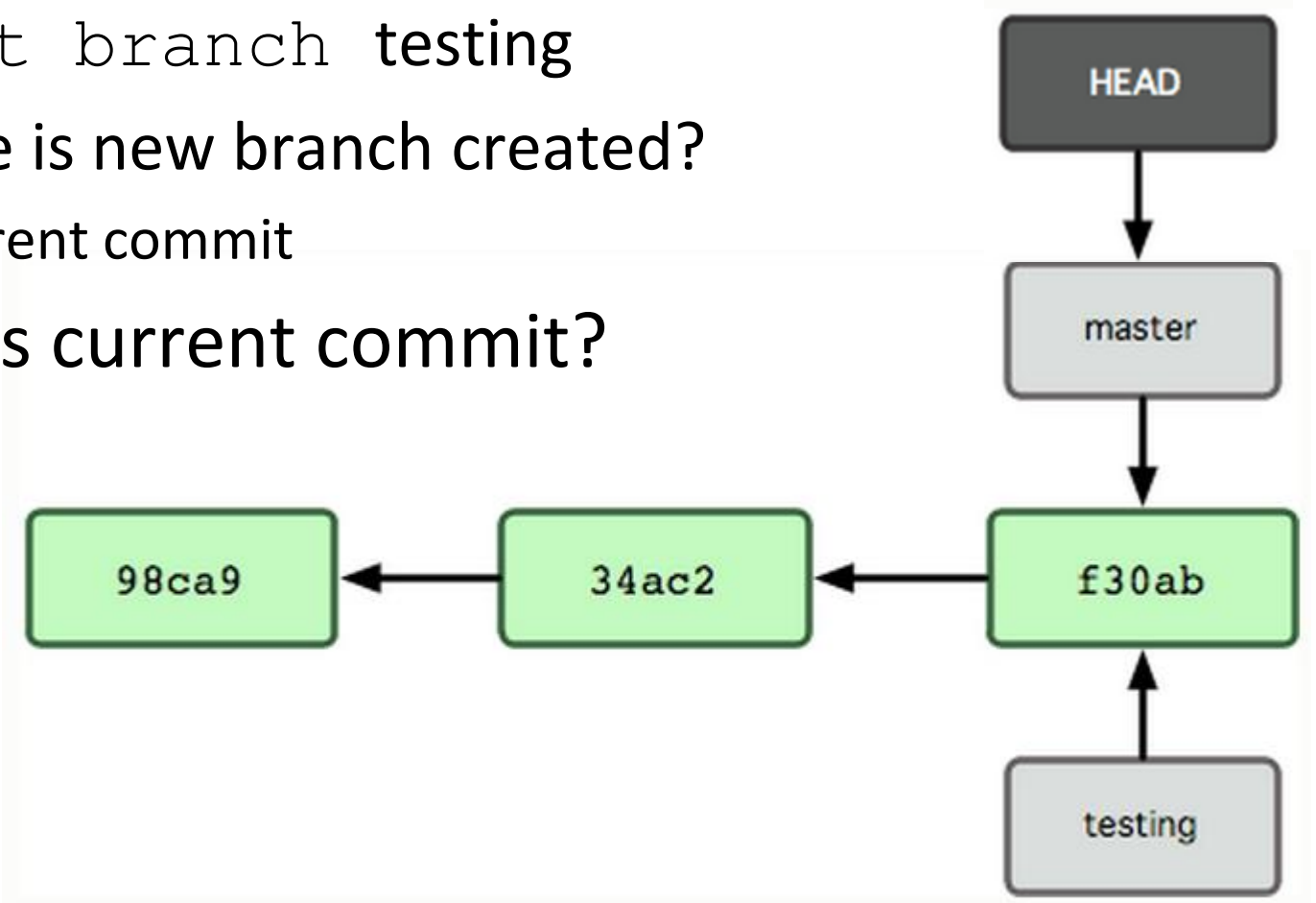
CS 35L

Week 9 – Section 7

New Branch

- Creating a new branch = creating new pointer
  - `$ git branch` testing
  - Where is new branch created?
    - Current commit
- Where is current commit?
  - HEAD

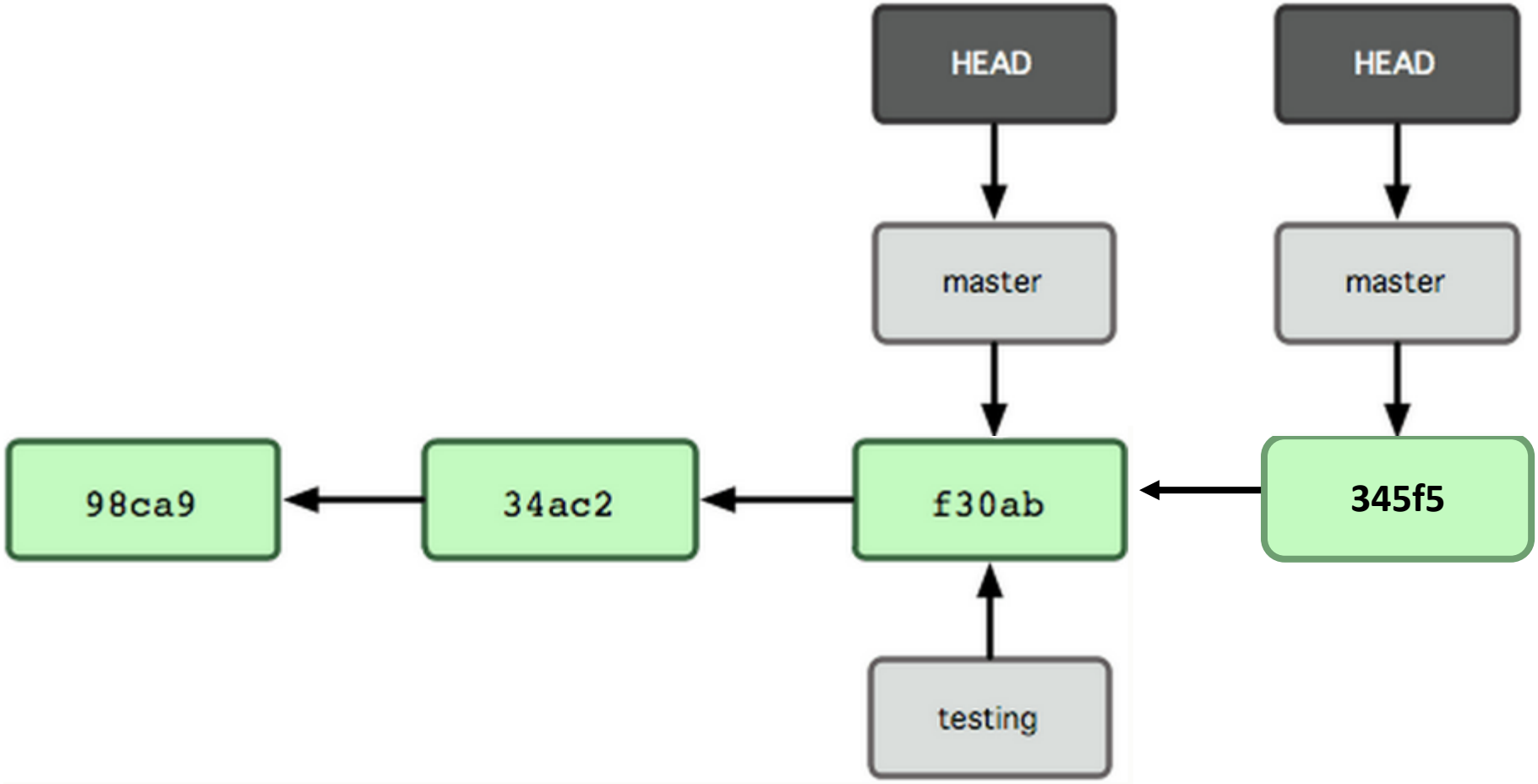


What Is a Branch?

