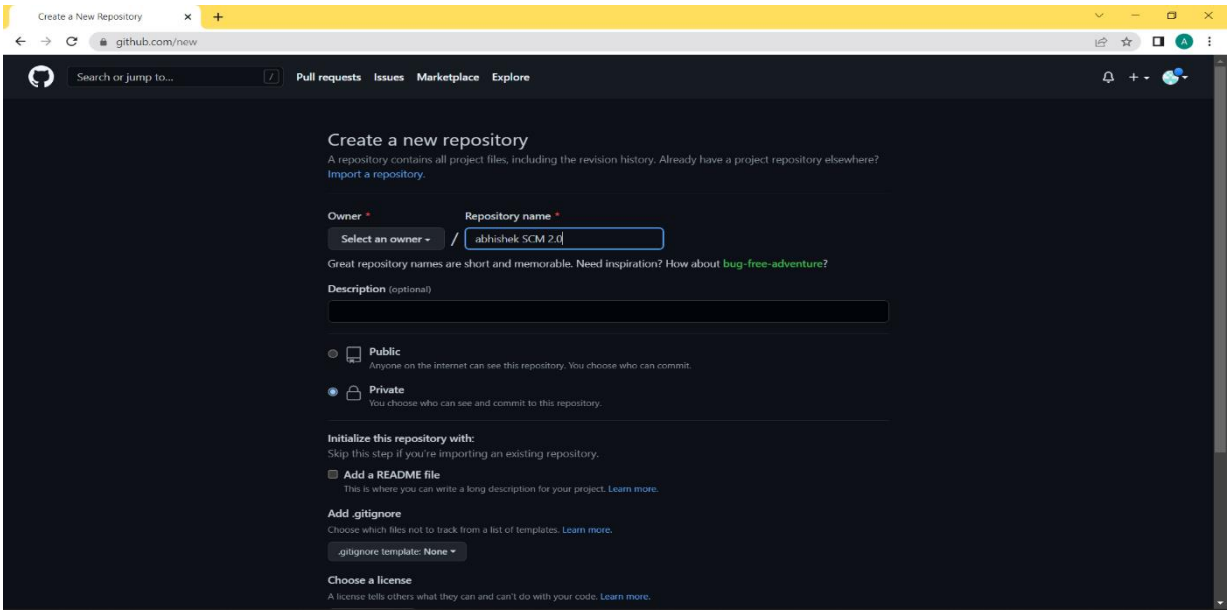
- Login to your GitHub account and you will land on the homepage as shown below. Click on Repositories option in the menu bar.



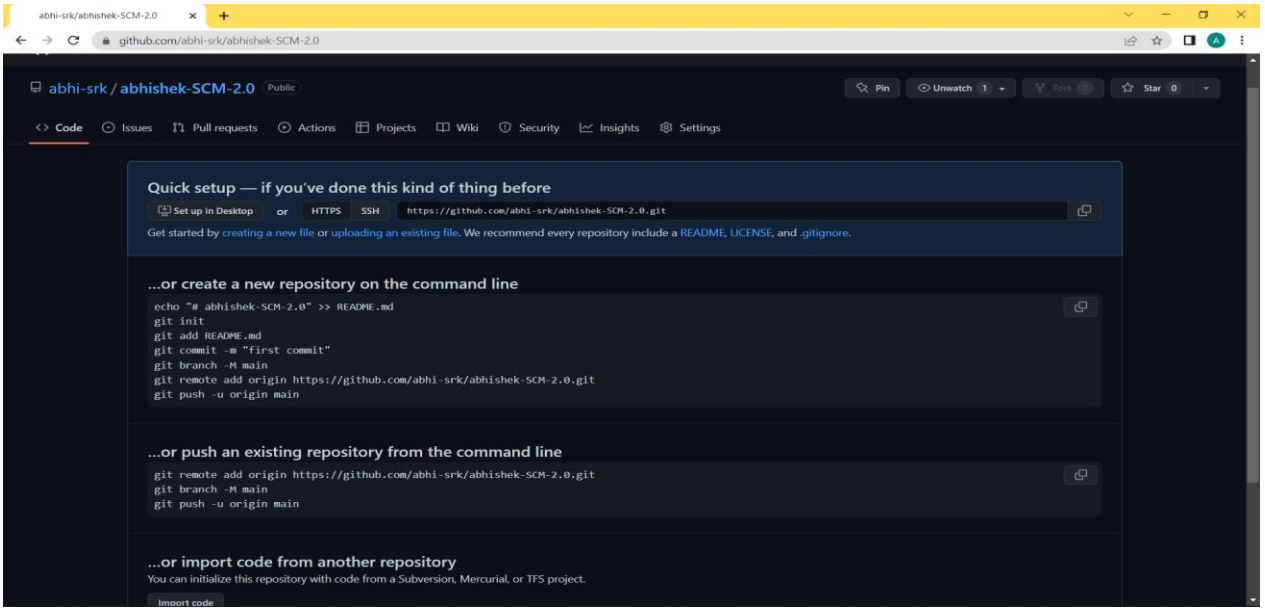
- Click on the ‘New’ button in the top right corner.



- Enter the Repository name and add the description of the repository.
- Select if you want the repository to be public or private.

