UCLA CS35L

Week 9

Wednesday

Reminders

- Assignment 8 due this Friday (5/29)
- Assignment 9 and Assignment 10 Report due next Friday (6/5)
    - NO Late Submissions for these

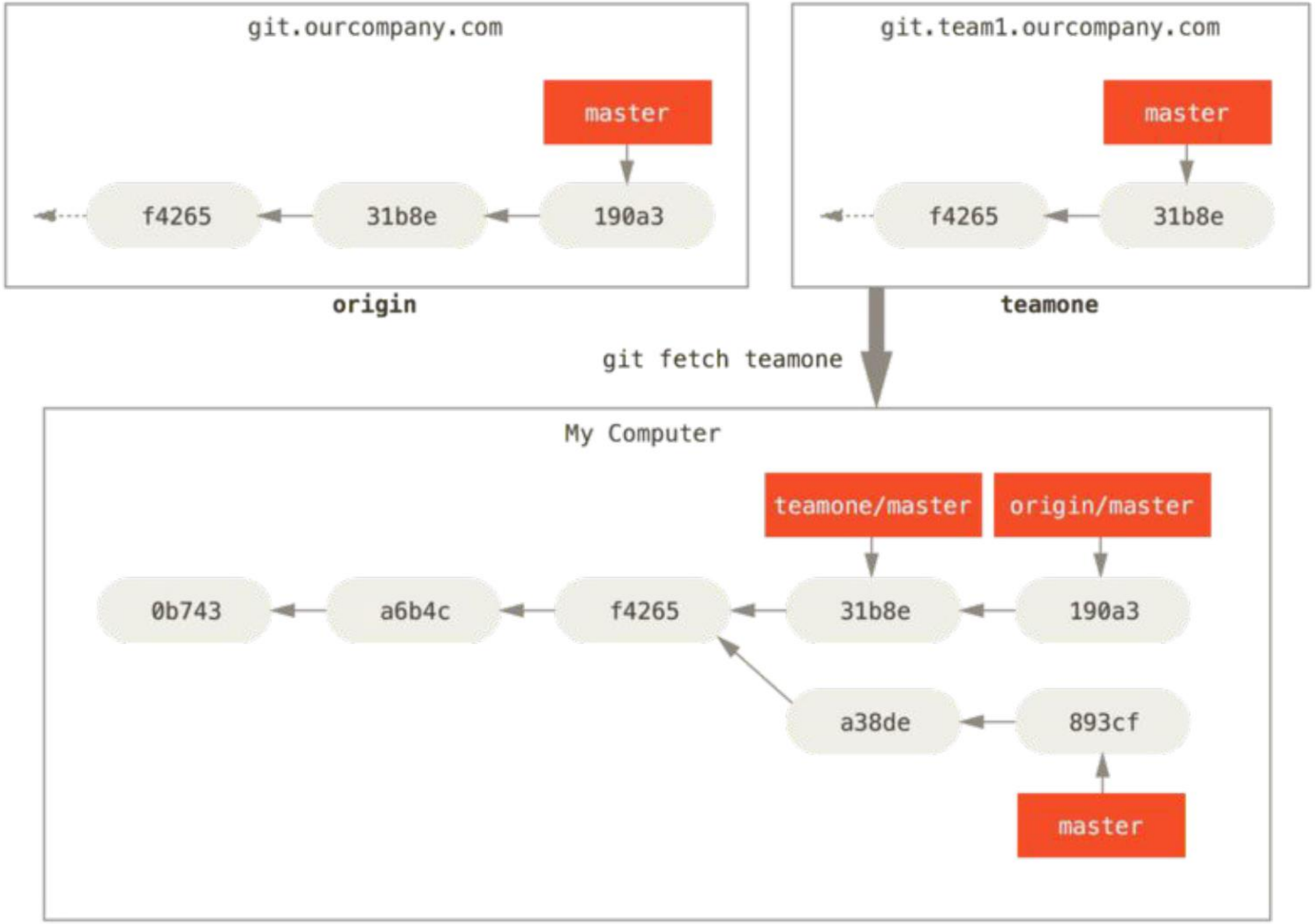
- Week 10 Assignment, first presenters are today!
- Reach out to me if:
    - You need to send in a recording due to timezone issues making it hard to present live
    - Your partner has not responded to you about preparing for the presentation/report
        - You will likely go solo and have a reduced report length

- Anonymous feedback for Daniel
    - https://forms.gle/tZwuMbALe825DBVn8

More Git Info

Working with multiple remotes

git remote add <name> <url> adds as a remote repo



Tracking Branches

- Checking out a local branch from a remote-tracking branch automatically creates a tracking branch. The branch it tracks is called the upstream branch
  - TLDR – a tracking branch is a local branch that knows it has a remote counterpart

```
git checkout –b <branch> <remote>/<branch>
```

Example - `git checkout –b b1 origin/b1`

- Creates a local branch b1, copied from and tracking the origin/b1 branch.