- A pointer to one of the commits in the repo (head) + all ancestor commits
- When you first create a repo, are there any branches?
  - Default branch named ‘master’
- The default master branch
  - points to last commit made
  - moves forward automatically, every time you commit

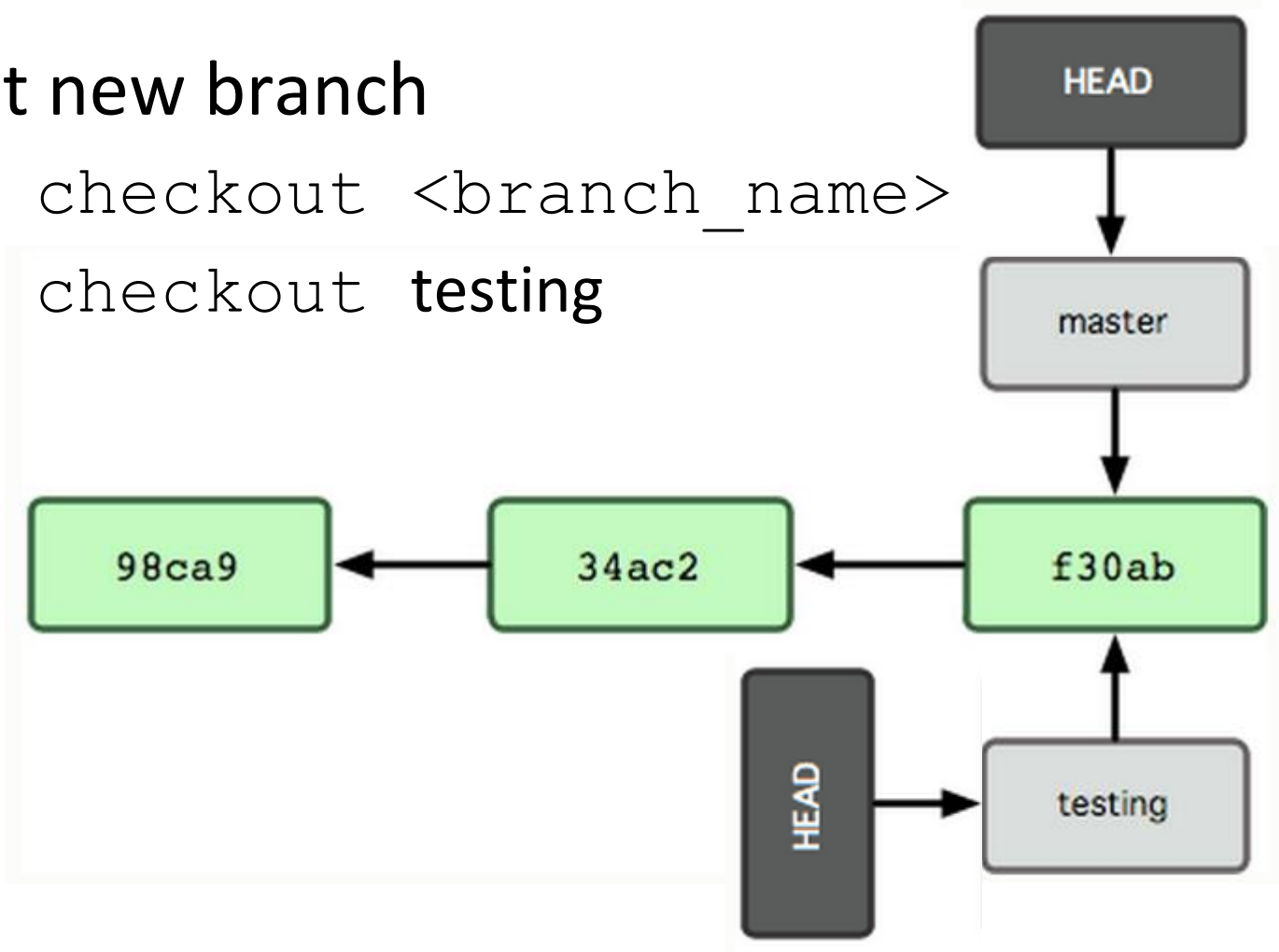
New Commit

- What happens if we make another commit?

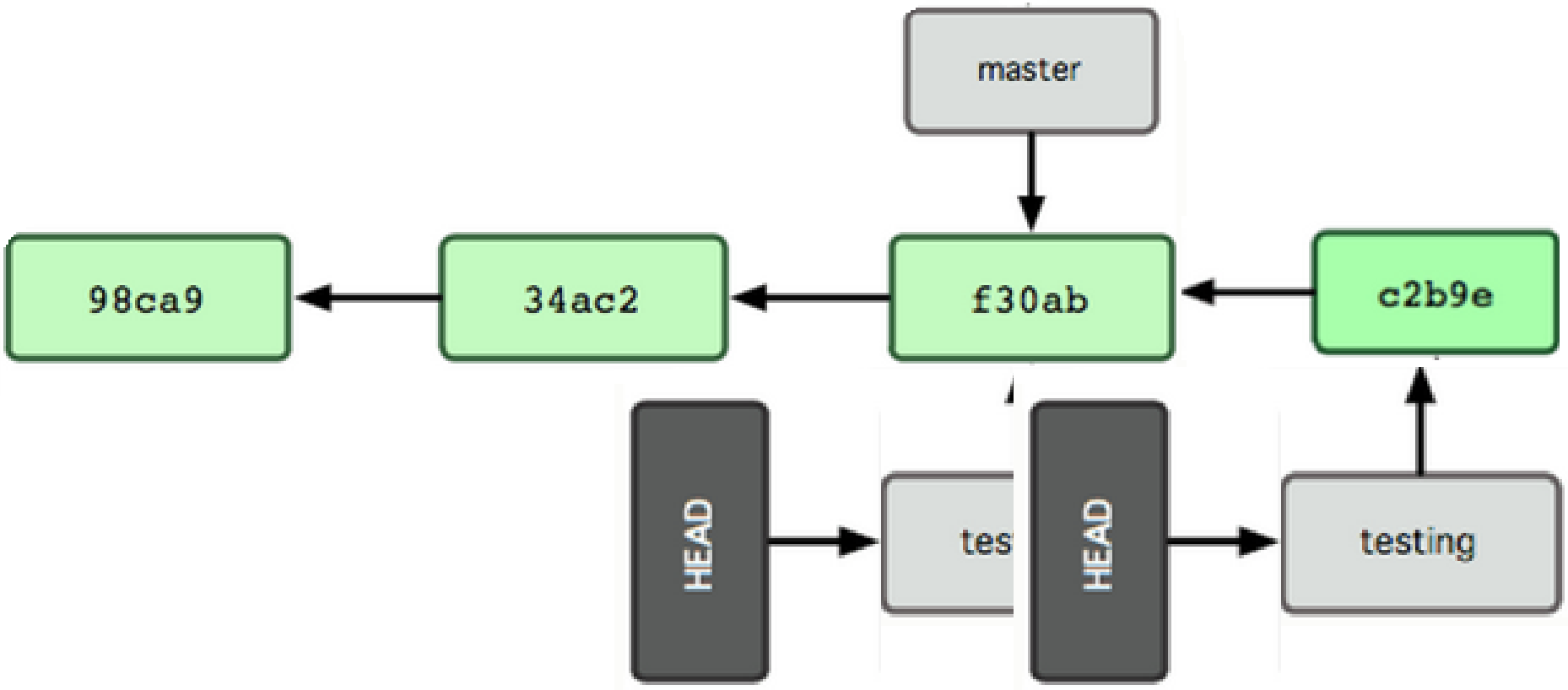


Switching to New Branch

- Check out new branch
  - `$ git checkout <branch_name>`
  - `$ git checkout` **testing**



Commit After Switch



Why Branching?

- Experiment with code without affecting main branch
- Separate projects that once had a common code base
- 2 versions of the project

Git Tag

- Used to tag specific points in a repository's history as being important.
- List existing tags: `git tag`
- Two types of tags (**lightweight**: name attached to hash, **annotated**: name, message, and other info attached to a hash).
- Annotated: `git tag –a <tag_name> -m "tag message"`
- Lightweight: `git tag <tag_name>`

Merging Branches

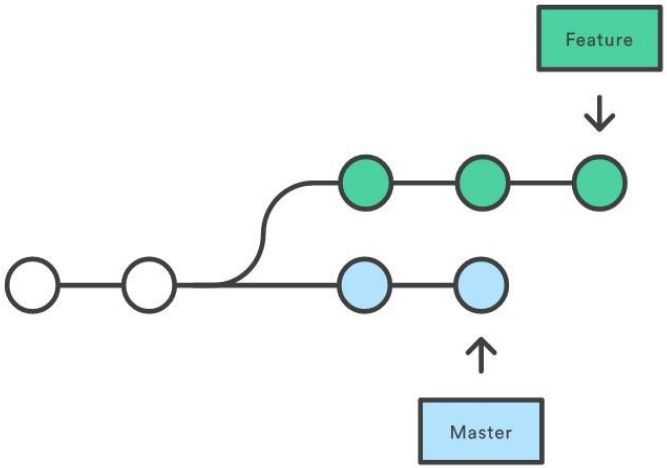
- We’ve seen these scenarios:
  - Fast-forward merge (two-way merge)
  - Three-way merge
  - Rebase

Git Rebase

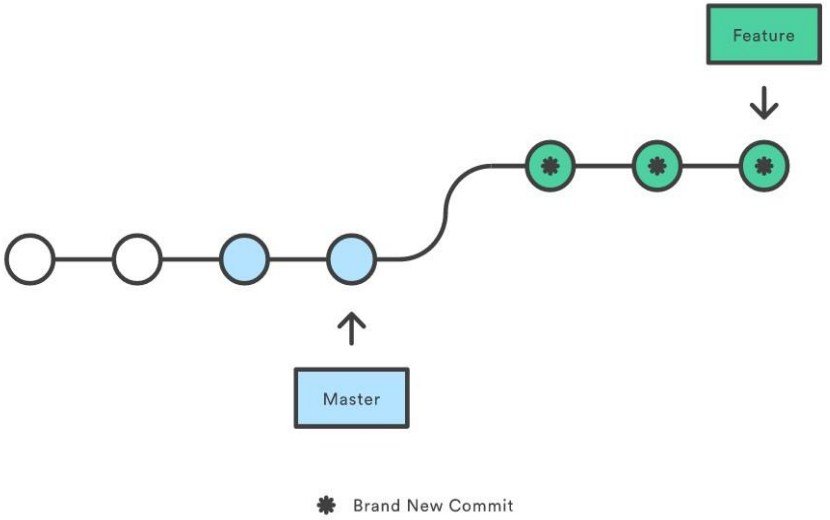
- Rewrites commit history.
- Loses context
- Never use this on public branches!
- How to rebase?

```
$ git checkout feature
$ git rebase master
```