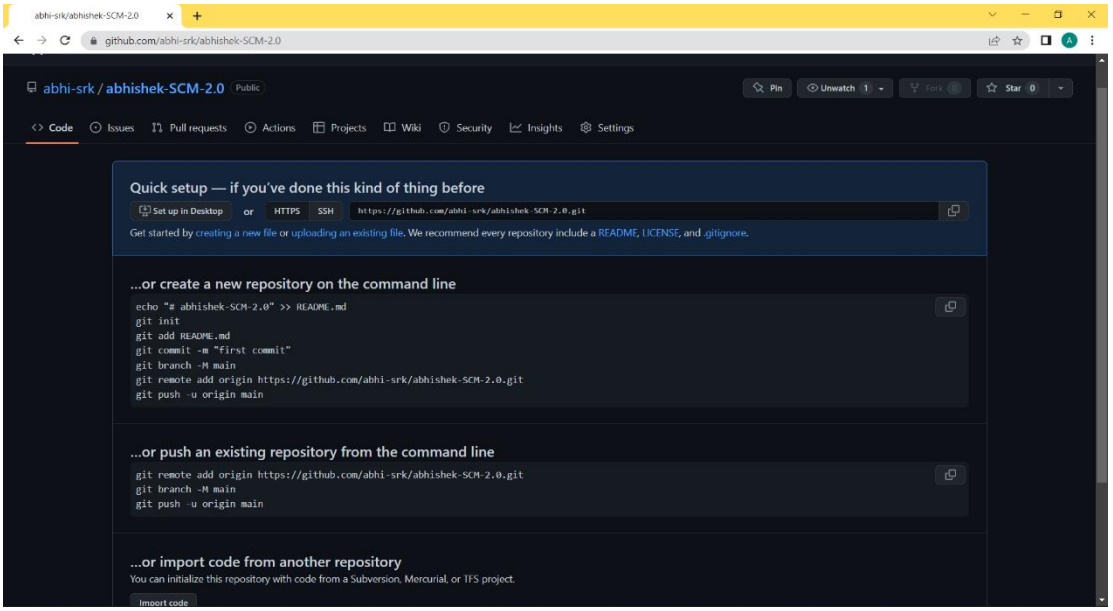


- If you want to import code from an existing repository select the

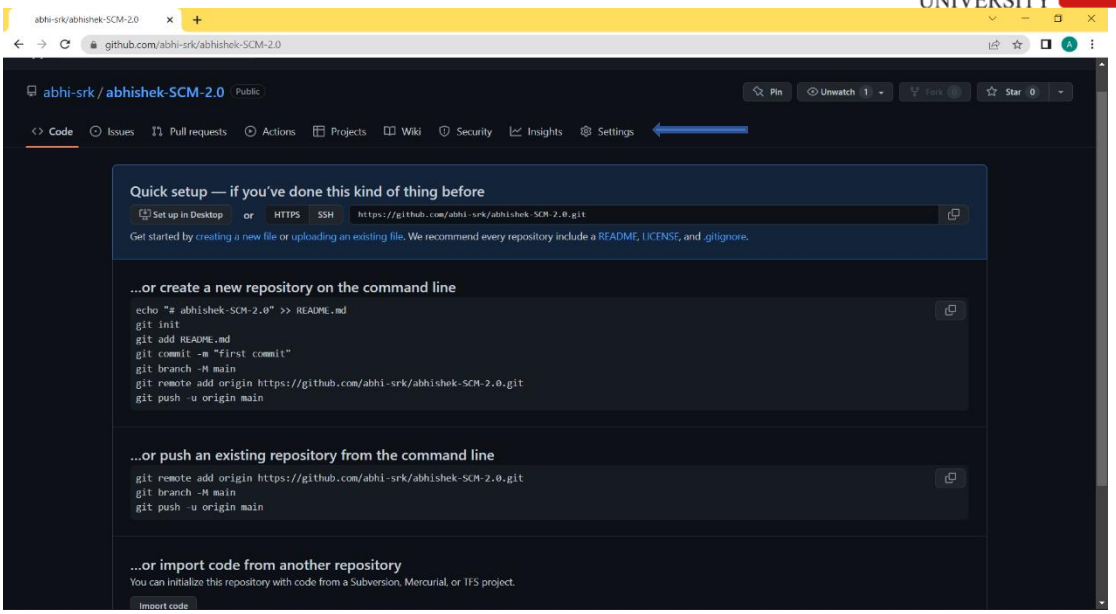


import code option.

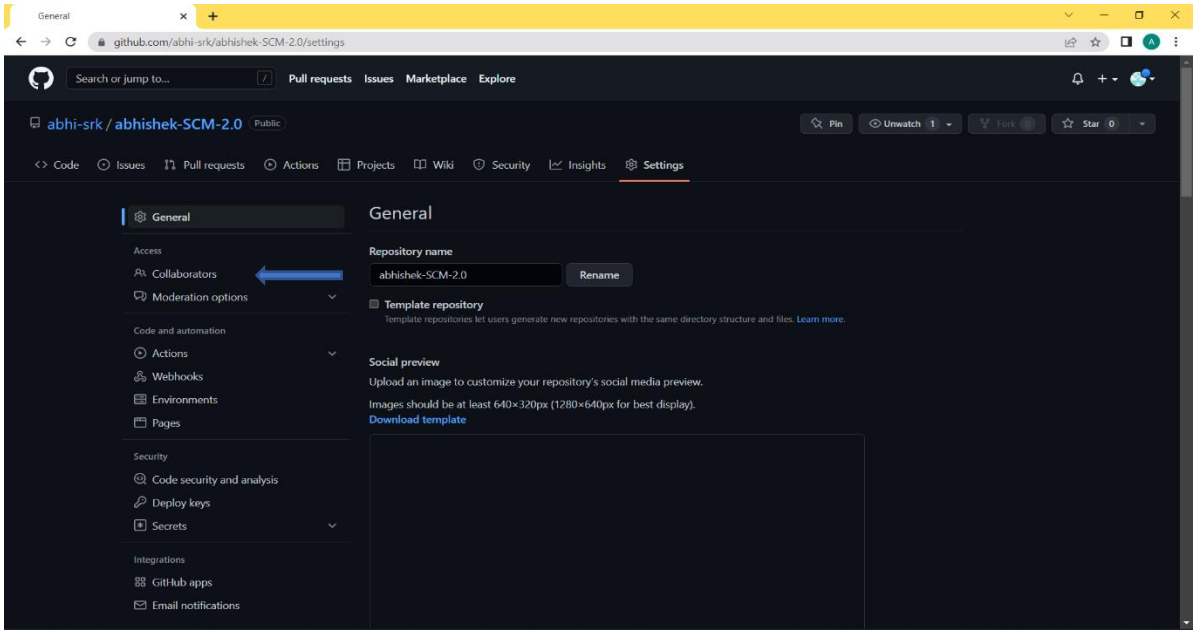
- To create a new file or upload an existing file into your repository select the option in the following box.



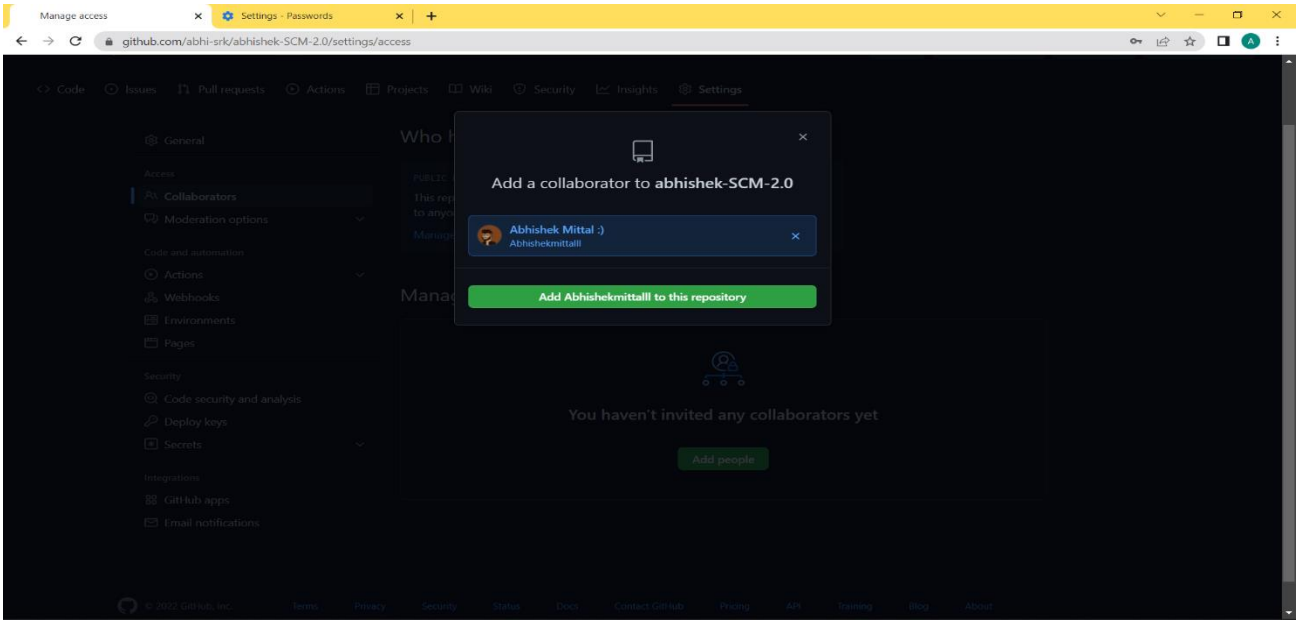
- Now, you have created your repository successfully.
- To add members to your repository, open your repository and select settings option in the navigation bar.



