Lab Manual for Git Lab

Prepared By Naveen Kumar B

Course Contents :

1. **Git Basics.**

2. **Git Installation**

3. **Git Basic Commands**

4. **Experiments**

    1. **Setting Up and Basic Commands**

Initialize a new Git repository in a directory. Create a new file and add it to the staging area and commit the changes with an appropriate commit message.

    2. **Creating and Managing Branches**

Create a new branch named "feature-branch." Switch to the "master" branch. Merge the "feature-branch" into "master."

    3. **Creating and Managing Branches**

Write the commands to stash your changes, switch branches, and then apply the stashed changes.

    4. **Collaboration and Remote Repositories**

Clone a remote Git repository to your local machine.

    5. **Collaboration and Remote Repositories**

Fetch the latest changes from a remote repository and rebase your local branch onto the updated remote branch.

    6. **Collaboration and Remote Repositories**

Write the command to merge "feature-branch" into "master" while providing a custom commit message for the merge.

7. **Git Tags and Releases**

Write the command to create a lightweight Git tag named "v1.0" for a commit in your local repository.

8. **Advanced Git Operations**

Write the command to cherry-pick a range of commits from "source-branch" to the

current.

9. **Analysing and Changing Git History**

Given a commit ID, how would you use Git to view the details of that specific

commit, including the author, date, and commit message?

10. **Analysing and Changing Git History**

Write the command to list all commits made by the author "JohnDoe" between "2023-01 01" and "2023-12-31."

11. **Analysing and Changing Git History**

Write the command to display the last five commits in the repository's history.

12. **Analysing and Changing Git History**

Write the command to undo the changes introduced by the commit with the ID "abc123".

Git Basics

Git is a distributed version control system (VCS) that is widely used for tracking changes in source code during software development. It was created by Linus Torvalds in 2005 and has since become the de facto standard for version control in the software development industry.

Git allows multiple developers to collaborate on a project by providing a history of changes, facilitating the tracking of who made what changes and when. Here are some key concepts and features of Git:

1. **Repository (Repo)**: A Git repository is a directory or storage location where your project's files and version history are stored. There can be a local repository on your computer and remote repositories on servers.

2. **Commits:** In Git, a commit is a snapshot of your project at a particular point in time.Each commit includes a unique identifier, a message describing the changes, and a reference to the previous commit.

3. **Branches:** Branches in Git allow you to work on different features or parts of your project simultaneously without affecting the main development line (usually called the "master" branch). Branches make it easy to experiment, develop new features, and merge changes back into the main branch when they are ready.

4**. Pull Requests** (PRs): In Git-based collaboration workflows, such as GitHub or GitLab, pull requests are a way for developers to propose changes and have them reviewed by their peers. This is a common practice for open-source and team-based projects.

5. **Merging**: Merging involves combining changes from one branch (or multiple branches) into another. When a branch's changes are ready to be incorporated into the main branch, you can merge them.

6. **Remote Repositories**: Remote repositories are copies of your project stored on a different server. Developers can collaborate by pushing their changes to a remote repository and pulling changes from it. Common remote repository hosting services include GitHub, GitLab, and Bitbucket.

7. **Cloning:** Cloning is the process of creating a copy of a remote repository on your local machine. This allows you to work on the project and make changes locally.

8**. Forking**: Forking is a way to create your copy of a repository, typically on a hosting platform like GitHub. You can make changes to your fork without affecting the original project and later create pull requests to contribute your changes back to the original repository. Git is known for its efficiency, flexibility, and ability to handle both small and large-scale software projects. It is used not only for software development but also for managing and tracking changes in various types of text-based files, including documentation and configuration files. Learning Git is essential for modern software development and collaboration.

Why we need git?

Git is an essential tool in software development and for many other collaborative and version controlled tasks. Here are some key reasons why Git is crucial:

1. **Version Control:** Git allows you to track changes in your project's files over time. It provides a complete history of all changes, making it easy to understand what was done, when it was done, and who made the changes. This is invaluable for debugging, auditing, and collaboration.

2. **Collaboration:** Git enables multiple developers to work on the same project

simultaneously without interfering with each other's work. It provides mechanisms for merging changes made by different contributors and resolving conflicts when they occur.

3**. Branching**: Git supports branching, which allows developers to create isolated

environments for developing new features or fixing bugs. This is essential for managing complex software projects and experimenting with new ideas without affecting the main codebase.

4. **Distributed Development:** Git is a distributed version control system, meaning that every developer has a complete copy of the project's history on their local machine. This provides redundancy, facilitates offline work, and reduces the reliance on a central server.

5. **Backup and Recovery:** With Git, your project's history is distributed across multiple locations, including local and remote repositories. This provides redundancy and makes it easy to recover from accidental data loss or system failures.

6**. Code Review:** Git-based platforms like GitHub, GitLab, and Bitbucket provide tools for code review and collaboration. Developers can propose changes, comment on code, and discuss improvements, making it easier to maintain code quality.

7. **Open Source and Community Development:** Git has become the standard for open

source software development. It allows anyone to fork a project, make contributions, and create pull requests, which makes it easy for communities of developers to collaborate on a single codebase.

8. **Efficiency:** Git is designed to be fast and efficient. It only stores the changes made to files, rather than entire file copies, which results in small repository sizes and faster operations.

9. **History and Documentation:** Git's commit history and commit messages serve as a

form of documentation. It's easier to understand the context and reasoning behind a change by looking at the commit history and associated messages.

10. **Customizability:** Git is highly configurable and extensible. You can set up hooks and scripts to automate workflows, enforce coding standards, and integrate with various tools.

Git Life Cycle

The Git lifecycle refers to the typical sequence of actions and steps you take when using Git to manage your source code and collaborate with others. Here's an overview of the Git lifecycle:

1. **Initializing a Repository:**

To start using Git, you typically initialize a new repository (or repo) in your project directory. This is done with the command git init.

2. **Working Directory:**

Your project files exist in the working directory. These are the files you are actively working on.

3. **Staging:**

Before you commit changes, you need to stage them. Staging allows you to select which changes you want to include in the next commit. You use the git add command to stage changes selectively or all at once with git add ..

4. **Committing:**

After you've staged your changes, you commit them with a message explaining what you've done. Commits create snapshots of your project at that point in time. You use the git commit command to make commits, like git commit -m "Add new feature".

5**. Local Repository:**

Commits are stored in your local repository. Your project's version history is preserved there.

6**. Branching:**

Git encourages branching for development. You can create branches to work on new features, bug fixes, or experiments without affecting the main codebase.

Use the git branch and git checkout commands for branching.

7**. Merging:**

After you've completed work in a branch and want to integrate it into the main

codebase, you perform a merge. Merging combines the changes from one branch into another. Use the git merge command.

8. **Remote Repository:**

For collaboration, you can work with remote repositories hosted on servers like

GitHub, GitLab, or Bitbucket. These repositories serve as a central hub for

sharing code.