A forked commit history

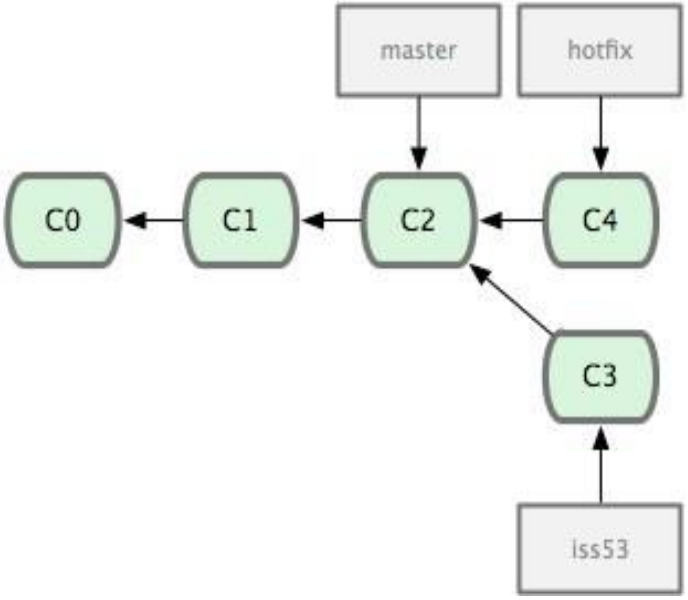


Rebasing the feature branch onto master



Source - https://www.atlassian.com/git/tutorials

Merging



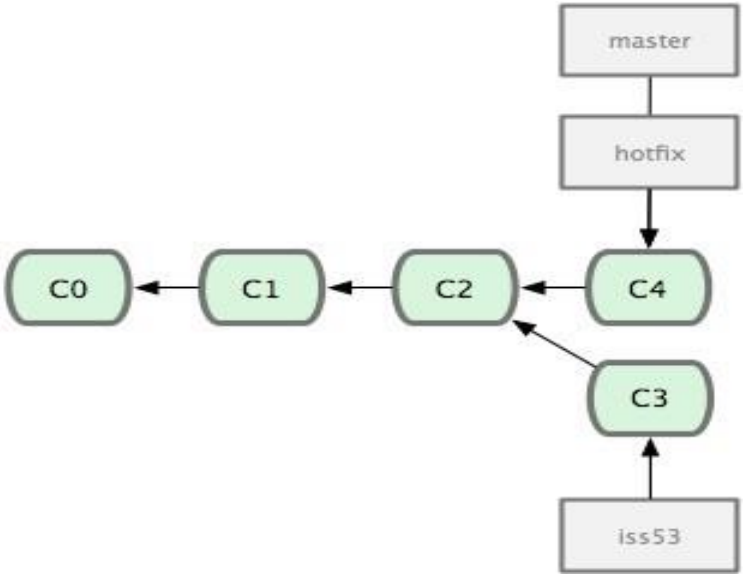


Image Source: git-scm.com

- Merging hotfix branch into master
  - git checkout master
  - git merge hotfix
- Git tries to merge automatically
  - Simple if it is a forward merge
  - Otherwise, you have to manually resolve conflicts

Merging

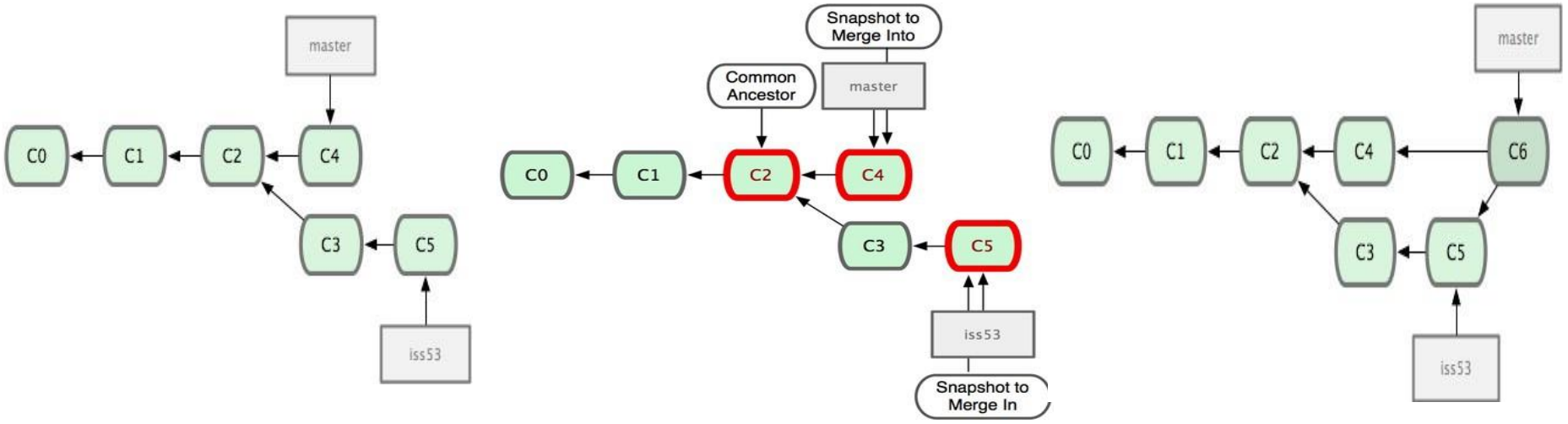


Image Source: git-scm.com

- Merge iss53 into master
- Git tries to merge automatically by looking at the changes since the common ancestor commit
- Manually merge using 3-way merge or 2-way merge
  - Merge conflicts - Same part of the file was changed differently

Merging

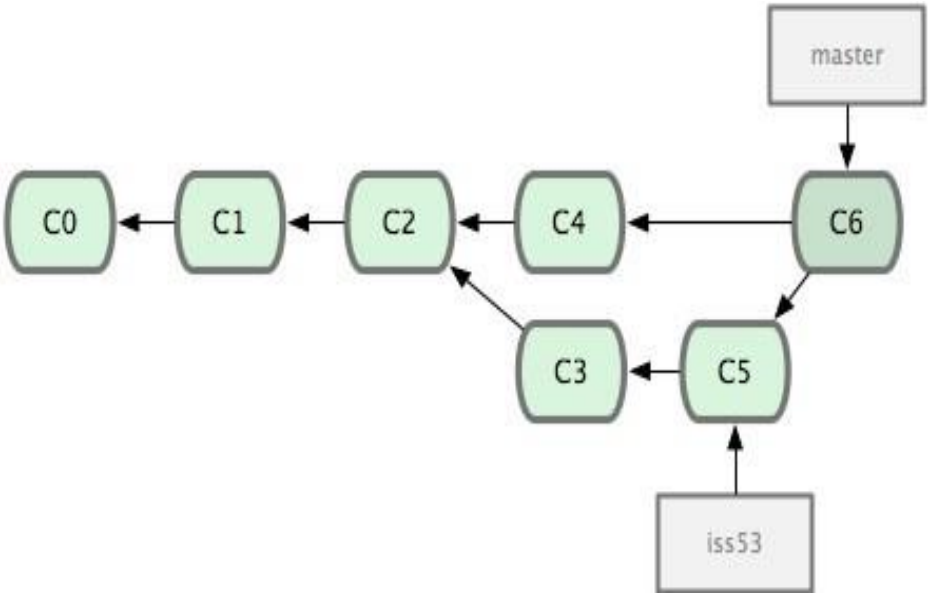


Image Source: git-scm.com

- Refer to multiple parents
  - git show hash
  - git show hash^2 (shows second parent)
- HEAD^^ == HEAD~2