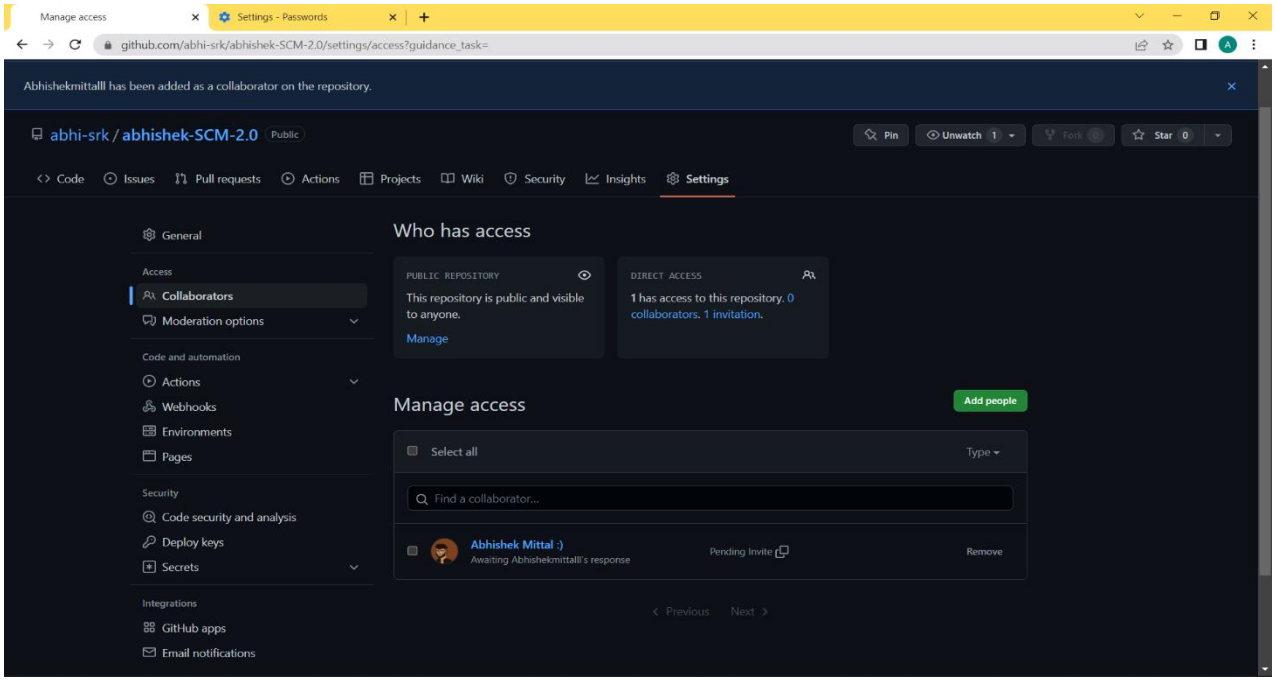
- Click on Collaborators option under the access tab.



- After clicking on collaborators GitHub asks you to enter your password to confirm the access to the repository.
- After entering the password you can manage access and add/remove team members to your project.
- To add members click on the add people option and search the id of your respective team member.



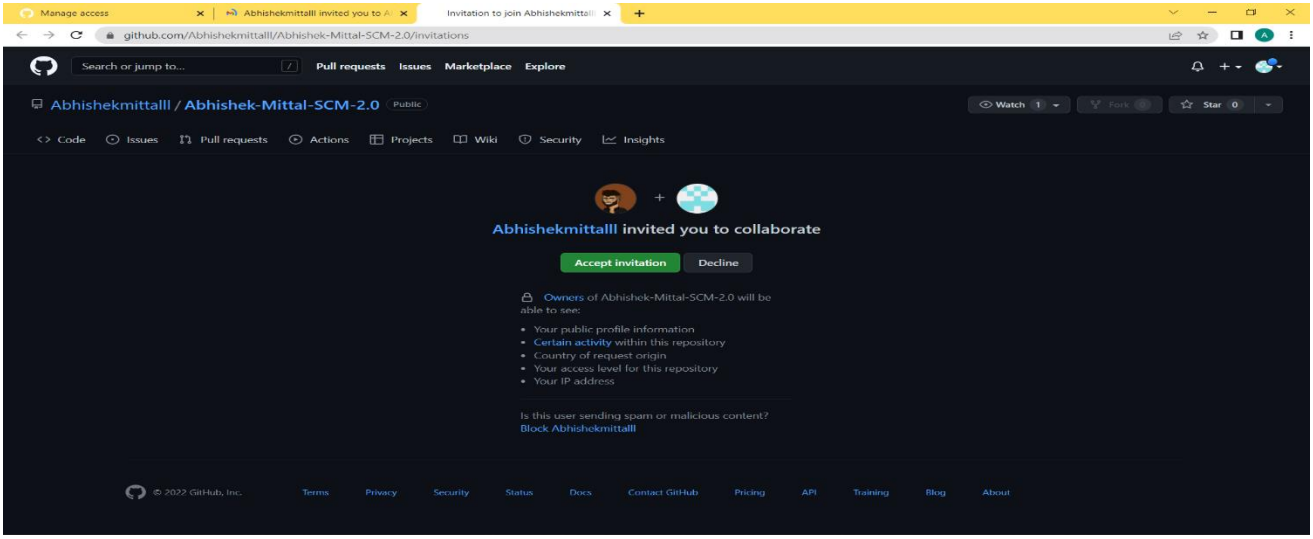
- To remove any member click on remove option available in the last column of member's respective row.



- To accept the invitation from your team member, open your email registered with GitHub.

- You will receive an invitation mail from the repository owner. Open the email and click on accept invitation.

- You will be redirected to GitHub where you can either select to accept or decline the invitation.



- You will be shown the option that you are now allowed to push

- Now all members are ready to contribute

Aim: Open and close a Pull Request

- To open a pull request we first have to make a new branch, by using git branch branchname option.

