Git & GitHub

CSC301

What is Git? Why do we use it?

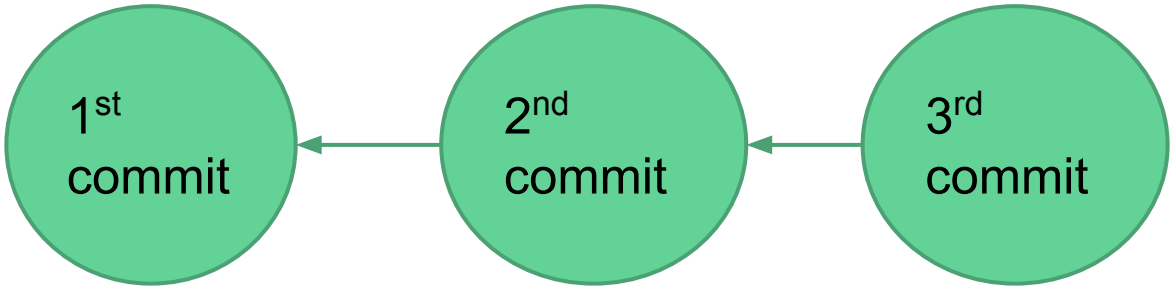
Version Control in CSC301

- Git
  - Standard Distributed Version Control System
  - Open-source
- GitHub
  - Hosting service for Git repositories
  - Web-based toolset for code/project management.
  - Free for public (and private) projects
  - Atlassian Bitbucket is a similar tool

Git Commits = Snapshots

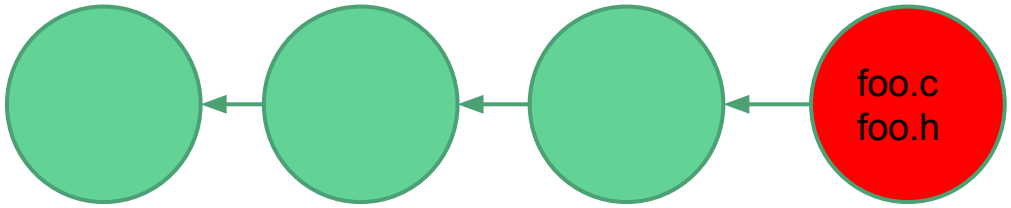
- Need to think differently about git
  - Forget about revisions to individual files
- Each *commit* is a snapshot of the full codebase
  - That's the *abstraction*. Under the hood, Git stores differences (to optimize space usage)
- Git repo (repository) is a graph of commits
  - A version of the code is a node in the graph
  - History is described by paths in the graph

Repo as graph of commits



Each *commit* represents a version of the code - a snapshot in time
A path of commits represents its history.

Repo as graph of commits



A new commit (i.e. snapshot) is created and added to the graph.

Then, you can commit **changes** to the repo

This is a simple linear graph of commits - next week we'll show more complicated graphs

Working Locally

1. Working directory

   The actual files on your machine.

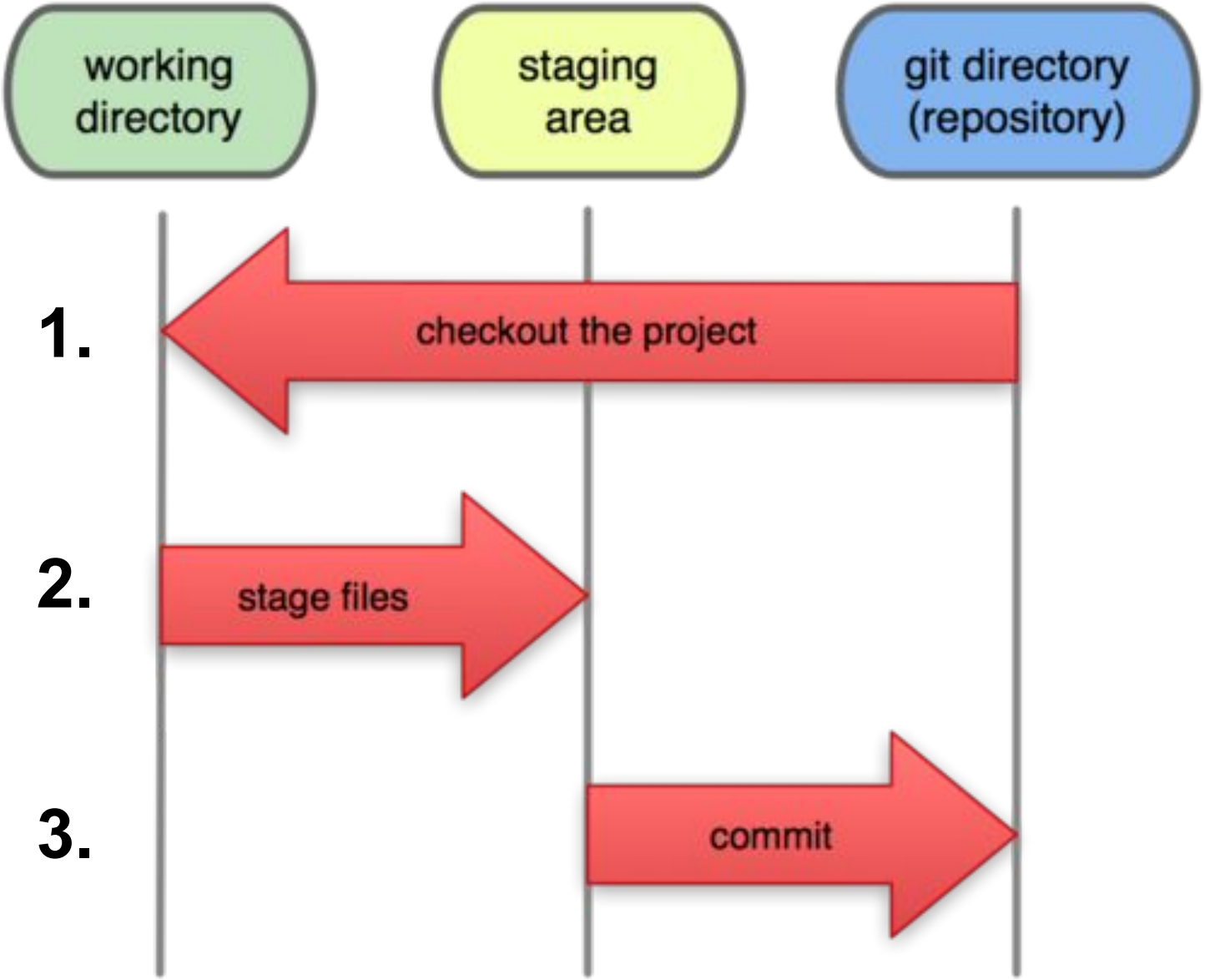
2. *Staging Area* (aka *index*)

   Intermediate storage for code changes.

3. Repository (aka *history*)

   The graph of commits.

Local Operations



Why the extra step?

- The extra step (i.e., *staging*, before committing) gives us more granularity
  - Choose which changes you want to commit
    - E.g.: Do not commit temporary changes made only for the purpose of local testing
  - When saving changes, can break them into multiple commits.
    Each commit should have its own concise, meaningful message.
  - **Goal**: Work in a traceable manner
- Warning: Do not forget to do them all in order!

Basic Git commands

- `init`
- `status`
- `add`
- `commit`
- `log`

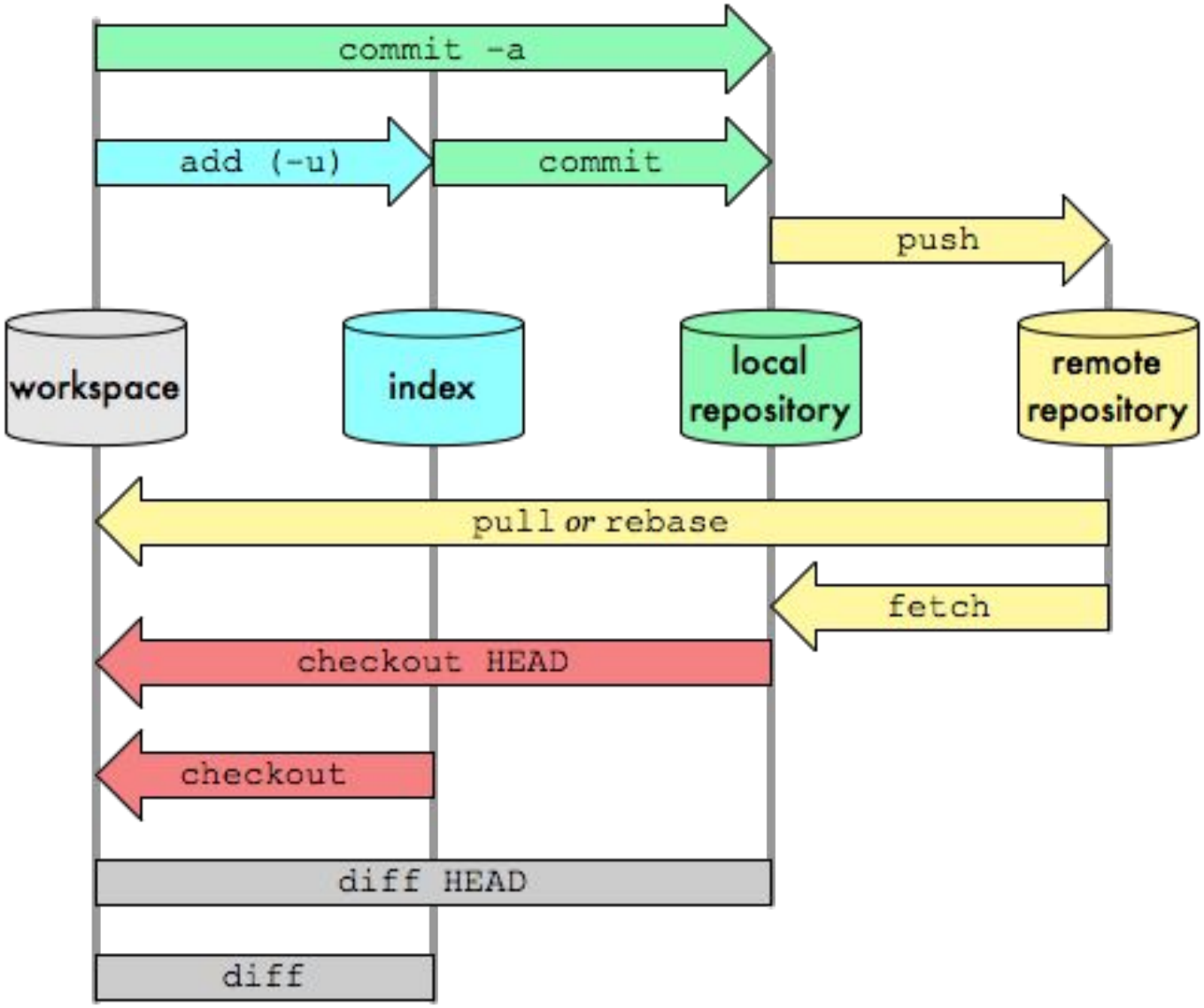
Think Outside Of The (One) Box

- The commands we just saw are local
  - That is, they are done entirely on your machine
- A more common scenario involves remote repos:
  - ***Clone*** some remote repo to your machine
  - ***Commit*** changes locally
  - When ready, ***push*** changes from your machine to the remote repo

Q: Where do we store remote repositories?