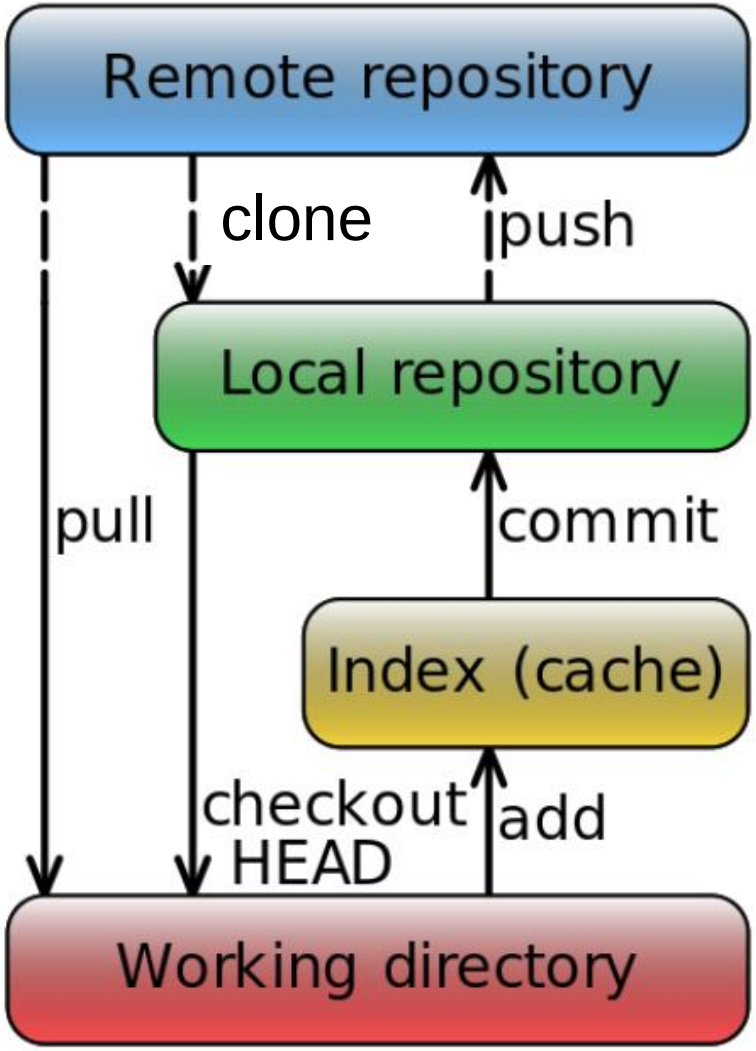
Undoing What Is Done

- **git checkout**
  - Used to checkout a specific version/branch of the tree
  - `git rebase master` (returns to current working version)
- **git revert**
  - Reverts a commit
  - Does not delete the commit object, just applies a patch
  - Reverts can themselves be reverted!
- **Git never deletes a commit object**
  - It is very hard to lose data

More Git commands

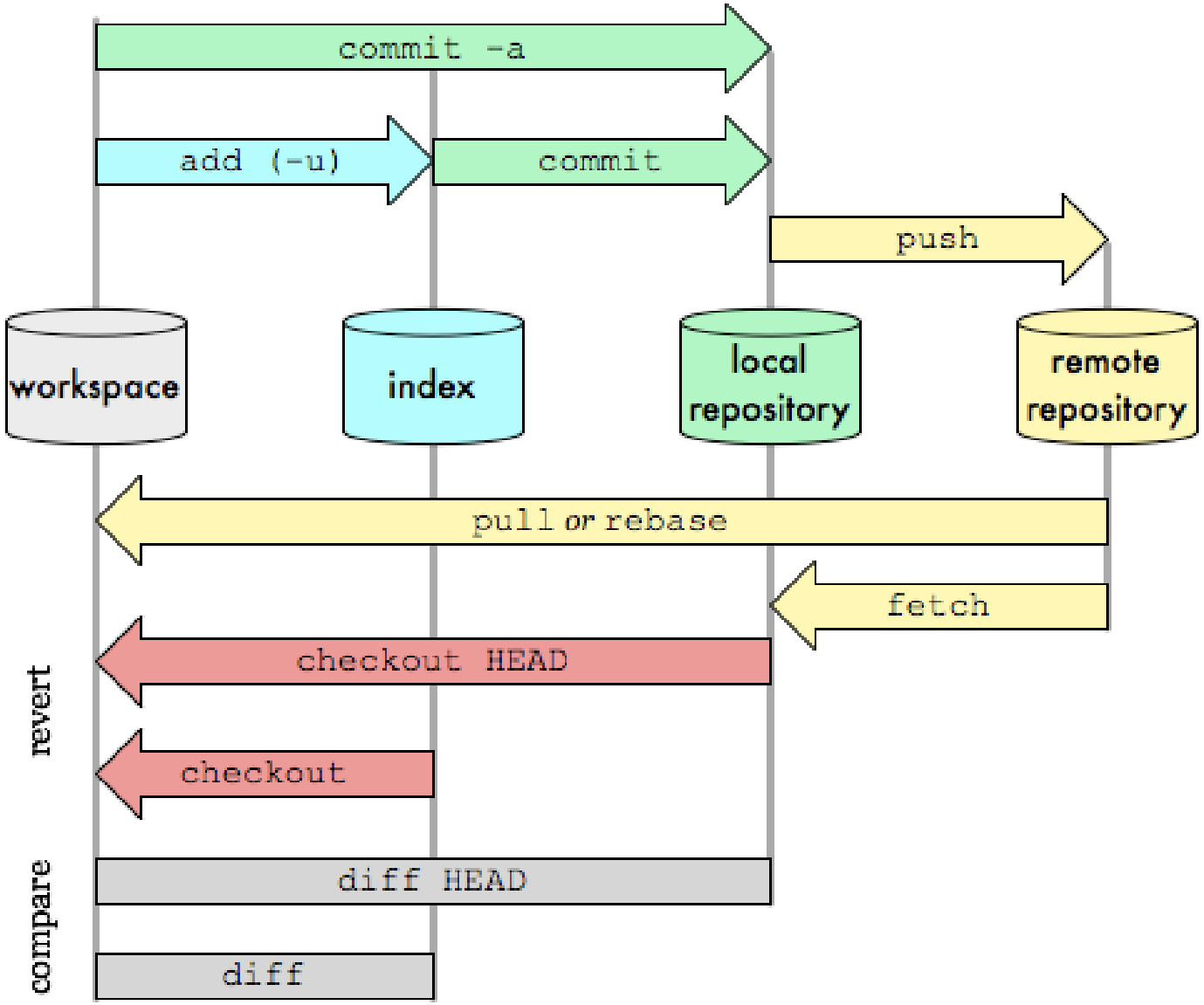
- Reverting
  - **`git checkout HEAD main.cpp`**
    - Gets the HEAD revision for the working copy
  - **`git checkout -- main.cpp`**
    - Reverts changes in the working directory
  - **`git revert`**
    - Reverts commits (this creates new commits)
- Cleaning up untracked files
  - **`git clean`**
- Tagging
  - Human readable pointers to specific commits
  - **`git tag -a v1.0 -m ‘Version 1.0’`**
    - This will name the HEAD commit as v1.0

Overview


Remote repository
clone
push
Local repository
pull
commit
Index (cache)
checkout
HEAD
add
Working directory

Git Data Transport Commands

http://osteele.com



More Git hints

- Git beginner's tutorial (highly recommended):
  - Click here
- Git cheat sheet:
  - Click here
- gitk introduction/tutorial:
  - Click here

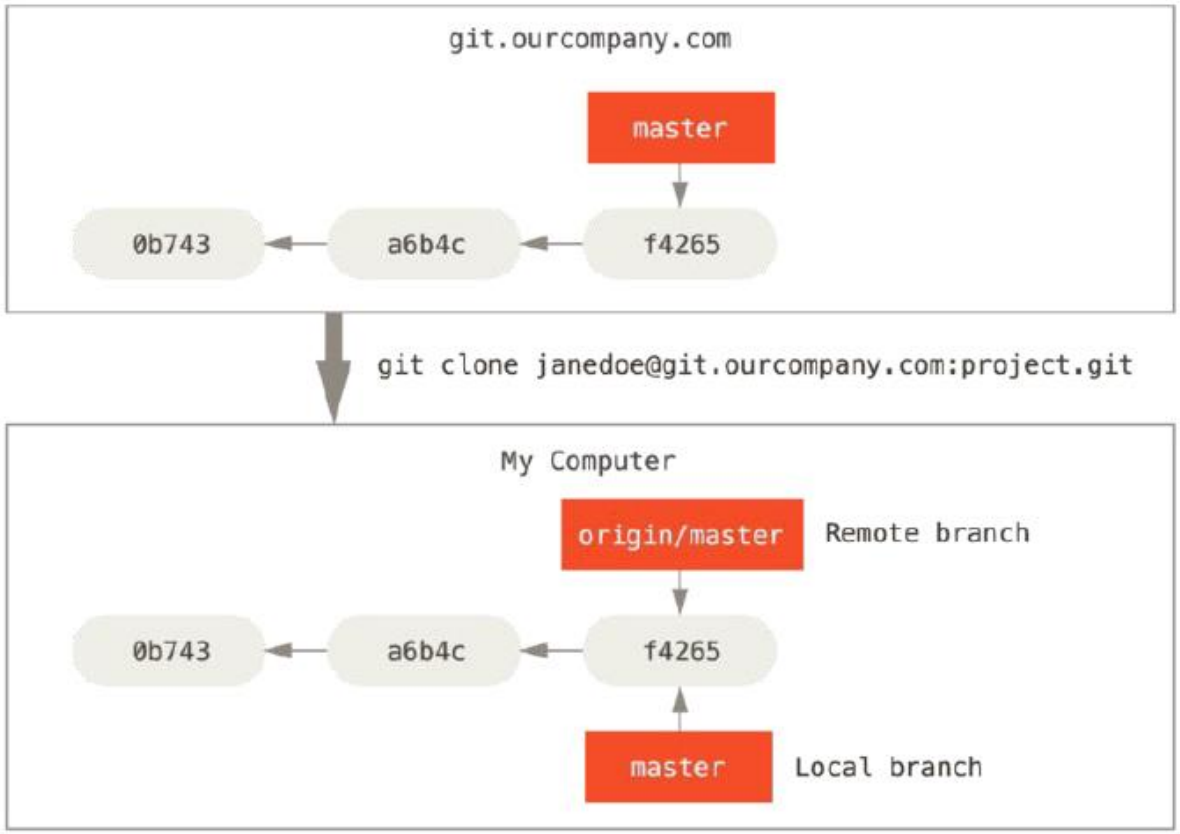
X11 forwarding must be configured properly for gitk!