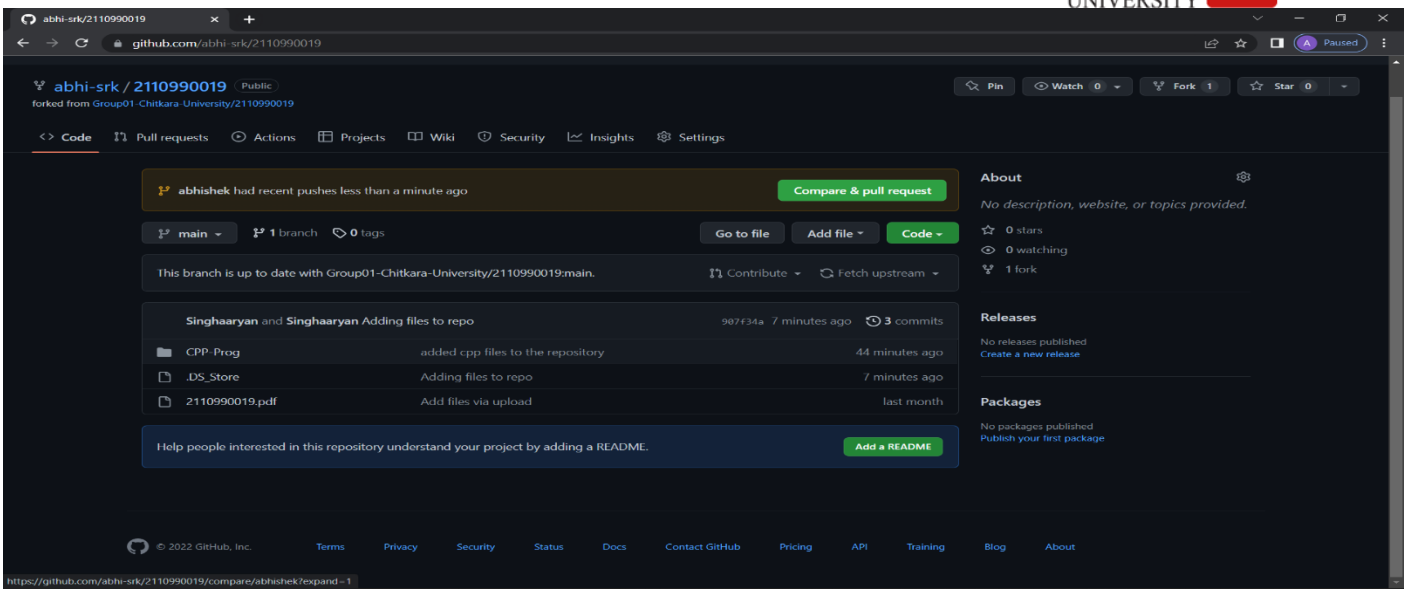
```
ABHISHEK KUMAR@LAPTOP-VC4G4TOF MINGW64 ~/OneDrive/Desktop/2110990019 (main)
$ git branch
* main

ABHISHEK KUMAR@LAPTOP-VC4G4TOF MINGW64 ~/OneDrive/Desktop/2110990019 (main)
$ git branch abhishek

ABHISHEK KUMAR@LAPTOP-VC4G4TOF MINGW64 ~/OneDrive/Desktop/2110990019 (main)
$ git branch
  abhishek
* main

ABHISHEK KUMAR@LAPTOP-VC4G4TOF MINGW64 ~/OneDrive/Desktop/2110990019 (main)
$ git checkout abhishek
Switched to branch 'abhishek'

ABHISHEK KUMAR@LAPTOP-VC4G4TOF MINGW64 ~/OneDrive/Desktop/2110990019 (abhishek)
$ git branch
* abhishek
  main
```

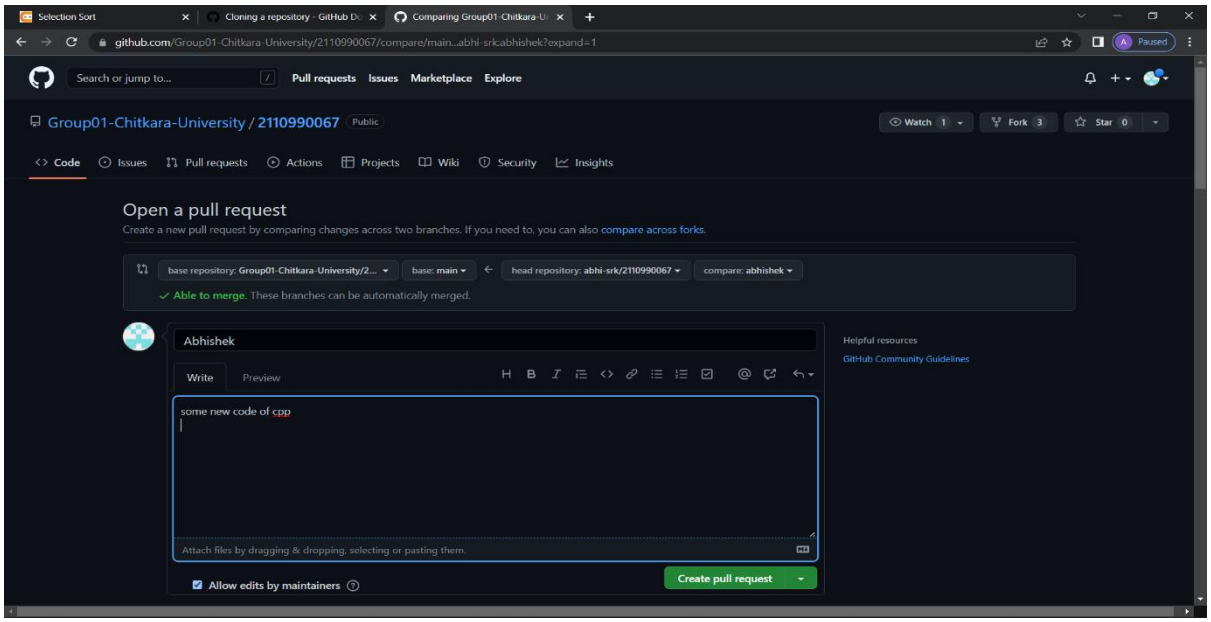
- After making new branch we add a file to the branch or make changes in the existing file.

- Add and commit the changes to the local repository.
- Use git push origin branchname option to push the new branch to the main repository.

