❖ What is Git and why do we need git?

Git is a **distributed version control system** that allows you to keep track of mor than just what's currently in a file.

- Git is a tool/software like IDE or VLC player. It is not a programming language.
- Git is not related to any programming language. It is a general tool.

There are two main reasons to learn git

1. Track changes to your code
2. Save your code to the cloud

Tracking changes to your own code is very important. It allows you to do a variety of things such as:

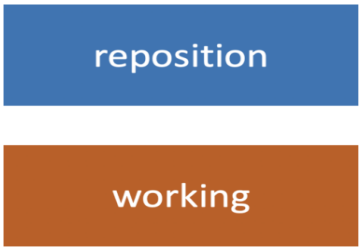
1. Maintain different versions of the same software
2. Make sure you never lose any code, recover code changes at any point of time.
3. Collaborate with others

Git tracks the changes you make to files, so you have a record of what has been done,

and you can revert to specific versions should you ever need to. **Git** also makes collaboration easier, allowing changes by multiple people to all be merged into one source.

❖ Explain the Architecture of Git

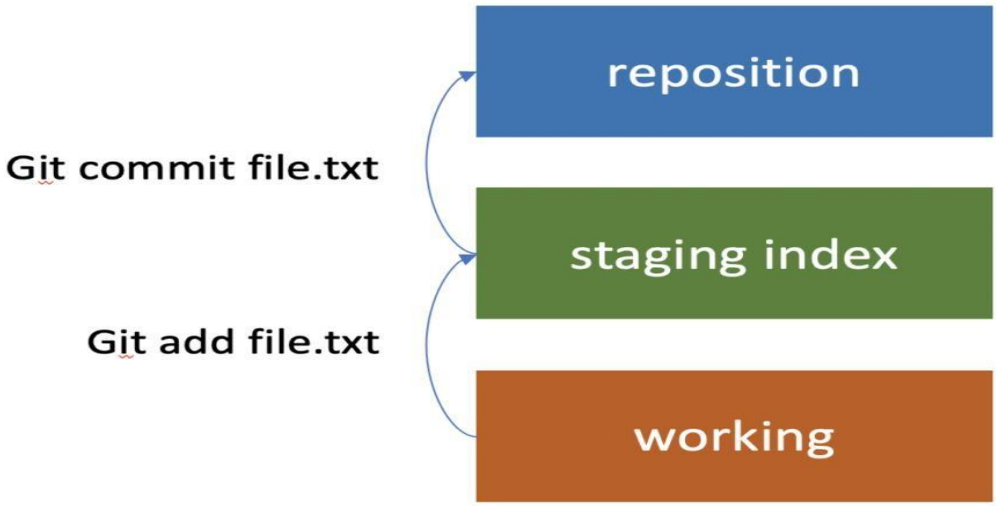
Let's start by taking a look at **a typical two-tree architecture**. They have a ***repository*** and a ***working copy***. These are the two trees.