Remote-Tracking Branches

- **Remote-tracking branches** are references to remote branches
- git automatically updates them whenever you communicate with the remote through a network (pull, fetch, …).
- Designed to reflect the last known state of the corresponding remote branches.
- Remote-tracking branches usually take the form <remote>/<branch>, e.g. origin/master

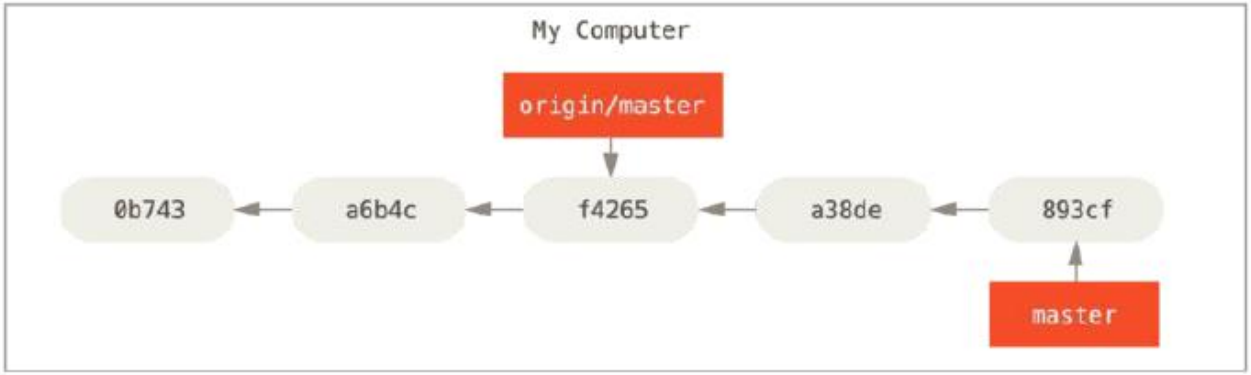
Git clone

- **git clone** clones a remote repository into a local copy.



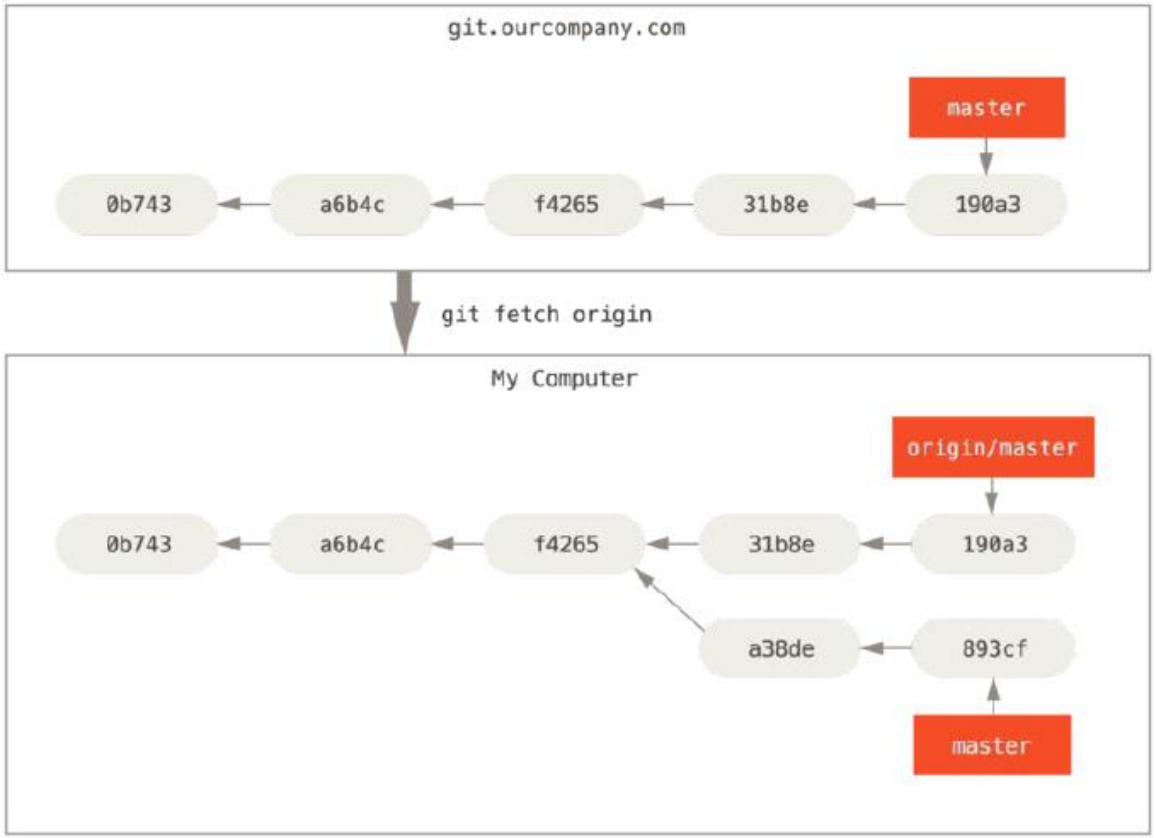
Git fetch

- In order for the remote-tracking branch **origin/master** to get an up-to-date view of the master branch in the remote server, one can use **git fetch <remote>**, which in this case will be **git fetch origin**, because the remote's name is origin.
- Note that there is nothing special about the name **origin**, it's just the default name given to a remote when it's cloned, and one can change the default to something else when cloning.



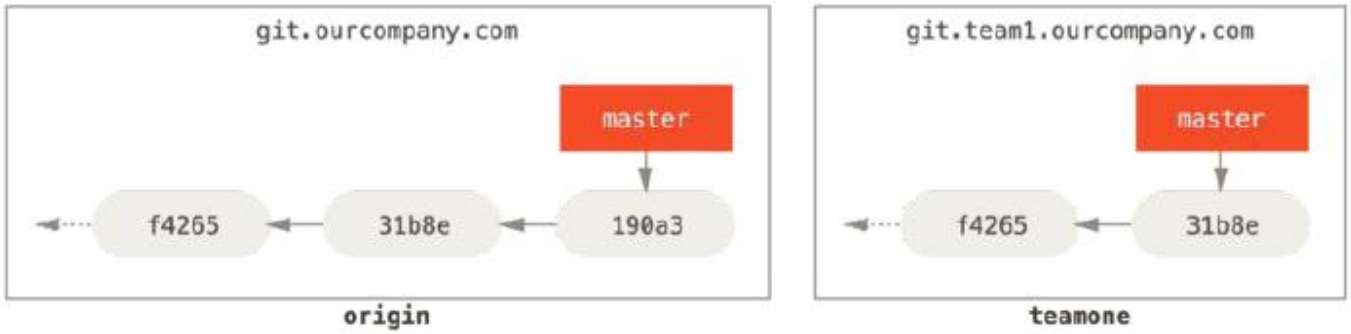
Git fetch

- Note that in this scenario the local master branch developed in parallel with the remote (origin's) master branch to demonstrate the distinction between the local master branch, the remote master branch, and the remote-tracking branch origin/master.
- In practice, one should develop in another local branch for easier bookkeeping.

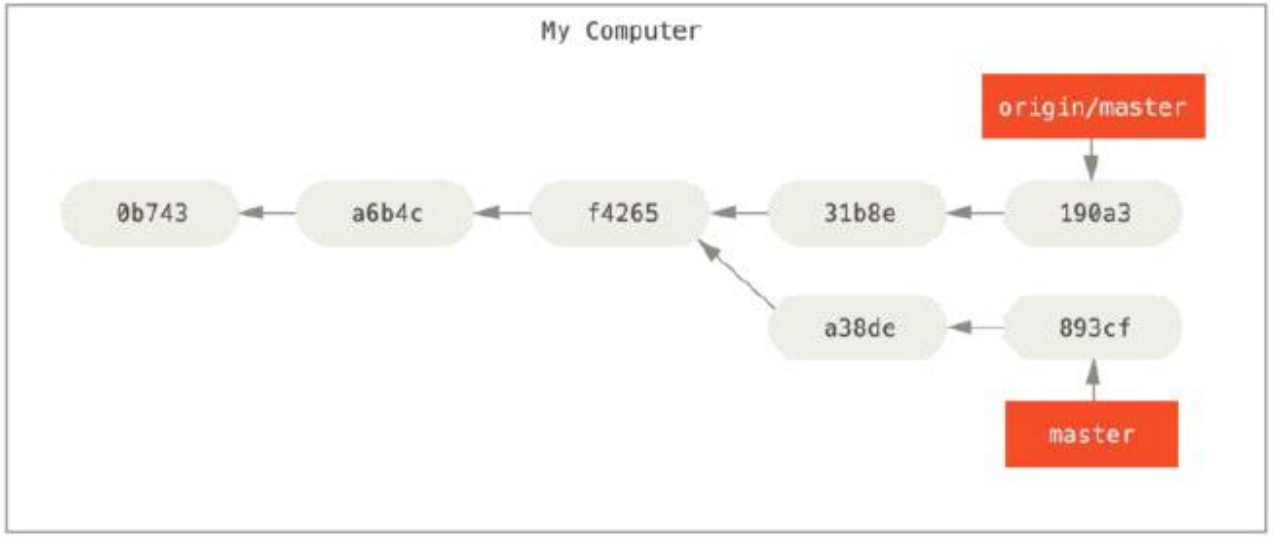


Multiple Remotes

- **git remote add <name> <url>** adds another remote repository located at <url> to the local repository, and will name the newly added remote as <name>.

