



Subject name: Source Code Management

Subject code: CS181

Cluster: Beta

Department: DCSE

Submitted by:

Garima Singh

2110990489

G8-B

Submitted to:

Dr. Monit Kapoor

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	G8 -B
Faculty Name	Dr. Monit Kapoor		

Table of Content -

S. No.	Content	Page No.
1	Setting up of Git Client	4 - 6
2	Setting up GitHub Account	7- 8
3	Generate logs	9 – 10
4	Create and visualize branches	11
5	Git lifecycle description	12 – 13
6	Add collaborators on GitHub Repo	14 – 15
7	Fork and Commit	16
8	Merge and Resolve conflicts created due to own activity and collaborators activity	17 – 20
9	Reset and revert	21-22
10	Create a distributed Repository and add members in project team	23 - 24
11	Open and close a pull request.	25 - 26
12	Each project member shall create a pull request on a team members repo and close pull requests generated by team members on own Repo as a maintainer.	27 - 28
13	Publish and print network graphs	29

Task 1.1

Practical No. 01

Aim: Setting up the git client.

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

When running the installer, various screens appear (Windows screens shown). Generally, you can accept the default selections, except in the screens below where you do not want the default selections:

In the Select Components screen, make sure Windows Explorer Integration is selected as shown:

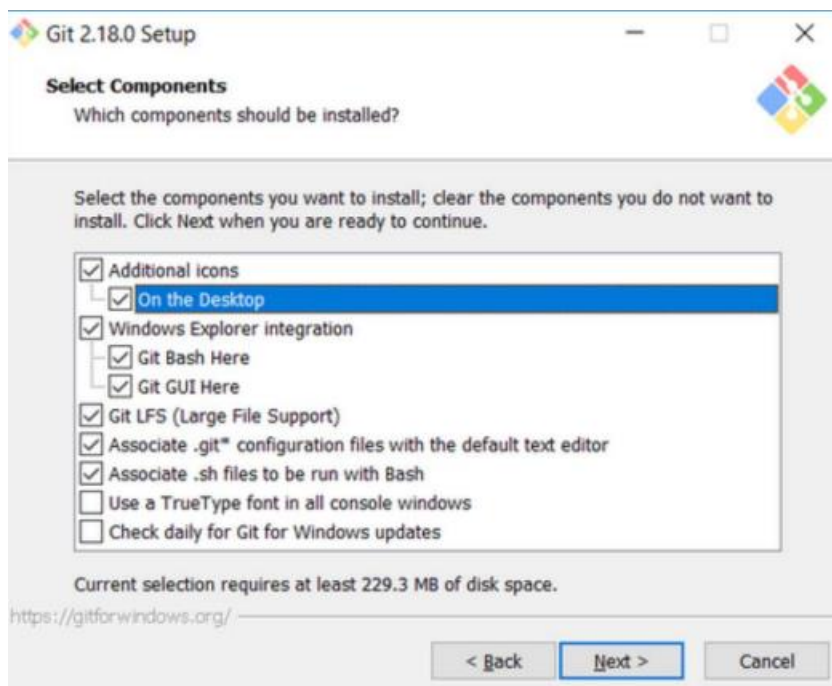
Setting up of Git Client

You can download Git for free from the following website:

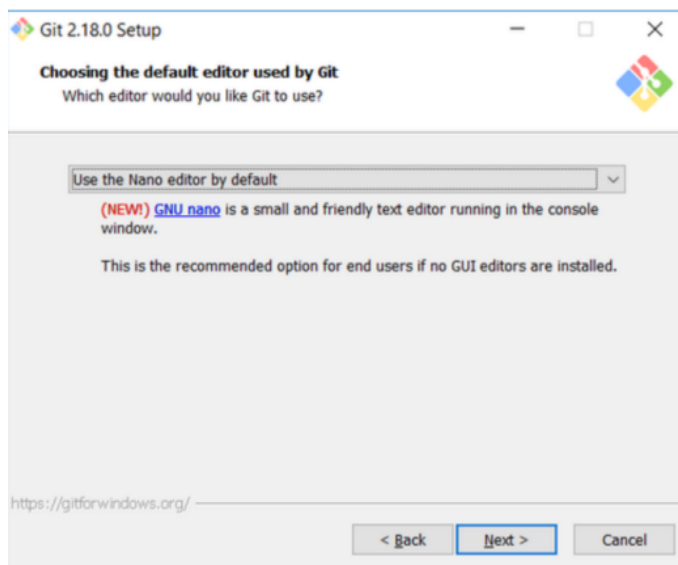
To start using Git, we are first going to open up our Command shell.

For Windows, you can use Git bash, which comes included in Git for Windows. For Mac and Linux you can use the built-in terminal.

The first thing we need to do, is to check if Git is properly installed:

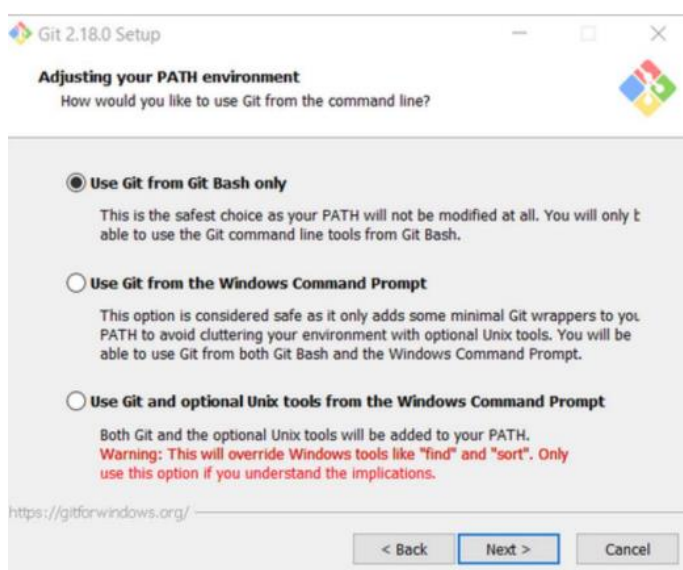


In the choosing the default editor is used by Git dialog, it is strongly recommended that you DO NOT select default VIM editor- it is challenging to learn how to use it, and there are better modern editors available. Instead, choose Notepad++ or Nano – either of those is much easier to use. It is strongly recommended that you select Notepad++



In the Adjusting your PATH screen, all three options are acceptable:

1. Use Git from Git Bash only: no integration, and no extra command in your command path.
2. Use Git from the windows Command Prompt: add flexibility – you can simply run git from a windows command prompt, and is often the setting for people in industry – but this does add some extra commands.
3. Use Git and optional Unix tools from the Windows Command Prompt: this is also a robust choice and useful if you like to use Unix like commands like grep.