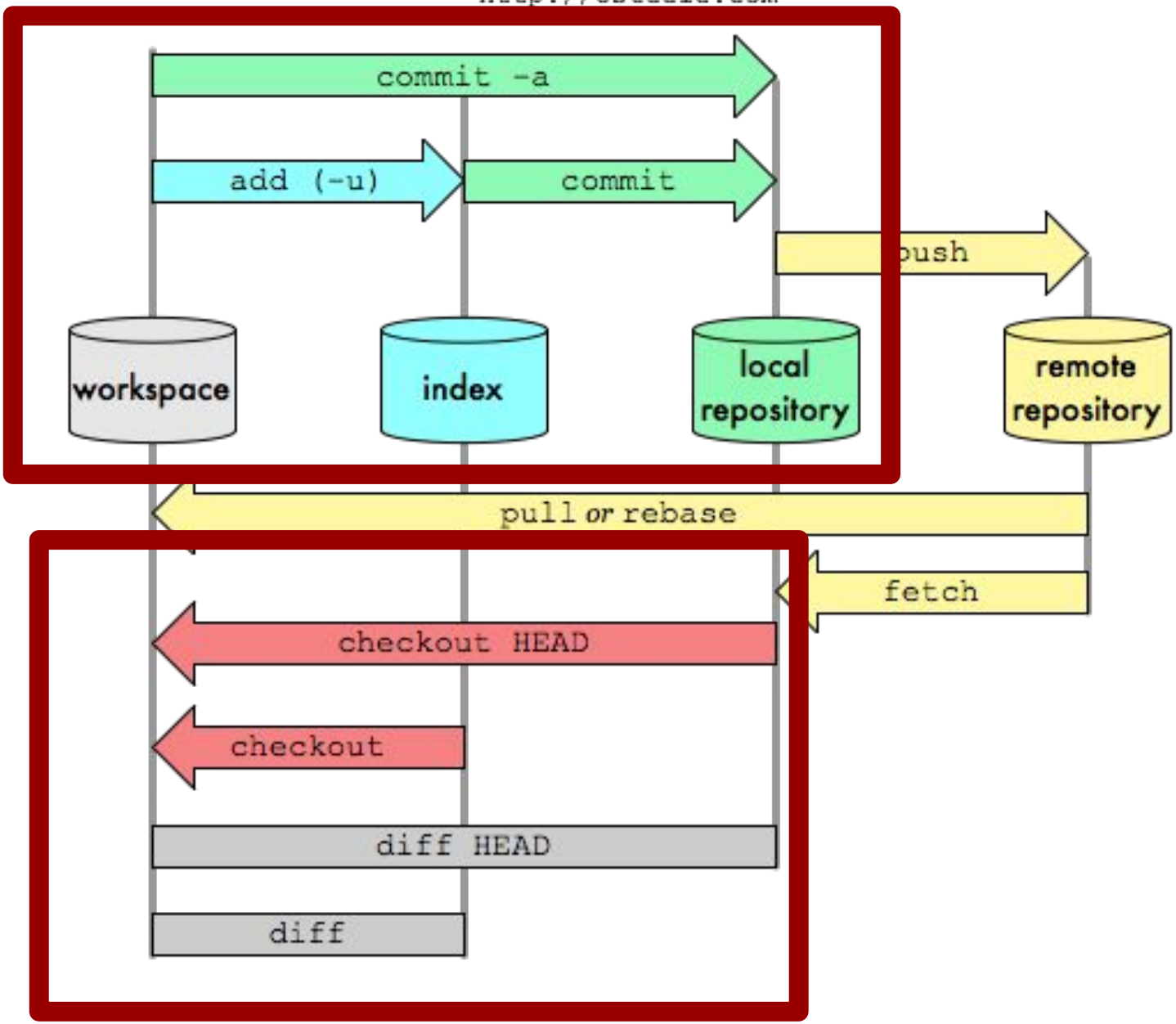
Git Data Transport Commands

http://osteele.com

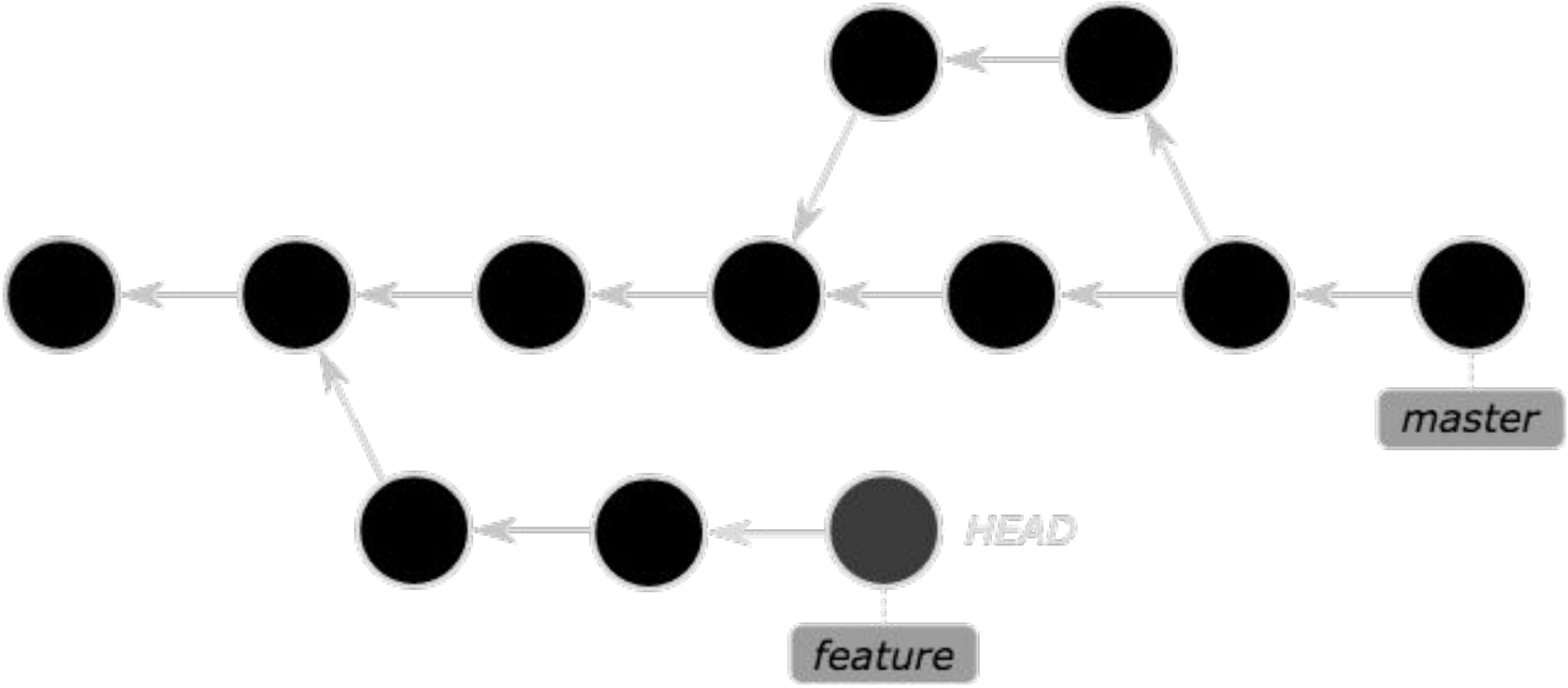


Git Data Transport Commands

http://osteele.com



The Git *commit graph*: a **DAG** structure



Git is really just a fancy **DAG** editor.

Recall: Repo as a graph of commits

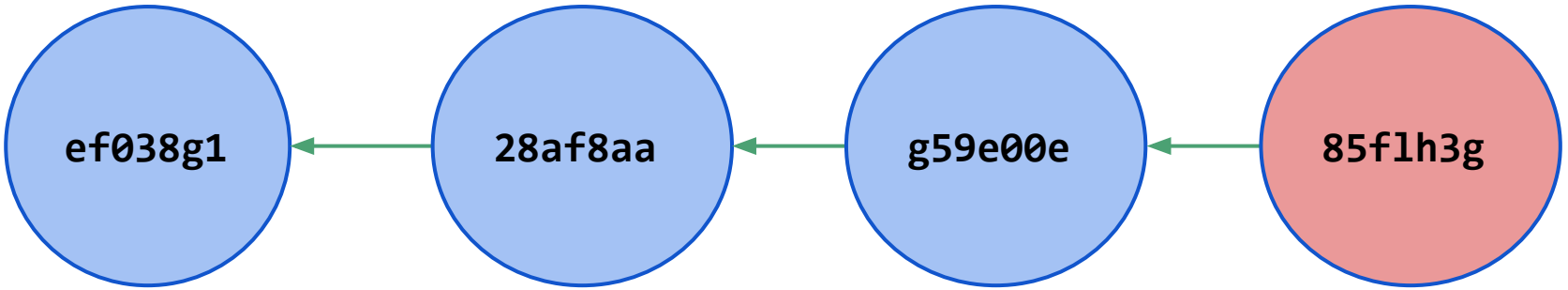
- Commits store the **changes**
  - Identified by `SHA-1` hash
- Each *commit* represents a snapshot of the codebase
- A path of commits represents its history

What’s in a commit hash?

```
sha1(
    commit message  => "initial commit"
    committer       => Ala Shaabana <ala.shaabana@utoronto.ca>
    commit date     => Sat Jan 10 10:56:57 2014 +0100
    author          => Ala Shaabana <ala.shaabana@utoronto.ca>
    author date     => Sat Jan 10 10:56:57 2014 +0100
    tree            => 9c435a86e664be00db0d973e981425e4a3ef3f8d
)
```

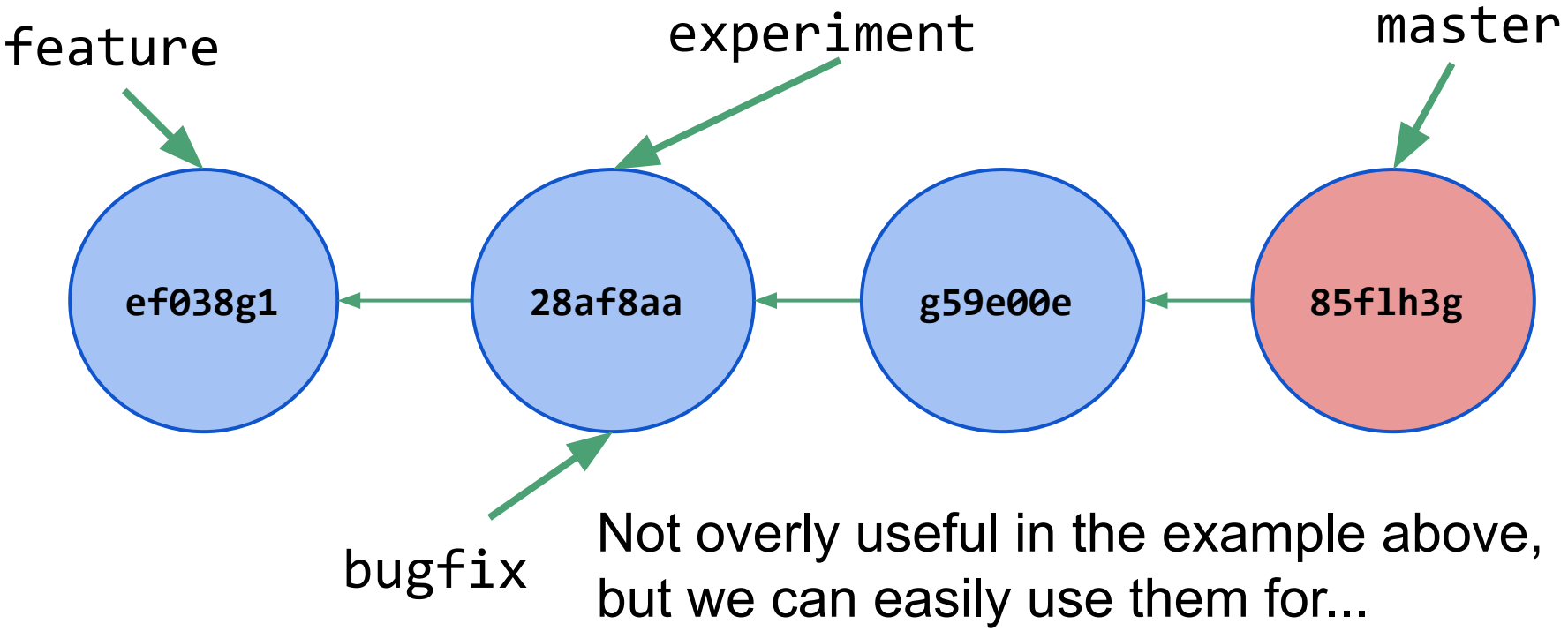
Adding to the history: linear graph

```
$ git add -A  #stage all additions/changes in index
$ git commit -m ‘fixed off by one error’
[master 85flh3g] fixed off by one error
```



References

- Easier to identify commits by references (refs)
- Simply labels of individual commits
  - Can put multiple labels on a commit



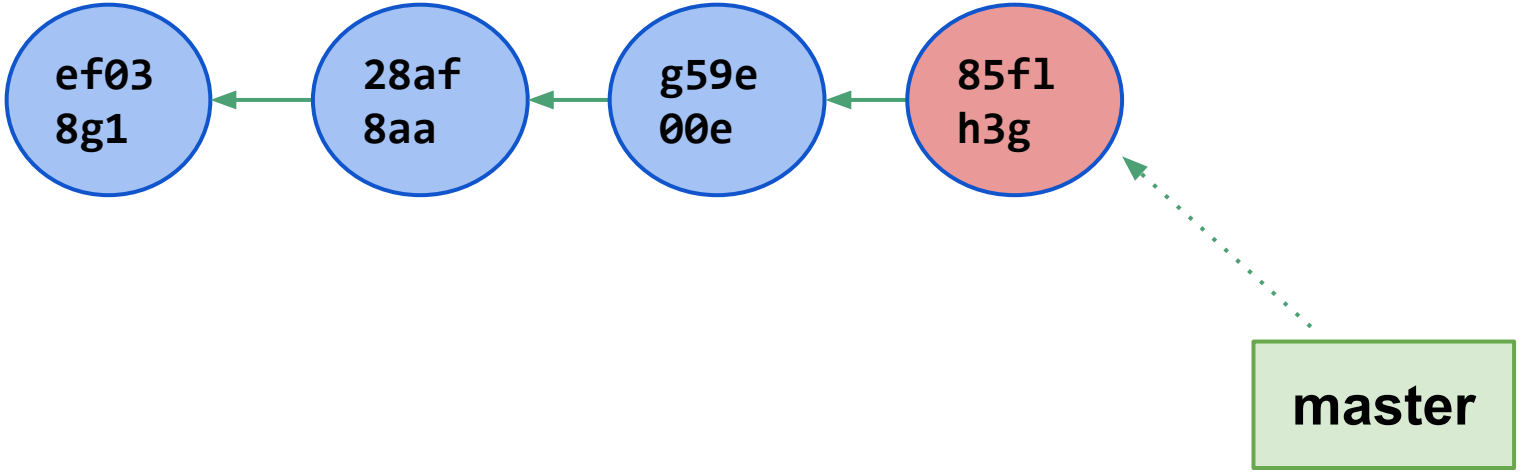
Not overly useful in the example above, but we can easily use them for...