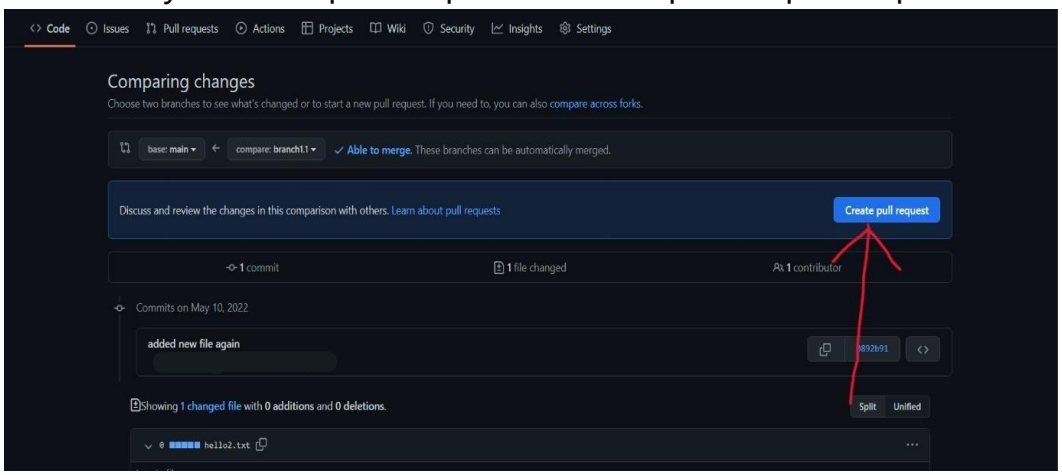
```
MINGW64:/c/Users/ABHISHEK KUMAR/OneDrive/Desktop/2110990019

ABHISHEK KUMAR@LAPTOP-VC4G4TOF MINGW64 ~/OneDrive/Desktop/2110990019 (abhishek)
$ git status
On branch abhishek
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        CPP-Prog/BookAllocation.cpp
        CPP-Prog/FindPeak.cpp
        CPP-Prog/bubblesort.cpp

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

ABHISHEK KUMAR@LAPTOP-VC4G4TOF MINGW64 ~/OneDrive/Desktop/2110990019 (abhishek)
$ git add .

ABHISHEK KUMAR@LAPTOP-VC4G4TOF MINGW64 ~/OneDrive/Desktop/2110990019 (abhishek)
$ git commit -m "Added some new cpp files in the Cpp folder"
[abhishek 71f766d] Added some new cpp files in the Cpp folder
 3 files changed, 118 insertions(+)
 create mode 100644 CPP-Prog/BookAllocation.cpp
 create mode 100644 CPP-Prog/FindPeak.cpp
 create mode 100644 CPP-Prog/bubblesort.cpp

ABHISHEK KUMAR@LAPTOP-VC4G4TOF MINGW64 ~/OneDrive/Desktop/2110990019 (abhishek)
$ git push origin abhishek
Enumerating objects: 8, done.
Counting objects: 100% (8/8), done.
Delta compression using up to 8 threads
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 1.50 KiB | 1.50 MiB/s, done.
Total 6 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
remote:
remote: Create a pull request for 'abhishek' on GitHub by visiting:
remote:      https://github.com/abhi-srk/2110990019/pull/new/abhishek
remote:
To https://github.com/abhi-srk/2110990019.git
 * [new branch]      abhishek -> abhishek

ABHISHEK KUMAR@LAPTOP-VC4G4TOF MINGW64 ~/OneDrive/Desktop/2110990019 (abhishek)
$
```

- After pushing new branch GitHub will either automatically ask you to create a pull request or you can create your own pull request.

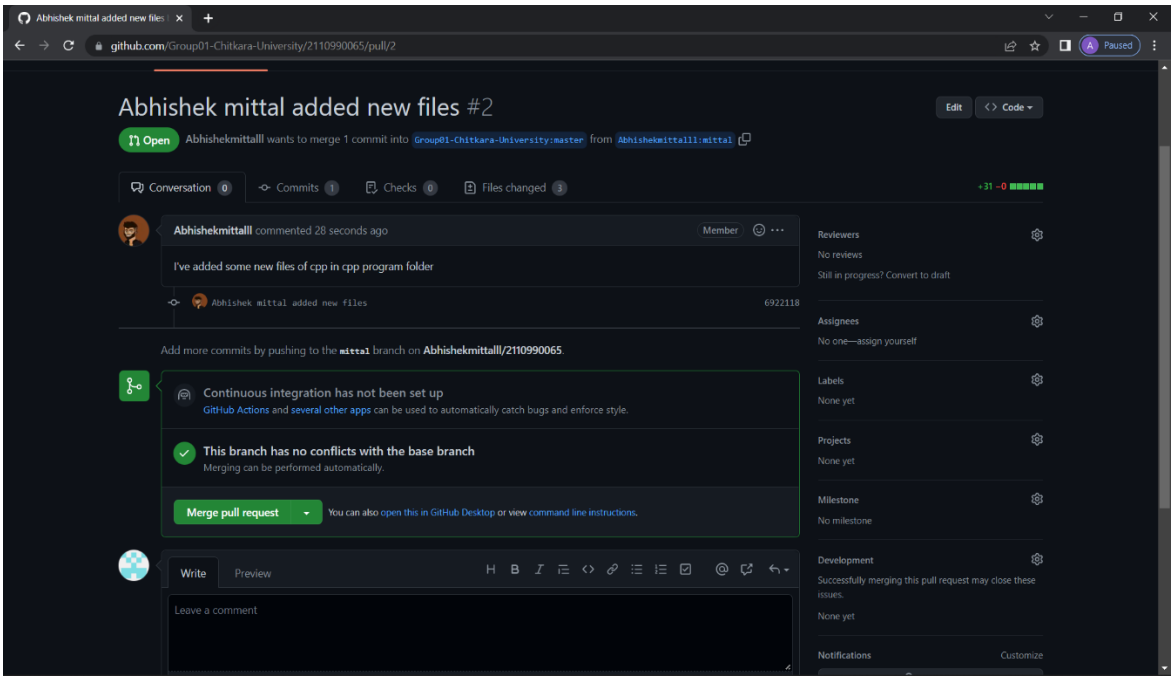




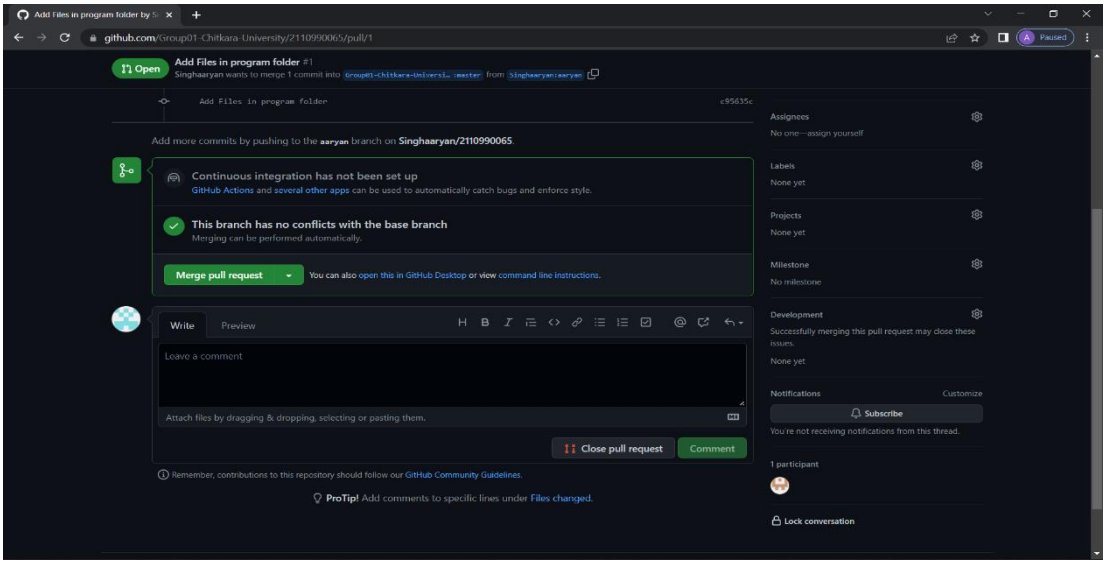
- To create your own pull request click on pull request option.