Command can be shortened too:

```
git checkout --track <remote>/<branch>
```

More tracking branches

```
git branch -vv
```

- Prints local branch and any tracking info

```
git push –u <remote> <branch>
```

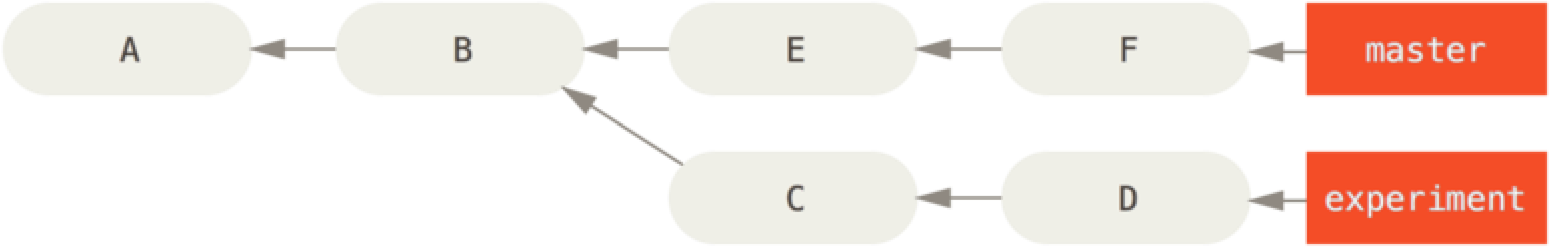
- Creates and sets the specified upstream branch as your current tracking branch

```
Git remote –v
```

- Prints remote repository information

Git Commit Ranges

Range: ..



```
git log master..experiment
```

- All commits reachable from experiment that aren’t reachable from master
- -> D, C

```
git log experiment..master
```

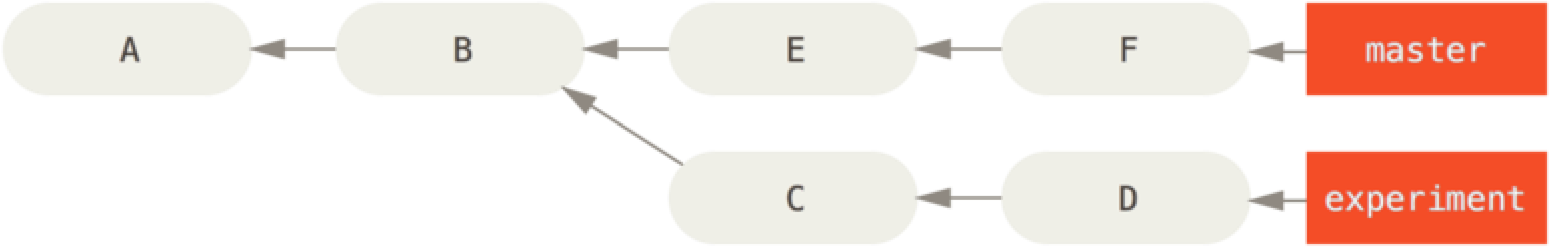
- -> F, E

Range: .. With remote

```
git log origin/master..HEAD
```

- Any commits in your current branch that aren’t in the master branch on your remote origin
- (Basically the commits you still need to push to master)

Range: …

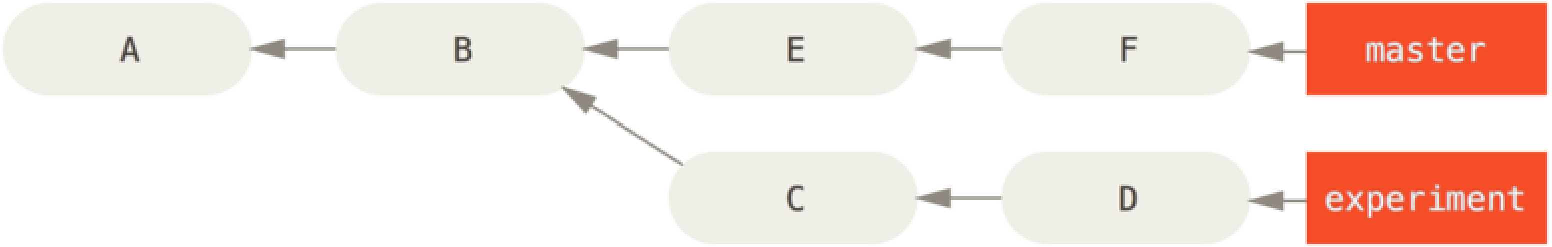


```
git log master…experiment
```

- TRIPLE DOT (…) means all commits reachable by either branch but not both

```
F
E
D
C
```

