More git

CS14 - S26

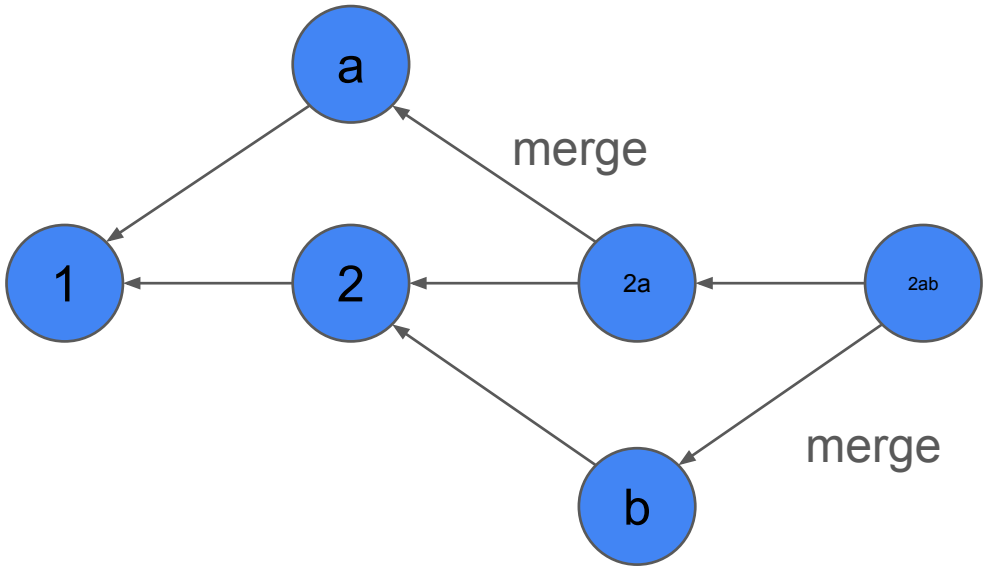
Rebasing

Using branches and merging allows different features to be developed without fear of messing with a shared base project.

However, always merging when branches/clones have small changes can lead to complex history graphs.

Rebasing

Suppose two features that touch different parts of a project are developed on two separate branches, `a` and `b`. Doing normal merges would show three branches in the history that required integration.



Rebasing

However, if the developments in `a`, `b`, and `main` are all separate parts of the project, then they could have been applied directly to main instead of branches. This would make the project history appear less complex.

Rebasing is an alternative to merging that appends commits from one branch directly to the end of another, as though those commits were made on the other branch.

Rebasing

`git rebase <branch> [<source>]` – replay commits in `<source>` (or `HEAD` if not specified) onto `<branch>`

Typically, this is followed by:

```
git switch <branch>
```

```
git merge <source>
```

Rebasing

Norms around using rebase and merge depend on the project and team.

You should never rebase commits that are already on a shared source repository (only on your personal clone), as this can break history and lose work for others.

Leveraging History

Often, you may want undo changes you have to a project, or restore some information that existed in the past.

git has three main commands that deal with restoring, undoing, and modifying history. `revert`, `restore`, and `reset`.

“`git revert` is about making a new commit that reverts the changes made by other commits”

“`git restore` is about restoring files in the working tree either from the index or another commit. This command does not update your branch.”

“`git reset` is about updating your branch, moving the tip in order to add or remove commits from the branch. This operation changes the commit history.”

git revert

`git revert <commit>` attempts to undo the changes done in `<commit>`, then makes a new commit logging the changes.

Suppose a critical bug is introduced by some commit that is already in the public repository. Using `revert` could undo the commit that introduces the bug without breaking the public history for others.

git restore

`git restore --source <source> <file>` will change `<file>` to the state it was in in `<source>`.

Suppose a few commits ago you accidentally deleted some work that you want to recover. You could save the current version of the file to a temporary copy, and then `restore` the old version.

git reset

`git reset [--mixed]` un-stages any files that have been `git add`ed, while preserving their contents.

`git reset --hard -- <file>` changes `<file>` to the state it was in at the `HEAD` commit. This can be used to discard work on the current commit.

`git reset --hard <commit>` **permanently undoes** previous commits, setting `HEAD` to `<commit>`. **Do not do this on commits that exist in a shared source repository.**