In the Configuring the line ending screen, select the middle option (Checkout-as-is, commit Unix-style line endings) as shown. This helps migrate files towards the Unix-style (LF) terminators that most modern IDE's and editors support. The Windows convention (CR-LF line termination) is only important for Notepad.



Once Git is installed, there is some remaining custom configuration you must do. Follow the steps below:

The first thing we need to do, is to check if Git is properly installed:

```
DELL@Dark-THunDeR MINGW64 ~ (master)
$ git version
git version 2.35.1.windows.2
```

If Git is installed, it should show something like `git version X.Y`

Now let Git know who you are. This is important for version control systems, as each Git commit uses this information:

```
DELL@Dark-THunDeR MINGW64 ~ (master)
$ git config --global user.name "Garima729"

DELL@Dark-THunDeR MINGW64 ~ (master)
$ git config --global user.email "garima0489.be21@chitkara.edu.in"
```

Change the user name and e-mail address to your own. You will probably also want to use this when registering to GitHub later on.

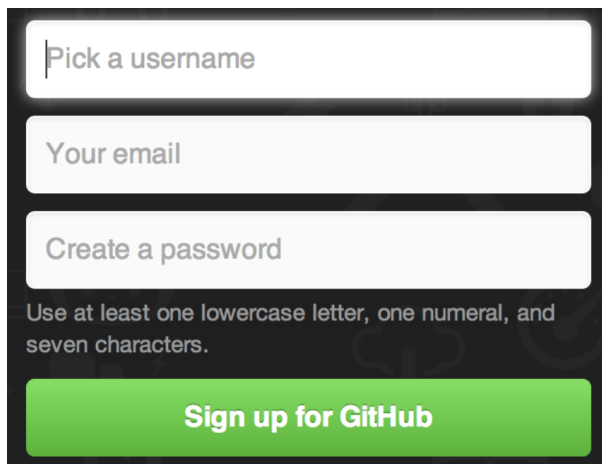
Practical No. 02

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.

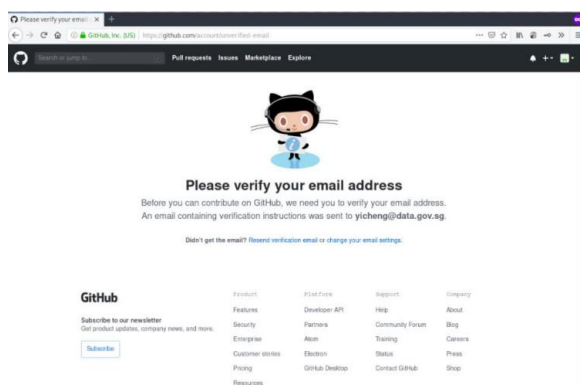
There are several types of accounts on GitHub. Every person who uses GitHub has their own user account, which can be part of multiple organisations and teams. Your user account is your identity on GitHub.com and represents you as an individual.

1. Creating an account: To sign up for an account on GitHub.com, navigate to <https://github.com/> and follow the prompts. To keep your GitHub account secure you should use a strong and unique password.

A screenshot of the GitHub sign-up form. It features three input fields: 'Pick a username', 'Your email', and 'Create a password'. Below the password field, there is a text requirement: 'Use at least one lowercase letter, one numeral, and seven characters.' At the bottom, there is a prominent green button labeled 'Sign up for GitHub'.

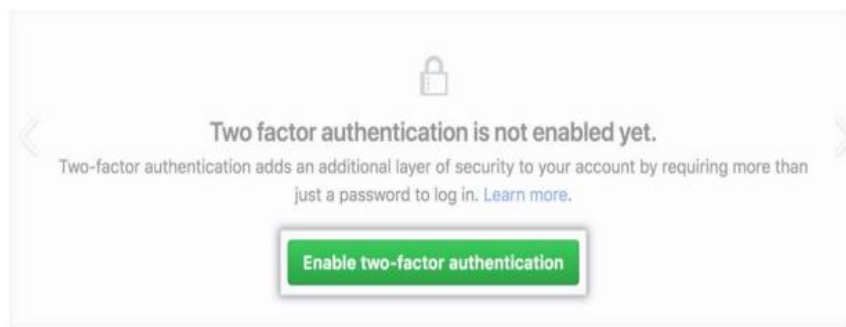
2. Choosing your GitHub product: You can choose GitHub Free or GitHub Pro to get access to different features for your personal account. You can upgrade at any time if you are unsure at first which product you want. For more information on all GitHub's plans, see "GitHub's products".

3. Verifying your email address: To ensure you can use all the features in your GitHub plan, verify your email address after signing up for a new account.



4. Configuring two-factor authentication: Two-factor authentication, or 2FA, is an extra layer of security used when logging into websites or apps. We strongly urge you to configure 2FA for safety of your account. For more information, see “About twofactor authentication.”

Two-factor authentication



5. Viewing your GitHub profile and contribution graph: Your GitHub profile tells people the story of your work through the repositories you’ve pinned, the organisation memberships you’ve chosen to publicize, the contributions you’ve made, and the projects you’ve created. For more information, see “About your profile” and “Viewing contributions on your profile.”

Practical No. 03

Aim: Program to generate logs

Basic Git init

Git init command creates a new Git repository. It can be used to convert an existing, undersigned project to a Git repository or initialize a new, empty repository. Most other Git commands are not available outside of an initialize repository, so this is usually the first command you'll run in a new project.

```
DELL@Dark-THunDeR MINGW64 /d (master)
$ git init
Reinitialized existing Git repository in D:/./.git/
```

Basic 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 status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached ..." to unstage)
    new file:   index.html
```

Basic Git add command

The git add command adds a change in the working directory to the staging area. It tells Git that you want to include updates to a particular file in the next commit. However, git add doesn't really affect the repository in any significant way—changes are not actually recorded until you run git commit

```
git add index.html
```

```
git add --all
```

Basic 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. Prior to the execution of git commit, The git add

command is used to promote or 'stage' changes to the project that will be stored in a commit. These two commands git commit and git add are two of the most frequently used.