9. **Pushing:**

To share your local commits with a remote repository, you push them using the

git push command. This updates the remote repository with your changes.

10. **Pulling:**

To get changes made by others in the remote repository, you pull them to your

local repository with the git pull command. This ensures that your local copy is

up to date.

11**. Conflict Resolution:**

Conflicts can occur when multiple people make changes to the same part of a

file. Git will inform you of conflicts, and you must resolve them by editing the

affected files manually.

12. **Collaboration:**

Developers can collaborate by pushing, pulling, and making pull requests in a

shared remote repository. Collaboration tools like pull requests are commonly

used on platforms like GitHub and GitLab.

13. **Tagging and Releases:**

You can create tags to mark specific points in the project's history, such as

version releases. Tags are useful for identifying significant milestones.

14**. Continuous Cycle:**

The Git lifecycle continues as you repeat these steps over time to manage the

ongoing development and evolution of your project. This cycle supports

collaborative and agile software development.

The Git lifecycle allows for effective version control, collaboration, and the management of complex software projects. It provides a structured approach to tracking and sharing changes, enabling multiple developers to work together on a project with minimal conflicts and a clear history of changes.

Git Installation

To install Git on your computer, you can follow the steps for your specific operating system:

1**. Installing Git on Windows:**

a. **Using Git for Windows (Git Bash):**

• Go to the official Git for Windows website: https://gitforwindows.org/

• Download the latest version of Git for Windows.

• Run the installer and follow the installation steps. You can choose the default settings for most options.

b. **Using GitHub Desktop (Optional):**

• If you prefer a graphical user interface (GUI) for Git, you can also install GitHub Desktop, which includes Git. Download it from https://desktop.github.com/ and follow the installation instructions.

2. **Installing Git from Source (Advanced):**

• If you prefer to compile Git from source, you can download the source code from the official Git website (https://git-scm.com/downloads) and follow the compilation instructions provided there. This is usually only necessary for advanced users.

After installation, you can open a terminal or command prompt and verify that Git is correctly installed by running the following command:

$ **git --version**

If Git is installed successfully, you will see the Git version displayed in the terminal. You can now start using Git for version control and collaborate on software development projects.

How to Configure the Git?

Configuring Git involves setting up your identity (your name and email), customizing Git options, and configuring your remote repositories. Git has three levels of configuration: system, global, and repository-specific. Here's how you can configure Git at each level:

1. **System Configuration:**

• System-level configuration affects all users on your computer. It is typically used for site-specific configurations and is stored in the /etc/gitconfig file.

To set system-level configuration, you can use the git config command with the --system flag (usually requires administrator privileges). For example:

$ **git config --system user.name "Your Name"**

$ **git config --system user.email "your.email@example.com"**

1. **Global Configuration:**

• Global configuration is specific to your user account and applies to all Git repositories on your computer. This is where you usually set your name and email.

To set global configuration, you can use the git config command with the --global flag. For example:

$ **git config --global user.name "Your Name"**

$ **git config --global user.email "your.email@example.com"**

You can also view your global Git configuration by using:

$ **git config --global --list**

Expt No: 01

Initialize a new Git repository in a directory. Create a new file and add it to the staging area and commit the changes with an appropriate commit message.

Make your own directory: mkdir "directory name" (eg: **mkdir naveen**)

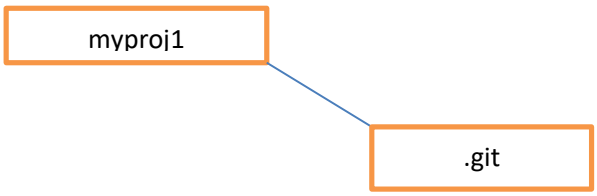
and always first change your working directory to your own created directory

cd "directory name" (eg: **cd naveen**)

1. create one directory with name myproj1: **mkdir myproj1**
2. go to created directory: **cd myproj1**
3. make directory myproj1 as local repo by initializing

new .git repository under myproj1: git init

see fig below:



4. create one file with name new.txt: **notepad new.txt**
5. check git status: **git status**

new.txt untracked (red color)

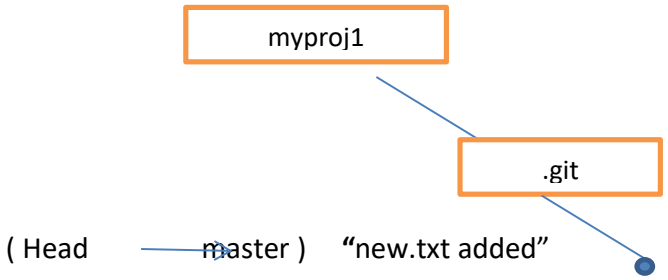
6. To track and record that file by .git add that file to staging area:

git add . or git add new.txt

7. check git status: **git status**

new.txt tracked (green color)

8. commit your updated .git: **git commit -m “new.txt added”**



9. To see commit history log: **git log --all or git log --oneline**

See output (write output in the record)

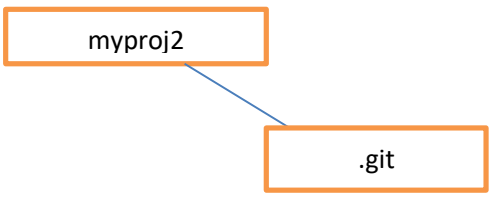
Expt No 02:

Create a new branch named "feature-branch." Switch to the "master" branch.

Merge the "feature-branch" into "master."

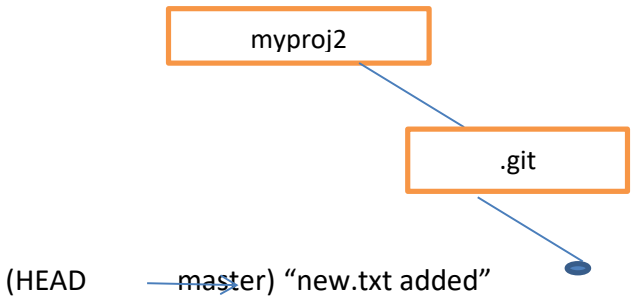
You can work for new problems by creating a new branch in your .git directory without affecting your master branch, after solution to the new problem is implemented in new branch, added it to staging area , switch to your master branch and make commit of updated .git.