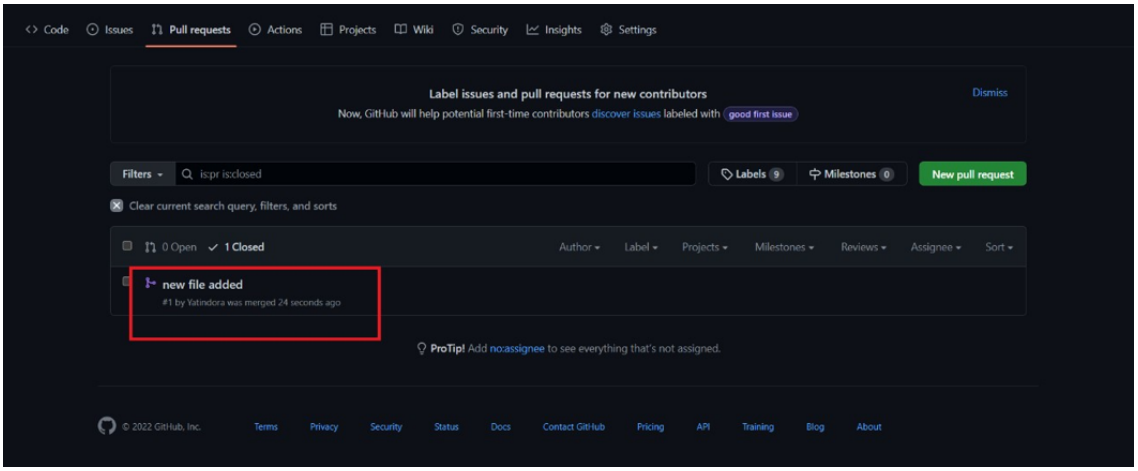
- GitHub will detect any conflicts and ask you to enter a description of your pull request.



- After opening a pull request all the team members will be sent the request if they want to merge or close the request.
- If the team member chooses not to merge your pull request they will close you're the pull request.
- To close the pull request simply click on close pull request and add comment/ reason why you closed the pull request.



- You can see all the pull request generated and how they were dealt with by clicking on pull request option.

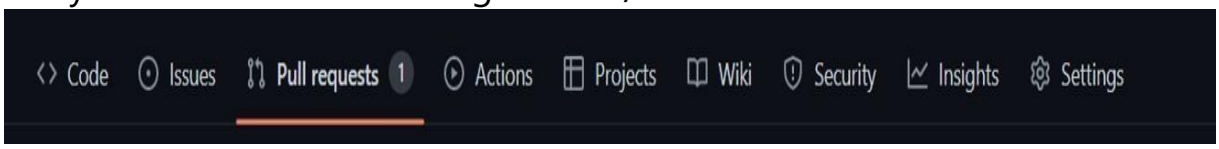


Aim: Create a pull request on a team member's repo and

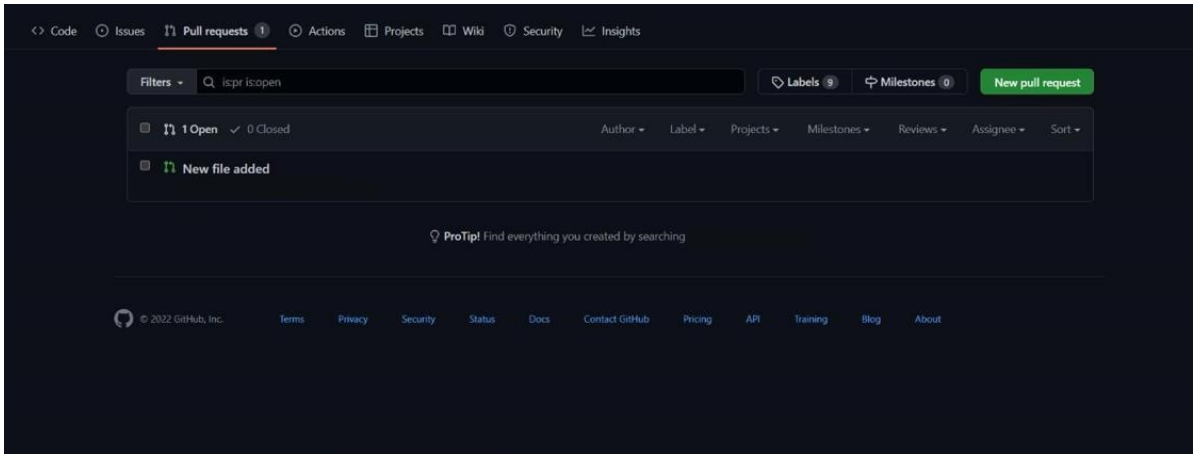
close pull requests generated by team members on own Repo as a maintainer

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:-

- 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.

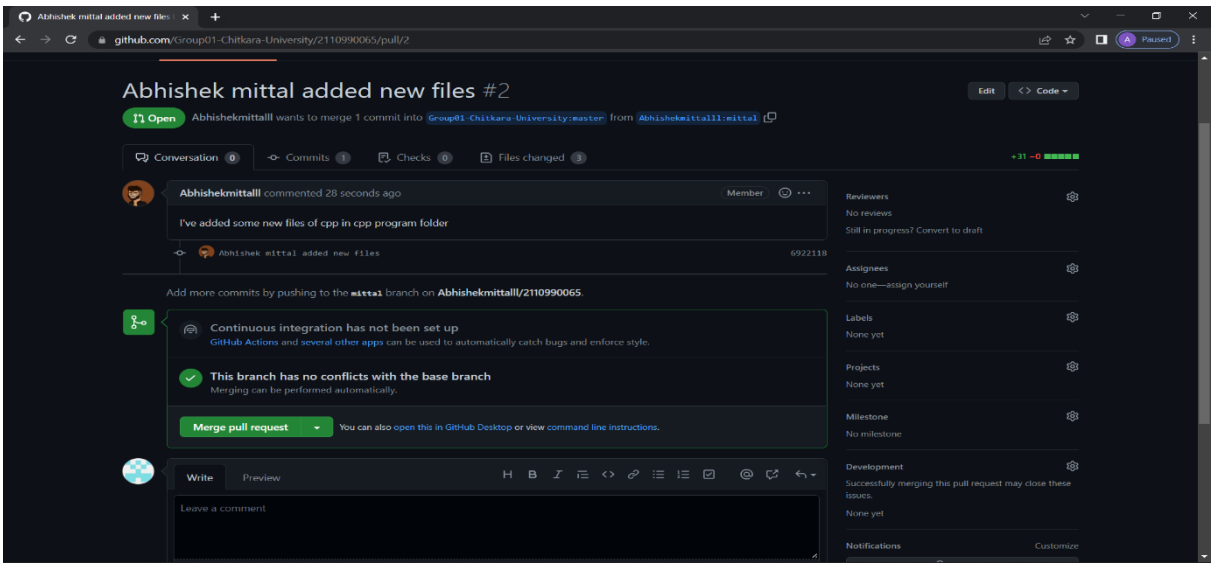


- They will notice a new notification in the pull request menu.
- Click on it. The pull request generated by you will be visible to them.