```
git commit -m "First release of Hello World!"
[master (root-commit) 221ec6e] First release of Hello World!
 3 files changed, 26 insertions(+)
 create mode 100644 README.md
 create mode 100644 bluestyle.css
 create mode 100644 index.html
```

Basic Git log

Git log command is one of the most usual commands of git. It is the most useful command for Git. Every time you need to check the history, you have to use the git log command. The basic git log command will display the most recent commits and the status of the head. It will use as:

```
$ git log
commit 51ff2e37eee2c7e4422dd52c607c859eb015cc11 (HEAD -> main, origin/main)
Author: Garima729 <garima0489.be21@chitkara.edu.in>
Date: Sat Apr 9 15:36:21 2022 +0530

    first commit

commit ec9f03f571828acad3dfaf1d89204c1de87cce26 (origin/master)
Author: Garima729 <garima0489.be21@chitkara.edu.in>
Date: Sat Apr 9 15:18:21 2022 +0530

    First sem iwt project
```

Practical No. 04

Aim: Create and visualize branches in Git

How to create branches?

The main branch in git is called master branch. But we can make branches out of this main master branch. We also can merge both the parent(master) and child (other branches).

git branch

lists your branches. a * will appear next to the currently active branch

git branch [branch-name]

create a new branch at the current commit

git checkout

switch to another branch and check it out into your working directory

git merge [branch]

merge the specified branch's history into the current one

Visualizing Branches: To visualize, we have to create a new file in the new branch “trial” instead of the master branch. After this we have to do three step architecture i.e. working directory, staging area and git repository.

A master branch is created after our first commit. Let's commit a file and check the branch list. Since master has been created, we can now create a branch.

```
DELL@Dark-THunDeR MINGW64 /d/project (main)
$ git status
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean

DELL@Dark-THunDeR MINGW64 /d/project (main)
$ git branch
* main

DELL@Dark-THunDeR MINGW64 /d/project (main)
$ git branch trial

DELL@Dark-THunDeR MINGW64 /d/project (main)
$ git checkout trial
Switched to branch 'trial'

DELL@Dark-THunDeR MINGW64 /d/project (trial)
$ git status
On branch trial
nothing to commit, working tree clean

DELL@Dark-THunDeR MINGW64 /d/project (trial)
$ touch test.txt

DELL@Dark-THunDeR MINGW64 /d/project (trial)
$ git add test.txt

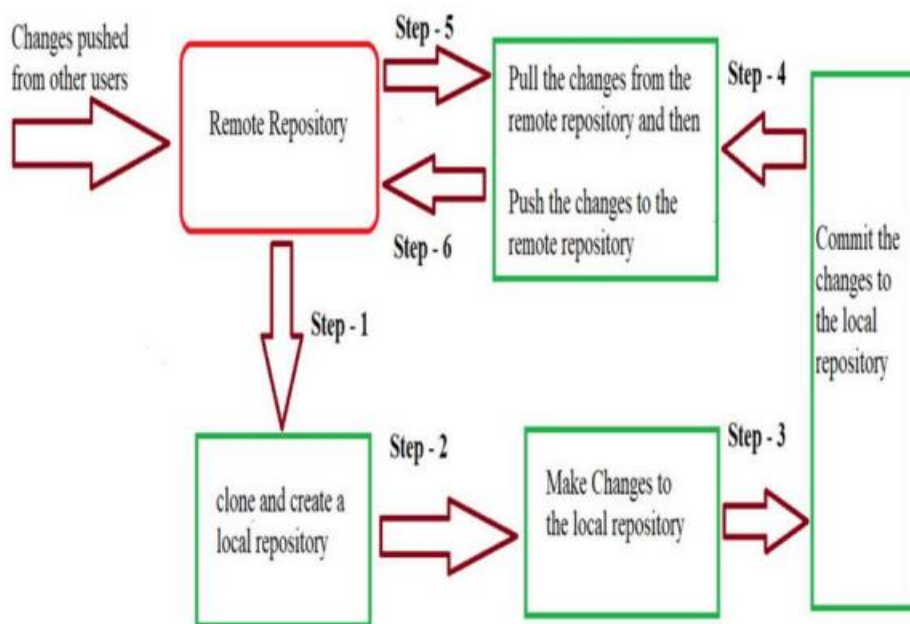
DELL@Dark-THunDeR MINGW64 /d/project (trial)
$ git commit -m "first branch"
[trial e6ccea5] first branch
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 test.txt

