__/__/____

Version Controlling: This is the process of maintaining multiple versions of the code into the Version Controlling system (VCS). The VCS accepts the project uploads from the entire team and creates an integrated project out of all these uploads. Next time when the developers download the code from the VCS, they can see the work done by the entire team. Version controlling system also preserve older and later versions of the code so that at any point of time the developers can toggle between any version.

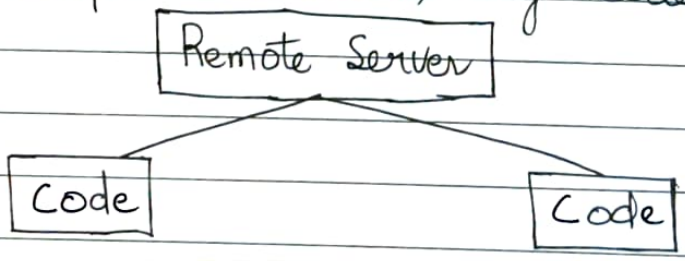
Version controlling system also keep a track of who's making what kind of changes.

VCS is of two types

1. Centralised Version Controlling
2. Distributed Version Controlling

Centralised Version Controlling:

In CVC we have a remote server where Version controlling happens. All the developers upload their code into this remote server. On the individual developers machines, only code is present

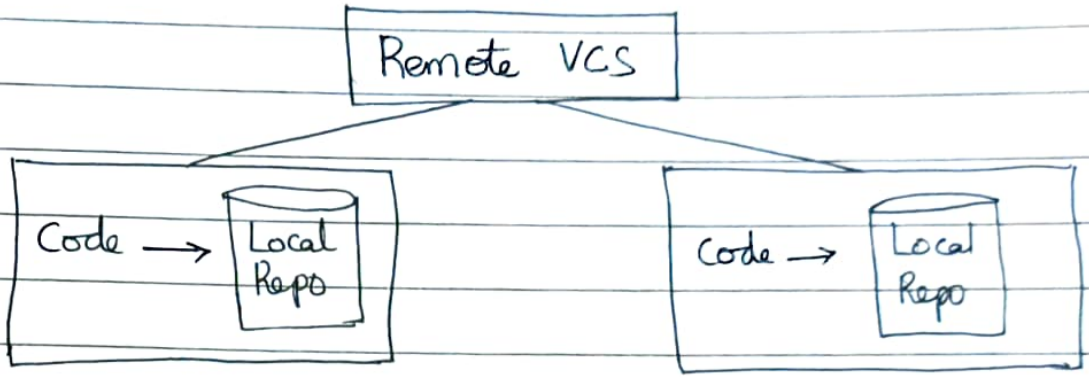


Ex: Apache Subversion

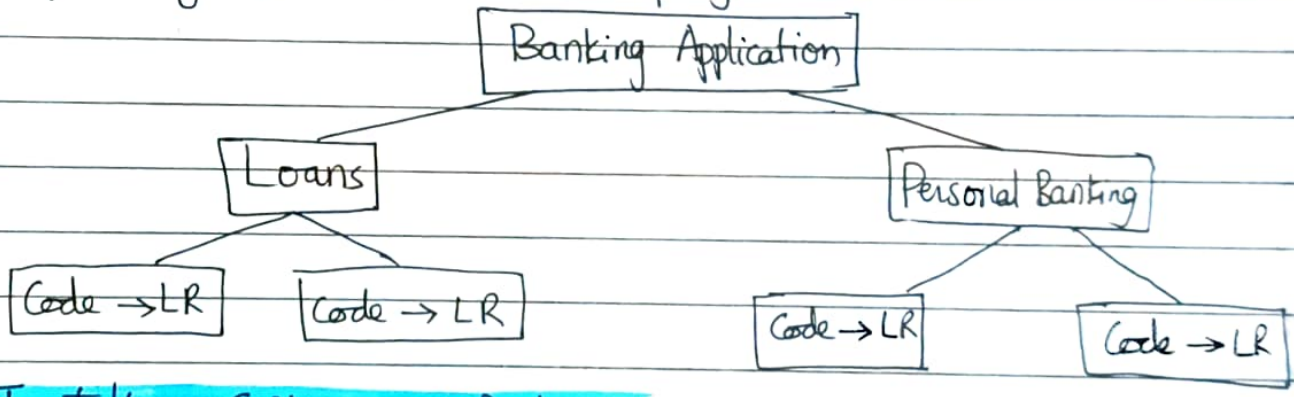
Distributed Version Controlling: In DVC we have a local repository installed on every developers machines. Initially the developers commit their code in the local repository where Version controlling happens

at the level of individual developers. Later it will be uploaded into the remote repository where version controlling happens at the level of the entire team