Range: ... --left-right



```
git log --left-right master…experiment
```

- --left-right will indicate which side the commit is reachable from

< F

< E

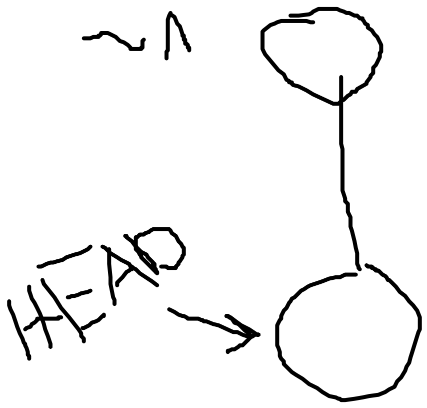
> D

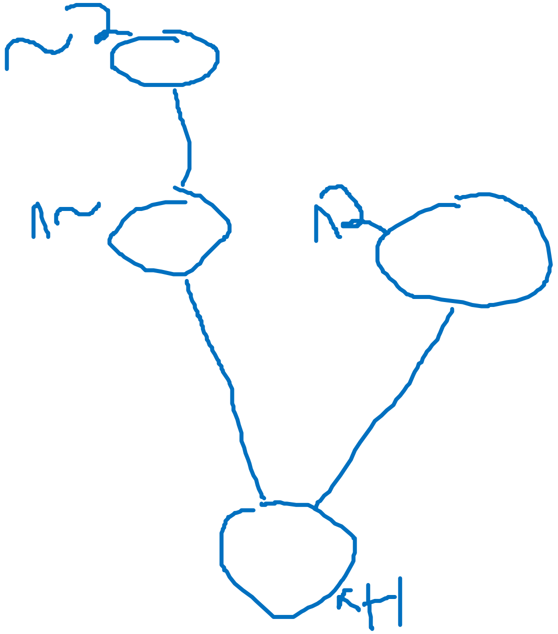
> C

Commit Syntax: ^ and ~

- ^ refers to parent of a commit
- ~ refers to first parent of a commit

- Examples
  - HEAD^ (parent) is equal to HEAD~ (first parent)
  - HEAD^2 (second parent) is *not* equal to HEAD~2 (first parent of the first parent)





Git Reset

git reset

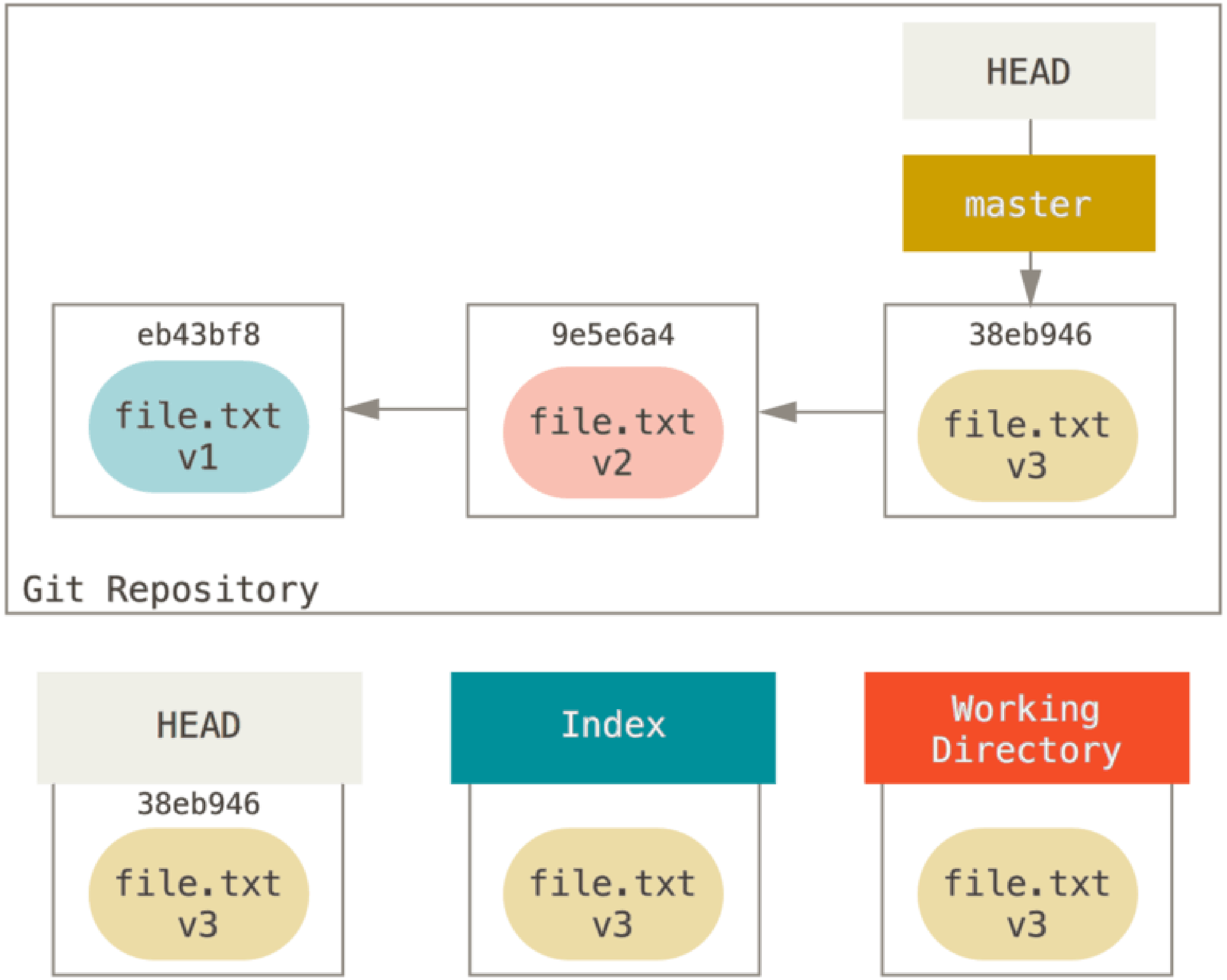
- Resets your HEAD pointer to the specified state
- Main options:
  - soft
  - default
  - hard

