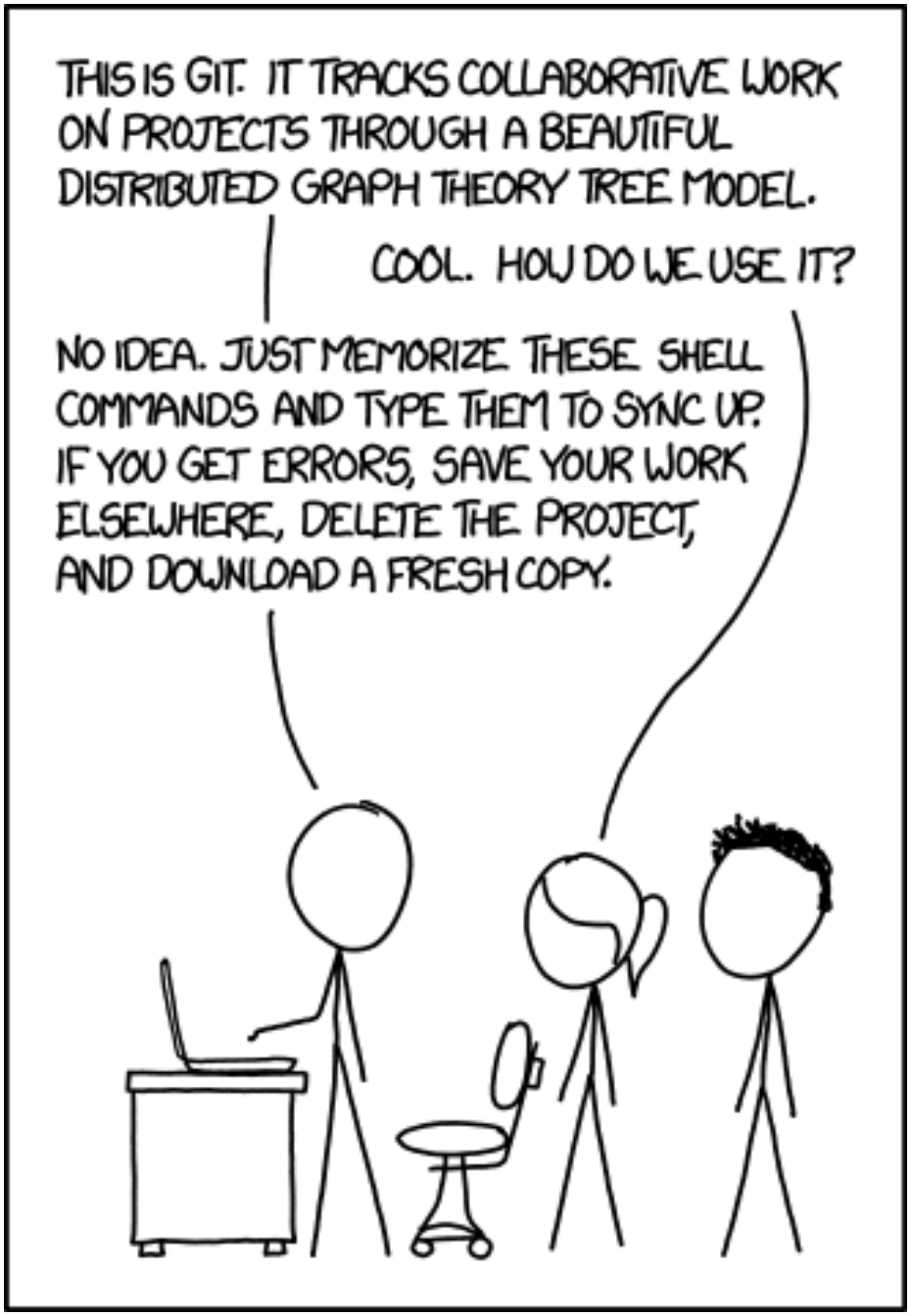

THIS IS GIT. IT TRACKS COLLABORATIVE WORK ON PROJECTS THROUGH A BEAUTIFUL DISTRIBUTED GRAPH THEORY TREE MODEL.
COOL. HOW DO WE USE IT?
NO IDEA. JUST MEMORIZE THESE SHELL COMMANDS AND TYPE THEM TO SYNC UP. IF YOU GET ERRORS, SAVE YOUR WORK ELSEWHERE, DELETE THE PROJECT, AND DOWNLOAD A FRESH COPY.