Branch Workflow

- Branching is used to create different **lines of development**
  - Features
  - Bug fixes
  - Experimental code
  - Releases
- Keep the master branch clean
  - Code that gets into master has usually been vetted in many ways (released, tested, etc.)

Branching

- A **branch** is just a reference to a commit in the graph.
  - *Not* another commit

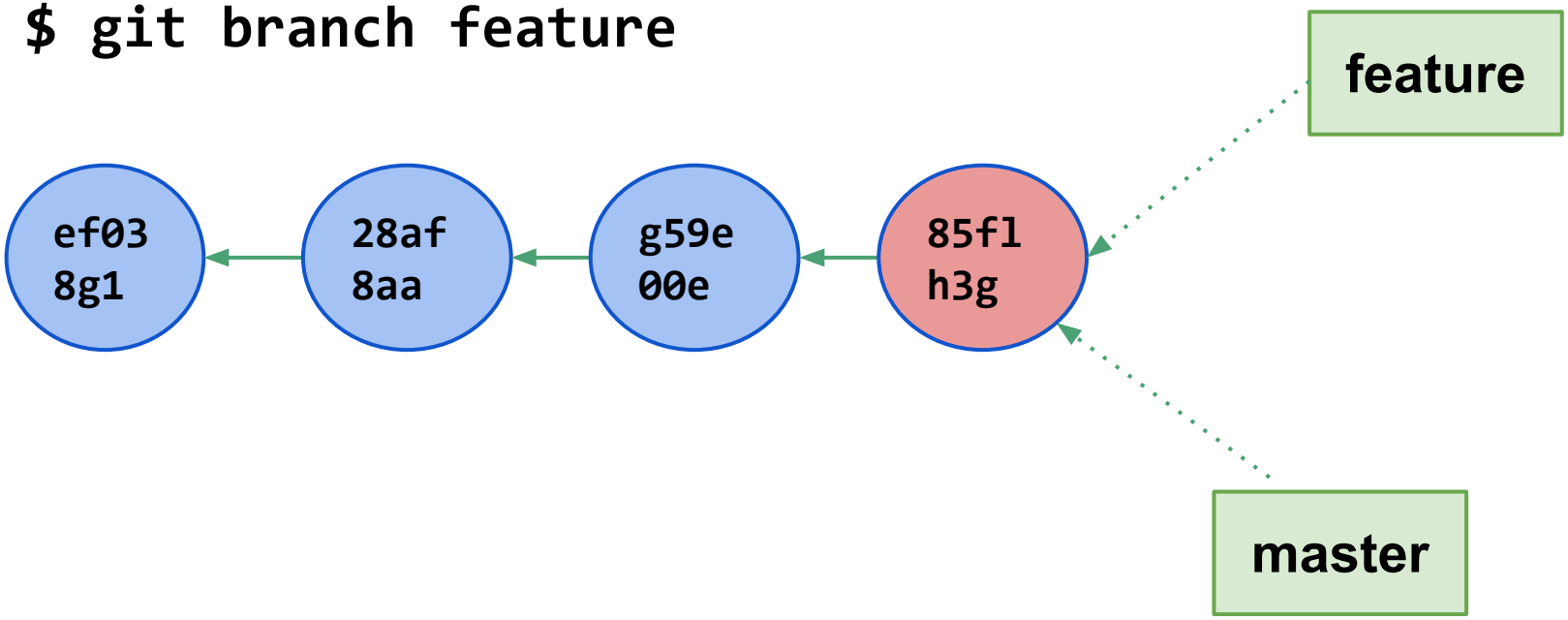


Default starting branch is **master**

Branching

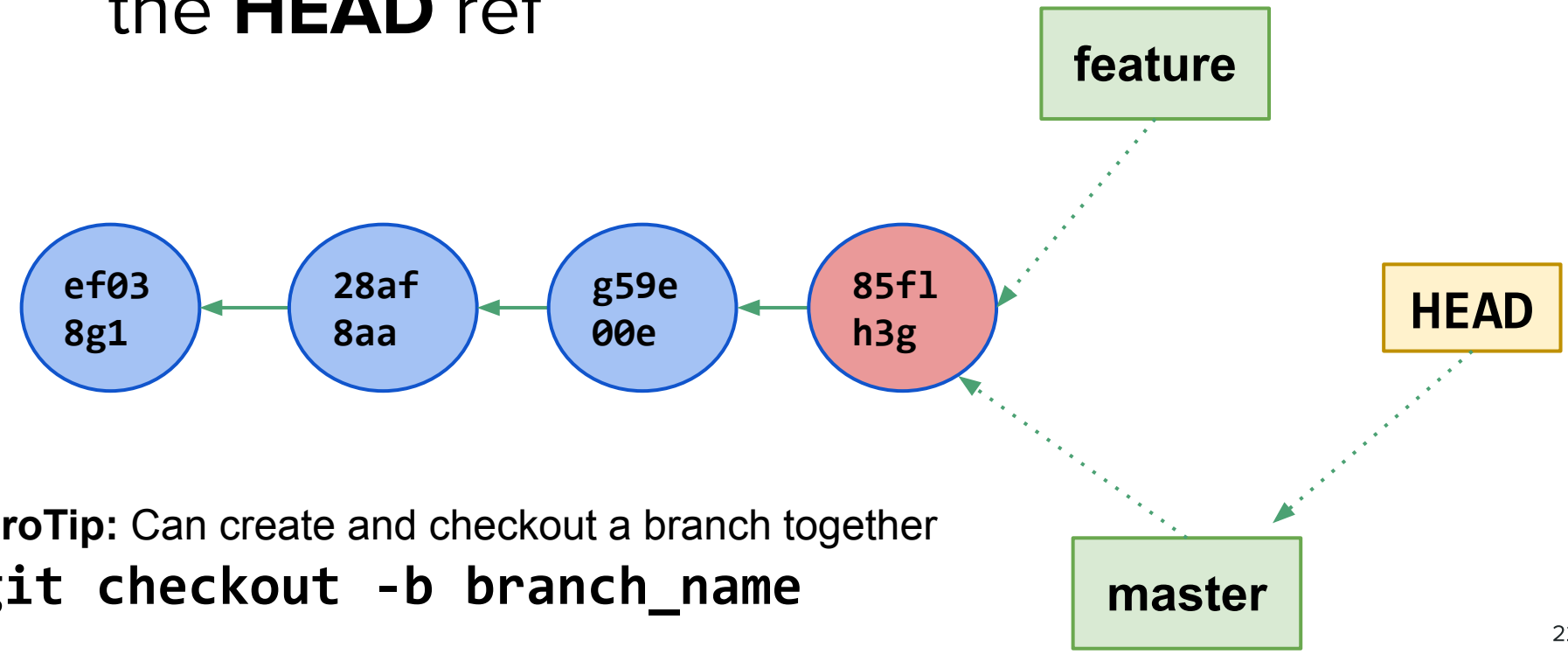
- Can create a new branch
  - Just adds another ref to latest commit

```
$ git branch feature
```



Checking out a branch

- To specify which branch we should work on, we must **checkout** the branch
- Checked out branch pointed to by the **HEAD** ref



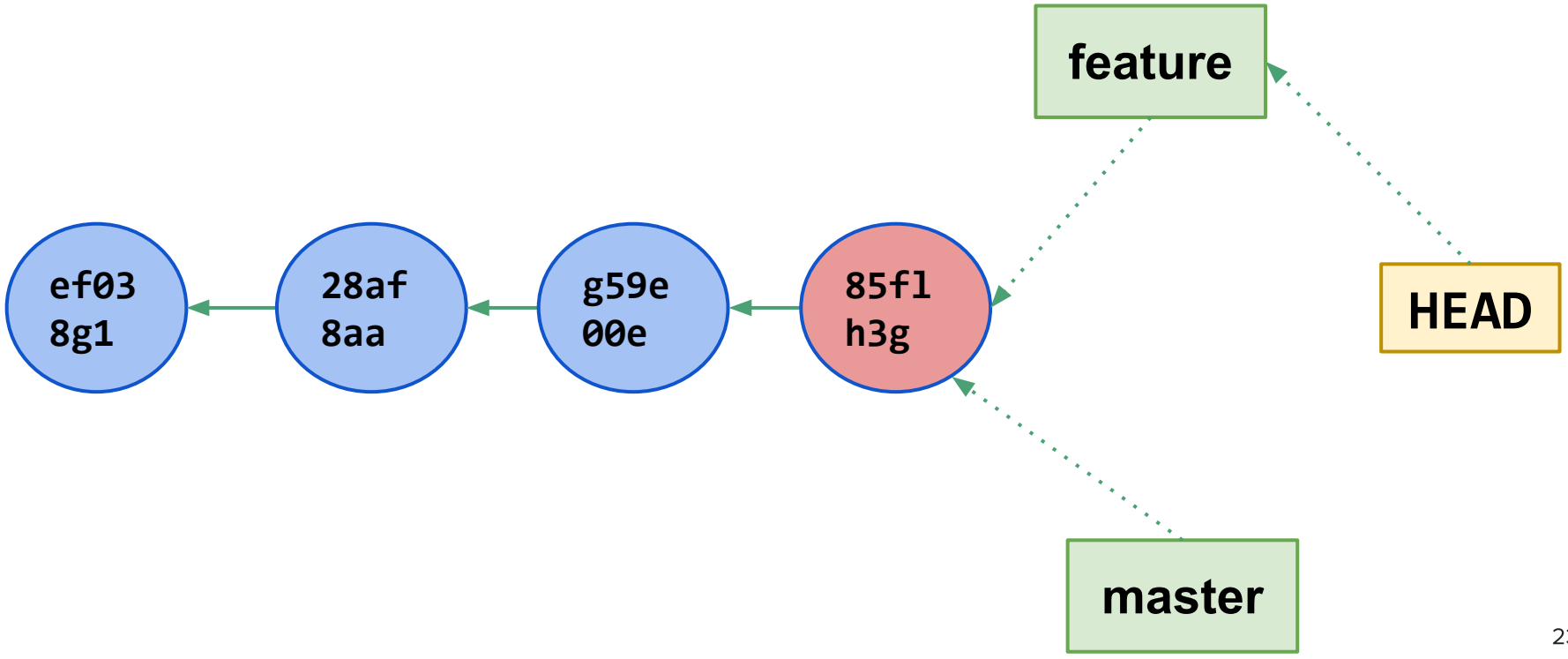
ProTip: Can create and checkout a branch together

```
git checkout -b branch_name
```