DELL@Dark-THunDeR MINGW64 /d/project (trial)
$ git branch
main
* trial
```

Practical No. 05

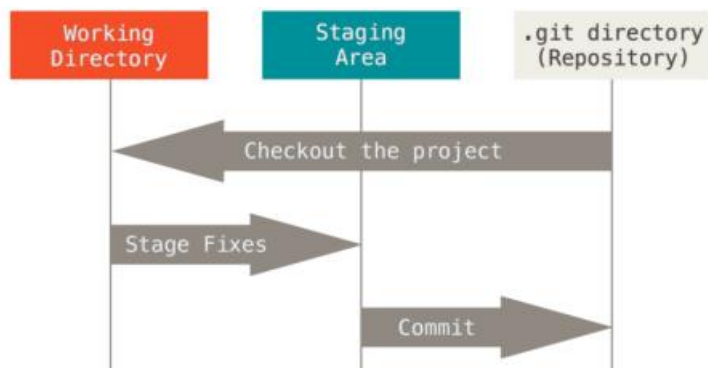
Aim: Git lifecycle description

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 Lifecycle description that git has and understand more about its life cycle. Let us see some of the basic steps that we have to follow while working with Git-



- **Step 1-** We first clone any of the code residing in the remote repository to make our own local repository.
- **Step 2-** We edit the files that we have cloned in our local repository and make the necessary changes in it.
- **Step 3-** We commit our changes by first adding them to our staging area and committing them with a commit message.
- **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.
- **Step 6-** If there are no changes we push our changes 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:



1. Working Directory

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

2. Staging Area

Now, to track files 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, temporary files, etc.

3. 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 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`

Task 1.2

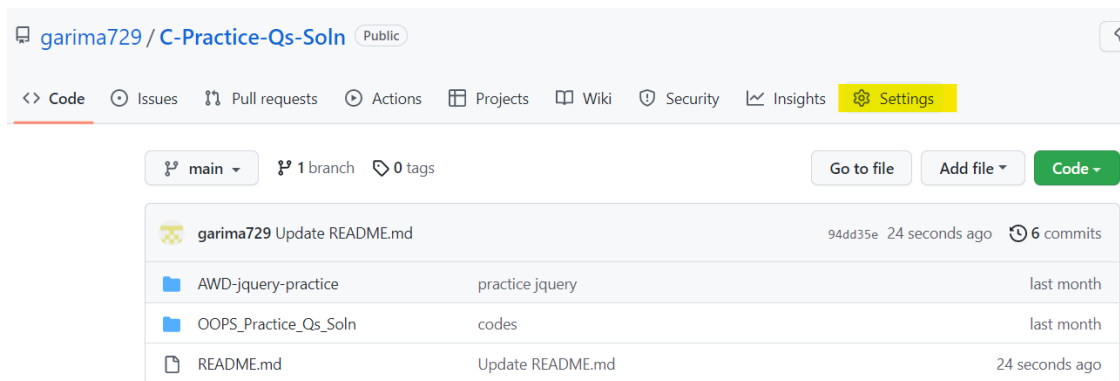
Practical No. 01

Aim: Add collaborators on GitHub Repository

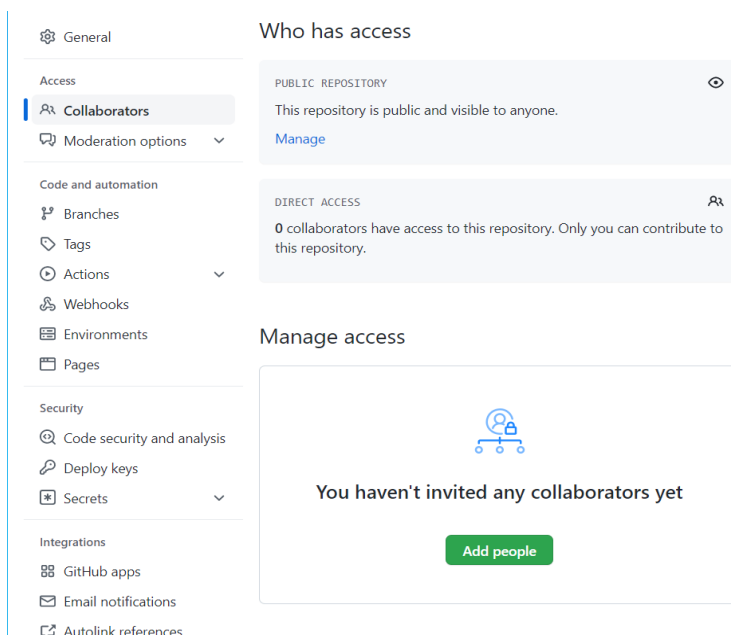
Even if you have a public repository in GitHub, not everyone has the permission to push code into your repository. Other users have a read-only access and cannot modify the repository. In order to allow other individuals to make changes to your repository, you need to invite them to collaborate to the project.

The following steps should be performed to invite other team members to collaborate with your repository.

1. Click on the **Settings** tab in the right corner of the GitHub page.

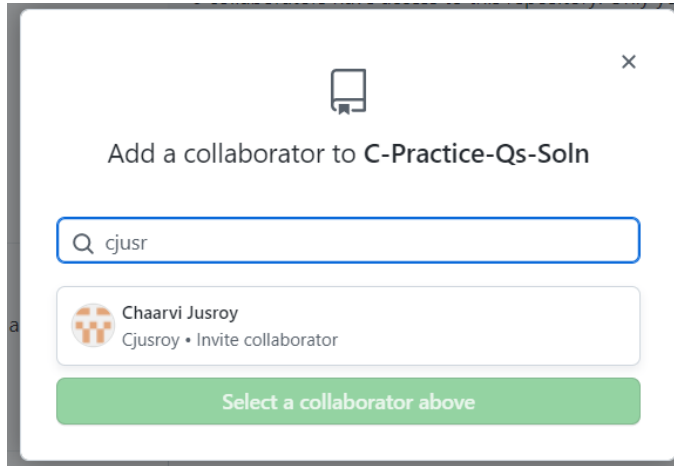


2. Go to **Collaborators** option under the Access tab. On the Manage Access page, you will see an **Invite collaborator link** as shown in the below diagram



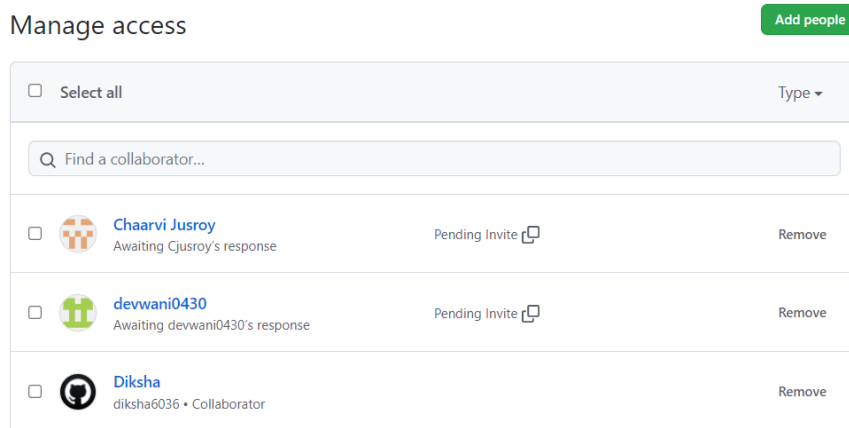
3. You can Invite collaborators by any of the following options –

- Username
- Full name
- Email



4. After you send the invite, the collaborator receives an email invitation. The collaborator has to accept it in order to get permission to collaborate on the same project.

5. The **Manage Access** option also allows a repository owner to view the invitations that are pending and not accepted.

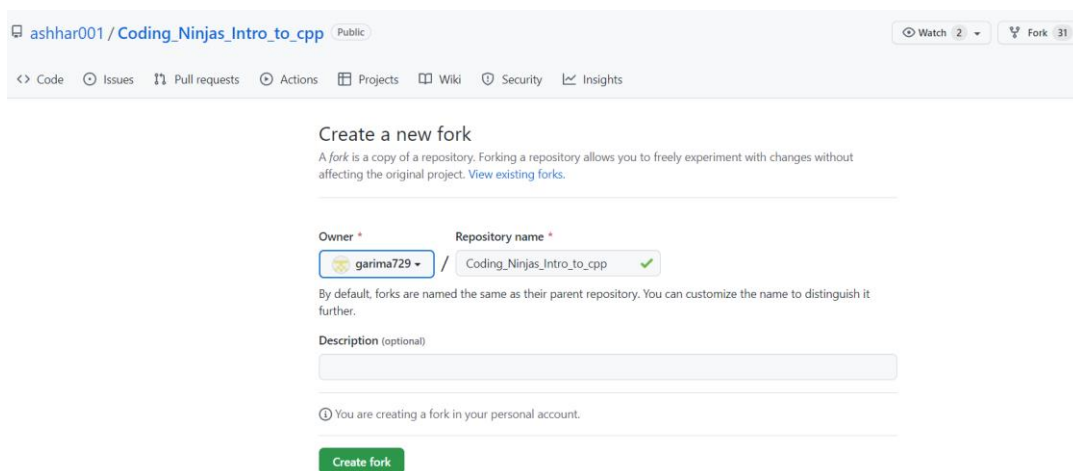


Practical No. 02

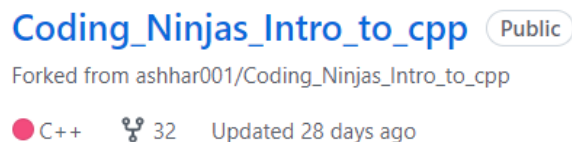
Aim: Fork And Commit

A **fork** is a copy of a repository. This is useful when you want to contribute to someone else's project or start your own project based on theirs. **fork** is not a command in Git, but something offered in GitHub and other repository hosts.

1. We can see the fork option at the top right corner of a repository page. By clicking on that, the forking process will start. It will take a while to make a copy of the project for other users. After the forking completed, a copy of the repository will be copied to your GitHub account. It will not affect the original repository.



2. The fork copy will look like as follows and we can add commits to it.



3. We can freely make changes and then create a pull request for the main project. The owner of the project will see your suggestion and decide whether they want to merge the changes or not.

Practical No. 03

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

Merging Branches

Once you've completed work on your branch, it is time to merge it into the main branch. Merging takes your branch changes and implements them into the main branch.

In real world, when we merge branches, we will run into conflicts quite often. Conflict happens because of the following reasons –

- When the same line of code is changed in different ways in two branches.
- A given file is changed in one branch but deleted in another branch.
- Same file is added twice in two different branches but the content of the file is different.

In these cases, git will stop the merge process as it cannot figure out how to merge the changes. In such scenarios we need to intervene manually and instruct how to proceed with the merging process.

resolve conflicts created due to own activity:

1. Create a repository with initial commit with the **hello.txt** file.

