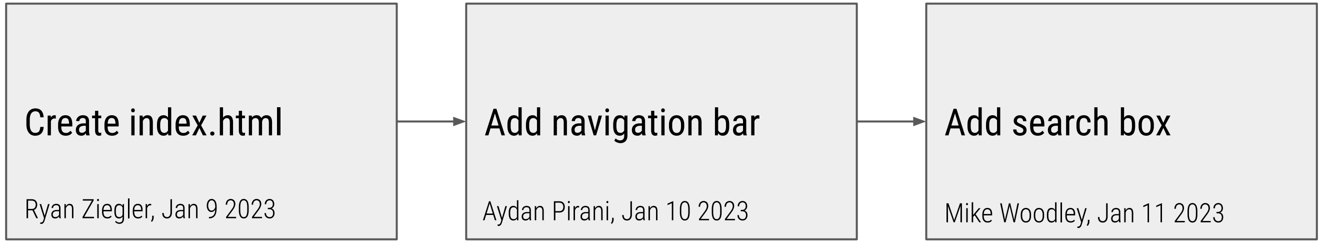
Introduction to Git

Before we begin

- If you’re viewing these slides after lecture, they will be dense, and are intended to serve as a reference
- We build up git from the most basic commands, many of which are important for conceptual understanding but that you don’t need to use everyday
- Everyday-use slides have a light green background
- *Rust added to team matching form!*
- *Team matching due TONIGHT!*

What is version control?

- Keep track of snapshots of your code: complete replicas at a given point in time
  - Extra information such as author, time, etc
- Why? Easy to undo a change, see when a bug was introduced, etc
- *Enables teamwork*: teammates can work in parallel and merge their snapshots together later



Introducing Git

- One of the most common VCS in use today
- Efficient for small to medium size files (i.e. code!)
  - Popular alternative: Perforce, often used in game dev
- Operates by adding a subfolder to your repository (the folder containing your code), and storing any data it needs in that subfolder
- Git != GitHub: Git is a tool that manages a local repository. GitHub is a place to store that repository that provides some management tools

Get Started

- Download and install Git if you haven't already: Installation Tutorial and Configuration
- Create a new, empty folder, and name it whatever you want
- Open up a terminal in the folder you created
- Run **`git init`**
- A folder called **`.git`** has been created, we'll cover what every subfolder does later
  - If you're in VSCode, you might have to go into settings and remove **`.git`** from "Files: Exclude" option

```
config
description
HEAD
hooks/
info/
    exclude
objects/
    info/
    pack/
refs/
    heads/
    tags/
```

Your first Git object

- Objects are snapshots of files or folders
- Create a file, `hello.txt`, in the project folder you created (*not in `.git`*), and add text to it
- Save the file, and run `git hash-object -w hello.txt`
- The output is a long string of characters (a <u>hash</u>) that uniquely identifies this snapshot
  - In my case, the output was `f50e67e27562a879238aeccdb35be90549c1ab6d`
- Observe the `.git/objects` folder: there's a new subfolder whose name is the first two characters of the hash (so, `f5` in my case). It has one file whose name is the rest of the above output (so, `0e67e27562a879238aeccdb35be90549c1ab6d` in my case)

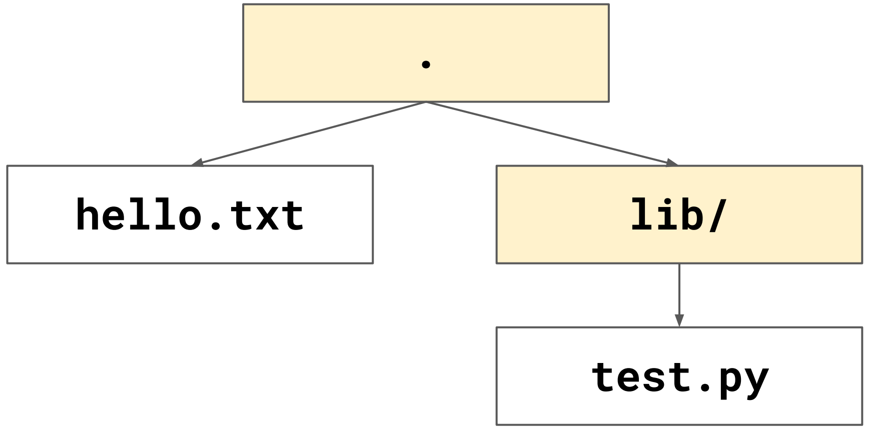
Modifying the file, and accessing versions

- Change the text of **`hello.txt`** to something else
- Run **`git hash-object -w hello.txt`** and observe the output: if you changed hello.txt, the hash will be different!
- Run **`git cat-file [hash from the previous slide] -p`**
  - Don't want to copy and paste? You only need to enter as many characters as are necessary (with minimum 4) for the hash to be uniquely identified (for example, if you have hashes **`abcdef`** and **`abcedf`**, you could run **`git cat-file abcd`** to get the first hash)
  - Observe that the output is what you originally entered into the file!
- Run **`git cat-file [hash from the second step on this slide] -p`**
  - The output is what you updated the file to be!

