1 How can new features be added to the main branch? What is a git conflict?

Create a new branch

git checkout -b new-feature main

work on feature

git add .

git commit -m “Implementation new feature”

push the branch

git push origin new-feature

create a pull request

review and merge

2 How do I remove a branch from GIT?

Delete local branch

git branch -d branch name

Delete remote branch

git push origin --delete branch\_name

3 What is the purpose of the git checkout command?

The git checkout command in Git is a versatile and fundamental command used to switch between different branches, navigate through commits, and manipulate the working directory. Its primary purpose is to update the state of the working directory and the HEAD pointer to reflect the specified state of the repository

git checkout <branch\_name>

git checkout -b <new\_branch\_name>

git checkout <commit\_hash>

git checkout -- <file\_path>

git checkout <commit\_hash>

4 What is the purpose of the git commit command?

The git commit command in Git is used to record changes to the repository's history by creating a new commit. When you make changes to files in your working directory and you're satisfied with those changes, you use git commit to save those changes to the repository's history. Here's the purpose and functionality of the git commit command:

Record Changes:

The primary purpose of git commit is to record the changes you've made to files in your working directory and staging area (if any) into the Git repository's history.

Create a Commit Object:

When you run git commit, Git creates a new commit object containing the current contents of the staging area (i.e., changes you've added using git add) and a commit message describing the changes.

Add a Commit Message:

When you commit changes, you're required to provide a commit message that describes the changes you're committing. A good commit message is descriptive and summarizes the purpose of the changes.

Track History:

Each commit in Git represents a snapshot of the repository at a particular point in time. Commits form a chronological history of changes, allowing you to track the evolution of the project over time.

Isolate Changes:

Commits in Git are isolated units of change. Each commit represents a logical unit of work, such as implementing a new feature, fixing a bug, or making a documentation update. This allows you to manage and review changes in a granular manner.

Rollback Changes:

Commits provide a mechanism for rolling back changes in case of errors or unintended consequences. You can revert to a previous commit to undo changes and restore the repository to a previous state.

Collaboration and Code Review:

Commits are essential for collaboration and code review workflows. Each commit represents a contribution from a developer, making it easier to review and discuss changes made to the codebase.

5 What is the purpose of the command 'git rm'?

The git rm command in Git is used to remove files from the working directory and the Git index (staging area), effectively deleting them from the repository. Here's the purpose and functionality of the git rm command:

Remove Files from the Working Directory and Staging Area:

When you run git rm <file>, Git removes the specified file from both your working directory and the staging area (index).

This means that the file is effectively deleted from your local repository, and it will no longer be tracked by Git.

Stage Deletion for Committing:

By removing files using git rm, you're staging the deletion of those files for the next commit.

After running git rm, you still need to commit the changes using git commit to permanently remove the files from the repository's history.

Options:

git rm supports various options to control its behavior:

-f or --force: Force removal of files even if they're modified or have unstaged changes.

-r or --recursive: Recursively remove files in directories.

--cached: Remove files only from the index (staging area) without deleting them from the working directory. This is useful for untracking files without actually deleting them from disk.

Remove Files Tracked by Git:

git rm is used specifically for removing files that are already tracked by Git. If you want to stop tracking a file but keep it in your working directory, you should use git rm --cached <file>.

Update Git Index:

After running git rm, Git updates its index to reflect the removal of the specified files. This ensures that the changes are properly recorded and included in the next commit.

6 What is the purpose of the git log command?

The git log command in Git is used to view the commit history of a repository. It displays a list of commits in reverse chronological order, showing the most recent commits first. Here's the purpose and functionality of the git log command:

View Commit History:

The primary purpose of git log is to display a chronological list of commits in the repository's history.

By default, git log shows information about each commit, including the commit hash (SHA-1), author, date, and commit message.

Commit Information:

For each commit, git log displays:

Commit hash: A unique identifier for the commit.

Author: The name and email address of the person who made the commit.

Date: The date and time when the commit was made.

Commit message: A descriptive message summarizing the changes made in the commit.

Navigating Commit History:

git log allows you to navigate through the commit history using various options:

Use the arrow keys to scroll through the log.

Press q to exit the log and return to the command prompt.

Use the -n or --max-count=<number> option to limit the number of commits displayed.