```
DELL@Dark-THunDeR MINGW64 /d/resolveconflict (master)
$ ls
hello.txt

DELL@Dark-THunDeR MINGW64 /d/resolveconflict (master)
$ git status
On branch master

No commits yet

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

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

DELL@Dark-THunDeR MINGW64 /d/resolveconflict (master)
$ git add hello.txt
warning: LF will be replaced by CRLF in hello.txt.
The file will have its original line endings in your working directory

DELL@Dark-THunDeR MINGW64 /d/resolveconflict (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   hello.txt
```

2. Create a new branch feature. Switch to the feature branch and create a new commit by editing the second line in the hello.txt file.

```

DELL@Dark-THunDeR MINGW64 /d/resolveconflict (master)
$ git branch feature

DELL@Dark-THunDeR MINGW64 /d/resolveconflict (master)
$ git switch feature
Switched to branch 'feature'

DELL@Dark-THunDeR MINGW64 /d/resolveconflict (feature)
$ echo hello feature >> hello.txt

DELL@Dark-THunDeR MINGW64 /d/resolveconflict (feature)
$ git status
On branch feature
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:   hello.txt

no changes added to commit (use "git add" and/or "git commit -a")

DELL@Dark-THunDeR MINGW64 /d/resolveconflict (feature)
$ git add .
warning: LF will be replaced by CRLF in hello.txt.
The file will have its original line endings in your working directory

DELL@Dark-THunDeR MINGW64 /d/resolveconflict (feature)
$ git commit -m 'hello feature'
[feature ddd8afe] hello feature
1 file changed, 1 insertion(+)

```

3. Now switch to the master branch and perform a new commit by adding a new line to hello.txt.

```

DELL@Dark-THunDeR MINGW64 /d/resolveconflict (feature)
$ git switch master
Switched to branch 'master'

DELL@Dark-THunDeR MINGW64 /d/resolveconflict (master)
$ cat hello.txt
hello

DELL@Dark-THunDeR MINGW64 /d/resolveconflict (master)
$ echo hello master>>hello.txt

DELL@Dark-THunDeR MINGW64 /d/resolveconflict (master)
$ git add .
warning: LF will be replaced by CRLF in hello.txt.
The file will have its original line endings in your working directory

DELL@Dark-THunDeR MINGW64 /d/resolveconflict (master)
$ git commit -m 'hello master'
[master f4a77a4] hello master
1 file changed, 1 insertion(+)

```

4. We will now try to merge the changes from the feature branch to the master branch.

```

DELL@Dark-THunDeR MINGW64 /d/resolveconflict (master)
$ git merge feature
Auto-merging hello.txt
CONFLICT (content): Merge conflict in hello.txt
Automatic merge failed; fix conflicts and then commit the result.

```

The output shows that the branch is now in an intermediate state of merging as the automatic merging has failed due to the conflict. We would need to interfere manually to complete the merging process.

5. Check the contents of the file hello.txt

```

DELL@Dark-THunDeR MINGW64 /d/resolveconflict (master|MERGING)
$ cat hello.txt
hello

<<<<<< HEAD
hello master
=====
hello feature
>>>>>> feature

```

From the contents of the file, it is clear that the first line is unchanged. The second line will be considered from both the branches. The output screen shows a divider with the "=====

notation. The first half of the divider contains the separator "<<<<<<< HEAD" which means that the contents are from the current branch where HEAD points to - master. The second half of the divider contains the separator ">>>>>>> feature" which means the contents are from the second branch - feature. Now we need to decide whether we need to keep the first half or the second half or both the changes.

6. We are deciding to keep both the changes so we modify the file manually by keeping only the contents and removing the divider "=====" notation, and the separators "<<<<<<< HEAD" and ">>>>>>> feature".

```
DELL@Dark-THunDeR MINGW64 /d/resolveconflict (master|MERGING)
$ vi hello.txt

DELL@Dark-THunDeR MINGW64 /d/resolveconflict (master|MERGING)
$ cat hello.txt
hello
hello master
hello feature
```

7. Now commit the changes and display the history.