- **Example** – **`git reset HEAD~`**
  - HEAD is pointing to the master branch
  - 3 commits – eb43bf8, 9e5e6a4, 38eb946
  - 38eb946 is the most recent

Example

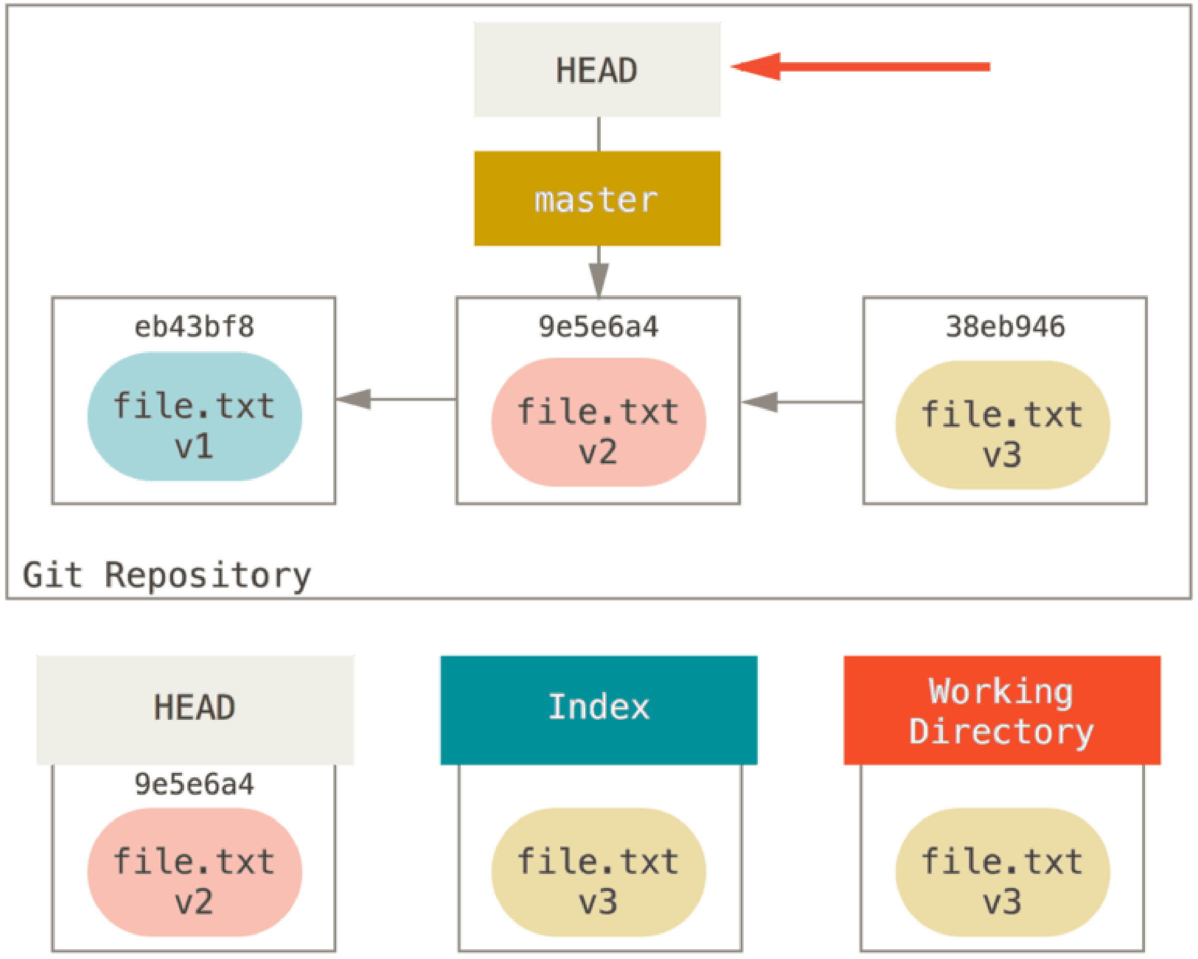
`git reset HEAD~`

- HEAD is pointing to the master branch
- 3 commits – eb43bf8, 9e5e6a4, 38eb946
- 38eb946 is the most recent
- file.txt v3 in Index



Step 1. Move HEAD (Soft)