Supporting subfolders

- What *is* a folder?
  - A tree! Recall we can define a tree recursively: a tree is either a leaf, or a non-leaf node with at least one child tree
  - Nodes correspond to folders, and leaves to files

```
hello.txt
lib/
    test.py
```



Supporting subfolders: the staging area

- Create a subfolder called **`lib`**, and add a file **`test.py`**, with whatever content you want
  - Make sure to create this folder inside your project directory, *not in* ***`.git`***
- Run **`git update-index --add lib/test.py`**
  - This adds **`lib/test.py`** to the *staging area*. The staging area is where you put files when you're ready to take a snapshot. You can put as many files as you want here.
- Run **`git update-index --add hello.txt`**
- Run **`git hash-object -w lib/test.py`**
  - When Git creates a snapshot of a directory structure, it expects file snapshots to already exist
- Run **`git write-tree`**, and note the hash it returned

Exploring the tree snapshot

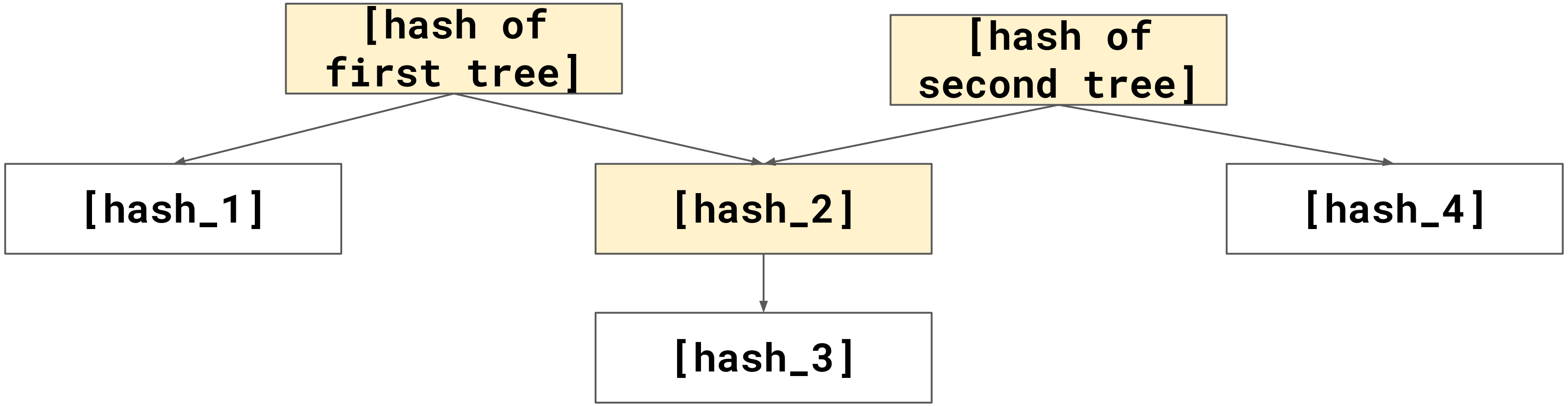
- Run `git cat-file [hash from the last slide] -p`
  - The output will look like:
    ```
    100644 blob [hash_1] hello.txt
    040000 tree [hash_2] lib
    ```
- Run `git cat-file [hash_2] -p`
  - The output will look like:
    ```
    100644 blob [hash_3] test.py
    ```
- We just followed pointers! The hashes either correspond to files (leaves), or to nodes in the directory tree (which, unsurprisingly, Git calls trees)

Modifying the tree

- Update the contents of `hello.txt`, and run git hash-file -w hello.txt to create a new snapshot of the file
- Run `git update-index --add hello.txt` and `git update-index --add lib/test.py`, and write a tree again with `git write-tree`
- Observe that the hash returned by write-tree is different than the one it returned before
- Run `git cat-file [hash from new write-tree] -p`
  - Observe that it follows the same structure as before, and *only* the hash for `hello.txt` changed, not the hash for lib!

Modifying the tree

- `hash_2` corresponds to `lib/` and `hash_3` to `test.py`
- `hash_1` and `hash_4` correspond to the first and second version of `hello.txt`, respectively



Digression: Packing

- We saw that the tree structure saves space when files aren't changed, but what happens when a file is changed only a little bit?
  - Don't want to make a new object, this will duplicate almost everything!
- Solution: packfiles! Every so often (and whenever you send your code over the network) Git will look at objects and compress them into a Packfile
- The Packfile compression includes *deduplication*, meaning that objects that share content will become smaller!
- Packing is transparent to the user, so you don't have to worry about doing it yourself!
- Packfiles are stored in `.git/objects/pack/` and Git stores any extra metadata in `.git/objects/info/`