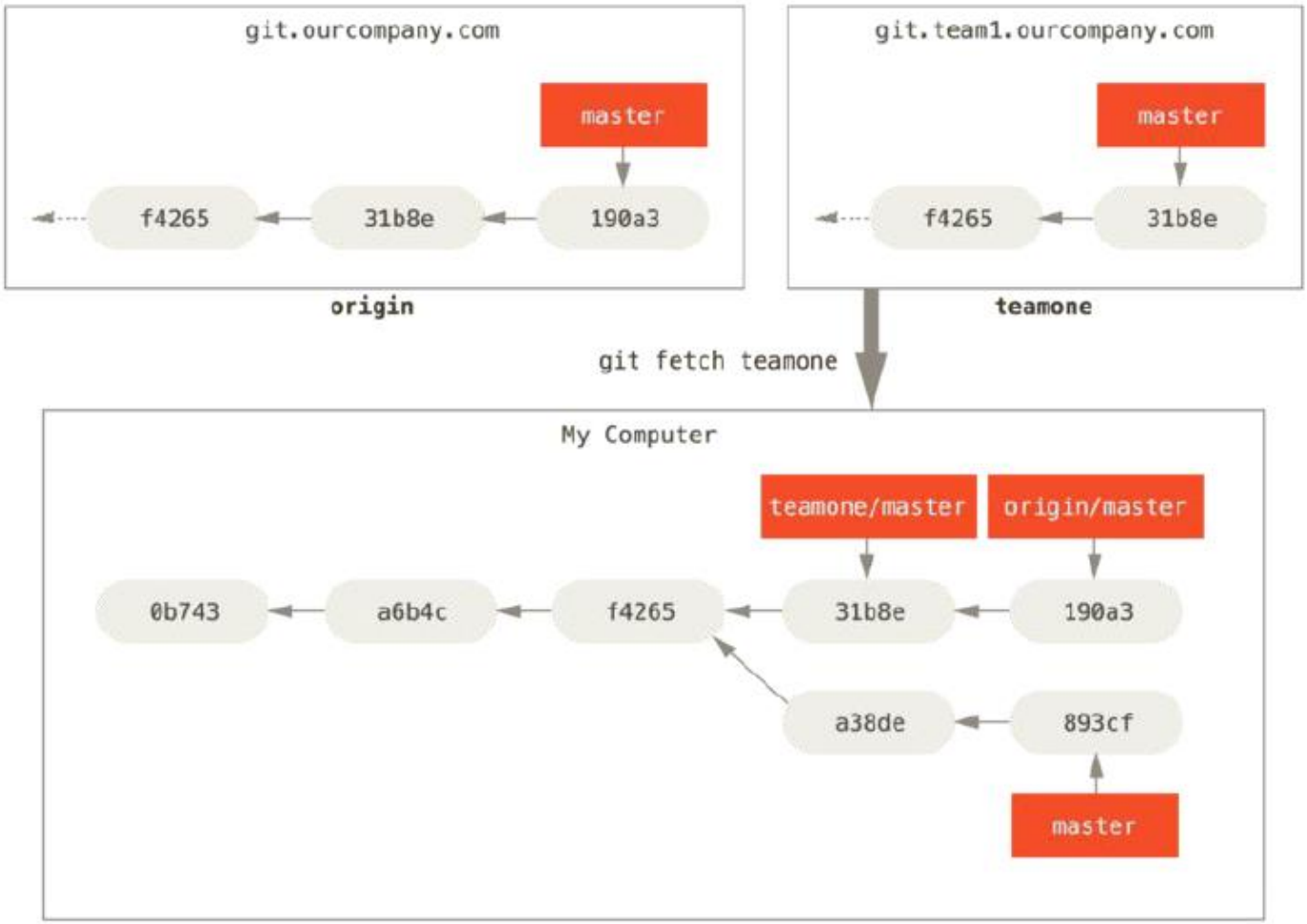



```
git remote add teamone git://git.team1.ourcompany.com
```



Multiple Remotes

- **git remote add <name> <url>** adds another remote repository located at <url> to the local repository, and will name the newly added remote as <name>.



Tracking Branches

- Checking out a local branch from a remote-tracking branch automatically creates a **tracking branch**, and the branch it tracks is called the **upstream branch**,
- i.e. a tracking branch is a local branch that knows it has a counterpart.
- e.g. **git checkout -b <branch> <remote>/<branch>**
- e.g. **git checkout -b b1 origin/b1** will create a local branch **b1**, copied from and tracking the **origin/b1** branch.

Tracking Branches

- **git checkout -b b1 origin/b1**
- can be simplified to **git checkout --track origin/b1**
- which can be simplified to **git checkout b1** IF b1 doesn't exist AND exactly matches a branch name on only one remote.

Tracking Branches

- To set up a local branch with a different name than the remote branch, you can use the first version with a different local branch name, e.g. **git checkout -b b1 origin/branch-one**
- An existing local branch <local-branch> can be set to track a remote branch <new-upstream>, by **git branch -u <new-upstream> [<local-branch>]**
- where the <local-branch> will default to the current branch if not specified.
- Lastly, **git branch -vv** displays the local branches along with the tracking info.

Git pull

- **git fetch** only fetches the changes in the remote repository, but does not modify the local working directory.
- **git pull** is essentially doing **git fetch** and then **git merge**, i.e. it not only fetches the remote changes, but also tries to incorporate those changes into the local working directory.
- By default, **git pull** pulls from the branch the current local branch is tracking.

Git push

- **git push <remote> <branch>** will update the <remote>'s <branch> with the local <branch>.
- e.g. **git push origin b1**
- However, this is a shortcut for typing **git push origin b1:b1** which specifies pushing the local b1 branch onto origin's b1 branch.
- Therefore, if the local and remote's branch names do not match, we can for example type **git push origin b1:branch-one** which will push the local b1 branch onto origin's branch-one branch.

Deleting Remote Branches

- **git push <remote> --delete <branch-name>**

Revision Selection

- Can refer to a commit through its short hash (first few Hex digits) or through its branch name (if its pointed to by that branch)

- **git show abcd ← hash**

  and

- **git show b1**

  are equivalent

Ancestry References

Suppose we have the following commits:

```
$ 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 ...
```

A caret **^** at the end of a reference refers to the parent of that commit. e.g. **c455717^** will refer to **f3fe695**

For merge commits such as **2e25043, 2e25043^** will refer to its first parent, **c455717** while **2e25043^2** will refer to its second parent **4950521**.

Moreover, since HEAD points to **2e25043, HEAD^** and **2e25043^** are equivalent.

Ancestry References

Suppose we have the following commits:

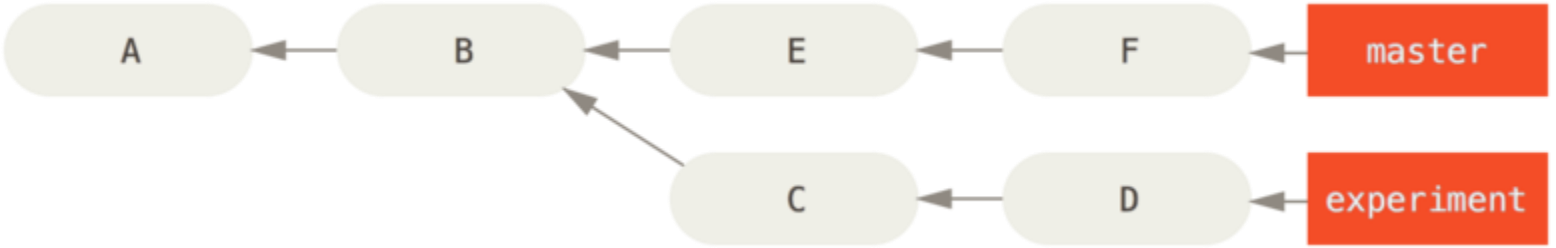
```
$ git log --pretty=format:'%h %s' --graph
* 2e25043 Merge pull request #18 from ...
|\
| * 4950521 fix sim by normalizing SNP columns
| * 69402cd generate g effects
|/
* c455717 tabulate_output.py
* f3fe695 Merge pull request #17 from ...
```

A tilde ~ also refers to the first parent, so HEAD^ and HEAD~ are equivalent.

HEAD~2 refers to the first parent of the first parent of HEAD, and the same pattern goes for HEAD~3 etc.