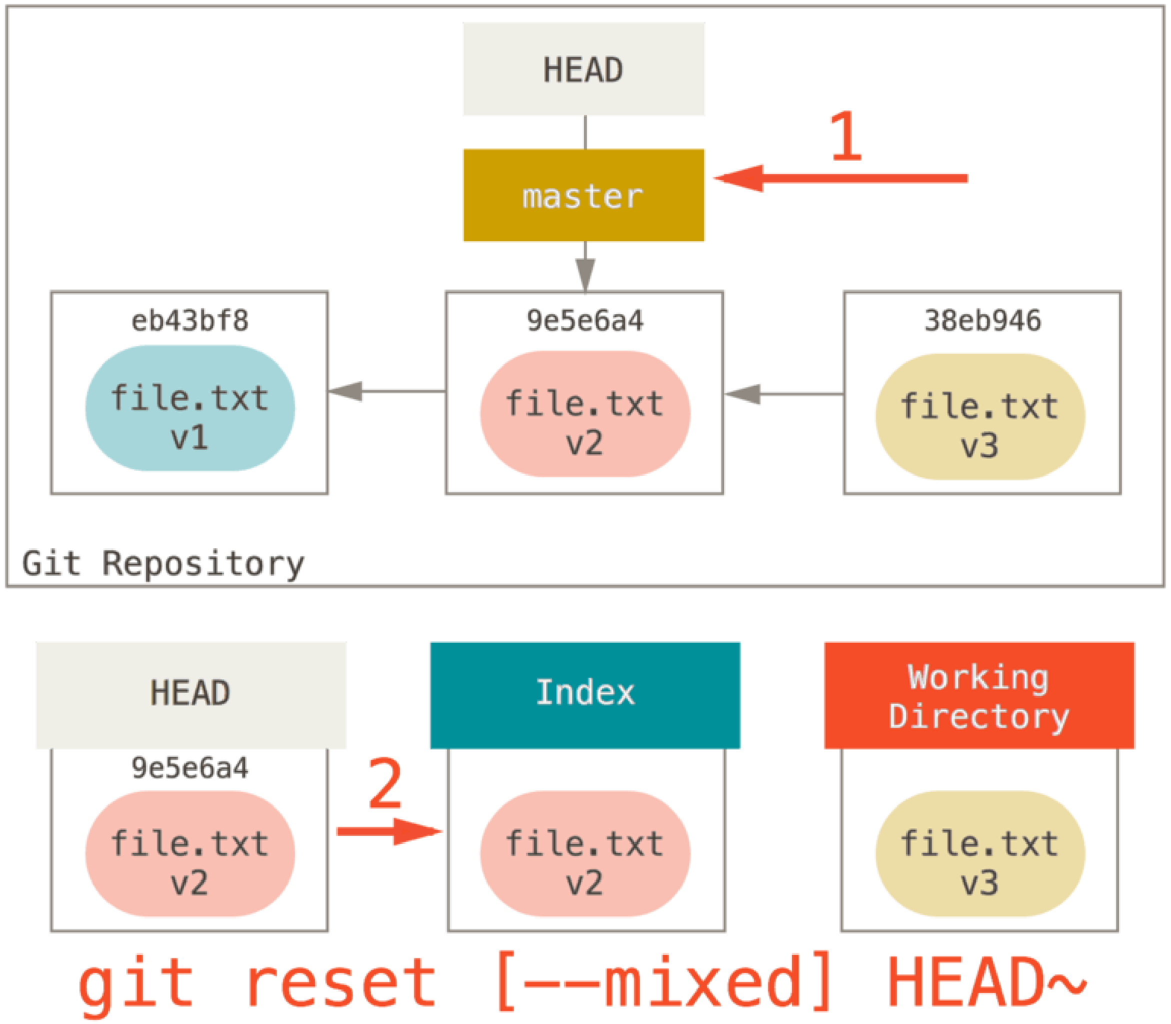
- Moves the branch that HEAD is pointing to (master) to the specified commit, 9e5e6a4.
  - Note the difference between this step and git checkout.
- At this point, git status will show the changes file.txt v3 as staged. This is because the staging area (index) didn’t change. You can run “git commit” from here if desired.
- The --soft option will stop here