1. **cd “your directory”** (eg: cd naveen)
2. **mkdir myproj2**
3. initialize git repository: **git init**

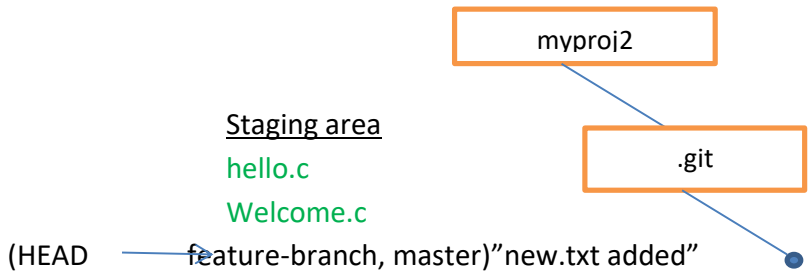


4. Create one file named new.txt: **notepad new.txt** (edit it , save it and close notepad)
5. To track and record that file by .git add that file to staging area:
   git add . or git add new.txt

6. commit your updated .git: **git commit -m “new.txt added”**

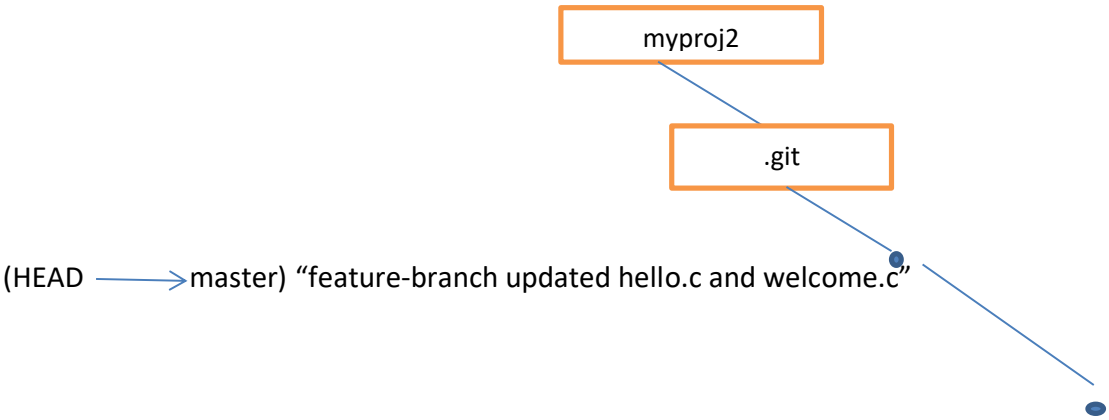


7. Create a new branch named “feature-branch”: **git branch feature-branch**
8. Switch to “feature-branch”: **git checkout feature-branch**
9. Create hello.c file: **notepad hello.c** (edit it , save it and close notepad)
10. Add hello.c to staging area: **git add . or git add hello.c**
11. Create welcome.c file: **notepad welcome.c** (edit it , save it and close notepad)
12. Add welcome.c to staging area: **git add . or git add welcome.c**
13. Check commit history: **git log --all**

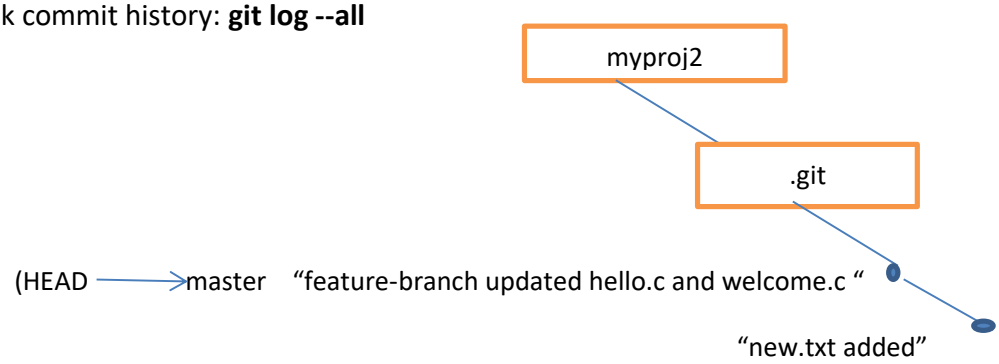


14. Switch back to “master” branch: **git checkout master**
15. Merge “feature-branch” into “master”: **git merge feature-branch**
16. Commit all updates in “master”:
    git commit -m “feature-branch updated hello.c and welcome.c”

17. Check commit history: **git log --all**



18. Remove “feature-branch”: **git branch -d feature-branch**
19. Check commit history: **git log --all**



Expt No 03:

Write the commands to stash your changes, switch branches, and then apply the stashed changes.

git stash is a command to save the changes or updates in a temporary storage stack data structure. We push changes into stack by push operation, whenever if you want to remove pushed changes from stack by pop operation.

Situation, sometime you are working for problem to implement wishing messages in “greetings.c”. Say first time you have implemented “greetings.c” to print “Hello World” message and you have committed. Again you may require opening same file “geetings.c” to implement “Welcome to git lab” and “Good Morning” messages. To update the same file “greetings.c’” better you plan to create a new branch “feature” switch to that branch to do updates to “greetings.c” without affecting to your old commits.

We do this Expt in two ways:

A. By using git stash (in general)
B. By using git stash push –m “message”

A.

1. **cd “your directory”** (eg: cd naveen)
2. **mkdir myproj3**
3. **cd myproj3**
4. **git init**
5. **notepad greetings.c**

```
int main()
{
    printf(“Hello World\n”);
    return 0;
}
```

(save it , close notpad)

6. **git add . or git add greetings.c**
7. **git commit –m “Hello msg impl”**
8. create a new “feature” branch: **git branch feature**
9. Switch to new branch “feature”: **git checkout feature**
10. open same greetings.c file: **notepad greetings.c**

```
int main()
{
    printf(“Hello World\n”);
    printf(“Welcome to git lab”);
    return 0;
}
```

(save it and close notepad)

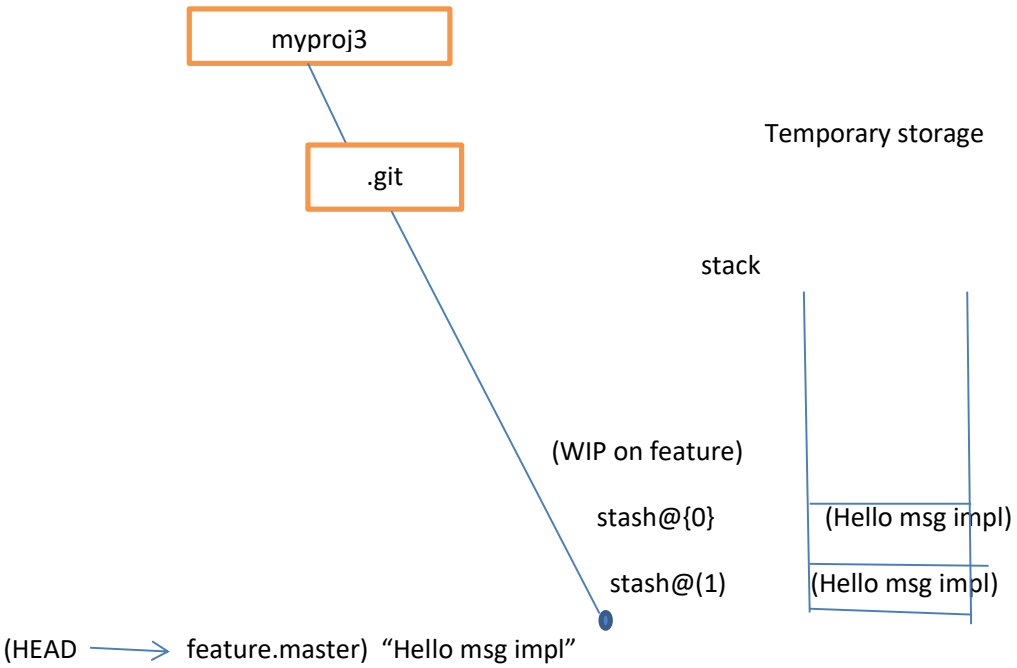
11. Add changes to staging area: **git add . or git add greetings.c**