Ex: Git created by Linus Torvalds



In DVC, we can maintain bare repositories for individual team which can be later merged with a centralised repository for the entire project



Installing Git on Windows:-

① Open https://git-scm.com/downloads
② Download git for windows
③ Installing git

Installing Git on Linux:-

Ubuntu
① Update apt repo — Sudo apt-get update
② Install git — Sudo apt-get install -y git

__/__/____

Setting username and email-id for git users

```
git config --global user.name "Your full name"
git config --global user.email "your mail"
```

To see the list of default configuration

```
git config --global --list
```

Git when working on the local machine has 3 components

1. Working directory (or) workspace
2. Staging Area
3. Local repository

Working directory is the folder where developer creates the code. Initially all these files are called untracked files

Staging area is the intermediate buffer area of git into which files are moved. These files are called staged files

Local repository is the place where version controlling happens. All the files present in the staging area will move into the local repository and these files are called committed files.

To make a project a git repository, go into the project

```
git init .
```

Check the status : git status (git shows all files in untracked section)

To add files to staging area

```
git add . (. represents current directory)
```

You can choose files of your choice to stage and commit

To remove files from staging area

```
git rm --cached filenames (or) git reset filename(s)
```

__/__/____

To move files from staging area to local repository
git commit -m "message"
To check the status of untracked section & staging area
git status
To check the commit history
git log (or) git log --oneline

.gitignore: This is a special file which is used to store private files information. Any files whose name is mentioned in .gitignore will not be accessed by git. This is generally used by developers to save private files and prevent them from getting exposed to the local repository.

① Create few files in working directory
touch a.txt, b.txt c.txt file1 file2
② Check the status of git
③ Create .gitignore file and add text files
cat > .gitignore
*.txt ^D
④ Check the status of git. It shows file1, file2 and .gitignore in untracked section. All the text files are not tracked by git anymore.

GIT BRANCHING: Branching is a feature available in most modern version control systems. Instead of copying files from directory to directory, git stores a branch as a reference to a commit. In this sense a branch represents the tip of a series of commits
To see list of branches : git branch
List of branches - local & remote : git branch -a
To create a new branch : git branch branch_name

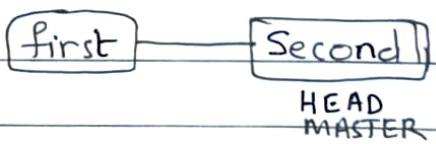
__/__/____

To move into a branch: git checkout branch_name
To create & move into branch: git checkout -b branch_name
To merge a branch into the master branch, first checkout to master and then merge the branch
git checkout master git merge branch_name
To delete a branch that is merged git branch -d branch name
this is also called soft delete (merged & then deleted. -d flag)
To delete a branch that is not merged: git branch -D branch_name
This is called hard delete (unmerged branch & deleted -D flag)
Note: whenever a branch is created, the commit history of the master branch till that point will be copied to the newly created branch.
Note: Irrespective of where a file is created or modified git always considers only the branch from where it is committed and that file belongs to that branch only.
Git MERGING: Whenever we perform a merge operation the commits present on the branch will be moved to the master branch based on timestamp of the commits.

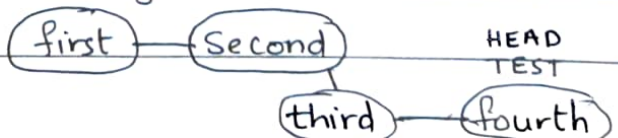
① Create few commits on the master
touch f1 → git add . → git commit -m "first"
touch f2 → git add . → git commit -m "second"

② Check the commit history of master
git log --oneline

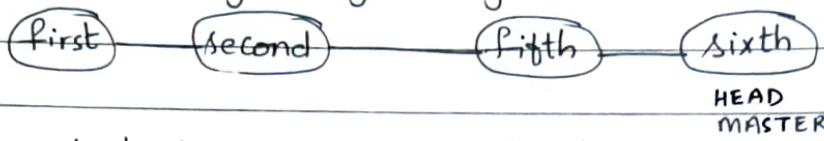
master branch



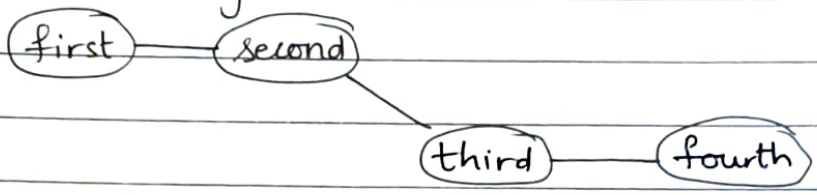
Create a test branch & make few commits git checkout -b test
touch f3 → git add . → git commit -m "third"
touch f4 → git add . → git commit -m "fourth"
check commit history
git log --oneline