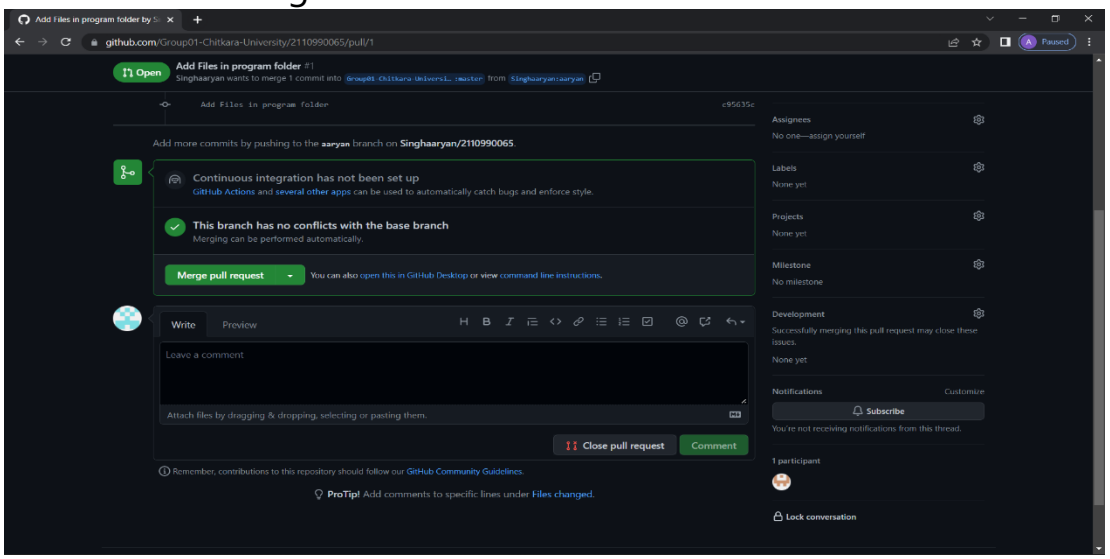


- Click on the pull request. Two option will be available, either to close the pull request or Merge the request with the main branch.

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

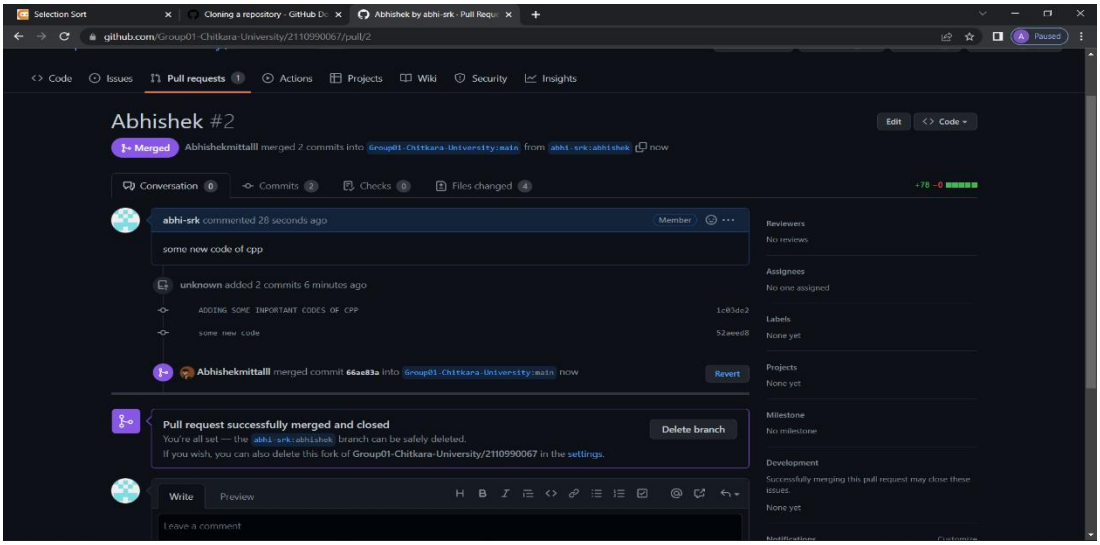


- By selecting close the pull request the pull request is not accepted and not merged with main branch.



- 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.



sidjain8427

- The result of closing the 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.

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.

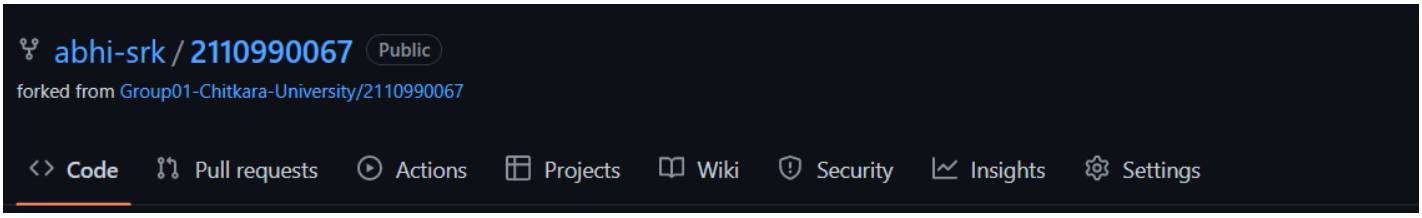
A repository's graphs give you information on traffic, projects that depend on the repository, contributors and commits to the repository, and a repository's forks and network. If you maintain a repository, you can use this data to get a better understanding of who's using your repository and why they're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

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

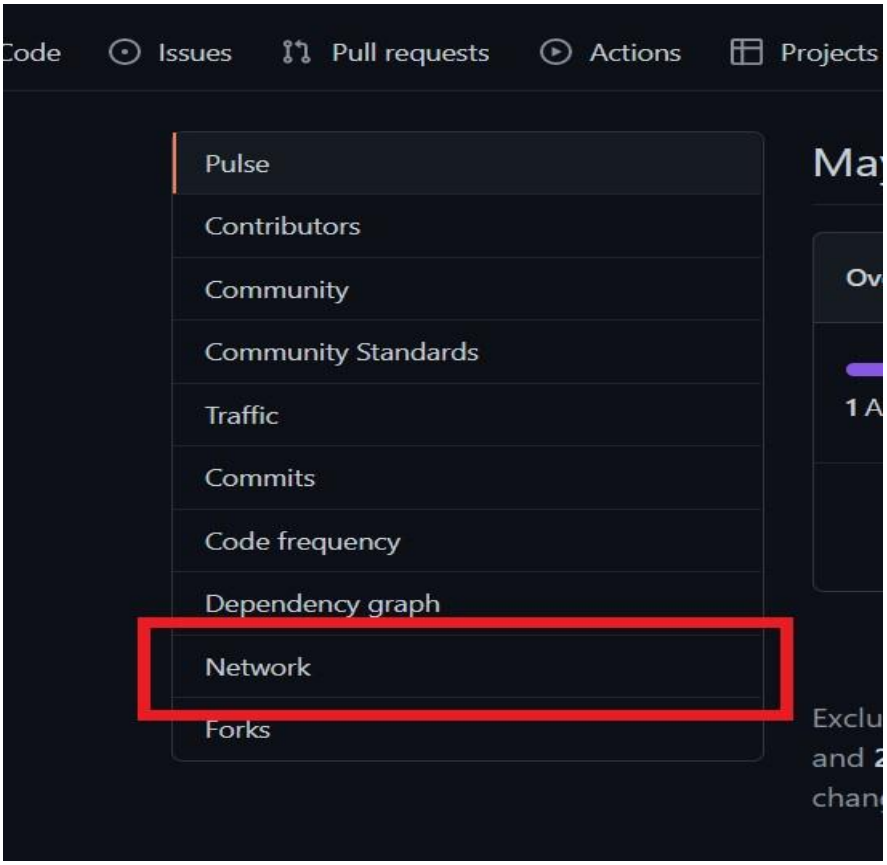
- Pulse
- Contributors
- Traffic
- Commits
- Code frequency
- Network