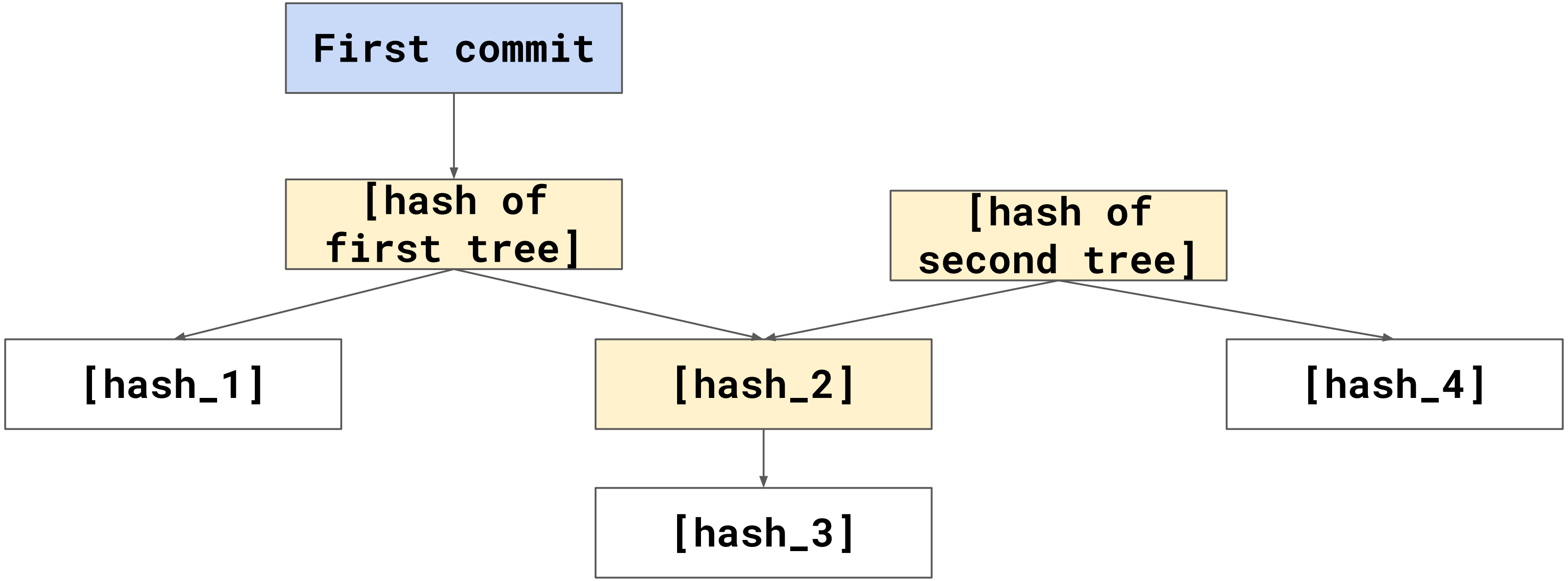
Making hashes more useful: introducing commits

- We have a bucket of unordered, undated, undescribed hashes. This makes change tracking and collaboration difficult. *We don't know where we are in history!*
- What are some familiar data structures for keeping track of linear items?
  - Arrays, **linked lists**, etc
- Git <u>commits</u> annotate tree hashes with metadata *and form a linked list*
- Create a commit for the first tree snapshot by running **`echo 'first commit' | git commit-tree [first tree hash]`**
- We'll get back the hash of another git object, and if we run **`git cat-file [hash from above] -p`** we'll see see something like this:

```
tree 9d7e203bef724a0899d559b98b7b71a637f98db2
author Ryan Ziegler <rzig408@gmail.com> 1673200367 -0600
committer Ryan Ziegler <rzig408@gmail.com> 1673200367 -0600

first commit
```

Making hashes more useful: introducing commits



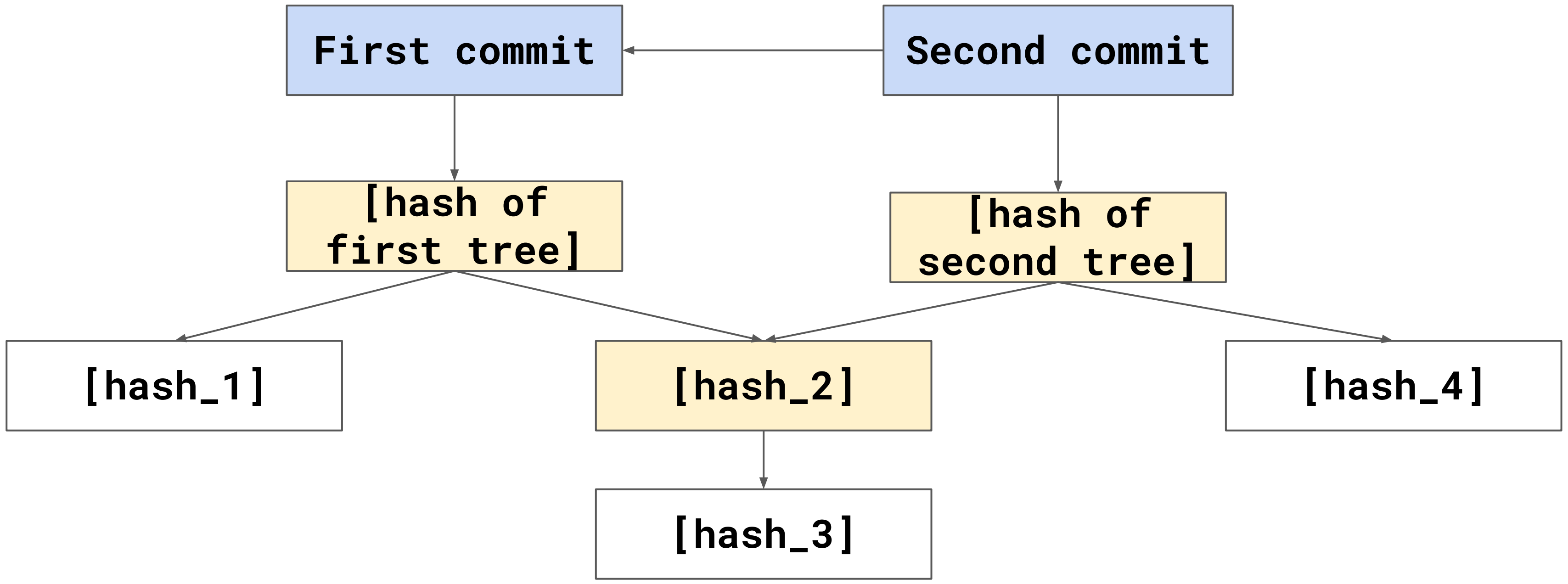
Our second commit

- Let’s make a commit for the second tree we saved. Run **`echo ‘second commit’ | git commit-tree [hash of second tree] -p [hash of previous commit]`**
- We just made a linked list! The second commit points to the previous commit, which has no previous commit. In other words, the second commit is the **head** of our linked list.
- We can confirm this by running **`git cat-file [hash of second commit] -p`**

```
tree ec07c06d2983bd9e96da039df30a700c8f3b9d56
parent 1e6464f844b12e6e3db91a920c0088a4357d6ba3
author Ryan Ziegler <rzig408@gmail.com> 1673200654 -0600
committer Ryan Ziegler <rzig408@gmail.com> 1673200654 -0600

second commit
```