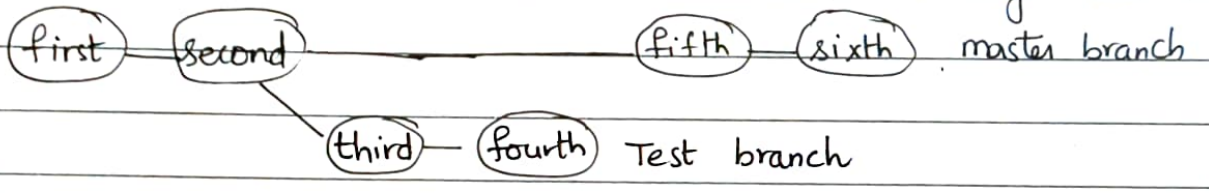
Now, move to master : git checkout master
Create more commits : touch f5 → git add . → git commit -m "fifth"
touch f6 → git add . → git commit -m "Sixth"
Check commit history : git log --oneline



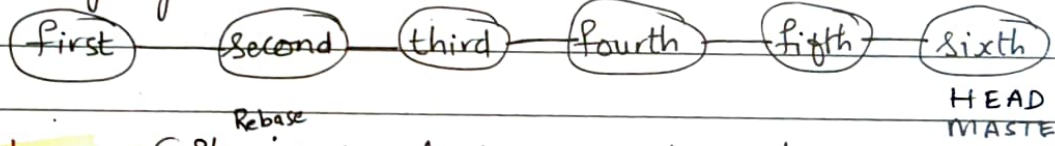
Check commit history on branch test



To merge the test branch with master —
first, checkout to master and then merge test



after merging —

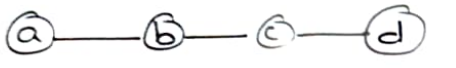


Git Rebase : Git Rebase is used for creating linear commits where the commits from branch will be projected as topmost commits.

Ex: Create few commits on master
touch f1 → git add . → git commit -m "a"
touch f2 → git add . → git commit -m "b"
git checkout -b test ← Create a branch & make few commits
touch f3 → git add . → git commit -m "c"
touch f4 → git add . → git commit -m "d"
Go back onto master and make few more commits
touch f5 → git add . → git commit -m "e"
touch f6 → git add . → git commit -m "f"

git log --oneline

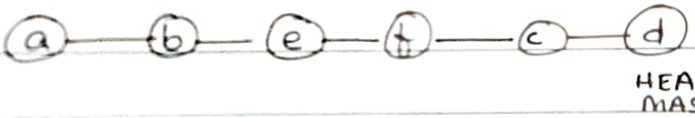
git log -- oneline



If I merge branch on master, my commit history would be

git merging does merge operation based on timestamp. What if I want commits on my branch as recent commits?

Checkout to branch: git checkout test
Rebase with master: git rebase master
Checkout to master: git checkout master
Merge the branch: git merge test
And now if I check the commit history —
git log --oneline



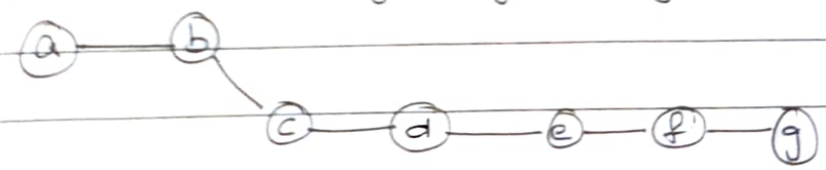
Git cherry-picking: Whenever we do a merge operation or a rebase operation, all the commits present on a branch will be moved to the master branch. If we have a scenario where the developer wants to pick up only few commits from branch and add them to master, then we can use cherry picking

① Create few commits on the master
touch f1 → git add . → git commit -m "a"
touch f2 → git add . → git commit -m "b"

② Check the commit history of master: git log --oneline

③ Create a branch & make few commits on branch
touch f3 → git add. → git commit -m "c" [Create 5 commits]
check branch history: git log --oneline

git checkout -b test



To add few commits on branch to master, __/__/____

```
git checkout master
git cherry-pick CommitID1 CommID2
```

Git RESET: This command can be used for moving the HEAD pointer from the topmost commit to any older commit. When we try to access the data, git always shows the data as present at the level of the HEAD commit.