Steps to acess network graphs of respective repository

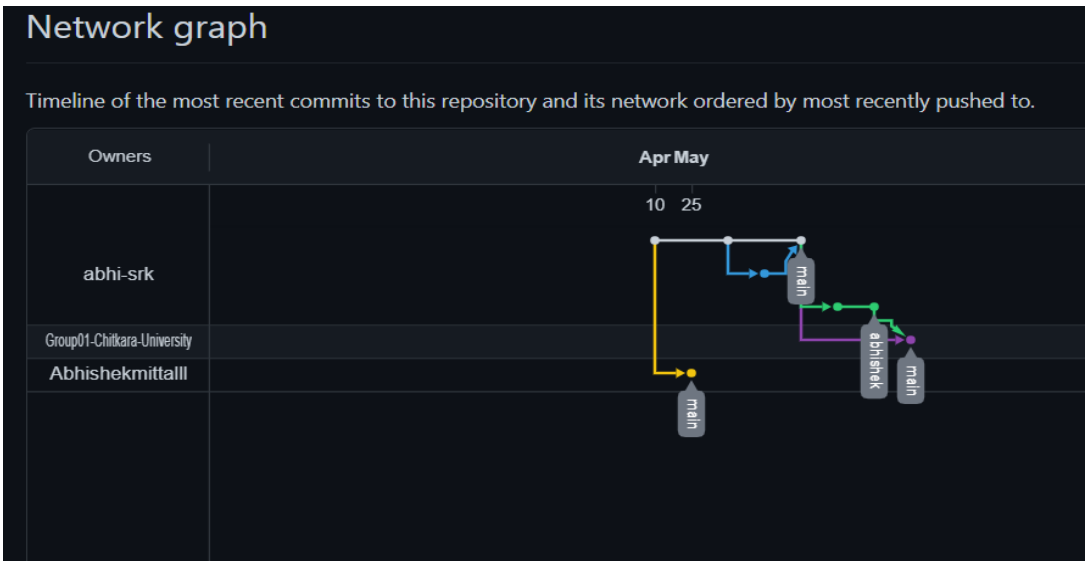
1. On GitHub.com, navigate to the main page of the repository. 2.Under your repository name, click Insights.



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.

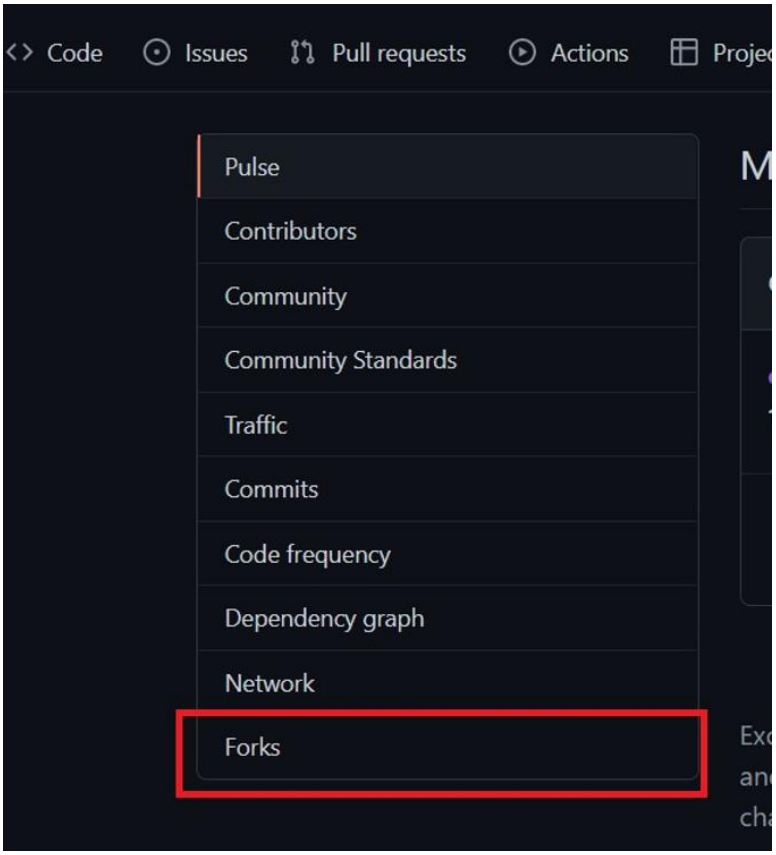


Listing the forks of a repository

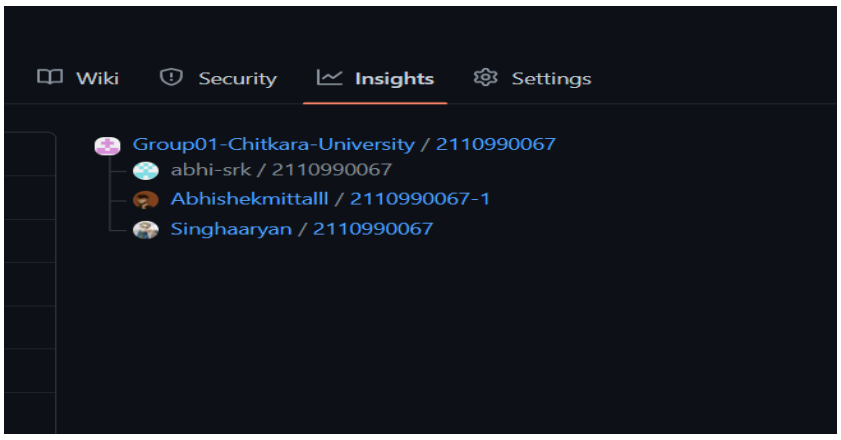
Forks are listed alphabetically by the username of the person who forked the repository

Clicking the number of forks shows you the full network. From there you can click "members" to see who forked the repo

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



3. In the left sidebar, click Forks.



Here you can see all the forks

Viewing the dependencies of a repository

You can use the dependency graph to explore the code your repository depends on.

Almost all software relies on code developed and maintained by other developers, often known as a supply chain. For example, utilities, libraries, and frameworks. These dependencies are an integral part of your code and any bugs or vulnerabilities in them may affect your code. It's important to review and maintain these dependencies.

6. Reference

S.No.	
1	https://www.geeksforgeeks.org/version-control-systems/
2	https://www.perforce.com/blog/vcs/what-is-version-control
3	https://medium.com/@lanceharvieruntime/version-control-why-do-we-need-it-1681f4888cec
4	https://en.wikipedia.org/wiki/Git
5	https://git-scm.com/
6	https://www.simplilearn.com/tutorials/git-tutorial/git-vs-github