Checking out a branch

- Checked out branch pointed to by the **HEAD** ref

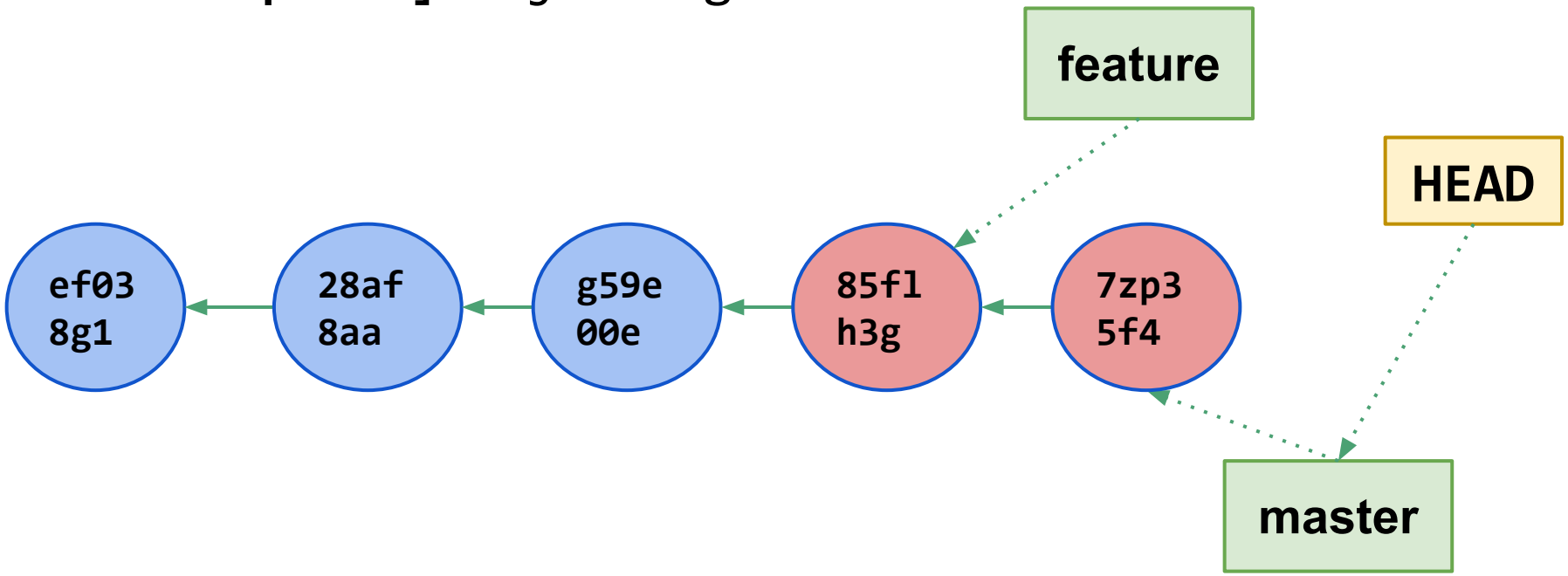
```
$ git checkout feature
```



Working on a checked out **master** branch

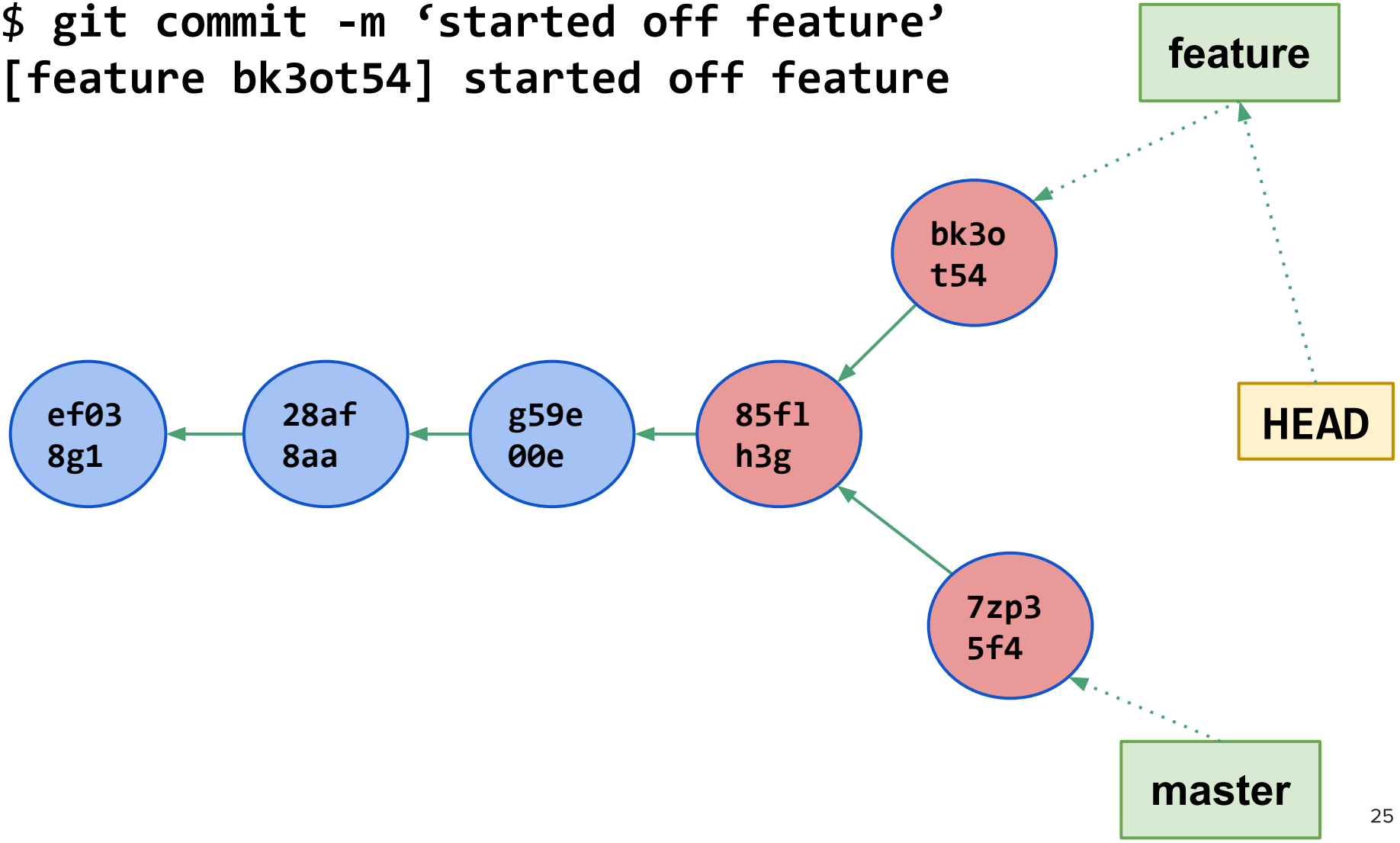
- In this case, any commit we make will be made to the master branch

```
$ git checkout master
$ git commit -m ‘major bug fix’
[master 7zp35f4] major bug fix
```



Diverging branches

```
$ git checkout feature
$ git commit -m ‘started off feature’
[feature bk3ot54] started off feature
```



Inevitably, our diverging branches will need to merge back together

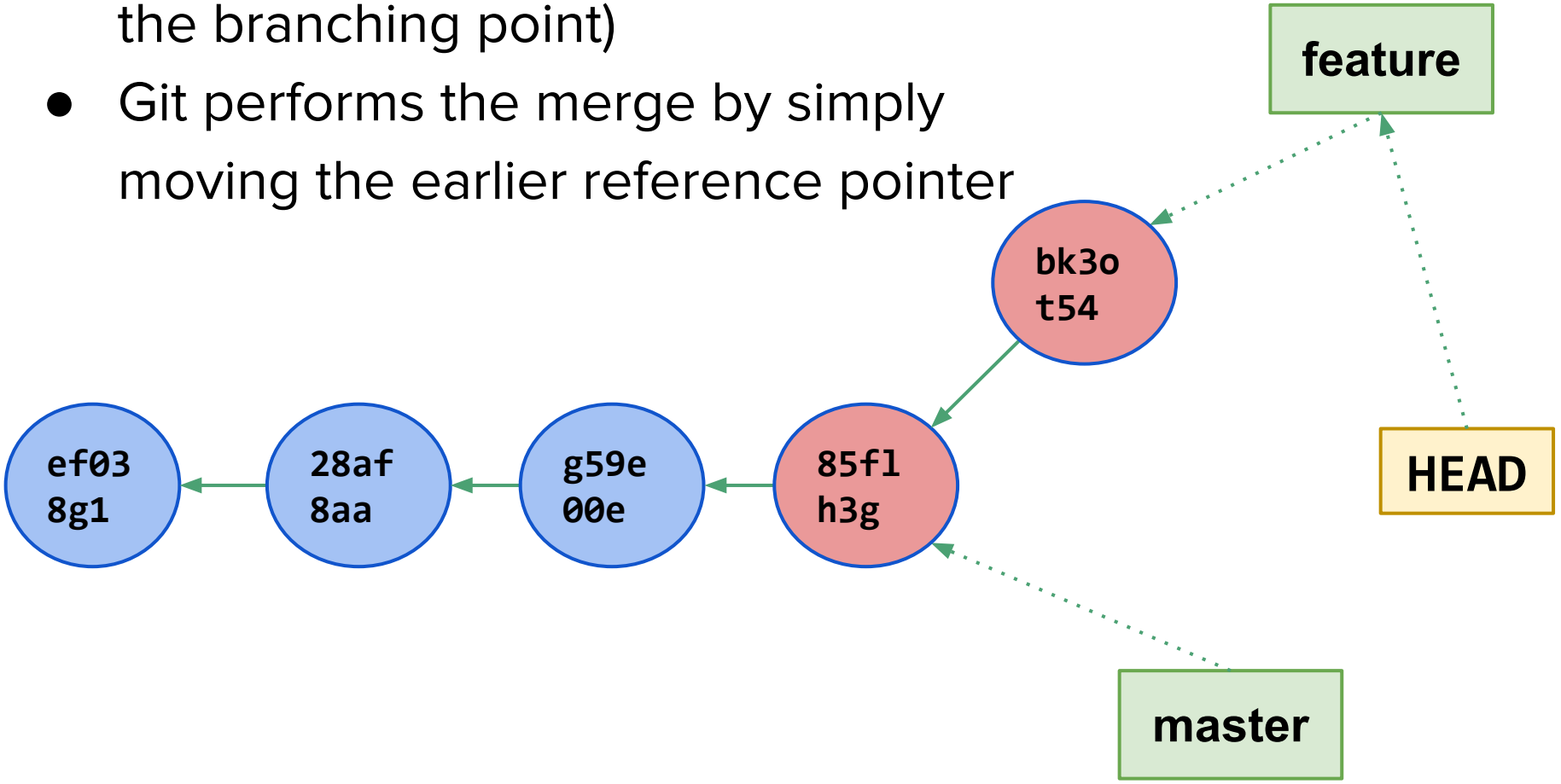
Merging Diverging Branches

3 cases for branch merging

1. **Fast forward** (the easy one)
    - One branch has changes
2. **Recursive**
    - Non-conflicting merge
3. **Fix conflicts manually**
    - Conflicting merge

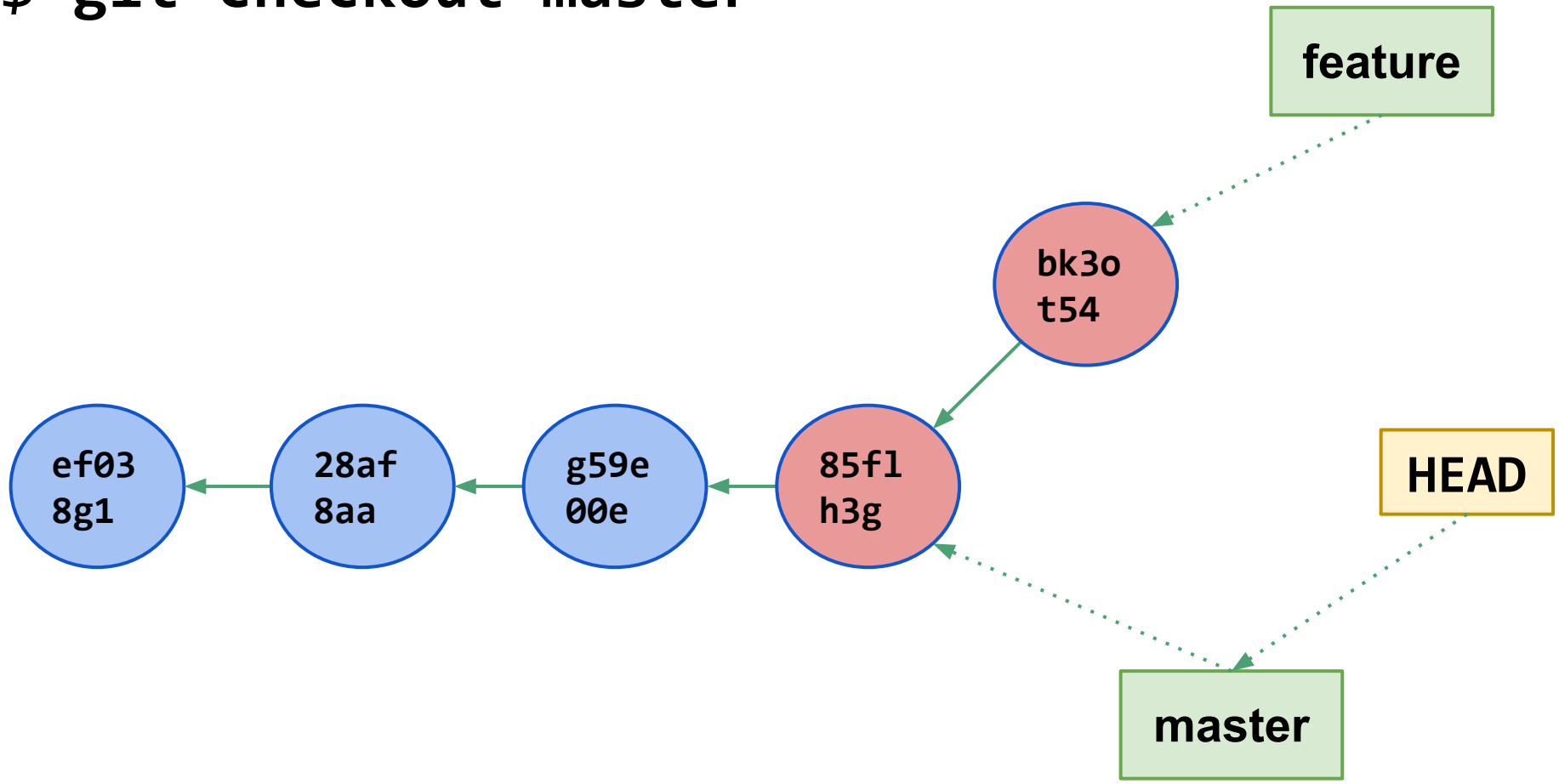
1. Fast Forward

- Changes only happened in one branch (since the branching point)
- Git performs the merge by simply moving the earlier reference pointer