Use the --since, --until, or --after options to filter commits by date.

Use the --grep=<pattern> option to filter commits by commit message.

Use the --author=<pattern> option to filter commits by author.

Graphical Visualization:

By default, git log displays commits in a text-based format. However, you can use additional options like --graph to visualize the commit history as a text-based graph, showing branching and merging.

Custom Formatting:

You can customize the output of git log using the --format option to specify a custom format string. This allows you to display specific information about each commit in the desired format.

Analyze History:

git log is useful for analyzing the history of a repository, understanding the evolution of the project, and tracking changes made by contributors over time.

It's commonly used for code review, debugging, and identifying the origins of specific changes or issues in the codebase.

7 What is the purpose of 'git add'?

The git add command in Git is used to stage changes for the next commit. When you make modifications to files in your working directory, Git initially does not track those changes. By using git add, you tell Git which changes you want to include in the next commit. Here's the purpose and functionality of the git add command:

Stage Changes:

The primary purpose of git add is to stage changes made to files in your working directory. This means that you're preparing those changes to be included in the next commit.

Add New Files:

When you create new files in your working directory, Git doesn't automatically track them. You need to use git add to start tracking the new files and stage them for the next commit.

Stage Modified Files:

If you've made modifications to existing files in your working directory, you can use git add to stage those changes for the next commit. Git will then include the modified versions of those files in the commit.

Stage Deleted Files:

If you've deleted files from your working directory, you can use git add to stage the deletion of those files. This tells Git to remove the files from the repository in the next commit.

Selective Staging:

git add allows for selective staging, meaning you can choose which changes to stage and which changes to leave out of the staging area. You can specify individual files, directories, or use patterns to match multiple files.

Interactive Staging:

With the git add -i or git add --interactive command, you can enter interactive mode, which allows you to interactively choose which changes to stage and which to leave unstaged.

Preview Changes:

Before committing your changes, you can use git add to preview the changes that will be included in the next commit. This helps you review and verify the changes before finalizing them

8 In GIT, what is a 'bare repository'?

In Git, a "bare repository" is a special type of repository that does not have a working directory. Unlike a typical Git repository, which contains the project's files and directories in the working directory, a bare repository only consists of the Git version control data (e.g., branches, commits, tags) and repository metadata. Here's a more detailed explanation of a bare repository:

No Working Directory:

A bare repository does not have a working directory where you can view or edit files. It only contains the Git data and repository structure.

Since there's no working directory, you cannot directly work with the files in a bare repository like you would with a regular repository.

Shared Repository:

Bare repositories are often used as shared or centralized repositories in collaborative workflows.

Multiple developers can push and pull changes to and from a bare repository without worrying about conflicts with working directories.

Remote Repository:

Bare repositories are commonly used as remote repositories that serve as a central hub for collaboration in distributed version control systems like Git.

When you clone a repository from a bare repository, you're essentially creating a local copy of the repository without the bare characteristics.

Convention:

By convention, bare repositories have a .git extension appended to their directory names to distinguish them from repositories with working directories.

For example, a bare repository for a project named "my-project" might be named "my-project.git".

No Direct Editing:

Since bare repositories do not have a working directory, you cannot directly edit files or view their contents in a bare repository.

However, you can perform administrative tasks and Git operations (e.g., creating branches, viewing commit history) using Git commands.

Efficient Storage:

Bare repositories are often more space-efficient than repositories with working directories because they do not store the actual file contents, only the Git metadata.

9 What's the difference between git remote and git clone?

git remote and git clone are two distinct Git commands that serve different purposes in the version control workflow:

git remote:

git remote is a command used to manage remote repositories associated with your local repository.

It allows you to view, add, rename, and remove remote repositories.

Commonly used subcommands include:

git remote add: Add a new remote repository.

git remote remove: Remove an existing remote repository.

git remote rename: Rename an existing remote repository.

git remote -v: View the names and URLs of remote repositories.

Example usage: git remote add origin <repository\_url>

git clone:

git clone is a command used to create a copy of an existing Git repository, including its history and files, onto your local machine.

It essentially duplicates a remote repository onto your local machine, allowing you to work on the project locally.

When you clone a repository, Git automatically sets up a remote connection to the original repository, typically named origin.

Example usage: git clone <repository\_url>