Commit Ranges

- Double dot syntax: Show range of commits reachable from one commit but aren't reachable from the other.



git log master..experiment will show commits in experiment not reachable from master. So the output will be

D

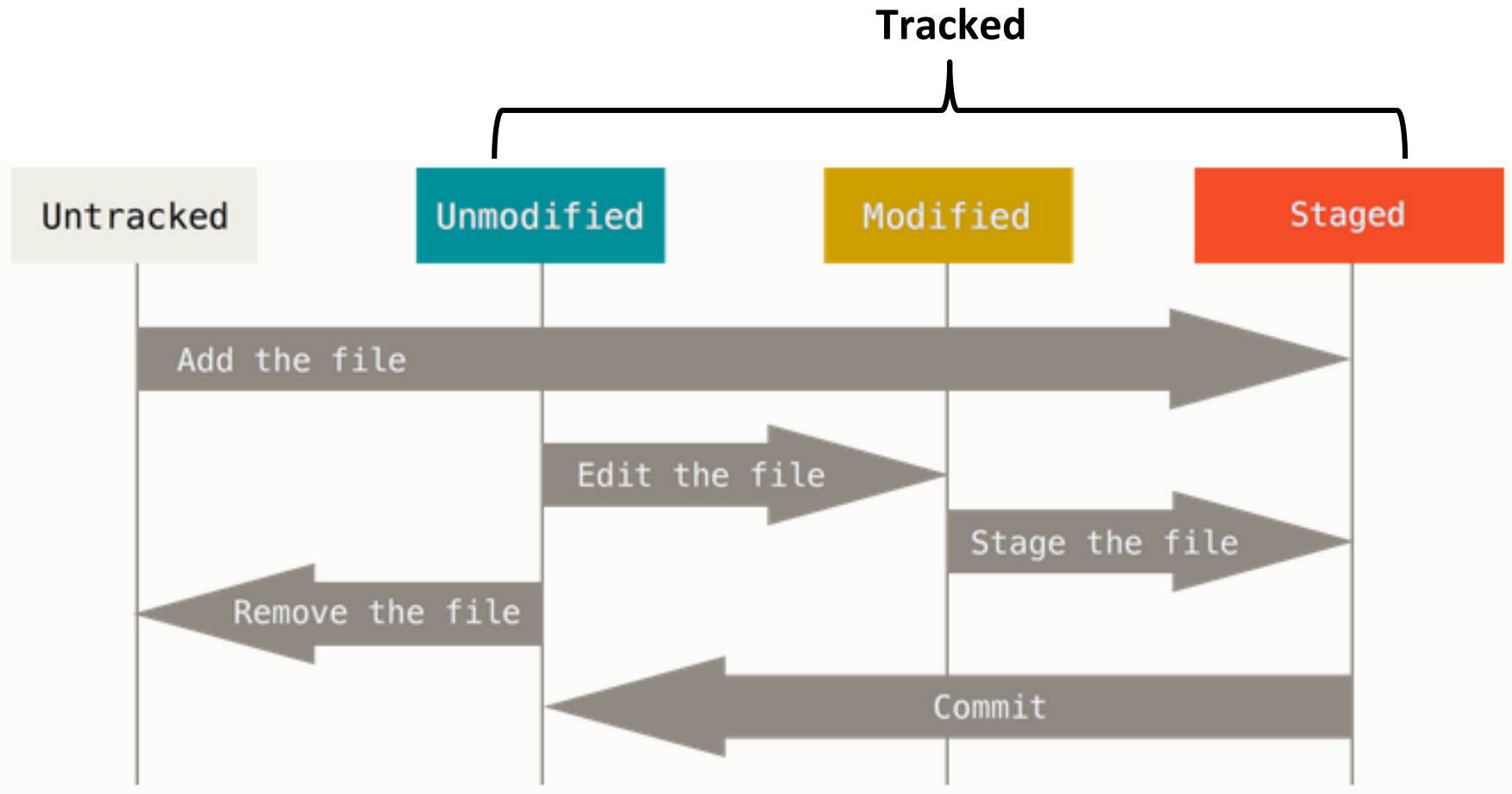
C

On the other hand, **git log experiment..master** will output

F

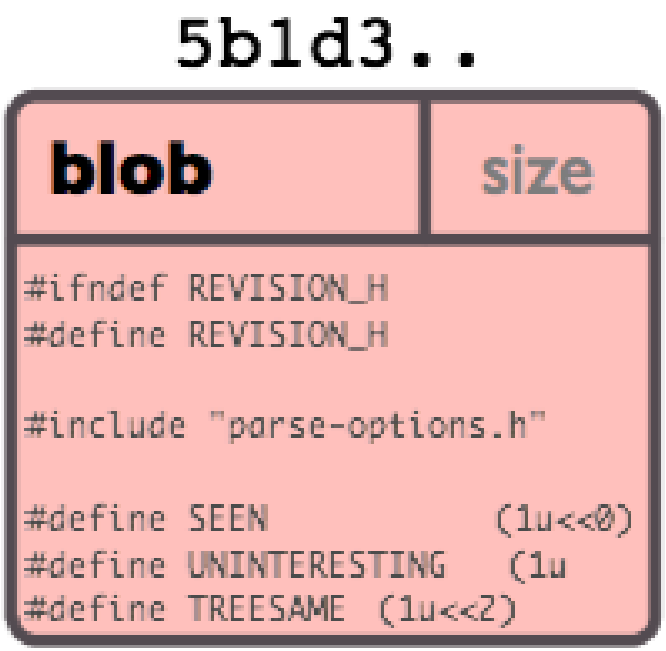
E

Git File Status Lifecycle



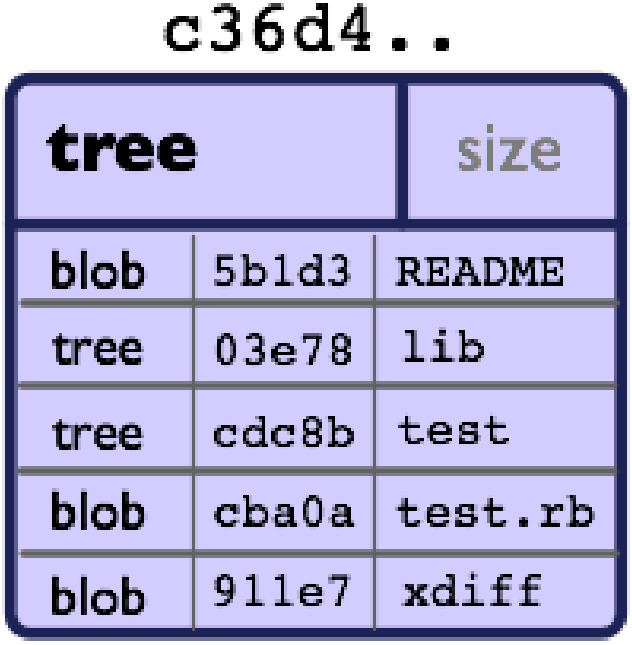
Git Objects - Blob

- Stores the contents of a file.
- Is a chunk of binary data.
- No filename and does not refer to anything else.
- If two files have the same contents, they share the same blob object.



Git Objects - Tree

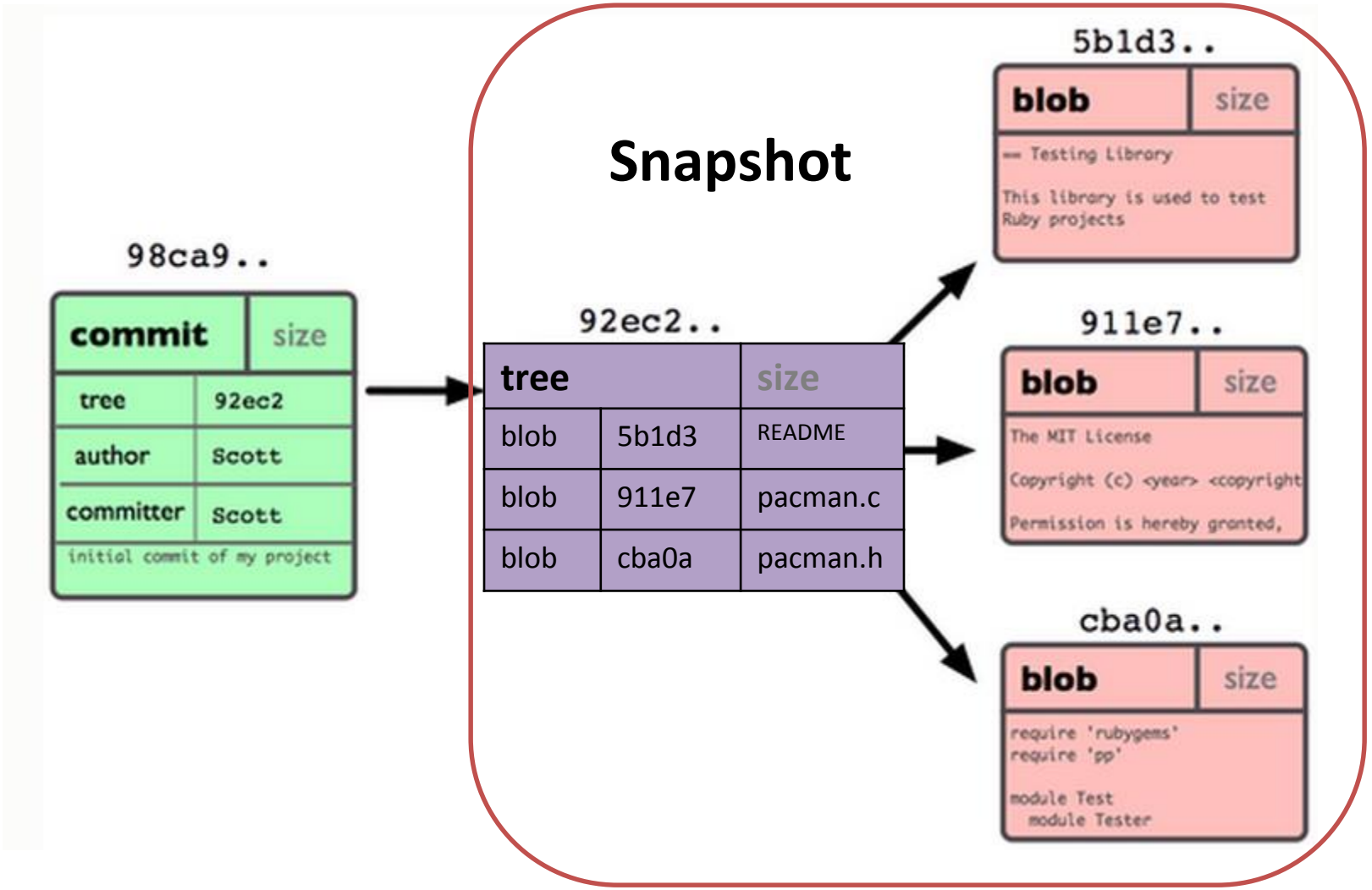
- Has pointers to blobs and other trees.
- Represents the contents of a directory or a subdirectory.
- Contains a list of entries with attributes (mode, object type, SHA1 hash, name).