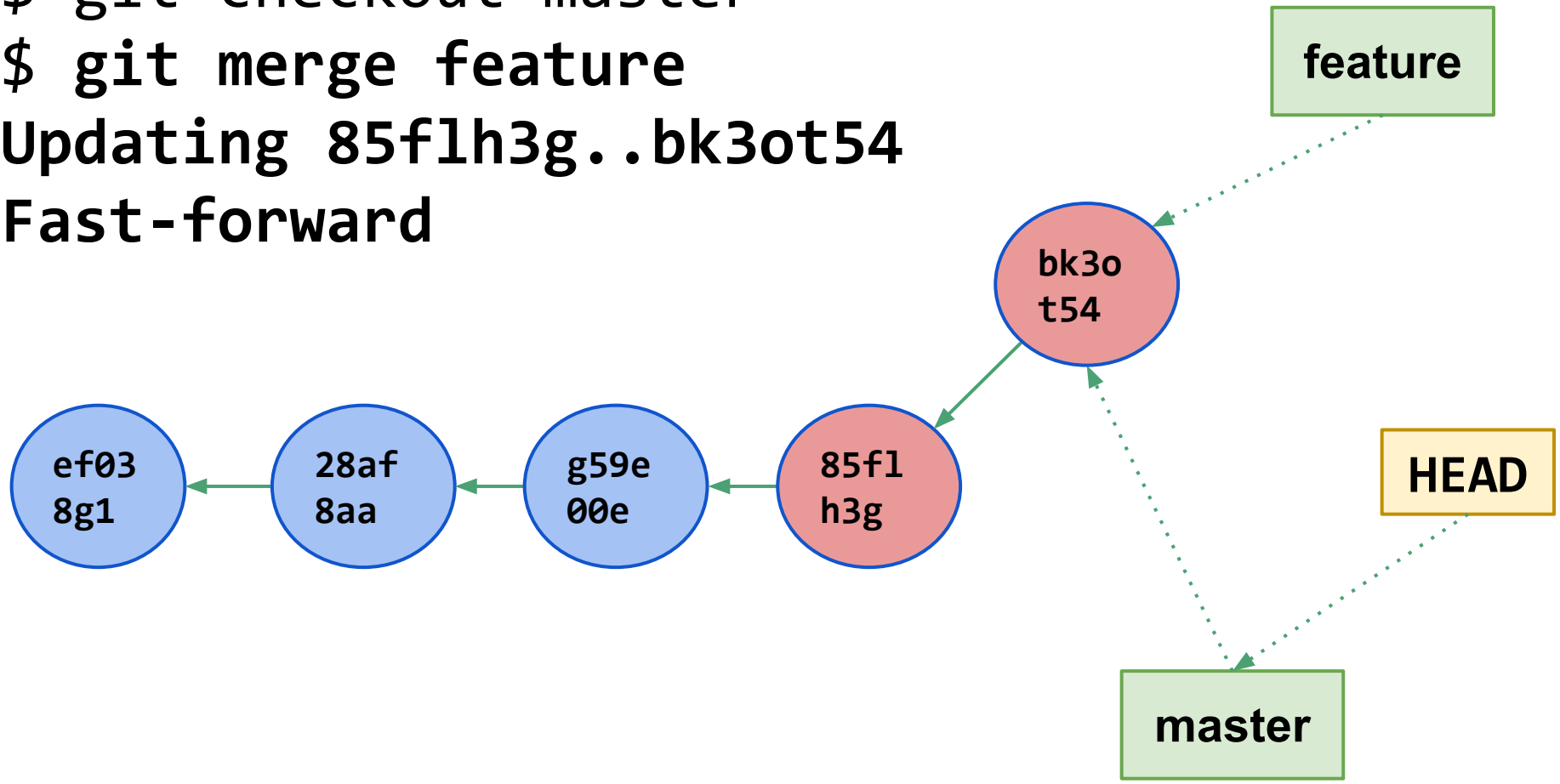
1. Fast Forward

```
$ git checkout master
```



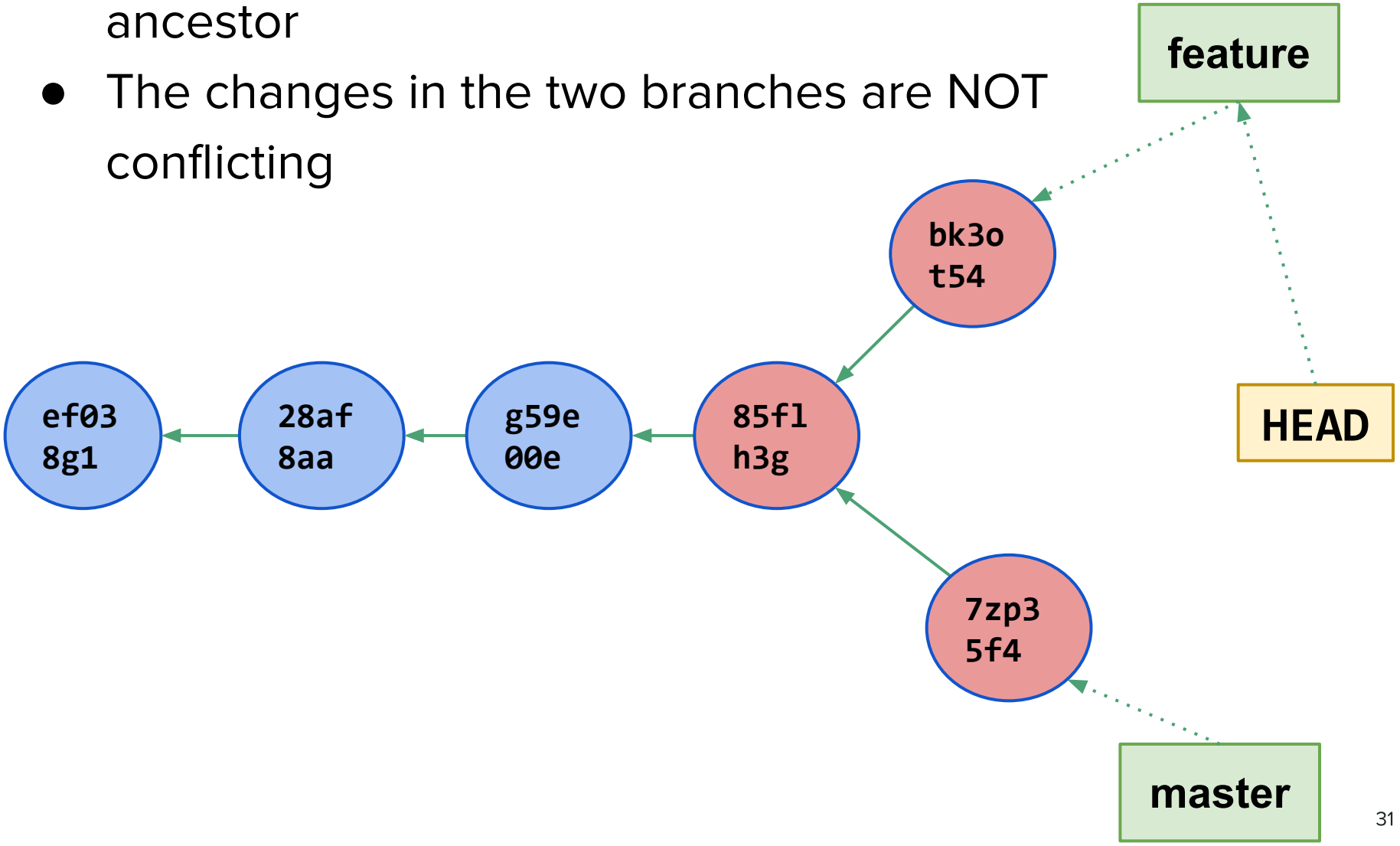
1. Fast Forward

```
$ git checkout master
$ git merge feature
Updating 85flh3g..bk3ot54
Fast-forward
```



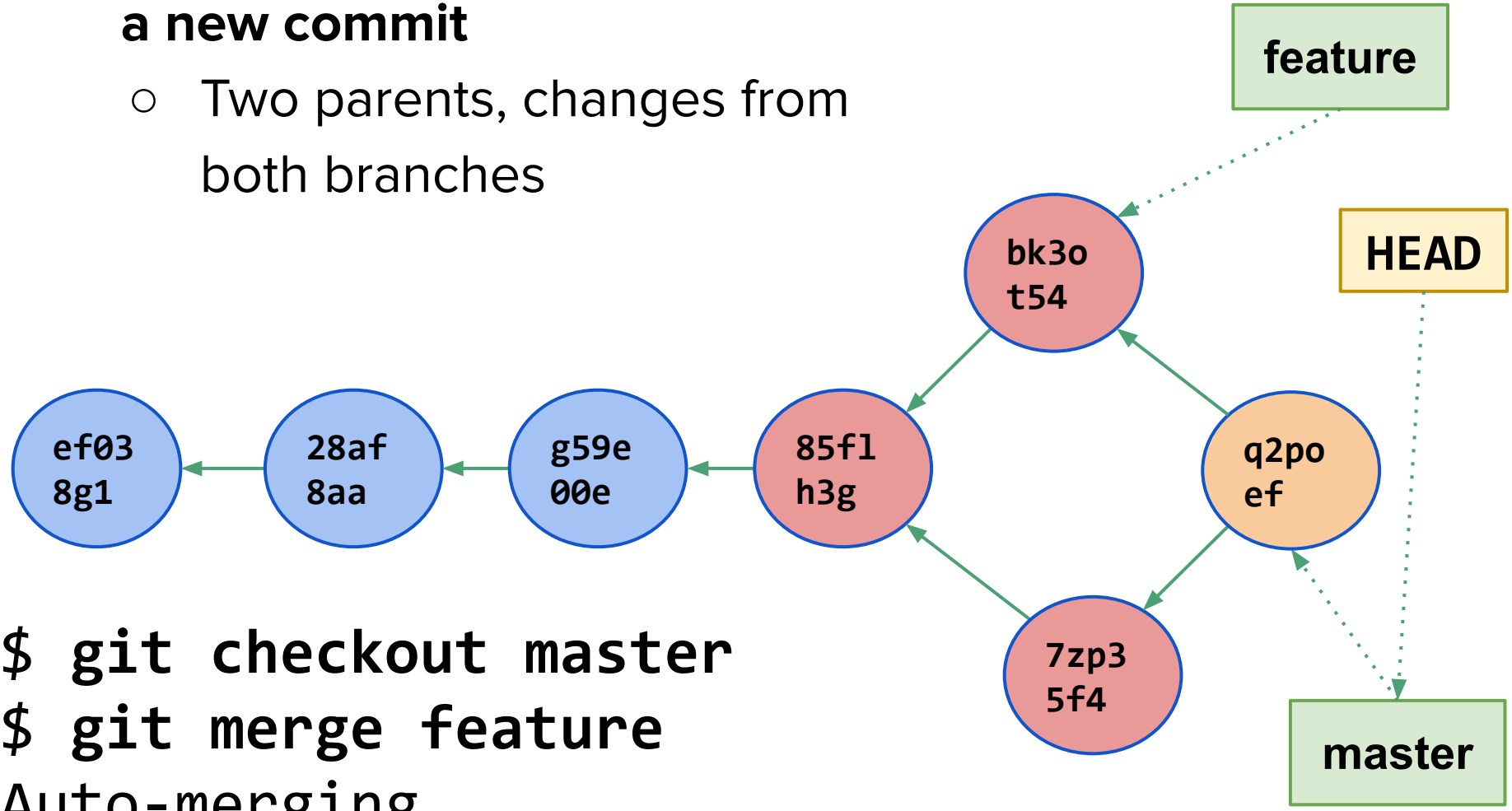
2. Recursive merge

- Branches have diverged from a common ancestor
- The changes in the two branches are NOT conflicting