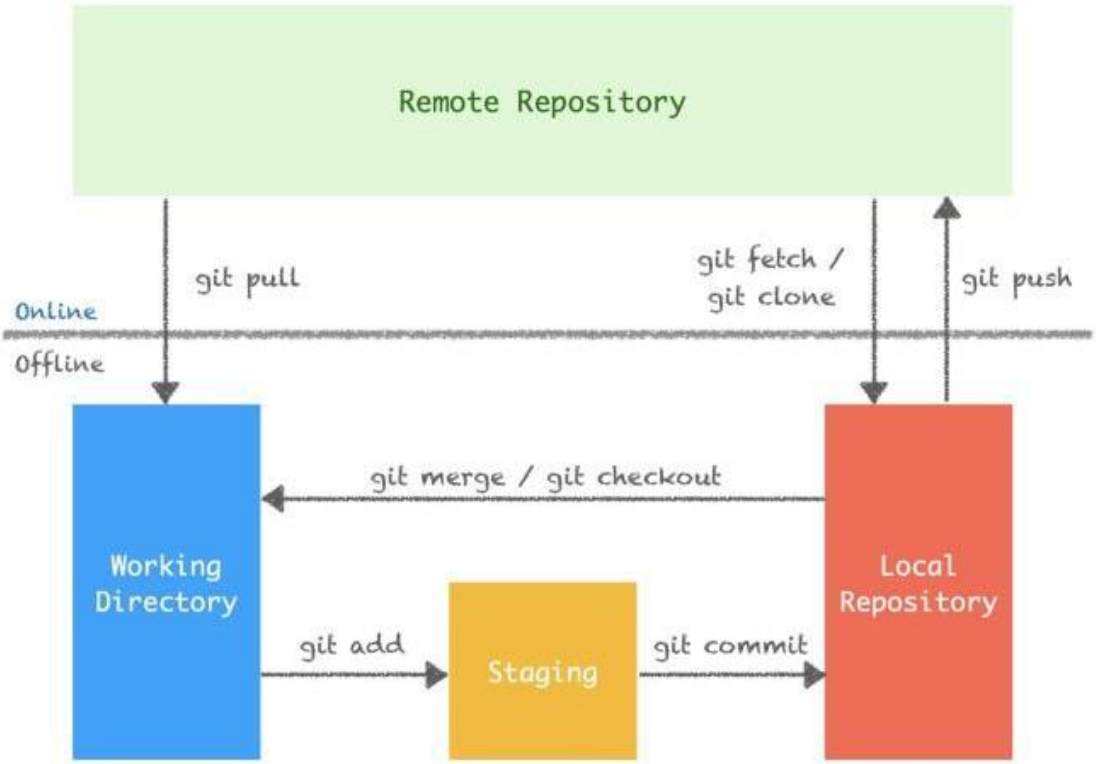
We call them trees because they represent a file/folder structure. The main project directory is at the top, and below it might be 4 or 5 different folders with a few files inside. Maybe a few more folders, each of those folders has a few more folders inside. You can imagine that if you map that out, each of those folders would branch out like the branches of a tree.

Git uses a three-tree architecture

It still has the repository and the working copy, but in between is another tree, which is the ***staging*** index. When we made our first commit, we didn't just perform a commit. First, we used the *add* command. We added, then we committed. It was a two-step process (added our files to the staging index, and then from there we committed to the repository).



As we're working with Git, it's useful to keep these three different trees in mind. There's our working directory, which contains changes that may not be tracked by Git yet, there's the staging index, which contains changes that we're about to commit into the repository, and then there's the repository, and that's what's actually being tracked by Git.



❖ **Explain the main tasks a user of git has to do to work in a collaborative way in a team!**

Create a New Repository on GitHub

Note: Only one repository is needed on GitHub, so only the team member who will host the repository on their GitHub account should perform this step.

On GitHub, under 'Your Repositories':

- Press the green ***New*** button.
- Fill in the Repository Name
- Provide an optional description

Other things to configure when setting up your repo:

- Select whether you want it to be private or public

- Select the checkbox to initialize the new repo with a README (The README is the front page of your repo and should summarize the project. Having a README in place will keep you from having an empty project. You can edit or replace the README later.)

- Select whether or not to include a .gitignore file. (File names and patterns listed in the .gitignore file will not be tracked by git. It is useful to list temporary files you know will be created during your project but that you don't want to track in your repo. E.g., Jupyter notebook checkpoint files.)

Clone the New Repository to your Local Machine(s)

Note: Each team member will need to perform this step.

On GitHub, navigate to the project repo:

- Press the green ***Clone or download*** button.
- Ensure that *clone with HTTPS* is selected.
- Copy the URL provided.

In your local terminal, from the top level folder under which you want to place your repo:

```
$ git clone paste-copied-url-here
```

NOTE: Be careful not to issue this command from a directory already being tracked by git. Do a *git status* first to verify. You see something like this message *fatal: not a git repository (or any of the parent directories): .git* This means it is safe to clone.