git reset --soft HEAD~

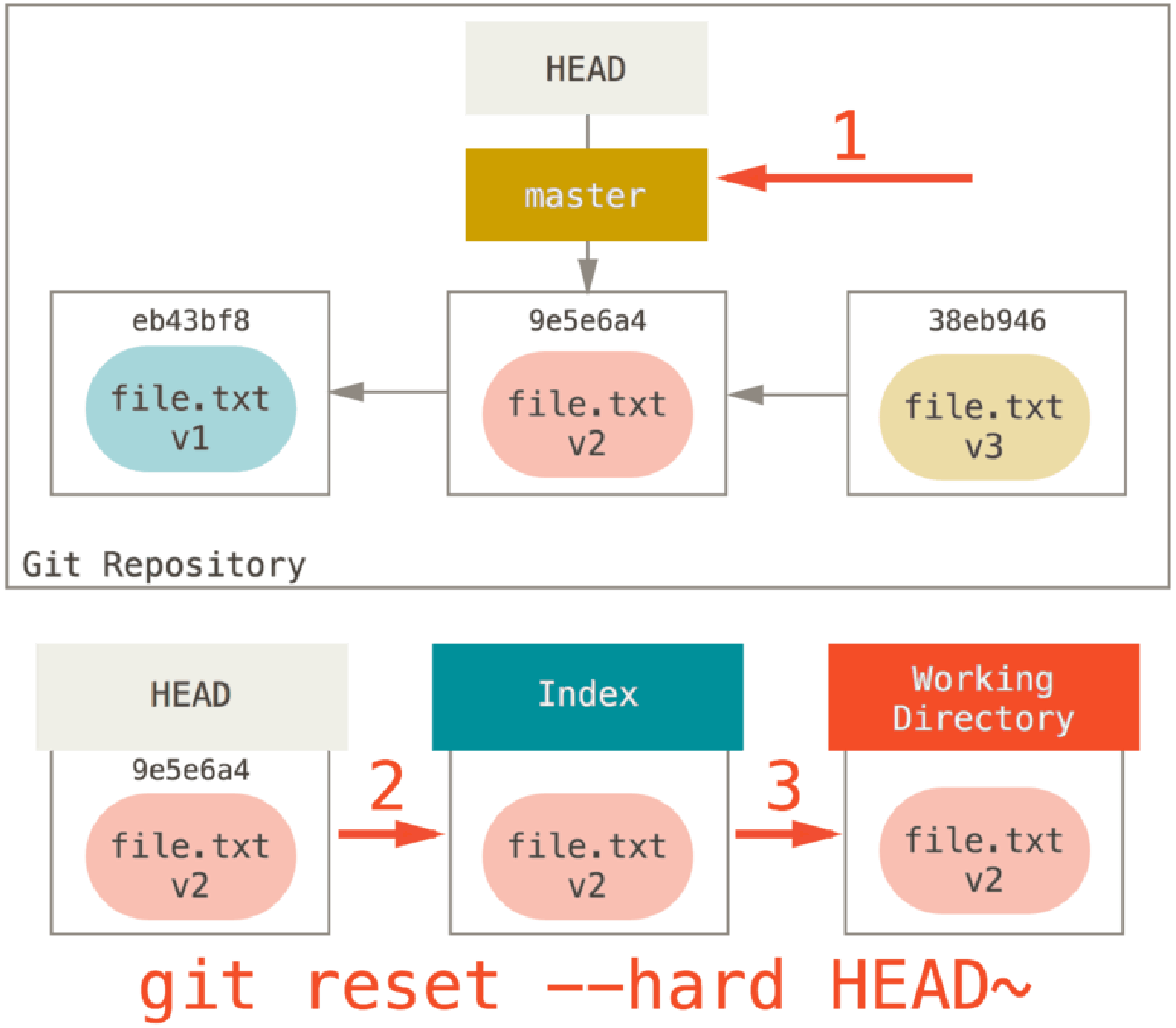
Step 2. Updating the Staging Area (Default)