2. Recursive merge

- Merge performed by **creating a new commit**
  - Two parents, changes from both branches

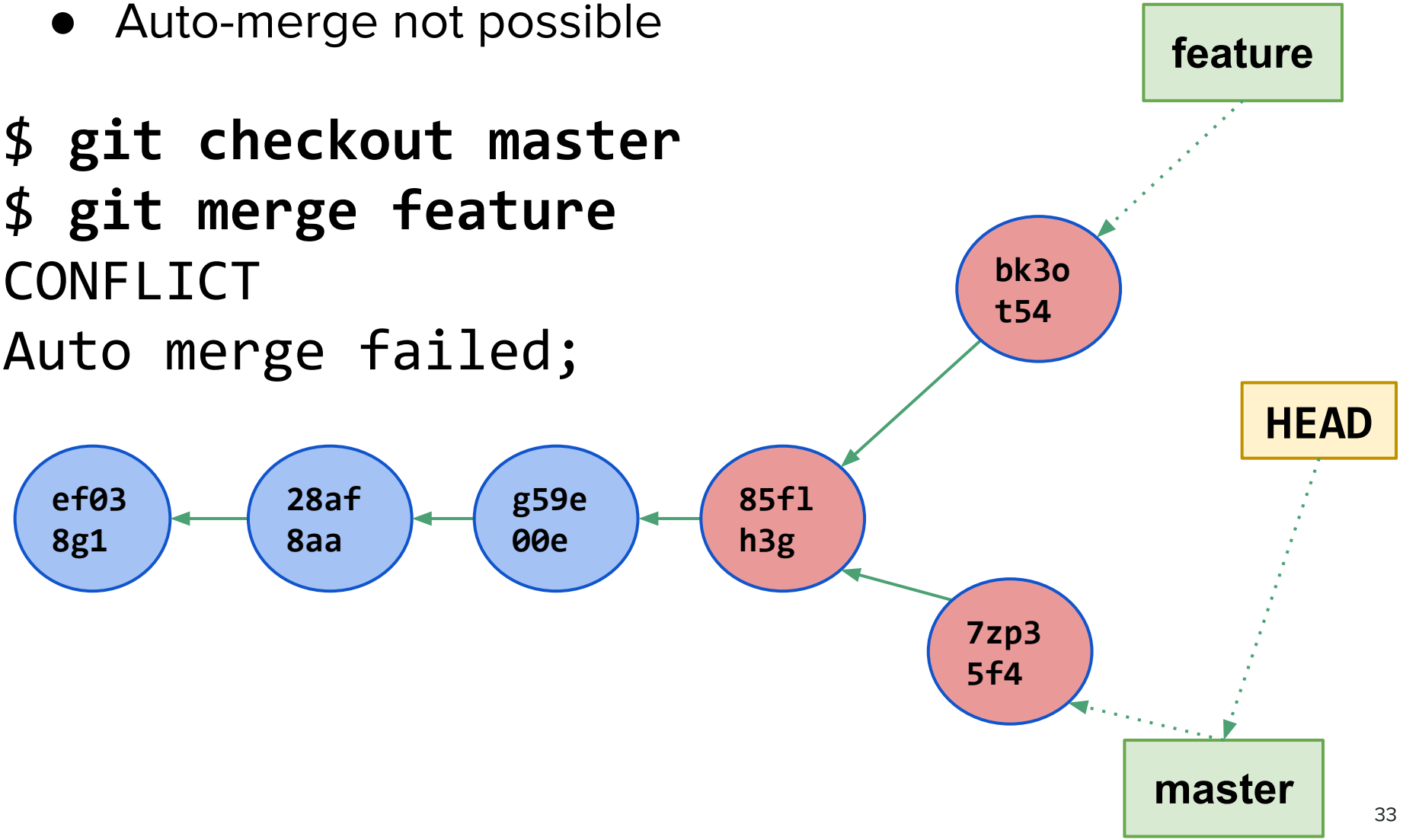



```
$ git checkout master
$ git merge feature
Auto-merging
```

3. Conflicting merge

- Changes in the two branches are **conflicting**
- Auto-merge not possible

```
$ git checkout master
$ git merge feature
CONFLICT
Auto merge failed;
```



3. Conflicting merge

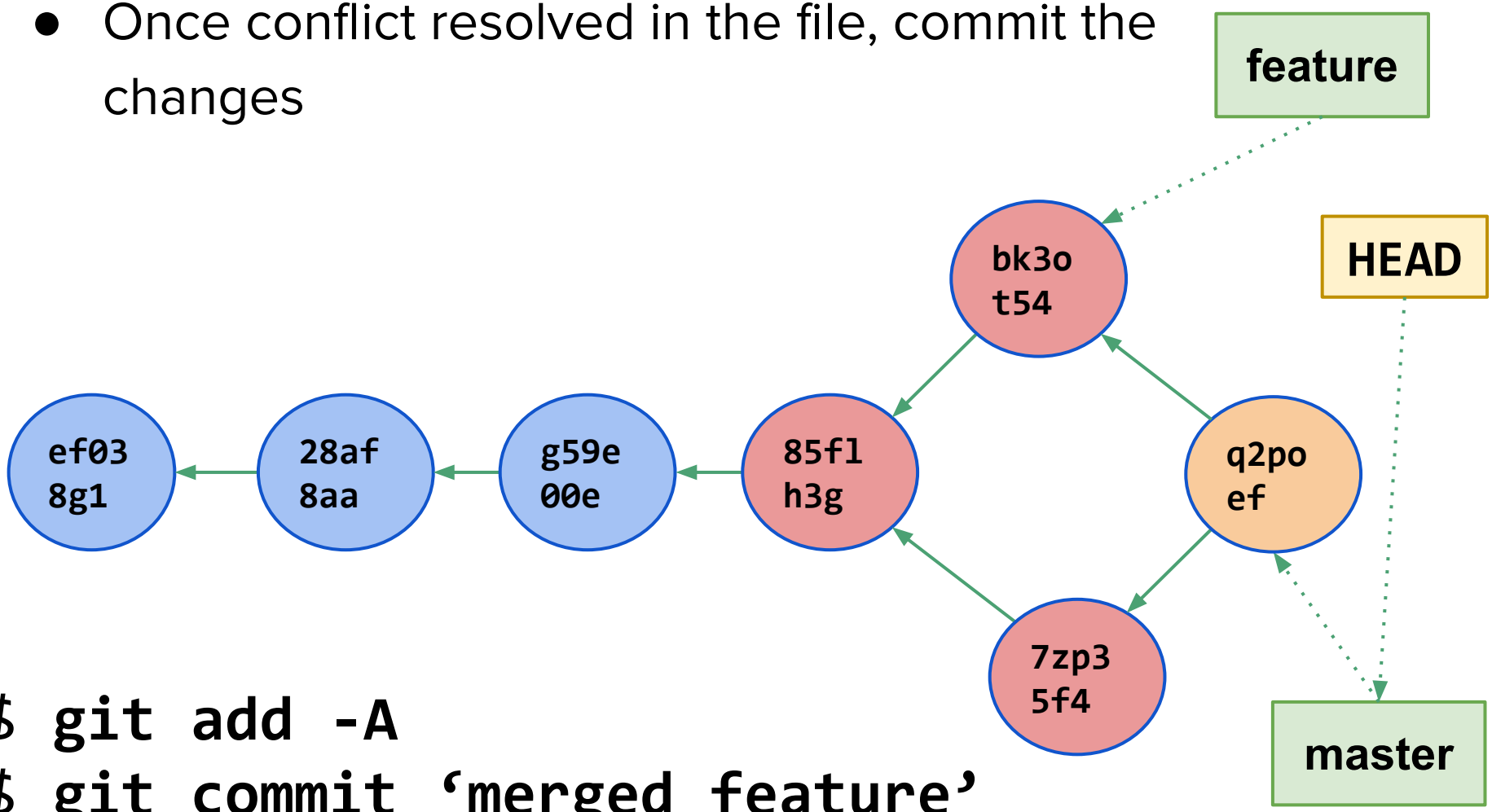
- Fix conflicts by hand - figure out what is ‘correct’
  between the two commits
- Conflicting files will generally look like:

```
<<<<<<< HEAD
<master code>
=======
<feature code>
>>>>>>> feature
```

- Remove the markers and lines

3. Conflicting merge

- Once conflict resolved in the file, commit the changes




```
$ git add -A
$ git commit ‘merged feature’
[master q2poef] merged feature
```

Other important git commands

- **`git stash`**
  - Use when you want to switch branches but not commit work on your current branch
  - Bring back stashed changes with `git stash apply/pop`
  - Use stashing if you're going to do anything that leaves unsaved changes vulnerable
    i.e. `git reset --hard <COMMIT>`
    (points branch to earlier commit)

Other important git commands

- `git log`
  - Provides various views of repo history
  - Nice graph view:

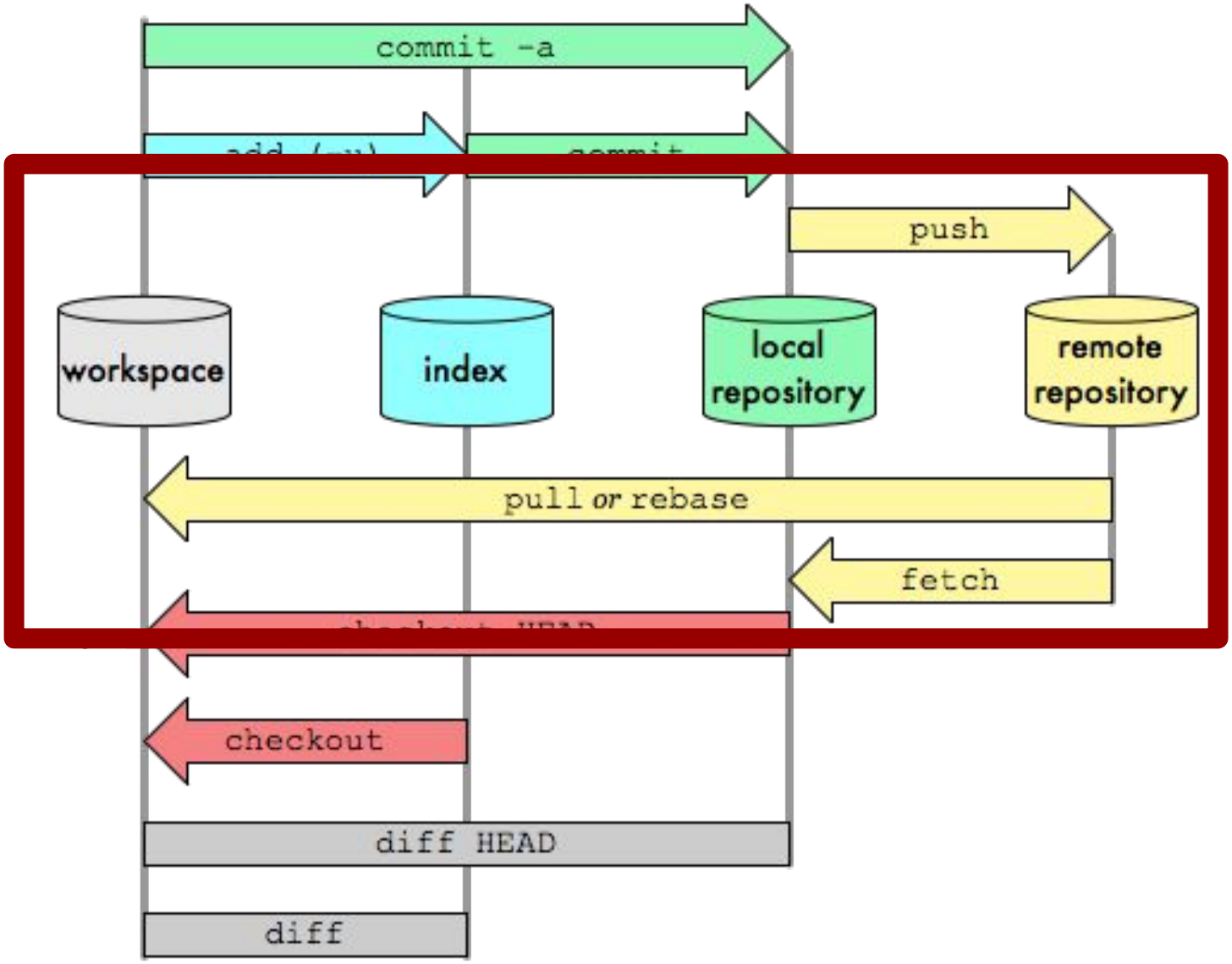
```
$ git log --graph --decorate
--pretty=oneline --abbrev-commit
```

Other important git commands

- `git diff`
  - Show changes
  - Between commits, files, indexes, etc.