```
DELL@Dark-THunDeR MINGW64 /d/resolveconflict (master|MERGING)
$ git add .

DELL@Dark-THunDeR MINGW64 /d/resolveconflict (master|MERGING)
$ git status
On branch master
All conflicts fixed but you are still merging.
(use "git commit" to conclude merge)

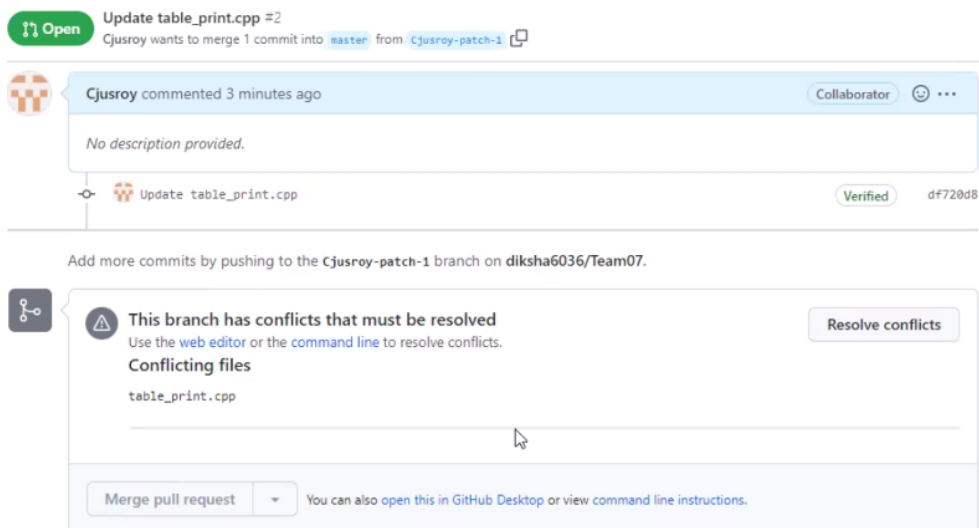
Changes to be committed:
  modified:   hello.txt

DELL@Dark-THunDeR MINGW64 /d/resolveconflict (master|MERGING)
$ git commit -m 'merge commit'
[master 46bd17f] merge commit

DELL@Dark-THunDeR MINGW64 /d/resolveconflict (master)
$ git log --oneline --all --graph
*   46bd17f (HEAD -> master) merge commit
| \
|  * ddd8afe (feature) hello feature
* | f4a77a4 hello master
|/
* 8d9c154 hello.txt
```

resolve conflicts created due to collaborator's activity:

1. Click on Resolve conflicts and you should see the entire display of the changed files in the pull request. You'll notice that GitHub has disabled the Mark as resolved button.

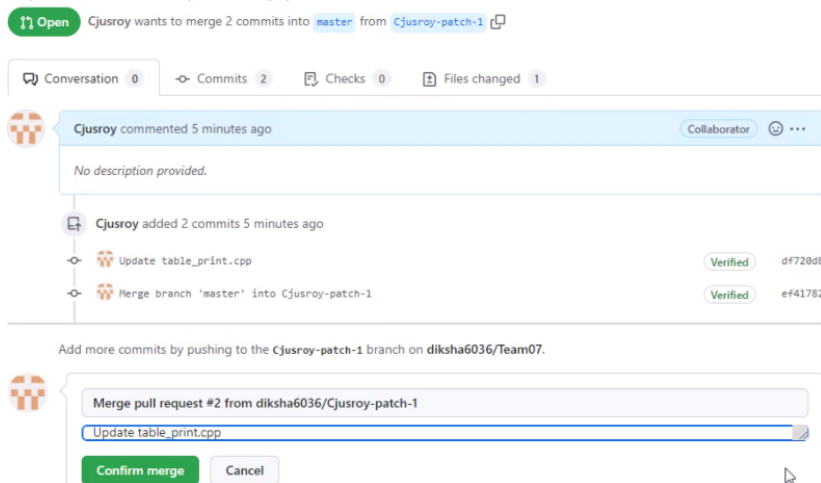


2. Resolve the conflicts in the file. Ensure that all traces of <<<<<<, >>>>>>, and ===== are removed.

3. If you do this correctly, you should see the button **Mark as resolved** become available for that particular file.

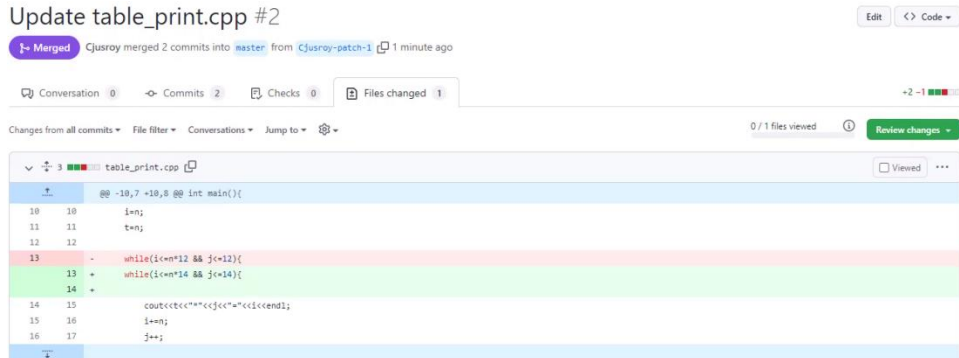
4. Now you can commit and confirm the merge

Update table_print.cpp #2



4. Once the merger is complete, you'll be able to review the changes.

Update table_print.cpp #2



Practical No. 04

Aim: Reset and Revert

reset is the command we use when we want to move the repository back to a previous **commit**, discarding any changes made after that **commit**.

1. We need to find the point we want to return to. To do that, we need to go through the **log**.