Our second commit

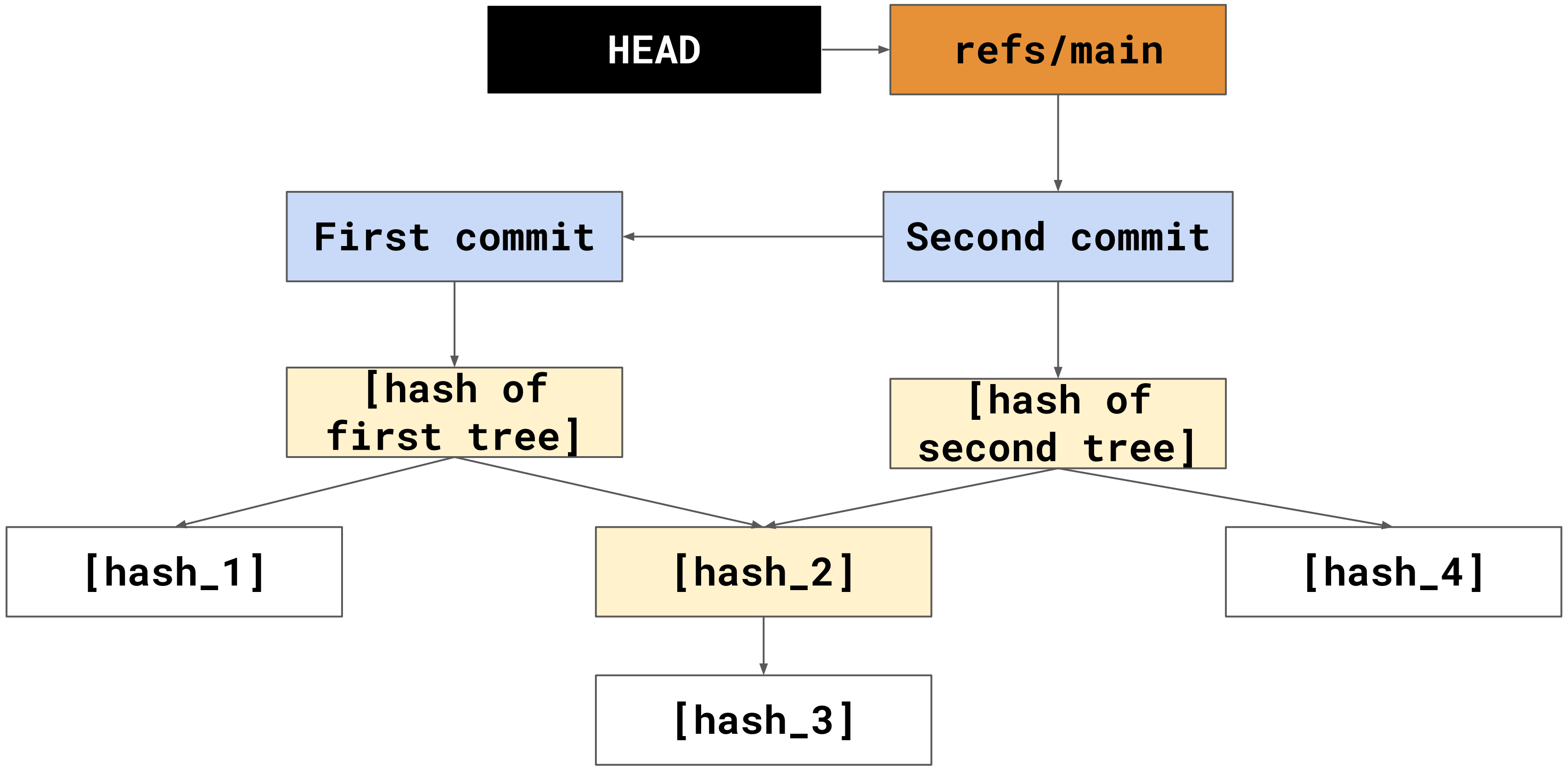


Setting the HEAD

- We need to tell Git where the head of our linked list is!
- Run **`echo [full hash of second commit] > .git/refs/heads/main`**
- This creates a file telling Git that the main reference of our repository is pointing to the second commit (we'll introduce branching and other references later)
- Run **`git log main`** to see the commit history for the main reference
- If we run git log right now, we'll get an error–Git doesn't know what reference it should be using. Tell it to use the main one by replacing the contents of `.git/HEAD` with the following:

  ```
  ref: refs/heads/main
  ```
- Now run **`git log`** to see the commit history we just created!
- *There is only one HEAD!*

Setting the HEAD



Making things easier

- Manually hashing files and committing trees and updating pointers is a lot of work! Fortunately Git gives us commands that do the legwork for us:
  - `git add [space separated paths to files or directories]`
    - Adds the specified files or directories to the staging areas
  - `git commit -m "[message]"`
    - Hash all the files, and the tree, and create a commit object with the message provided, then update the pointer pointed to by `HEAD` to point to the new commit. After all this, unstage any staged files.
- Modify `lib/test.py` and then run `git add lib/test.py` and `git commit -m “update test”`
- Run `git log` again, and you’ll see your new commit