Create and Checkout a Local Branch for Your Work

Each contributor should create their own local branch for their work. This new branch will be a copy of the main branch.

Navigate to the top directory of the repo you just cloned. The ‘master’ branch should be active.

Check which branch is active using the command:

```
$ git branch
```

Create a new branch, giving it a name consistent with the naming conventions developed by your team.

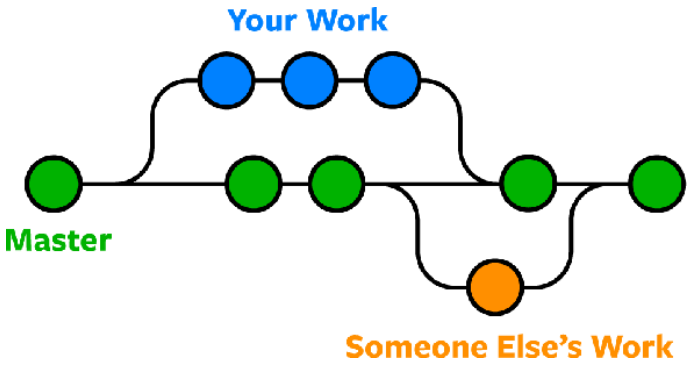
```
$ git branch your_branch_name
```

Check out the new branch. It will create a copy of the cloned master and set the current working branch.

```
$ git checkout your_ branch_name
```

Alternately, you can check out *and* create the new branch in one command:

```
$ git checkout -b your_branch_name
```



Save Incremental Changes Locally

It is a good idea to save versions of your work frequently. Do this with the *‘git commit’* command. The process involves staging the work you want to ‘check in’, using the *‘git add’* command, then using *‘git commit’* to tell Git to save the staged changes. Git keeps track of each commit batch and allows you to roll back to any prior commit if necessary. You can repeatedly *‘git add’* and *‘git commit’* in your local environment without ever pushing to GitHub. You may want to occasionally *‘git push’* unfinished code to GitHub (without doing a pull request) to have a backup copy, but it is good practice to merge to the master branch only code that has been tested and ready for

release.

Use the ***'git status'*** command to check what files have changed, what files git is tracking, and what files have been staged for commit:

```
$ git status
```

Git will notice new files and let you know they are not yet being tracked.

Use the *'git add'* command to add a file to staging. Do this to add a new file to Git or to add a modified file to staging:

```
$ git add your_file_name
```

A word of caution. You *can* do *'git add -a'* to add *all* the files in the tracked directory, including subdirectories, to staging. However, there may be files you do not want to track, such as temporary files created by you or by the software you use. For this reason, I would caution against using the -a option.

Remove Files from Git To remove a file. This removes the file from git *and* the branch:

```
$ git rm your_file_name
```

Commit the Staged Files When you have added the files you wish to have Git save to staging, you are ready to issue the *'git commit'* command.

```
$ git commit -m'descriptive message here'
```

Push Changes to GitHub

Here is where team communication comes into play. Has anyone updated the main branch on GitHub since you created your branch? Are there likely to be conflicts? If so, it is much easier for you to 'rebase' your local code by fetching a new copy of the master branch and resolving any conflicts locally. If others are doing this at the same time it could get even messier. To avoid problems, let your team-mates know what you are doing and coordinate pushes and pulls.

Pushing your code to GitHub If you need to rebase, follow the steps below before coming back to this step.

```
$ git push -u origin your-branch_name
```

This pushes your changes upstream (-u) to the repo pointed to by origin and creates a copy of your branch in the repo on GitHub.

To push again to this same branch, use the *'git push'* command from within the active branch, without any additional parameters. (If you are doing this subsequent times, GitHub already knows about the branch because you created it previously.) You would do this if you wanted to push more work and use GitHub as a backup, before doing the Pull Request.

```
$ git push
```

Note: Once your branch has been merged into the master, you don't want to use that branch name again. If you are going to continue working on the project, create a new branch as described previously.

Create a Pull Request to ask for the code to be merged into the main (master) branch on GitHub.