```
DELL@Dark-THunDeR MINGW64 /d/myproject (master)
$ git log --oneline
8580d87 (HEAD -> master, emergency-fix) updated
9abc93a coacoac
```

2. We want to return to the **commit**: 9abc93a coacoac. We **reset** our repository back to the specific commit using **git reset commithash** (*commithash* being the first 7 characters of the commit hash we found in the **log**):

```
DELL@Dark-THunDeR MINGW64 /d/myproject (master)
$ git reset 9abc93a
Unstaged changes after reset:
D    bluestyle.css
D    index.html
```

3. Check the **log** again:

```
DELL@Dark-THunDeR MINGW64 /d/myproject (master)
$ git log --oneline
9abc93a (HEAD -> master) coacoac
```

revert is the command we use when we want to take a previous **commit** and add it as a new **commit**, keeping the log intact.

1. Make a new **commit**, where we have "accidentally" deleted a file:

```
DELL@Dark-THunDeR MINGW64 /d/myproject (master)
$ git commit -m 'regular update'
[master 93b333f] regular update
1 file changed, 39 deletions(-)
delete mode 100644 practice3_awd.html
```

2. Now we have a part in our **commit** history we want to go back to. Let's try and do that with **revert**. We want to revert to the previous commit: 93b333f

```
DELL@Dark-THunDeR MINGW64 /d/myproject (master)
$ git log --oneline
93b333f (HEAD -> master) regular update
1ae20f3 git revert practice
9abc93a coacoac
```

3. We revert the latest commit using `git revert HEAD` (revert the latest change, and then commit), adding the option `--no-edit` to skip the commit message editor (getting the default revert message):

```
DELL@Dark-THunDeR MINGW64 /d/myproject (master)
$ git revert HEAD --no-edit
[master 23fde81] Revert "regular update"
Date: Mon May 23 03:03:52 2022 +0530
1 file changed, 39 insertions(+)
create mode 100644 practice3_awd.html
```

4. Now check the **log** again