12. save the changes by stashing: **git stash**

13. Again open same file greetings.c: **notepad greetings.c**

```
int main()
{
    printf("Hello World\n");
    printf("Good Morning\n");
    return 0;
}
```
(save it and close notepad)

14. Add changes to staging area: **git add . or git add greetings.c**
15. Save the changes by stashing: **git stash**



16. Check to see stash list: **git stash list**
17. Switch to the "master" branch : **git checkout master**
18. Apply the stash changes to "master": **git stash apply**

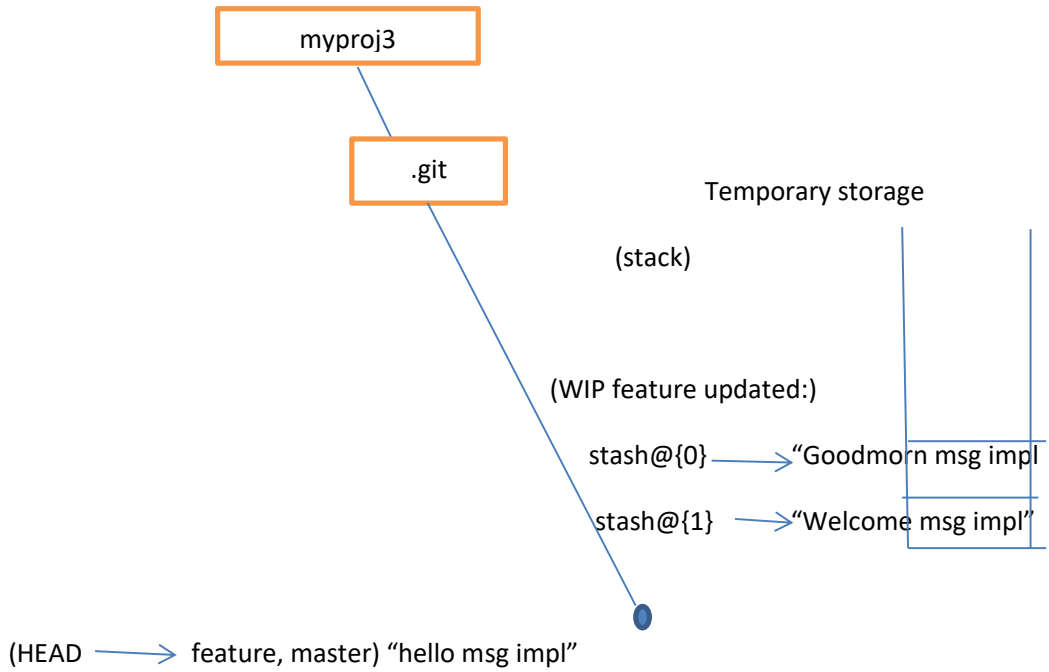
This command applies all stashed changes to master but shows "changes not staged For commit" and greetings.c file modified in red color. (stack content not cleared, check it by: **git stash list**)

19. Add to staging area: **git add . or git add greetings.c**

20. Commit the changes: **git commit -m “feature updated by welcome and good morn message”**
21. Clear stack stashes by : **git stash clear**
22. check commit history log: **git log --all** (see output)

B.

1. Follow same of method A from 1 to 11
2. point 12, change like this: **git stash push -m “WIP feature updated : welcome msg impl”**
3. do same as like point 13 and 14
4. point 15, change like this: **git stash push -m “WIP feature updated: Good morn msg impl”**



5. check to see stash list: **git stash list**
6. Switch to “master” branch: **git checkout master**
7. Apply the stashed changes and remove stack stashes**: git stash apply**

   This command applies all stashed changes to master but shows “changes not staged

   For commit” and greetings.c file modified in red color. (stack content not cleared,

   check it by: **git stash list**)

8. Add it to staging area: **git add . or git add greetings.c**

9. Commit the changes in “master”:

   git commit -m “feature updated: welcome and Good morn msg impl”

10. Check commit history: **git log --all** (see output)

Expt No 04:

Clone a remote Git repository to your local machine.

Cloning means getting or downloading remote git repository from github.com server to your local machine.

To create remote git repository at github.com server you must first create PAT (personnel access token).

To create PAT follow these steps:

1. login to **github.com** with your registered email-id and password
2. click on **profile** account icon in your github home page and choose **settings**
3. click on **developer settings** in left side settings window
4. Choose **personnel access token (classic)** in generate new token dropdown window
5. Give **PAT name** in the text box and make **NO Expiration** in dropdown window
6. Check on check box **Repo, workflow, write packages** and **delete packages**
7. Finally click on **generate new token** button.

The above steps gives you PAT key , copy the key and save it in a text file under your own directory.

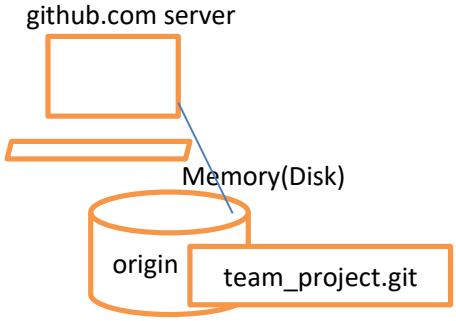
To create repository follow these steps:

1. Go to your home page in **github.com**
2. Click on **create repository** button
3. Give **repository name** in the text box (say : team_project)

The url link (url is **uniform resource locator**) of remote git repository is shown below:

https://github.com/your username/team_project.git

you have created an empty remote git repository



This remote git repository is identified by default name : **origin**
and default branch name for origin(team_project) is **main**
Cloning remote git repository to your local machine:

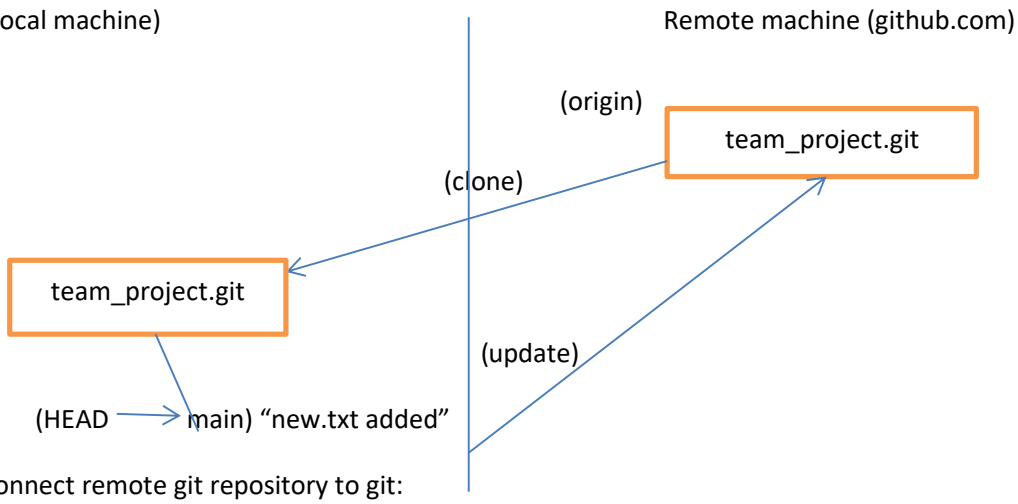
1. Go to your own directory : **cd “your directory”** (eg: cd naveen)

2. Clone git repository: **git clone "url link of remote git repository"**
   (eg: git clone https://github.com/navinaveen-b/team_project.git)
   When we clone this, git says you have cloned an empty remote git repository
3. List the content of your own directory: **ls**

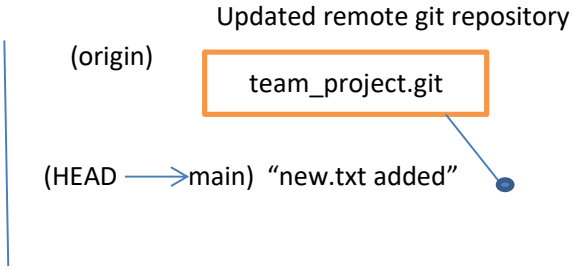
   You will see team_project folder under your own directory

Updating remote git repository directly in your local machine

1. Go to cloned team_project folder: **cd team_project**
2. You are working in remote git repository in your local machine. So, your branch name: "main"
   Update remote git repository branch main by creating one file and committing it.
3. Create one file named new.txt: **notepad new.txt** (edit it, save it and close notepad)
4. Add to staging area: **git add . or git add new.txt**
5. Commit changes: **git commit -m "new.txt added"**



6. Connect remote git repository to git:
   git remote add origin "url link of remote git repository"
   (eg: git remote add origin https://github.com/navinaveen-b/team_projet.git)
   The above command Error remote origin already exists, because you are working in team_project.git (igmore the Error)
7. Update the remote git repository:
   git push -u origin main



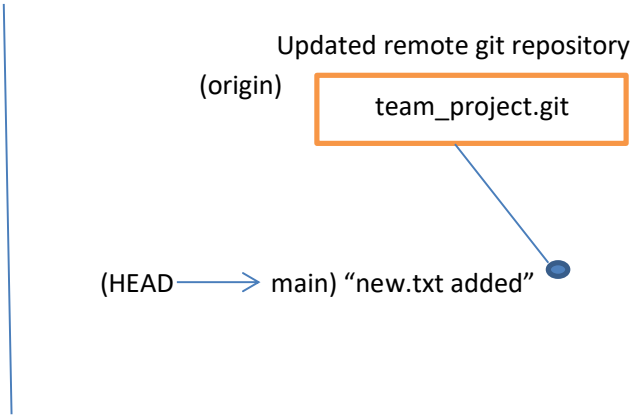
Expt No 05:

Fetch the latest changes from a remote repository and rebase your local branch onto the updated remote branch.

Rebasing means applying all your local repository commits on top of the remote repository branch by doing this we creating new base level of our branch.

First: Assume you have updated remote git repository team_project.git accordingly as from Expt No 04.

The updated remote git repository as ahown below

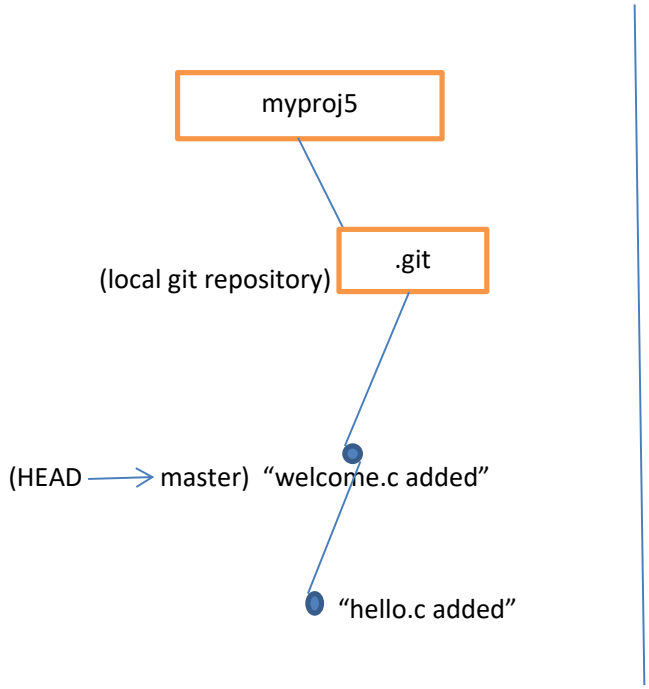


Second: you start working in your local git repository and make some commits by adding some files.

Say, hello.c and welcome.c

1. Go to your own directory: **cd "your directory"**
   (eg: cd naveen)
2. **mkdir myproj5**
3. **cd myproj5**
4. **git init**
5. **notepad hello.c** (edit it, save it and close notepad)
6. **git add . or git add hello.c**
7. **git commit -m "hello.c added"**
8. **notepad welcome.c** (edit it, save it and close notepad)
9. **git add . or git add welcome.c**
10. **git commit -m "welcome.c added"**

Your local git repository as shown below:

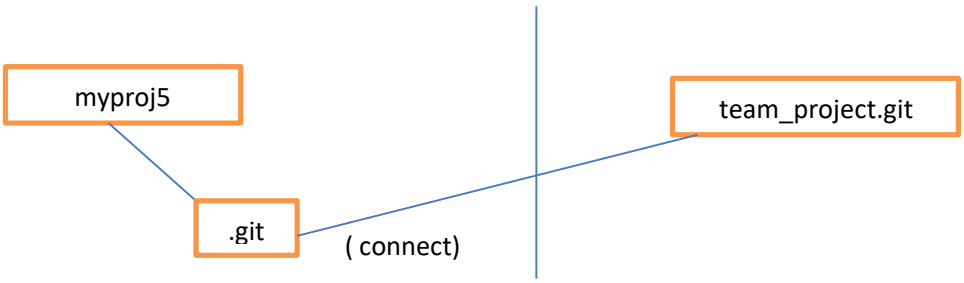


Fetch the remote git repository under your local repository

First connect remote git repository to the git

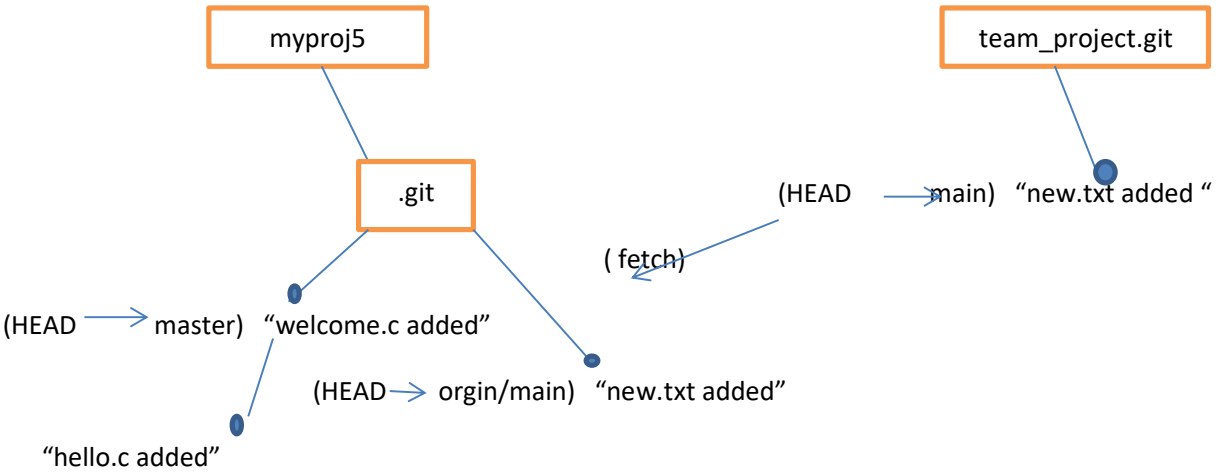
git remote add origin "url link of remote git repository"

(eg: git remote add origin https://github.com/navinaveen-b/team_project.git)



Fetch the remote branch: **git fetch origin** **(**it shows you fetched main identified as origin/main)

Now your local repository have two branches "master" and "origin/main" is as shown below:

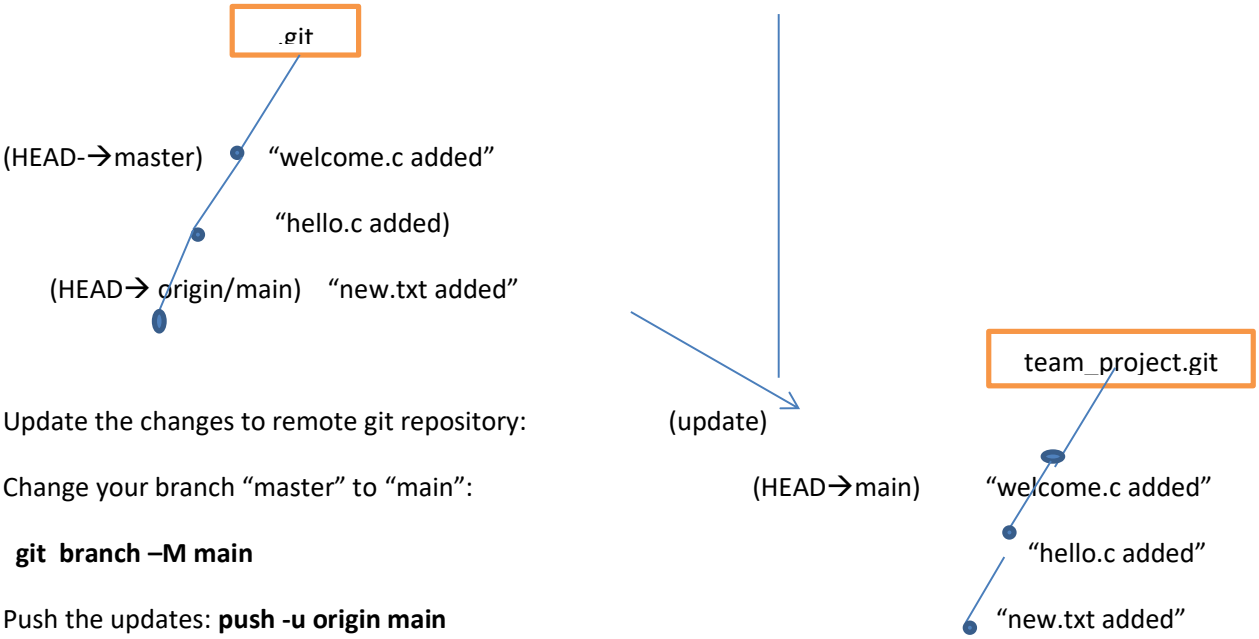


Rebase your local repository commits on to remote branch

git rebase orgin/main

check the commit history log : **git log --all**

After rebasing your branch tree is as shown below:



Update the changes to remote git repository:

Change your branch "master" to "main":

git branch –M main

Push the updates: **push -u origin main**

Expt No 06:

Write the command to merge "feature-branch" into "master" while providing a custom commit message for the merge.

Whenever you solving new problems in a new branch (feature-branch),

you may solved printing of

1. hello message by creating “hello.c” and committed with “feature-branch updated”
2. welcome message by creating “welcome.c” and committed with “feature-branch updated”

When you merge this “feature-branch” into “master” branch, to make identified in “master” while merging “feature-branch” we want to merge “feature-branch” with custom commit message.

First, start working in “master”

1. Go to your own directory: **cd “your own directory”** (eg: cd naveen)
2. **mkdir myproj6**
3. **cd myproj6**
4. **git init**
5. **notepad new.txt** (edit it, save it and close notepad)
6. **git add . or git add new.txt**
7. **git commit -m “new.txt added”**

second, start working in feature-branch

1. create new branch “feature-branch”: **git branch feature-branch**
2. Switch to “feature-branch” : **git checkout feature-branch**
3. **notepad hello.c** (edit it , save it and close notepad)
4. **git add . or git add hello.c**
5. **notepad welcome.c** (edit it, save it and close notepad)
6. **git add . or git add welcome.c**
7. **git commit -m “feature-branch: updated”**

Third, Merge “feature-branch” into “master” with custom commit message

1. Switch to master: **git checkout master**
2. Add changes to staging area: **git add .**
3. Commit the changes in “master” by merging “feature-branch” with custom commit message: **git merge –no-ff feature-branch -m “feature-branch updated adding hello.c and welcome.c”**
4. Check commit history log**:** **git log --all** (see output)

Expt No 07:

Write the command to create a lightweight Git tag named "v1.0" for a commit in your local repository.

Assume you may be implemented some programs for your project. Say, hello.c and welcome.c and committed. Assume you have implemented “welcome.c” by using older compiler version 1.0 using print function like this: print(“Welcome to Git Lab”); so, welcome.c can be compiled using compiler version 1.0. SO, you to give a tag for the welcome.c commit as : V1.0.