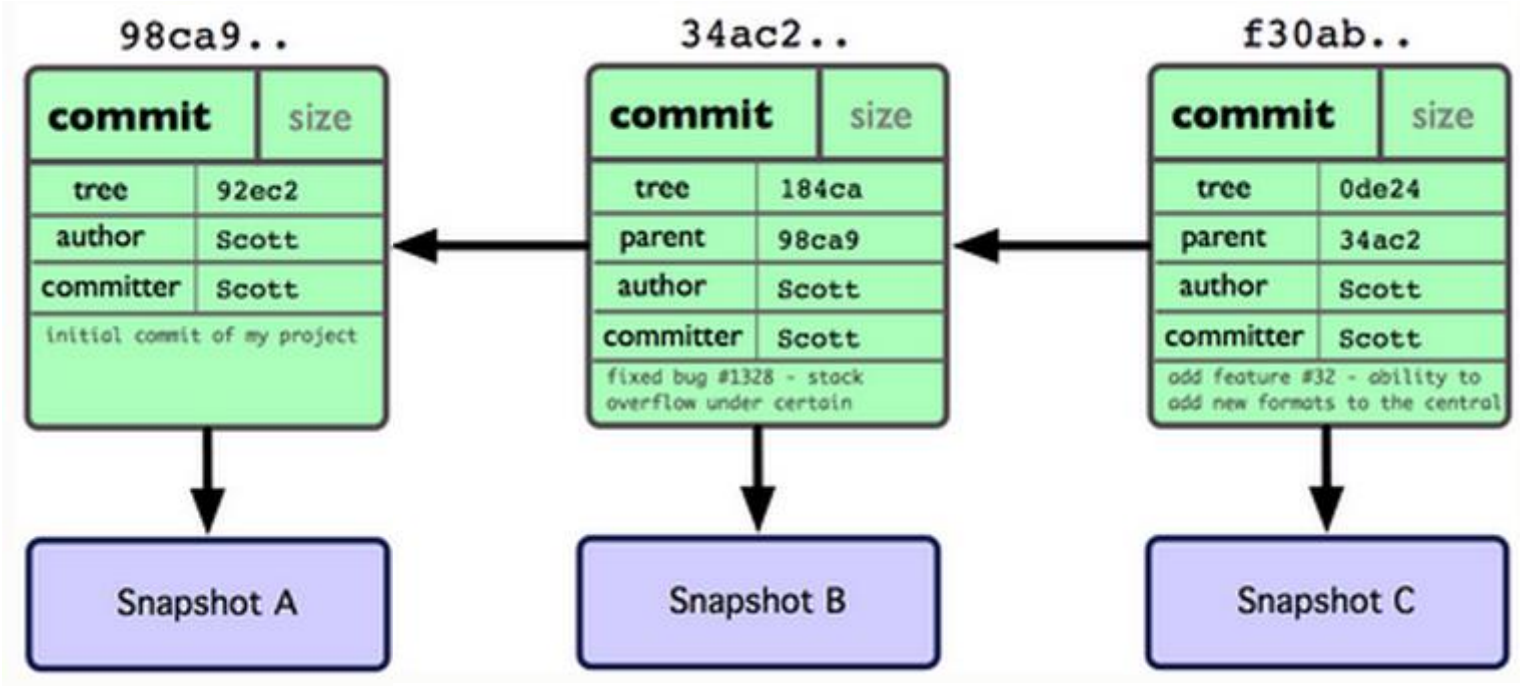
Git Objects - Commit

- Stores information relating to a “snapshot” of our repository.
- Has a pointer to a **tree** that represents the contents of our repository at a certain point in time.
- Has pointers to parent commit objects.
- Has other metadata (author, comment, …)

Git Repo Structure



After 2 More Commits…

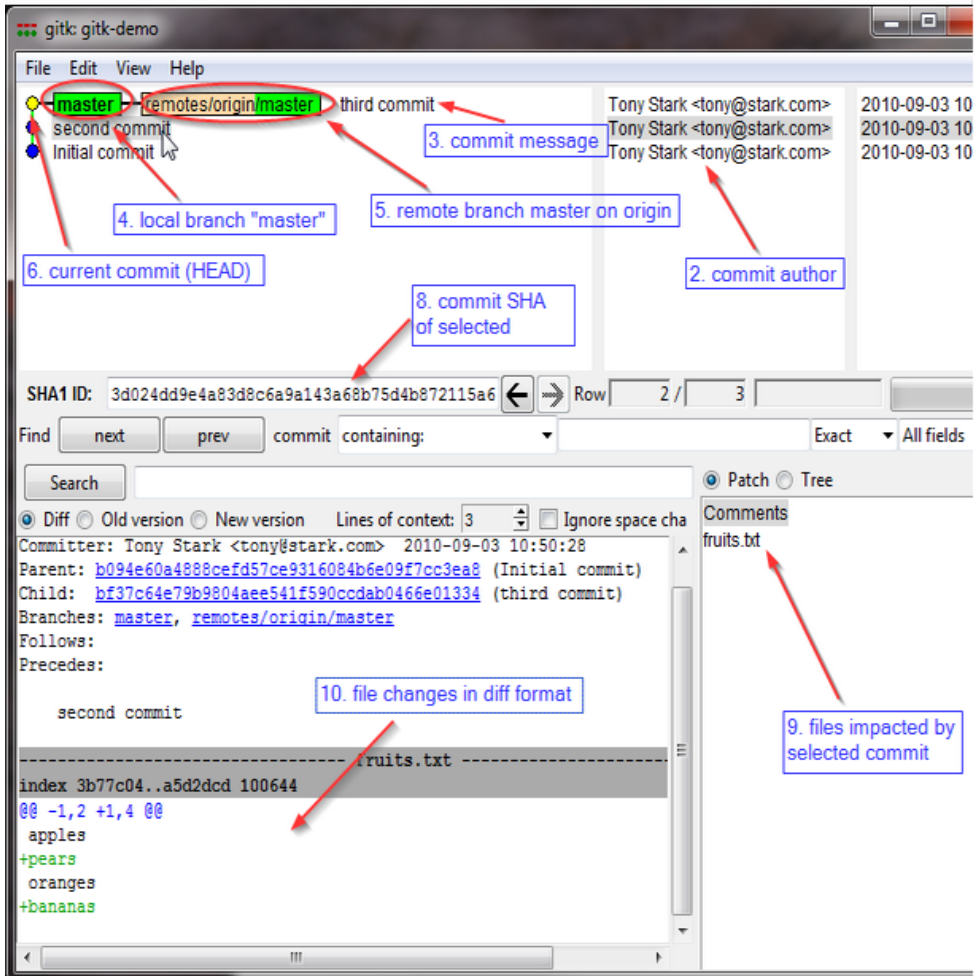


Git Objects - Tags

- Contains object name, object type, tag name, …
- Used to tag specific points in a repository's history as being important.
- Lightweight tags are not represented by tag objects.

Gitk

- A repository browser
  - Visualizes commit graphs
  - Used to understand the structure of the repo
  - Tutorial: http://lostechies.com/joshuaflanagan/2010/09/03/use-gitk-to-understand-git/



Gitk

- SSH into the server with X11 enabled
  - ssh -X for OS with terminal (OS X, Linux)
  - Select “X11” option if using putty (Windows)
- Run gitk in the ~eggert/src/gnu/emacs directory
  - Need to first update your PATH
    - $ export PATH=/usr/local/cs/bin:$PATH
  - Run X locally before running gitk
    - Xming on Windows

Useful Links

- Git Tutorial
  - By topic
- Git Beginner's Tutorial (alternative)
  - Step by step tutorial + testing terminal
- Git Visual Guide
  - For visualizing what each command does
- Git From The Bottom Up
  - For understanding how Git is structured and the details of how it tracks changes