```
DELL@Dark-THunDeR MINGW64 /d/myproject (master)
$ git log --oneline
23fde81 (HEAD -> master) Revert "regular update"
93b333f regular update
1ae20f3 git revert practice
9abc93a coacoac
```

Task 2

Practical No. 01

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

Distributed Repository:

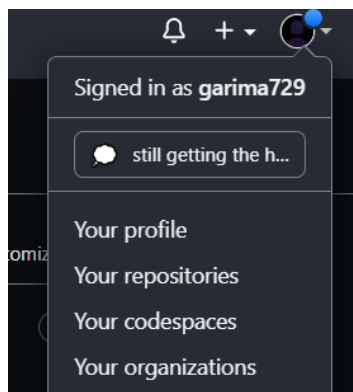
In centralized approach, there is a single source or a central copy of your repository. You have to grant permission to other team members/ add them as collaborators to this central repository, for them to start contributing to your central repository.

Whereas in distributed approach everyone clones ‘a copy of the source repository’ also known as ‘forking’. In this approach, all the collaborators will ‘fork’ the source repo. Upon forking every member gets their own copy of the source repository in their GitHub account. Collaborators can then go ahead and clone their ‘forked repository’ on their local machines.

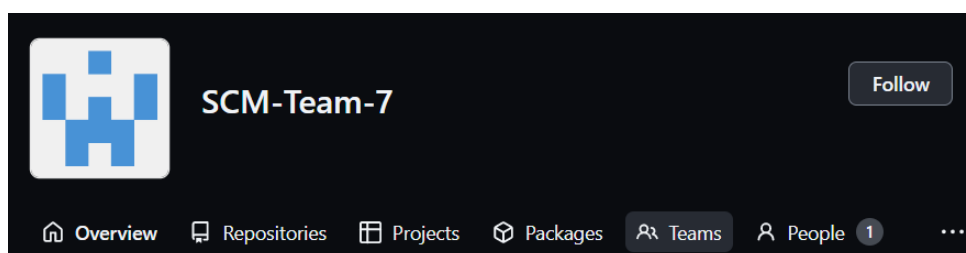
Creating a team

In github, you can create independent or nested teams to manage repository permissions and mentions for groups of people. Only organization owners and maintainers of a parent team can create a new child team under a parent. Owners can also restrict creation permissions for all teams in an organization.

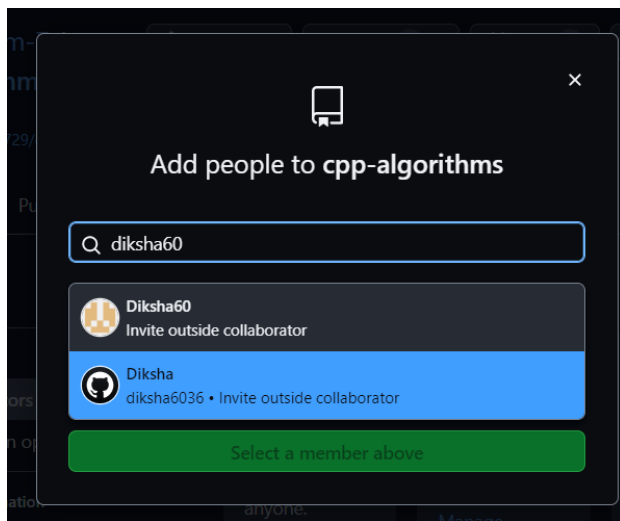
1. In the top right corner of GitHub.com, click your profile photo, then click Your organizations.



2. Under your organization, click on Teams.



3. Create a new team, send invite to the required members and give the team access to organization repositories.

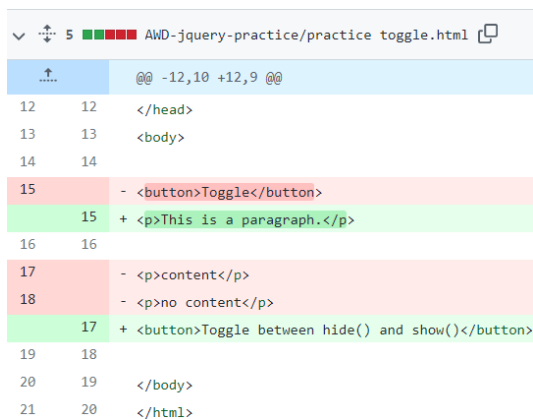


Practical No. 02

Aim: Open and close a pull request

Pull Requests are the heart of GitHub collaboration. With a pull request you are proposing that your changes should be merged (pulled in) with the master.

Pull requests show content differences, changes, additions, and subtractions in colors (green and red).



The screenshot shows a diff view for the file 'toggle.html'. The changes are as follows:

Line	Original	Changes
12	</head>	
13	<body>	
14		
15	- <button>Toggle</button>	
15		+ <p>This is a paragraph.</p>
16		
17	- <p>content</p>	
18	- <p>no content</p>	
17		+ <button>Toggle between hide() and show()</button>
19		
20	</body>	
21	</html>	

1. As soon as you have a commit, you can open a pull request.

☐ Commit directly to the `main` branch.

☒ Create a new branch for this commit and start a pull request. [Learn more about pull requests.](#)

`toggle jquery (updated)` Will be created as `toggle-jquery-(updated)`

[Propose changes](#)

2. You can merge any changes into the master by clicking a “Merge pull request” button.

☒ This branch has no conflicts with the base branch

Merging can be performed automatically.

[Merge pull request](#) You can also [open this in GitHub Desktop](#) or view [command line instructions](#).

3. After merging you can delete the branch by clicking a “Delete branch” button.

Pull request successfully merged and closed

You're all set—the `readme-edits` branch can be safely deleted.

 Delete branch

4. If you choose “close pull request” the pull request will be closed without attempting to merge the source branch into the destination branch. This option does not provide a way to delete the source branch as part of closing the pull request, but you can do it yourself after the request is closed.

Leave a comment

Attach files by dragging & dropping, selecting or pasting them.

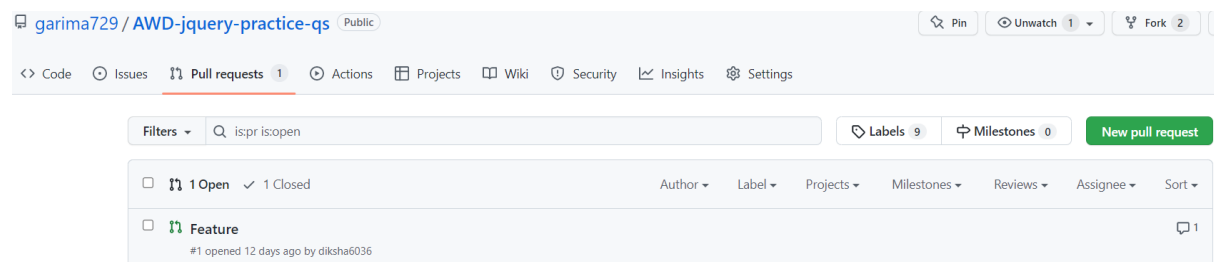
 Close pull request

Comment

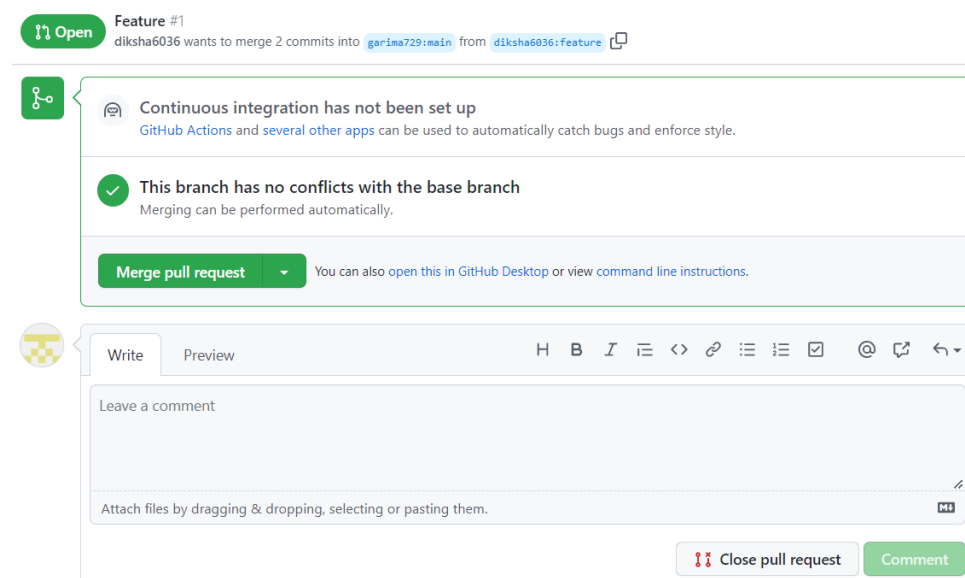
Practical No. 03

Aim: Each project member shall create a pull request on a team members repo and close pull requests generated by team members on own Repo as a maintainer.

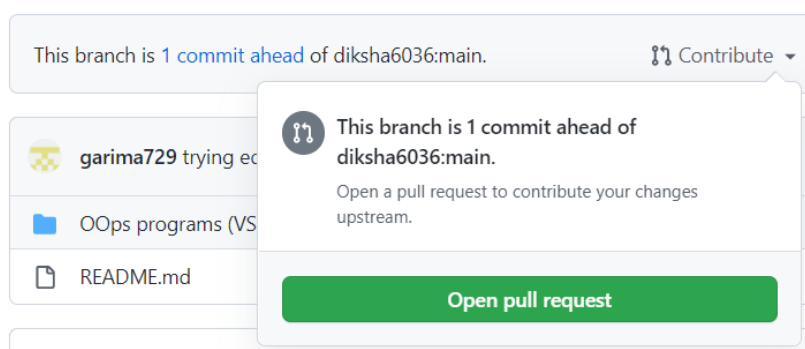
1. Here a pull request has been opened by Diksha (team member).



2. Now I, as the maintainer of the repo, can merge or close the pull request.



3. Here I have opened a pull request on my team member's repo.



4. It can be seen under the “pull requests” section. Now the maintainer can either merge or close it.

diksha6036 / Cpp-practise-Questions Public

Code Issues Pull requests 3 Actions Projects Wiki Security

Filters

🔗 3 Open ✓ 0 Closed

🔗 **task 1.2 pull request**
#3 opened 3 minutes ago by garima729

🔗 **Editor**
#2 opened 2 days ago by Cjusroy

🔗 **Feature**
#1 opened 2 days ago by garima729

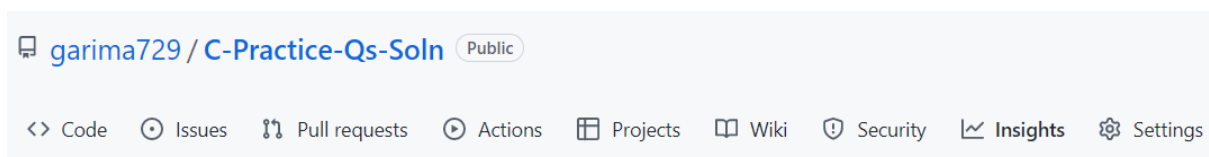
Practical No. 04

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.

Accessing the network graph

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 Network.