- The staging area is modified to be the same as the commit stage.
- The working directory wasn't modified, so "git status" will show that file.txt has been modified still.
- This is where git reset will stop by default



Step 3. Updating the Working Directory (hard)

- The working directory is overwritten to match the staging area.
  - The current content of file.txt will match the v2 state.
- One of the few ways to actually lose data in git, can be dangerous
- Only happens if you specify --hard

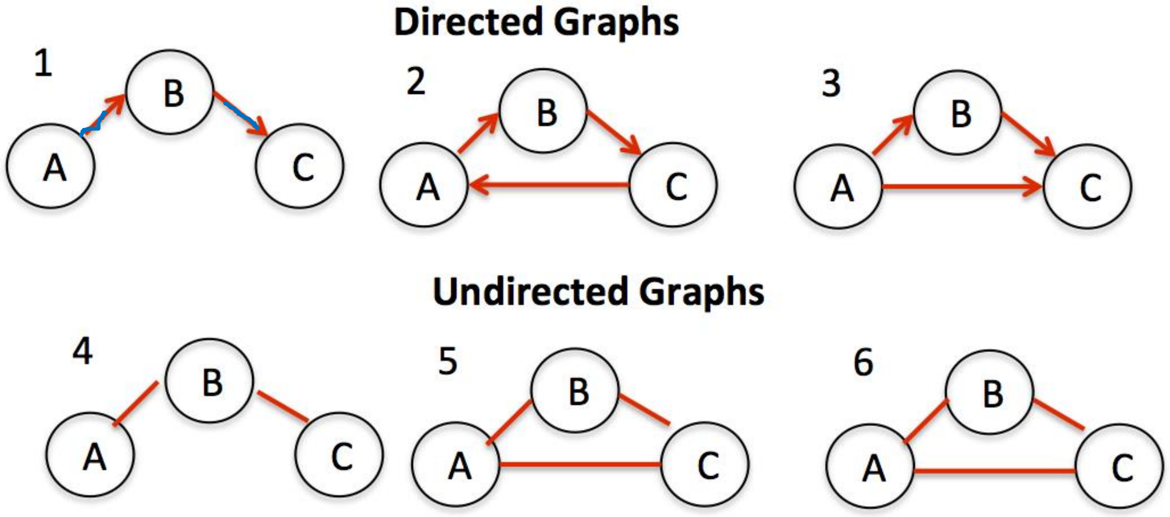


git reset --hard HEAD~

Git Theory

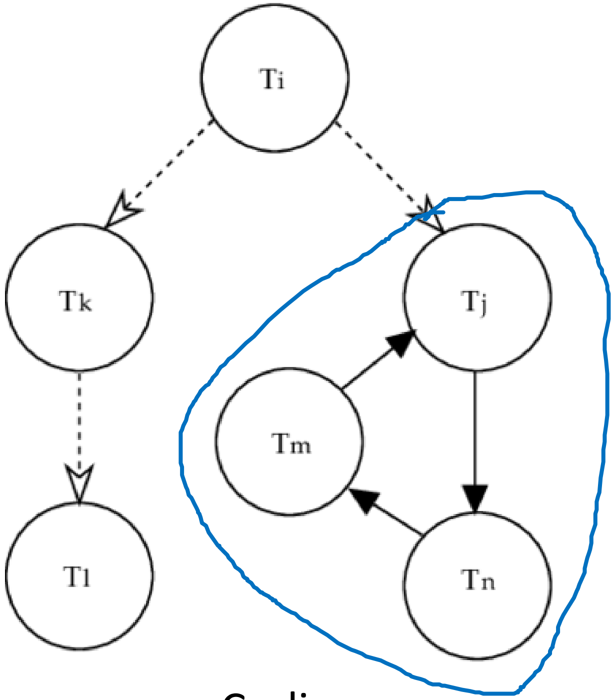
Directed vs Undirected Graph

- Graphs are a collection of nodes (vertices) connected by edges
- Directed Graphs have "direction"
  - The edges have arrows
- Undirected Graphs show general connections
  - The edges do NOT have arrows
- What kind of Graph do you think Git is?