Git Data Transport Commands

http://osteele.com



Git: From **Local** to **Remote**

- So far we've seen everything we can do with **local commands**
  - Entirely on your machine

Git: From **Local** to **Remote**

- **Remote** repo
  - A version of your codebase hosted ‘somewhere else’
- Accessible via URLs (HTTPS or SSH)
- You can have **multiple** named remotes
  - Default is called ***origin***
- Associate remote URL with name:
  - **`git remote add origin <REMOTE_URL>`**

Working with Remotes

- Basic workflow
  - ***Clone*** **some remote repo** to your machine
  - ***Commit*** changes locally
  - When ready, ***push*** changes from your machine to the remote repo

Basic **remote** commands

(mostly review, but we should understand what they do)

- **`clone`**
- **`pull`**
- **`push`**

Basic **remote** commands

- **`git clone`**
  - Create a **copy** of a repo
  - Gives name ***origin*** to remote from which you cloned

```
$ git clone https://github.com/my_repo.git
$ git remote -v
origin https://github.com/my_repo.git (fetch)
origin https://github.com/my_repo.git (push)
```

Basic **remote** commands

- **`git pull`** (and `fetch`)
  - Download new data from a remote
  - **`fetch`** gets the data
  - **`pull`** also attempts to merge your local commits with the remote (fetch + merge)
    - You will usually use pull - but beware of conflicts
      - **Make sure your working directory is clean through `git status` first!**

Basic **remote** commands

- **`git push <remote> <branch>`**
  - Push your local commits ‘upstream’
  - Must have write access to the remote

e.g.

```
git push origin master
```

`git push` behaviour set in git config

(**`simple`** pushes the current branch)

Commit ≠ Push

- It is important to understand the distinction between commit and push
  - `git commit` creates a node in the commit graph of your **local** repo
  - `git push` creates node(s) in the commit graphs of some **remote** repo
- Allows for more granularity
  - Make **small frequent commits** while working locally on your code
    - **Commit early, commit often** on local
    - **Don’t be afraid to branch!**
  - When your work is ready (and tested) **push all commits** to a remote repo

Choose commits strategically

- Commit history should tell a coherent story.
- Commits should be properly sized and must have a good commit message
  - Should contain changes to one specific task/ticket/fix
  - Commit when milestones reached. Not at the end/start of day.
    - Ex: significant feature or bug fix
  - Should be easy to review and revert if needed

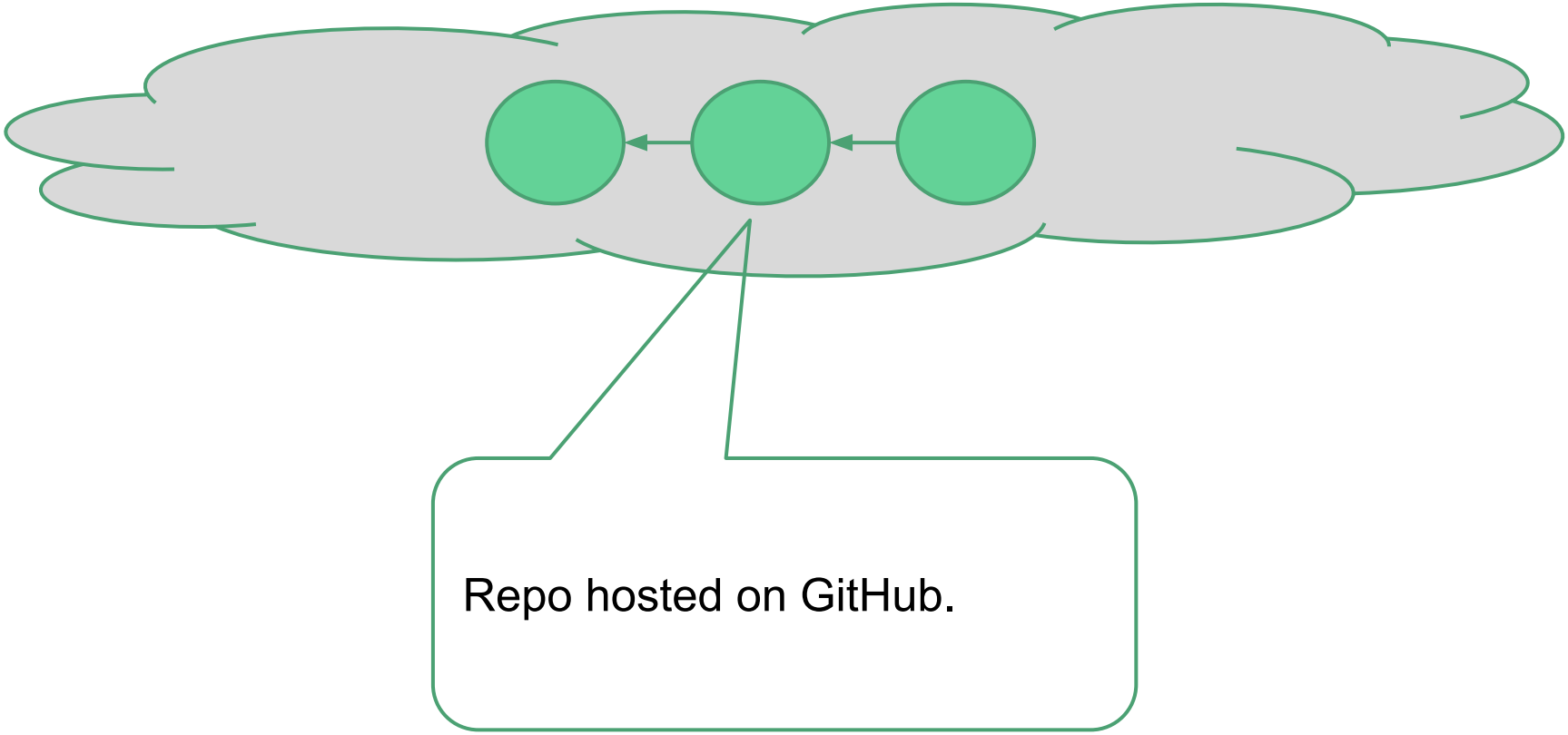
GitHub

- GitHub is a hosting service for Git repos
- Collaborative website and rich toolset on top of Git
- Free for public (and private) projects
- Industry standard for OSS development
- Other options are Bitbucket, gitLab, private servers

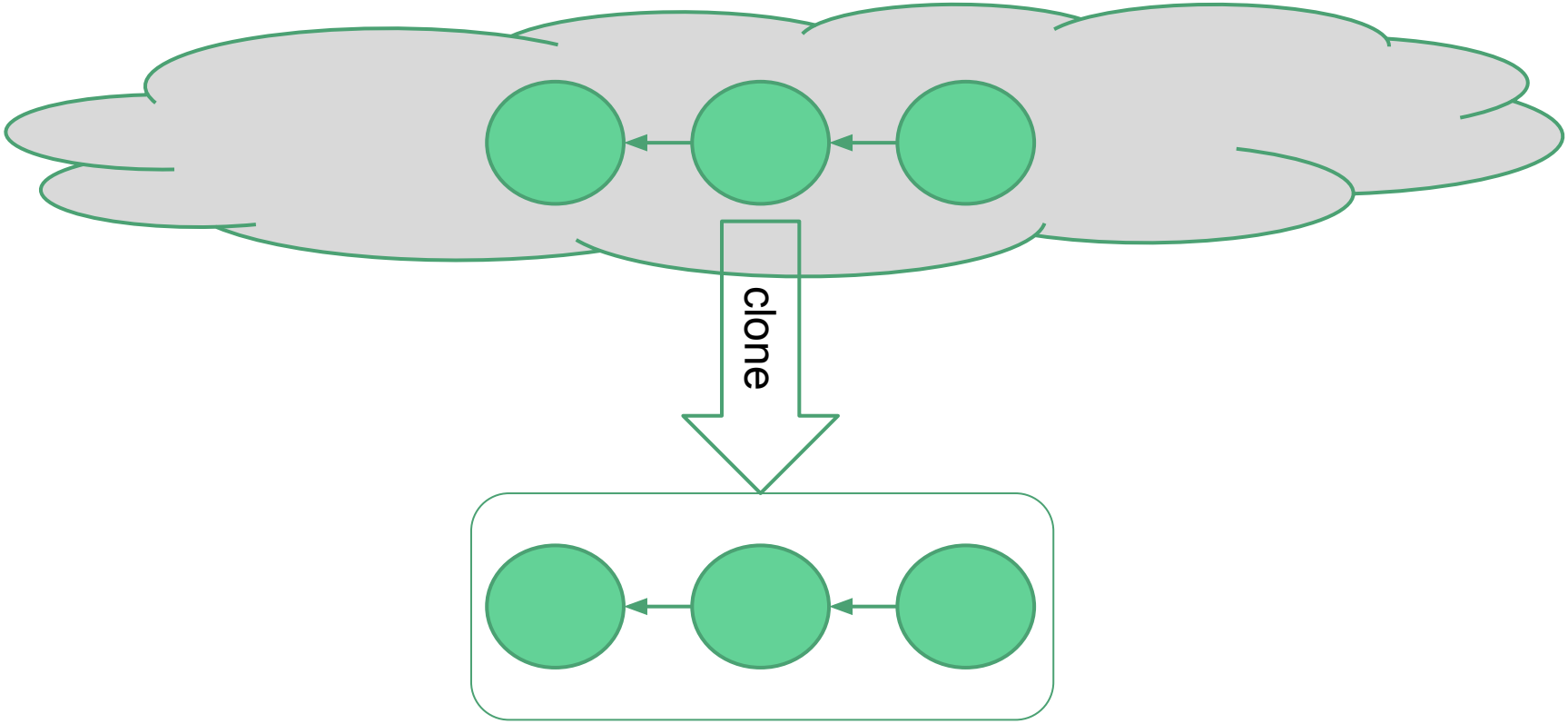
How to do real work?

- How do you contribute work to a repo you have no write permission for?
- Create a **pull request**, and let someone who has write permission merge
  - This is how open-source software works
  - This is how you will submit your individual coding assignments in this course

Common GitHub Workflow

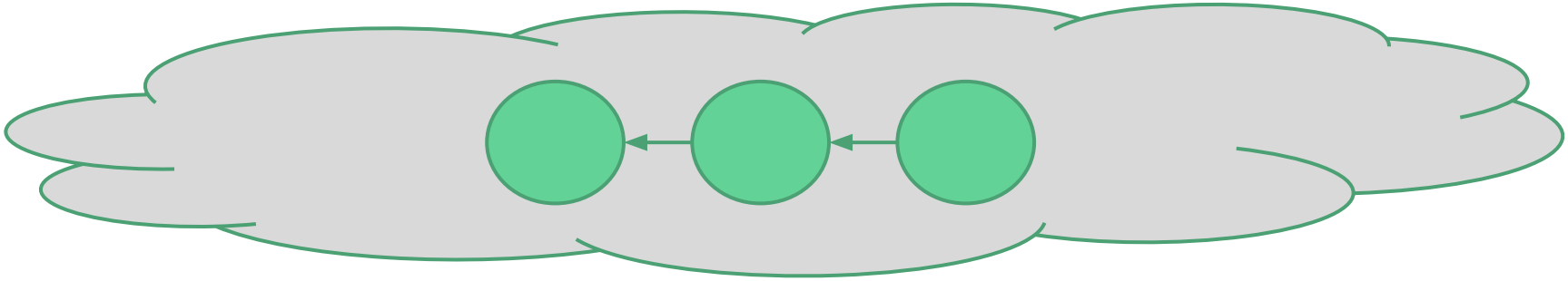


Common GitHub Workflow

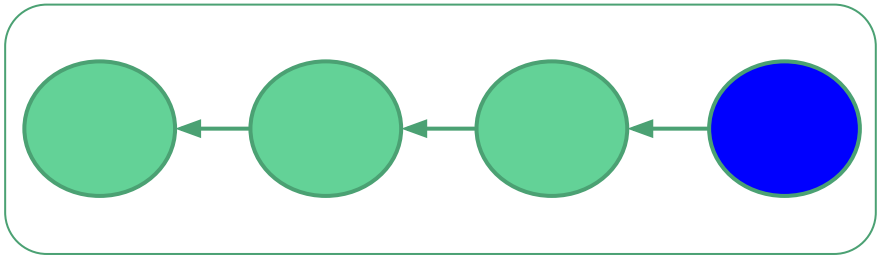


Create a local clone, so you can use your favorite IDE (e.g. IntelliJ), run the code to test your changes, etc.
Note: This is **exactly** like you've seen before