Diffs and some history

- Commits aren't useful unless we can see what changed between them
    - For example we want to see what specific changes introduced a bug
- Run **`git diff [old commit hash or ref] [new commit hash or ref]`**
- Git started as a VCS built for the Linux kernel development workflow
    - Developers share patches (see example below, which is output from git diff of first two commits) via email, and can apply them to their local repositories

```
diff --git a/hello.txt b/hello.txt
index 6769dd6..9553a1e 100644
--- a/hello.txt
+++ b/hello.txt
@@ -1 +1 @@
-Hello world!
\ No newline at end of file
+Howdy world!
\ No newline at end of file
```

Best practices for commits

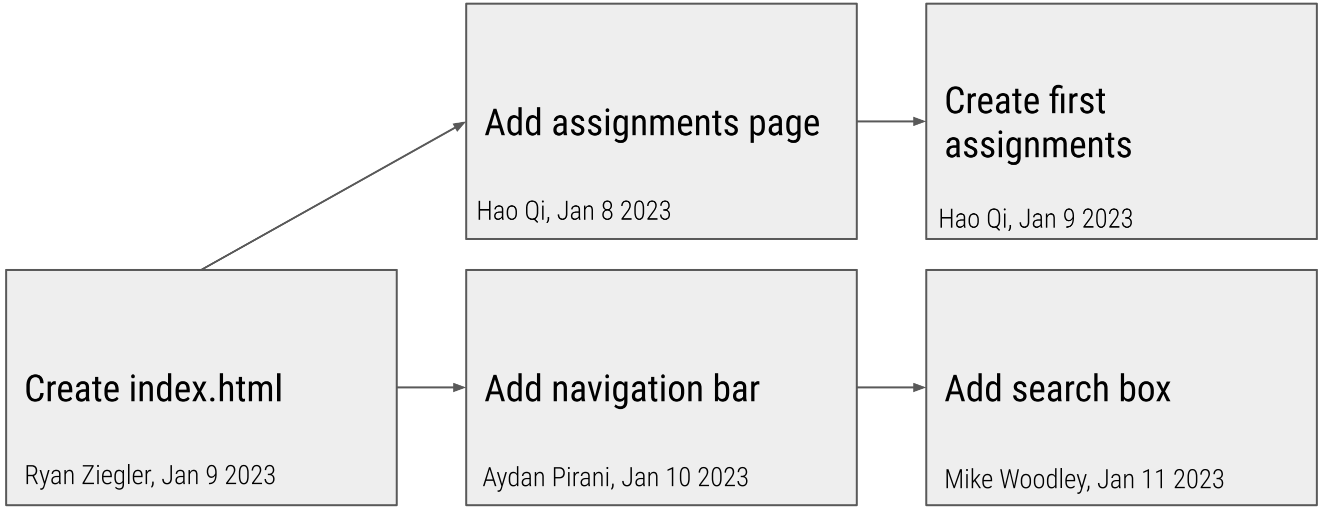
- Commits were originally *emails*, and this has inspired many of the best practices in use today
- Commit messages were (and still are, for Linux Kernel developers) email subjects: they should be short, descriptive summaries of the change
  - Good: “Update driver class to support silent command line option”
  - Bad: “Update driver”
- Commits should also be *small*: ideally <250 lines unless you have good reasons not to (such as a project-wide find and replace)
  - If you can’t summarize the commit into a tweet, it should probably be broken up into smaller commits

Undoing commits

- Undoing a commit is as simple as moving pointers around!
- `*HEAD := pointer that HEAD points to`
- `git reset --hard [commit hash]`
  - Make `*HEAD` point to the provided commit, and update all files to match the tree that commit points to
- `git reset --soft [commit hash]`
  - Make `*HEAD` point to the provided commit, but keep files as they are
- `git reset --hard HEAD~`*n*
  - Make `*HEAD` point to the commit *n* behind it, and update all files to match the tree that commit points to (`HEAD~1` will undo the most recent commit)
- `git reset --soft HEAD~`*n*
  - Make `*HEAD` point to the commit *n* behind it, but keep files as they are

Collaboration and organization

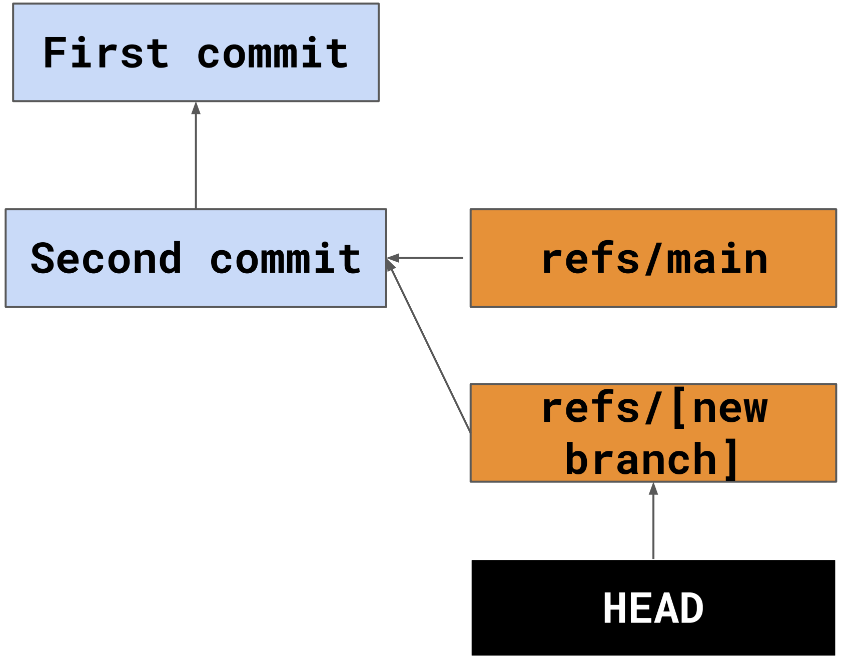
- Linked list model is *linear* but we don't necessarily want this: generally any commit that the main ref points to should be something you'd be comfortable giving a user
    - We want to *branch* off of the main reference and make a sequence of commits, and then add them back in to main
    - Advantage of this strategy: multiple people can work simultaneously on different branches without stepping on each others toes