1. Create a file and commit it
2. Keep modifying the file and make few more commits
3. Check commit history and to move into any of the older commits, git reset --hard commit_id (older commit id)

Git Amend: This is used for modifying the commit rather than creating new commit. Whenever the developer makes any modification, he generally commits it. But sometimes for every minor modification, there might be no necessity for creating a commit. This can be done using the git-amend command.

EX; Create few commits

```
touch file1
git add .
git commit -m "a"
touch file2
git add .
git commit -m "b"
```

Check the commit history - git log --oneline

Create new file. But add them to latest commit "b"

```
touch file3
git add .
git commit --amend -m "b"
```

Check the commit history - git log --oneline

There will be only two commits. Third commit of file3 gets amended to commit "b"

Note: The git amend command internally creates a new commit. The older commit that it has modified will be removed from the git active tree structure and it becomes an orphaned commit

git log shows only the list of active commits. To see all the commits active and orphaned, go with the command "git reflog"

Git Stash: Stash is an area of git where once the files are pushed, git can no longer access it. This is used when the developer want to leave some unfinished work and they want to start working on some other functionality and further commands of git should work only on this new functionality. Later when the developers want to resume the work with the older functionality, he can "stash it and git will start accessing those files.

① To send all the files present in staging areas to stash

```
git stash
```

② To send all files present in untracked section & staging area into the stash section

```
git stash -u
```

③ To send all files present in untracked section and staging area and also the .gitignore into stash section

```
git stash -a
```

④ To see the list of stashes done

```
git stash list
```

⑤ To unstash a latest stash

```
git stash pop
```

⑥ To unstash an older stash

git stash pop stash@{stash_number}

-gitignore is used for hiding private files. Any file whose name is mentioned in .gitignore will not be accessed by git but the .gitignore file itself can be accessed by git and it can move into staging area and local repository and remote repository to hide .gitignore file also we can use "git stash -a"

Rearranging the commit history:-

This can be done using git rebase command. The first commit is called as initial commit and that cant be modified. All the remaining commits rearranged in whichever format we want.

① Create few commits on the master branch

```
touch f1
git add .
git commit -m "a"
    touch f2
    git add .
    git commit -m "b"
        touch f3
        git add .
        git commit -m "c"
    touch f4
    git add .
    git commit -m "d"
touch f5
git add .
git commit -m "e"
```

② Check the commit history
git log --oneline
ⓐ — ⓑ — ⓒ — ⓓ — ⓔ

Note: The first commit 'a' cant be disturbed so we can rearrange only four commits

③ To rearrange the commits
git rebase -i HEAD~4

This command will open the commit history in Vi editor where we can rearrange commits in any order that we want

④ Check the commit history
git log --oneline

Git Squash:

Merging of multiple commits is called squash.
This is useful when the developer wants to merge minor commits with each other. This will shorten the commit history and less no. of commits will be pushed into the remote github.
Note: 1st commit which is initial commit cannot be squash

⑤ To squash — git rebase -i HEAD~5
Whichever commits have to be merged, for them remove the "pick" word and replace it with "squash" save and quit.

⑥ Check the commit history → git log --oneline

Git Tagging:-

Tags are used for placing bookmarks on commits
This is only used for identifying important commits which are related to software releases.
Tags are categorized into two types
① Lightweight tags
② Annotated tags

__/__/____

Lightweight tag is just a bookmark whereas annotated tag is also a bookmark and it contains information about who tagged it, why they tagged it, when it was tagged etc.,

To create a lightweight tag

```
git tag tag_name
```

To see the list of all tags

```
git tag
```

To create a annotated tag

```
git tag -a tag_name -m "some message"
```

To see the detailed info about annotated tag

```
git show annotated_tag_name
```

Note: Tags are by default created for the latest commit

To create a tag for an older commit

```
git tag -a tagname -m "some message" olderCommitid
```

To delete a tag locally

```
git tag -d tag_name
```

To push all the tags into the remote repository

```
git push --tags
```

To delete tags from the remote github

```
git push origin :tag-name
```

Cleaning git repository:-

```
git clean
```

removes untracked files from git repository * undoable

```
git clean -n
```

A dry run, doesn't remove anything but notifies

```
git clean -f
```

Removes untracked files. clean.requireForce = True

```
git clean -f <path>
```

Untracked file in the path gets deleted.

__/__/____

git clean -df <path>

removes both untracked files and folders as well

git clean -xf

deletes the files in .gitignore