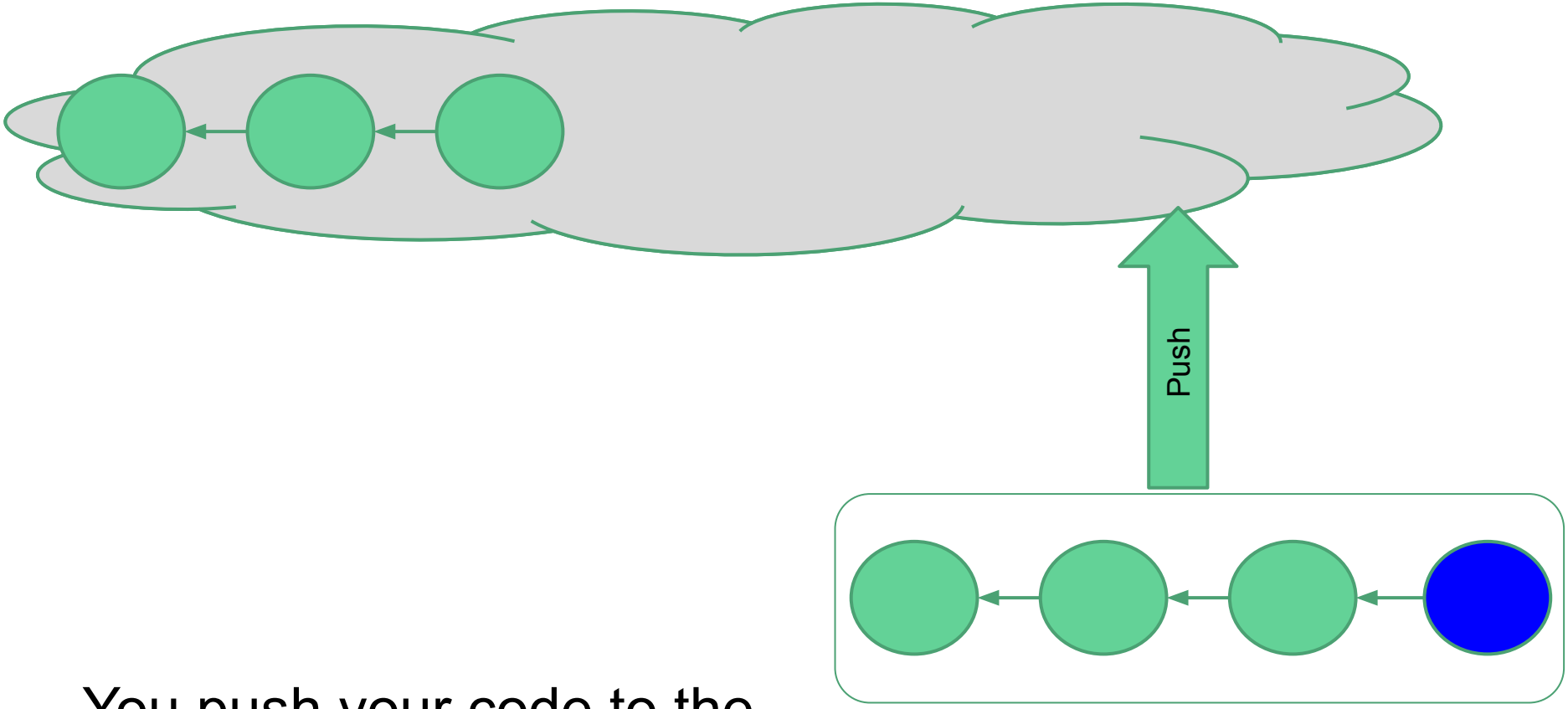
Common GitHub Workflow



Make and commit some changes (locally) ...

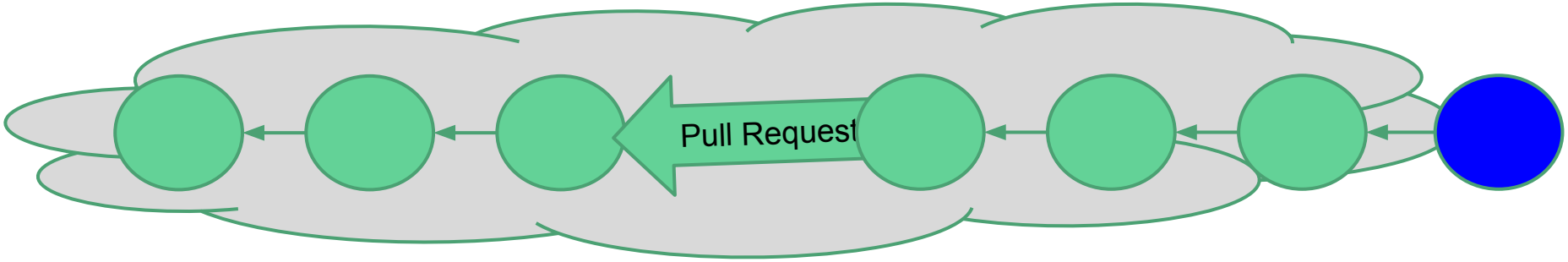


Common GitHub Workflow



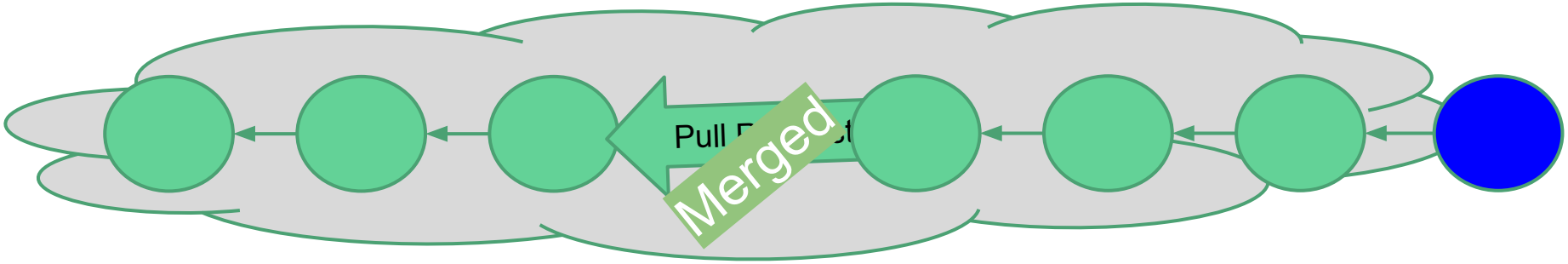
You push your code to the remote repo

Common GitHub Workflow



Create a pull request containing changes from your branch to the main branch

Common GitHub Workflow



Someone on your team *merges your pull-request* into the main repo (***merge privilege***)

GitHub - Pull Request

- Discussion is part of the pull-request
- Automatically warn about conflicts
- Can merge pull-requests directly from GitHub
- GitHub didn’t invent pull-requests
  - Git has built-in support
  - GitHub just simplified the process and added convenient web UI on top of it

Pull Request great for code review

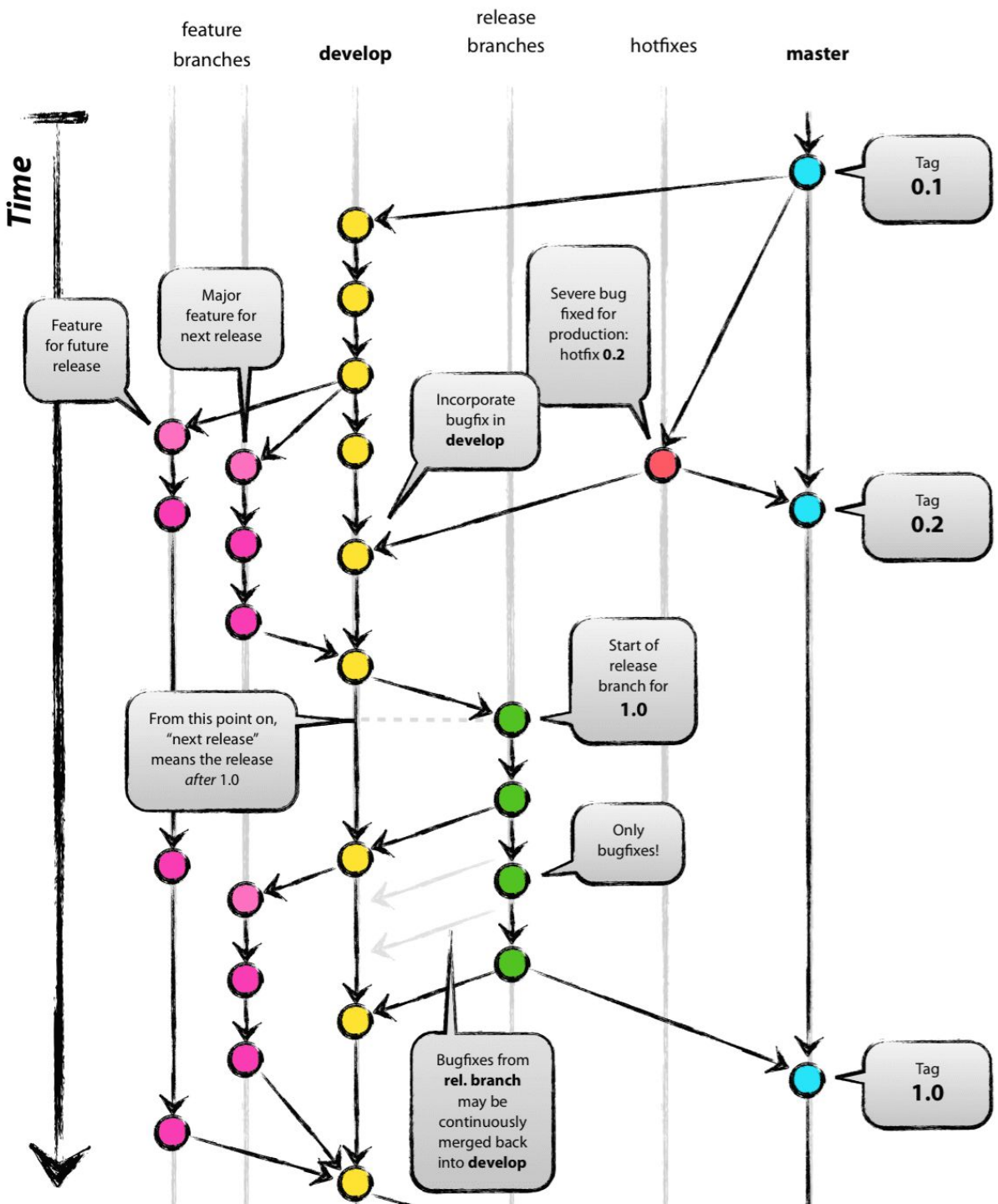
1. Rather than simply merging your feature branch locally, **push it to GitHub.**
2. Submit a pull request from the feature branch to master (or develop) branch,
    - Previously, we saw a pull request across forks.
        - That is, **master branch** of one repo to **master branch** of another repo.
    - Works equally effective **between branches** in the same repo.
3. We will be using Pull Requests so you must push your branches upstream

Common Workflow for Pull Request

- You create a Pull Request (PR)
- Your teammate(s) review the PR, makes comments, etc.
- You fix whatever needs to be fixed, based the review(s)
- Once everybody is happy, someone on the team merges the PR

Gitflow: Common Release Management Workflow

[Learn more here]



GitHub, Fork

- **Forking** = Copying the repo directly **on GitHub**
  - The fork is a **separate GitHub repo**, associated with your GitHub account (i.e., you can read/write to it)
  - Allows multiple teams to work on their own copy of the same code without needing other instances
- Forking vs. cloning
  - **Forking** = Creating a copy of the repo on Github
  - **Cloning** = Creating a copy of the code on your local machine

Resources

Many great resources for learning Git and GitHub

- A Simple Guide
- Training page on GitHub and BitBucket
- An interactive tutorial
- Pro Git - A whole book on Git
- Git for Computer Scientists