1. Go to your own directory: **cd “your own directory”** (eg: cd naveen)
2. Create project folder: **mkdir myproj7**
3. Move to project folder: **cd myproj7**
4. Initialize git directory: **git init**
5. Create “hello.c” file: **notepad hello.c**

```
int main()
{
    printf(“Hello World\n”);
 }
```
(save it and close notepad)

6. Add “hello.c” to staging area: **git add . or git add hello.c**
7. Commit the changes: **git commit -m “hello.c added”**
8. Create “welcome.c” file: **notepad welcome.c**

```
int main()
{
    print(“Welcome to Git Lab\n”);
 }
```
(save it and close notepad);

9. Add “welcome.c” to staging area: **git add . or git add welcome.c**
10. Commit the changes: **git commit -m “welcome.c added”**
11. Check to see all commit history log: **git log --oneline**

    Output shown below:

    91bb7dd (HEAD -> master) welcome.c added

    ec21a6d hello.c added

(Green letters shown above are called ID number for commits)

12. Now apply tag for commit ID “91bb7dd” as V1.0: **git tag V1.0 91bb7dd**
13. Verify the tag by: **git tag** (see output)
14. Check to see to which commit tag points to: **git show V1.0** (see output)

    (output shows all the details of commit)

Expt No 08:

Write the command to cherry-pick a range of commits from "source-branch" to the current branch.

git cherry-pick <start-commit>^..<end-commit>

git cherry-pick command picks range of commits from start commit-id to end commit-id from one branch to another branch.

<start-commit> is starting commit-id and <end-commit> is ending commit-id.

Assume, your source branch is "master" and current branch is "feature"

First start working in master

1. Go to your own directory: **cd "your directory"** (eg: cd naveen)
2. Create project folder: **mkdir myproj8**
3. Move to project folder: **cd myproj8**
4. Initialize git directory: **git init**
5. Create a file named new.txt: **notepad new.txt** (edit it, save it and close notepad)
6. Add to staging area: **git add . or git add new.txt**
7. Commit the changes: **git commit -m "new.txt added"**
8. Check commit history log in "master": **git log --oneline** (see output)

output shows as shown below:

34d79df (HEAD -> master) new.txt added

Second create branch "feature" and start working there

1. Create "feature" branch: **git branch feature**
2. Switch to "feature" branch: **git checkout feature**
3. Create file named mytext1.txt: **notepad mytext1.txt** (edit it , save it and close notepad)
4. Add it to staging area: **git add . or git add mytext1.txt**
5. Commit the changes: **git commit -m "mytext1.txt added"**
   Repeat 3,4 and 5 to create files mytext2.txt,mytext3.txt,mytext4.txt and mytext5.txt and commit.
6. Check commit history log in "feature": **git log --oneline**
   Output shows as shown below:

9adbe88 (HEAD -> feature) mytext5.txt added
78e3fbe mytext4.txt added
23dbc7a mytext3.txt added
dbce617 mytext2.txt added

(These 5 commits is from "feature")

6647dda mytext1.txt added
34d79df (master) new.txt added (This commit is from “master”)

Third switch to “master” and start working there

1. Switch to “master”: **git checkout master**
2. To cherry-pick range of commits as shown above in red color from “feature” to “master” branch:
   git cherry-pick dbce617^..78e3fbe
3. Check now commit history log in “master”: **git log --oneline**

   output shows as shown below:

   7273594(HEAD -> master) mytext4.txt added
   3e1717e mytext3.txt added
   8256ff7 mytext2.txt added
   34d79df new.txt added

Expt No 09:

Given a commit ID, how would you use Git to view the details of that specific commit, including the author, date, and commit message?

When user committed any changes in the git repository , git records or stores details like

Author : which user as committed . that is user configured to the git

(eg: git config --global user.name “navinaveen-b”

Date: when was committed

Commit Message: what commit message is given

By knowing commit-id we can view the above details. All commit-id can be viewed by the commad

git log –oneline

1. Go to your own directory: **cd “your directory”** (eg: cd naveen)
2. Create project folder: **mkdir myproj9**
3. Move to project folder: **cd myproj9**
4. Initialize git repository: **git init**
5. Cerate file “hello.c”: **notepad hello.c** (edit it , save it and close notepad)
6. Add it to staging area: **git add . or git add hello.c**
7. Commit the changes: **git commit -m “hello.c added”**

8. Create file “welcome.c”: **notepad welcome.c** (edit it, save it and close notepad)
9. Add it to staging area: **git add . or git add welcome.c**
10. Commit the changes: **git commit -m “welcome.c added”**
11. Check the commit history log: **git log --oneline**

    Output shows as shown below:
    a7eab84 (HEAD -> master) welcome.c added
    db8900c hello.c added
12. To see the details of commit-id db8900c: **git show db8900c**

    Output shows as shown below:

    commit db8900c1374ffe2ea7dc0851c8e5fbf15989825a
    Author: navibaveen-b <navinaveen_b@yahoo.co.in>
    Date: Wed Nov 12 15:23:24 2025 +0530

        hello.c added

    git a/hello.c b/hello.c
    new file mode 100644
    index 0000000..b8236ec
    --- /dev/null
    +++ b/hello.c
    @@ -0,0 +1 @@
    +printf("Hello World\\n");

    \\ No newline at end of file

13. If you want only metadata: **git show --no-patch db8900c**

    Output shows as shown below:

    commit db8900c1374ffe2ea7dc0851c8e5fbf15989825a
    Author: navinaveen-b <navinaveen_b@yahoo.co.in>
    Date: Wed Nov 12 15:23:24 2025 +0530

    hello.c added

Expt No 10:

Write the command to list all commits made by the author "JohnDoe" between "2023 -01-01" and "2023-12-31."

The question is on author “JohnDoe” is the user name , assume it that your user name which you configured to git. (eg: my user name is navinaveen-b and I am the author). The question is to find all the commits made by you between the dates 20239Y-01M-01D to 2023Y-12M-31D.

Assume you have worked for the project “myproj10” for the period 2025-11-01 to 2025-11-30 of November, during this period you may committed several changes to your project. Now suppose if you want find the commits made by you in the 2nd week of November 2025 (ie 2025-11-07 to 2025-11-14).

1. Go to your directory: **cd “your directory”** (eg: cd naveen)
2. Create project folder “myproj10”: **mkdir myproj10**
3. Move to project folder: **cd myproj10**
4. Initialize git repository: **git init**
5. Create and commit one file “hello.c” on 2025-11-14:
   notepad hello.c (edit it , save it and close notepad)
6. Commit the changes: **git commit -m “hello.c added”**
7. Add it to staging area: **git add . or git add hello.c**
8. Create and commit one more file “welcome.c” on 2025-11-21:
   notepad welcome.c (edit it , save it and close notepad)
9. Add it to staging area: **git add . or git add welcome.c**
10. Commit the changes: **git commit -m “welcome.c added”**
11. Check commit history log: **git log --all**

    Output shows as shown below:

    commit a7eab84650a7eec545059d39a03575368cbcd3e1
    Author: navinaveen-b navinaveen_b@yahoo.co.in
    Date: Fri Nov 07 15:24:23 2025 +0530

      welcome.c added

    commit db8900c1374ffe2ea7dc0851c8e5fbf15989825a
    Author: navinaveen-b navinaveen_b@yahoo.co.in
    Date: Fri Nov 21 15:23:24 2025 +0530

      hello.c added
12. Now, check commits made by author “navinaveen-b” since from 2025-11-07 to 2025-11-21:

git log --author=”navinaveen-b” --since= “2025-11-07” --until=”2025-11-21”

output shows as shown below:

commit a7eab84650a7eec545059d39a03575368cbcd3e1
Author: navinaveen-b navinaveen_b@yahoo.co.in
Date: Fri Nov 07 15:24:23 2025 +0530

welcome.c added

commit db8900c1374ffe2ea7dc0851c8e5fbf15989825a
Author: navinaveen-b navinaveen_b@yahoo.co.in
Date: Fri Nov 21 15:23:24 2025 +0530

hello.c added

Expt No 11:

Write the command to display the last five commits in the repository's history.