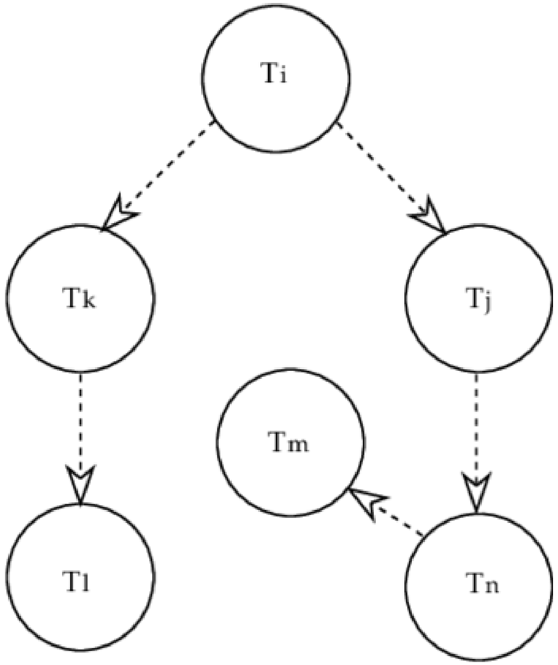


Cyclic vs Acyclic Graph

- Directed Graphs can be either Cyclic or Acyclic
- Cyclic Graphs have “cycles”
  - Somewhere inside the graph is a loop.
  - I can get from node Tj back to itself
- Acyclic Graphs do not
  - You can never reach the same node through itself
- A Directed Acyclic Graph is a **DAG**



Cyclic



Acyclic

Git and Graphs

- Git is a DAG, or a Directed Acyclic Graph
- That means we can process the information like a graph data structure
- One thing we can do is sort the order of commits, based on their dependencies (Topological Sort)

Git log as graph

```
git log --graph --pretty=format:“%h %s”
```

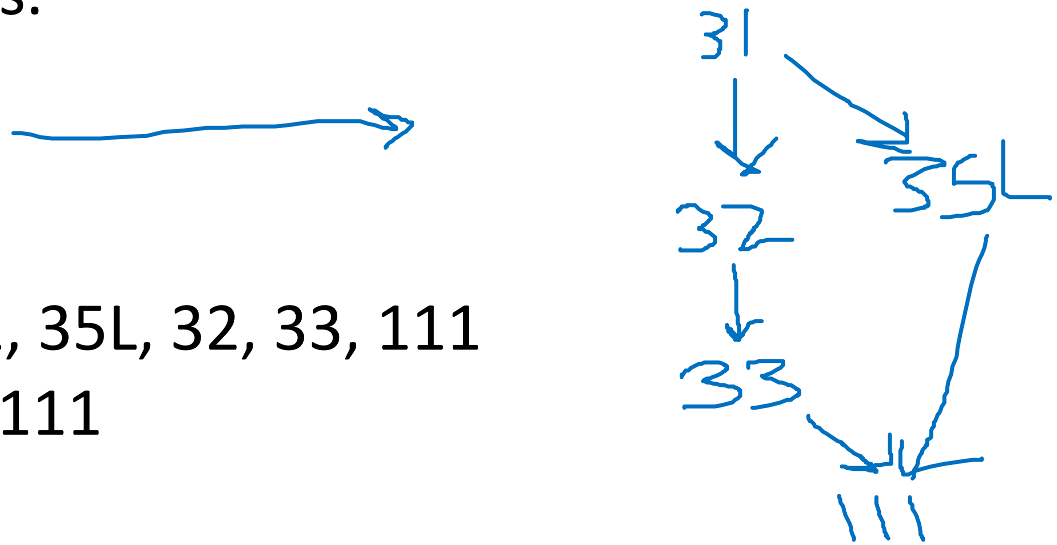
- Will show git log as a graph, with abbreviated commit has (%h) and message (%s)
- More options for the --pretty format can be found here

```
$ git log --pretty=format:'%h %s' --graph
* 2e25043 Merge pull request #18 from ...
|\
| * 4950521 fix sim by normalizing SNP columns
| * 69402cd generate g effects
|/
* c455717 tabulate_output.py
* f3fe695 Merge pull request #17 from ...
```

Topological Order

- A topological order is the sorted order of a graph such that if there is an edge from v1 -> v2 than v1 < v2.

- A classical example, is class planning. The classes and their prereqs below can be sorted by their dependencies (edges) and you can find the right order to take these classes.
  - CS 31 -> CS 35L
  - CS 31 -> CS 32
  - CS 32 -> CS 33
  - CS 32, 33, 35L -> CS 111



One Topological Order is 31, 35L, 32, 33, 111

Another is: 31, 32, 33, 35L, 111