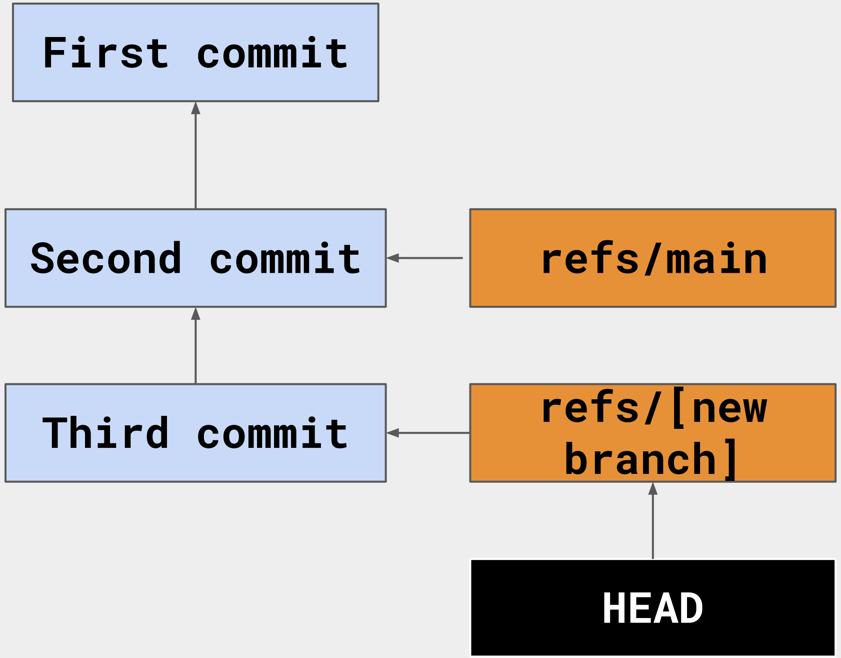
Making a new branch

- Run `git checkout -b [name of branch]`
  - In CS222, you should prefix branch names with your name, for example `ryan/[name of branch]`
- Now we see a new file has been created in `.git/refs/heads/ryan/[name of branch]`: if we open the file, it'll point to the same commit that main points to
- We can see that the HEAD has been updated to point to `refs/heads/ryan/[name of branch]`, meaning any commits we make will change what `ryan/[name of branch]` points to, *not* what main points to
- Modify `lib/test.py`, add it to staging, and create a commit
- Run git log: git will tell you that `ryan/[name of branch]` points to the new third commit, but main still points to the second commit