This question is to display last recent 5 commits from commit history log.

1. Go to your directory: **cd “your directory”** (eg: cd naveen)
2. Create project folder: **mkdir myproj11**
3. Move to project folder: **cd myproj11**
4. Initialize git repository: **git init**
5. Create a file named “mytext1.txt” : **notepad mytext1.txt** (edit it, save it and close notepad)
6. Add it to staging area: **git add . or git add mytext1.txt**
7. Commit the changes**:** **git commit -m “mytext1.txt added”**

   Repeat 5,6 and 7 for the files mytext2.txt,mytext3.txt,mytext4.txt,mytext5.txt and mytext6.txt.
8. Check commit history log: **git log --all**

   Output shows as shown below:

   commit 94006cc4032e04dbceb8742a886edbc772455682 (HEAD -> master)
   Author: teamprojectabcd <teamprojectabcd@gmail.com>
   Date: Fri Nov 21 16:11:35 2025 +0530

   mytext6.txt added

   commit 5c254a86f635909b5818cce35f2e7730d31d0ff3
   Author: teamprojectabcd <teamprojectabcd@gmail.com>

Date: Fri Nov 21 16:11:07 2025 +0530

    mytext5.txt added

commit 36fdc1ac0156dae5bdbb2504383d7e31f72abad8
Author: teamprojectabcd <teamprojectabcd@gmail.com>
Date: Fri Nov 21 16:10:41 2025 +0530

    mytext4.txt added

commit ecdd0738a8858cf1b2a9d30420c1c1197d9b51c3
Author: teamprojectabcd <teamprojectabcd@gmail.com>
Date: Fri Nov 21 16:10:13 2025 +0530

    mytext3.txt added

commit 7fadc52abf51e741ba10ae9e2f603eacb9ad7658
Author: teamprojectabcd <teamprojectabcd@gmail.com>
Date: Fri Nov 21 16:09:37 2025 +0530

    mytext2.txt added

commit 3c8d055d45d2bf639a5adaa5141d81d7ed7513a0
Author: teamprojectabcd <teamprojectabcd@gmail.com>
Date: Fri Nov 21 16:09:10 2025 +0530

    mytext1.txt added

9. To display last recent 5 commits: **git log -n 5**

   Output shows as shown below:

commit 94006cc4032e04dbceb8742a886edbc772455682 (HEAD -> master)
Author: teamprojectabcd <teamprojectabcd@gmail.com>
Date: Fri Nov 21 16:11:35 2025 +0530

    mytext6.txt added

commit 5c254a86f635909b5818cce35f2e7730d31d0ff3
Author: teamprojectabcd <teamprojectabcd@gmail.com>

Date: Fri Nov 21 16:11:07 2025 +0530

    mytext5.txt added

commit 36fdc1ac0156dae5bdbb2504383d7e31f72abad8
Author: teamprojectabcd <teamprojectabcd@gmail.com>
Date: Fri Nov 21 16:10:41 2025 +0530

    mytext4.txt added

commit ecdd0738a8858cf1b2a9d30420c1c1197d9b51c3
Author: teamprojectabcd <teamprojectabcd@gmail.com>
Date: Fri Nov 21 16:10:13 2025 +0530

    mytext3.txt added

commit 7fadc52abf51e741ba10ae9e2f603eacb9ad7658
Author: teamprojectabcd <teamprojectabcd@gmail.com>
Date: Fri Nov 21 16:09:37 2025 +0530

    mytext2.txt added

Expt No 12:

Write the command to undo the changes introduced by the commit with the ID "abc123".

Sometimes we do implement solutions to several problems in a same implementation file, but solution implemented and committed contain bug/error. We can revert that commit by removing update changes by using git revert “commit-id” . git revert command removes wrong updates made by commit-id and creates a new commit-id to come back to correct old commit.

Assume you have two problems to print “Hello World” and “Welcome to Git Lab” messages. First you implemented “Hello World” msg and committed in a file “hello.c” by using statement printf(“Hello World\n”);. Second time you have implemented “Welcome to Git Lab” msg and committed in the same file “hello.c” by using statement print(“Welcome to Git Lab\n”);. The second update changes and commit contains a bug/error (shown in red color). So we can revert this commit by removing wrong updates and create a new commit to the proper old commit state.

1. Go to your directory: **cd "your own directory"** (eg: cd naveen)
2. Create a project folder: **mkdir myproj12**
3. Move to project folder: **cd myproj12**
4. Initialize git repository: **git init**
5. Create a file "hello.c": **notepad hello.c**

```
#include<stdio.h>
int main()
 {
     printf("Hello World\n");
     return 0;
  }
```
(save it and close notepad)

6. Add it to staging area: **git add . or git add hello.c**
7. Commit the changes: **git commit -m "hello.c: "hello msg impl"**
8. Open same file "hello.c" and update: **notepad hello.c**

```
#include<stdio.h>
int main()
{
     printf("Hello World\n");
     print("Welcome to Git Lab\n");
     return 0;
  }
```
(save it and close notepad)

(updates shown in red color)

9. Add it to staging area: **git add . or git add hello.c**
10. Commit the updated changes: **git commit -m "hello.c: Welcome msg impl"**
11. Check the commit history log: **git log --oneline**

    Output shows as shown below:

    3631796 (HEAD -> master) hello.c Welcome msg impl
    0232b4f hello.c Hello msg impl

The commit-id shown in bold contain error/bug , So we want to remove the update changes and revert back to old commit for proper state.

12. To revert the commit "3631796": **git revert 3631796**

    This command opens "hello.c" file in notepad and asks for enter revert commit message. Just do save the file and close notepad
13. Check finally commit history log: **git log --oneline**

    Output shows as shown below:

8506b19 (HEAD -> master) Revert "hello.c Welcome msg impl"
3631796 hello.c Welcome msg impl
0232b4f hello.c Hello msg impl

(New coomit-id "8506b19" in green color proper commit state removing updates of "3631796")

14. Check by opening "hello.c" file: **notepad hello.c**
(you can see the line print("Welcome to Git Lab\n") is removed)