Making a new branch



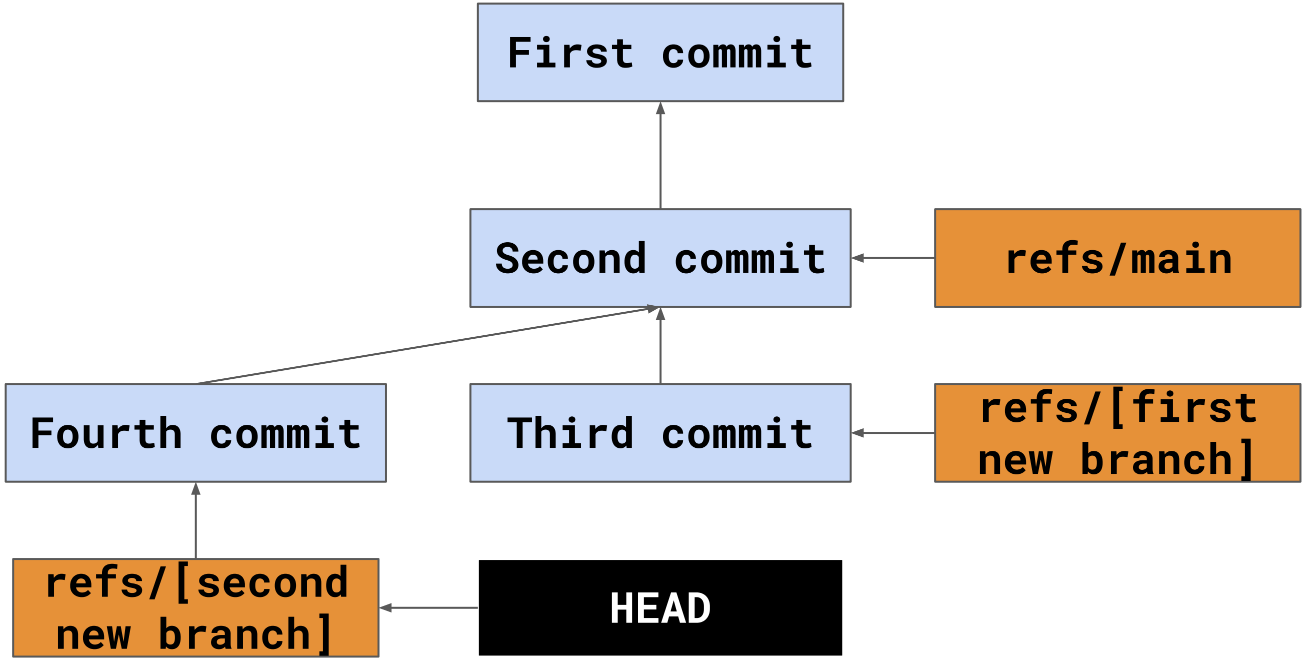
After making a commit on branch



Switching branches

- Run **`git checkout [name of branch]`**. Git will:
    - Update **`HEAD`** to point to **`refs/[name of branch]`**
    - Modify your files to match the tree that **`refs/[name of branch]`** points to
        - If you've modified files but didn't commit the changes, Git will attempt to keep those modifications when you checkout the other branch, and will tell you if it can't. *We recommend that you always commit before switching branches unless you are very experienced!*
- Let's switch back to main by running **`git checkout main`**
- If we run **`git log`**, we'll see that the third commit isn't shown
- Make another new branch, and on it, modify **`hello.txt`** and make a commit with those changes

Switching branches



Merging branches

- We now have two branches with changes–this is similar to what would happen if two people were working on different parts of the project at once
- Switch to the main branch (**`git checkout main`**)
- Merge the first branch into main by running **`git merge [name of first branch]`**, and you'll see the following output:

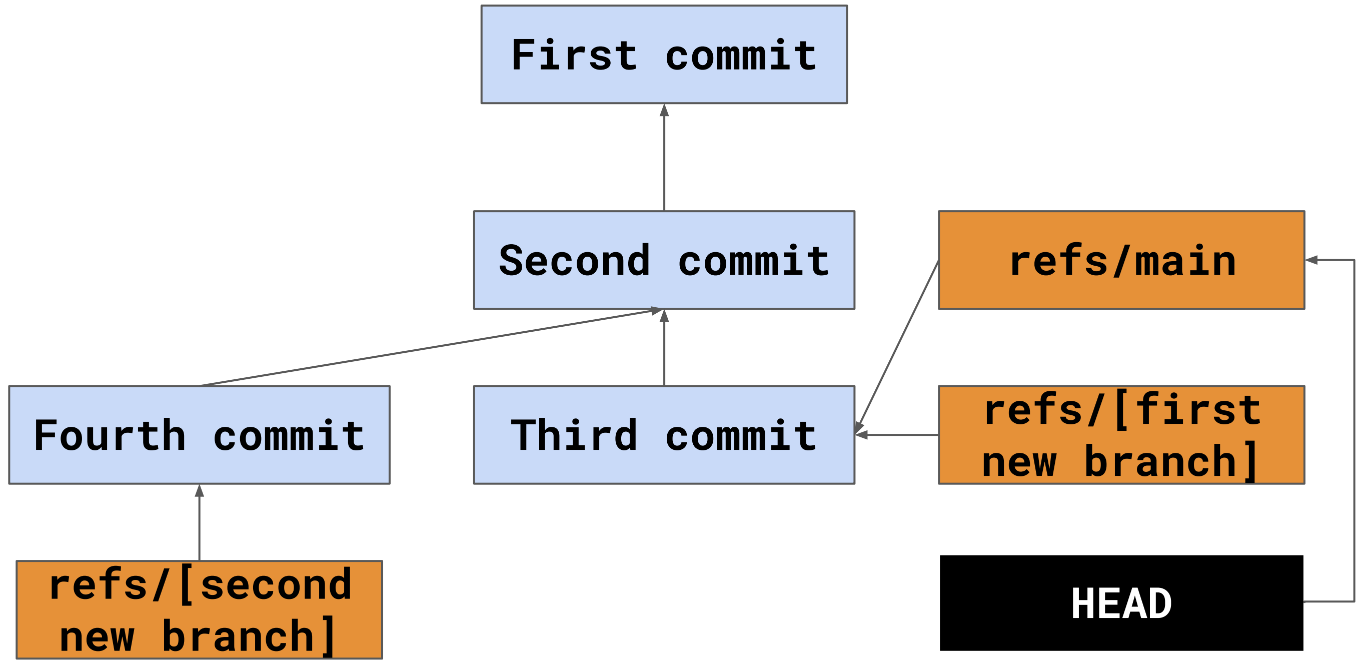
```
Updating 4f4e5e9..8d6e41d
Fast-forward
 lib/test.py | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

- Run **`git log`**, and we'll see that the main pointer points to the same thing as the **`[name of first branch]`** pointer

Digression: how merging works

- TLDR: merging is hard, for full detail read the docs: https://git-scm.com/docs/merge-strategies
  - First check for files being moved/renamed
  - Next check in-file changes
    - If same file changed by both branches, attempt to apply both changes
    - If not possible, let user resolve the conflicts
      - This is why we suggest you don't work on the same file on multiple branches at the same time
  - If there were conflicts, the merge changes will be saved as a singular commit, otherwise it is treated as copying all the commits from source to target

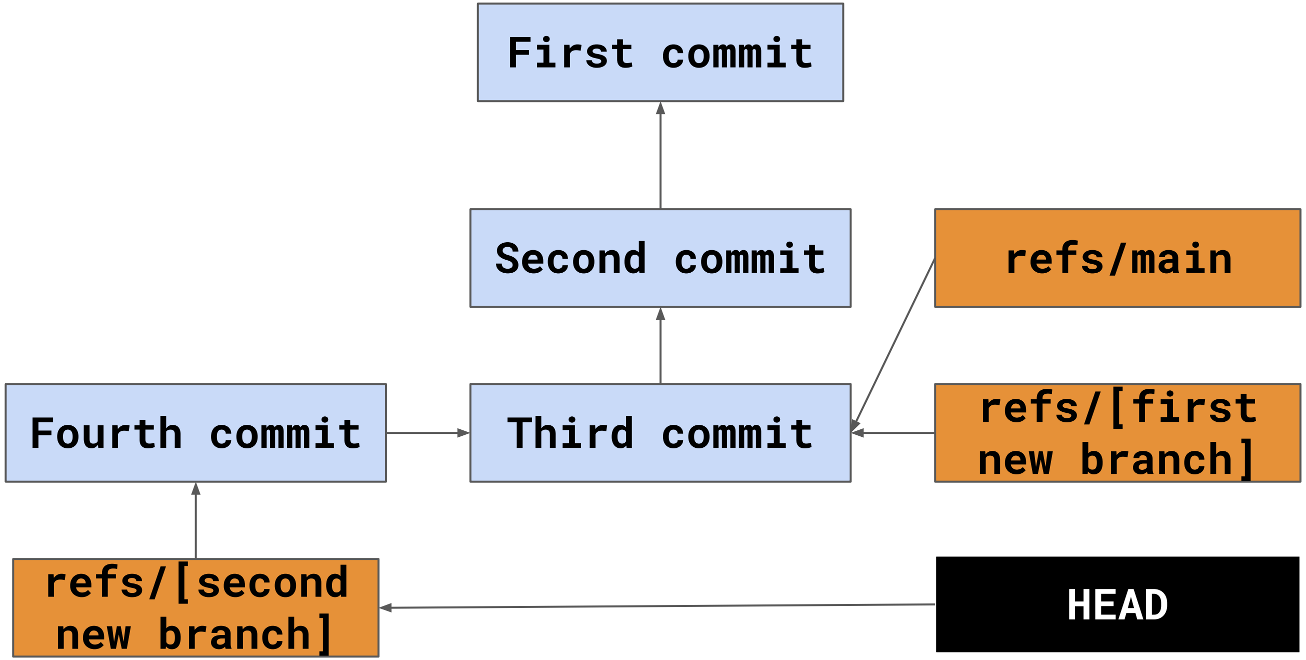
Merging branches



Rebasing branches

- The second branch we created is now “behind” main (1 commit ahead, 1 commit behind)
- Switch to this branch (**`git checkout [name of second branch]`**) then run **`git rebase main`**. Git will:
  - Move the ref for this branch back to the commit where the branch began
  - Apply all commits from main
  - Rewrite all commits on the current branch to be as if they branched off of the most recent commit from main

Rebasing branches



Finishing up, team workflows

- Switch back to main (**`git checkout main`**) and merge the second branch (**`git merge [name of second branch`**])
- Once you're done with a branch, you can delete it with **`git branch -D [name of branch]`**--this amounts to simply deleting the ref file!
- Generally you aren't working on the same computer as the rest of your team. This is where GitHub comes in: it is a platform that can store the .git folder, and facilitates merging through a feature called Pull Requests, with support for code reviews by teammates.

One more thing: `.gitignore`

- **There are some files you should not track with git: *never ever ever* commit files containing passwords, API keys, etc**
  - Sensitive data should be stored in `.env` files: check language/library docs for details on how to access data from `.env` files
- Best practice to avoid tracking dependencies (such as `node_modules` folder): keeps repository size down
- `.gitignore` file specifies files and folders to ignore

Git tips and tricks

- When working in a team, you should create a new branch for each new bugfix, feature, etc. Branches should be self-contained.
- You should avoid modifying the same file on two different branches at the same time, doing this will cause merge difficulties!
- Commit messages should be short and descriptive
- More commits is better than fewer: save your progress often!
- Experience is your friend: if you're new to Git, it might be a lot at first, but as you use it you'll understand more
    - Feel free to ask any Git questions in office hours, we're here to help!

The folders we didn’t cover

- There’s a few files and subfolders in `.git` that this presentation doesn’t cover because CS222 won’t use them
  - `hooks/`: contains files that run when certain git actions occur, for example running a script to make sure tests pass before you make a commit
  - `info/`: contains the exclude file, which has the same function as `.gitignore`. `.gitignore` is most commonly used, so don’t use this
  - `description`: file containing a description of the repository. You don’t need to fill this out
  - `config`: file containing git configuration options. You probably don’t need to edit this

Summary - part 1

- Set up git for your project by running git init in project folder
- Add files to staging area with `git add [space separated list of files and/or folders]`
- Create a commit with `git commit -m "[message]"`
    - Messages should be short and descriptive (no longer than a tweet), commits should modify < ~250 lines as a rule of thumb (but there are exceptions, use your best judgement)

Summary - part 2

- Check out a new branch with `git checkout -b [branch name]`
  - All branches in 222 should start with your name
- Check out an existing branch with `git checkout [branch name]`
- See what branch you’re currently on with `git branch`
- Compare two commits/ref with `git diff [first thing] [second thing]`
- Merge branch `b` into branch `a` by running `git merge b` on branch `a`
- Rebase branch `b` onto branch `a` by running `git rebase b` on branch `a`
- Delete branch [name] by running `git branch -D [name]`

Useful resources

- Git SCM book: https://git-scm.com/book (main reference used to make this guide)
- Git cheatsheet: https://gist.github.com/rzig/a37930960417055cea942dc8ddc18469
- Oh shit git: https://ohshitgit.com/