Git: (Distributed) Version Control

Lecture 2

The Need for Version Control

- Track evolution of a software artifact
  - Development is often non-linear
    - Older versions need to be supported
    - Newer versions need to be developed
  - Development is non-monotonic
    - May need to undo some work, go back to an older version, or track down when a mistake was introduced
- Facilitate team-based development
  - Multiple developers working on a common code base
  - How can project be edited simultaneously?

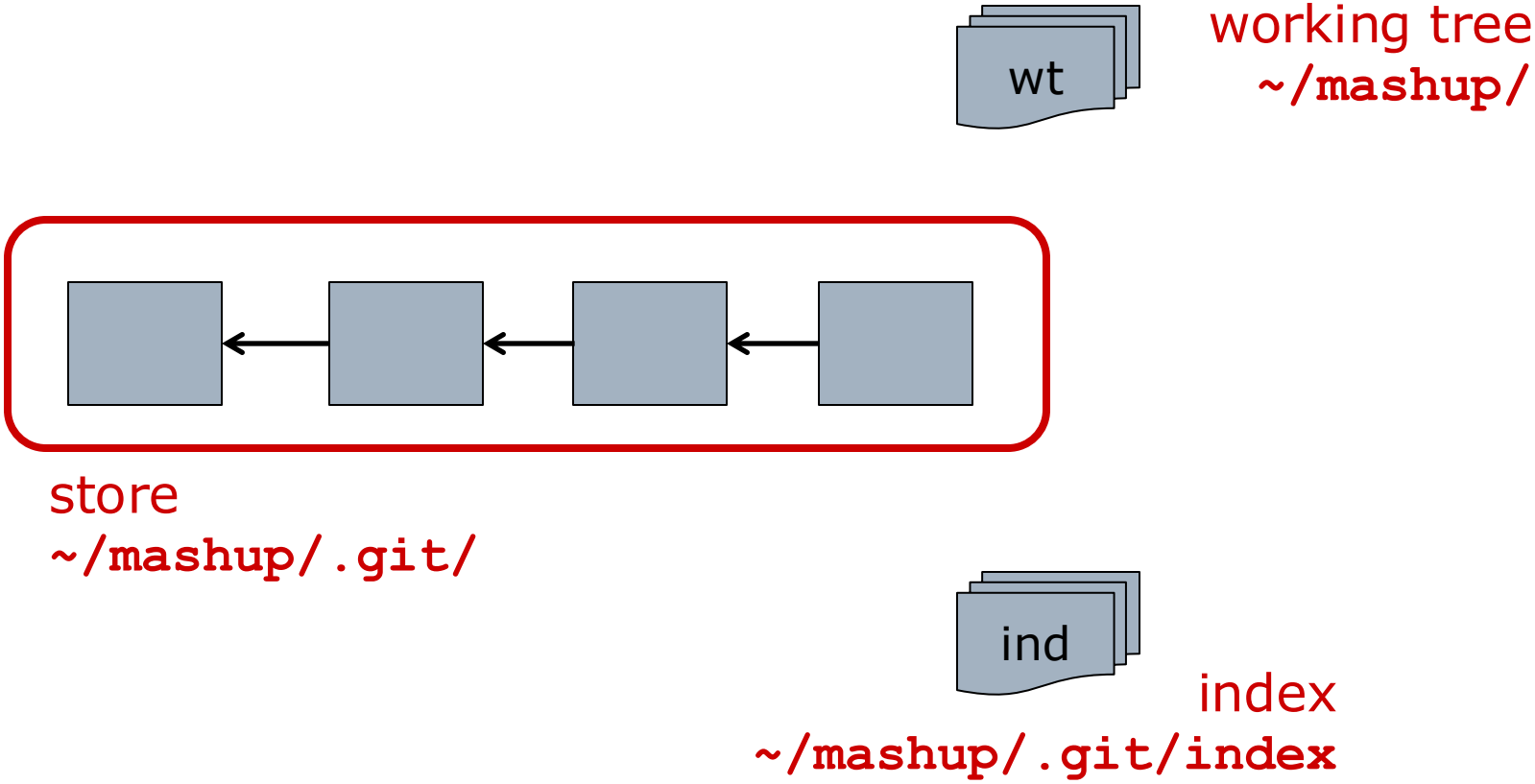
Key Idea: A Repository

- *Repository*= working tree + store + index
  - Warning: "Repo" often used (incorrectly) to mean just the store or just the working tree
- *Working tree* = project itself
  - Ordinary directory with files & subdirectories
- *Store* = history of project
  - Hidden directory: don't touch!
- *Index* = virtual snapshot
  - Gateway for moving changes in the working tree into the store (aka stage, cache)
- *History* = DAG of *commits*
  - Each node in graph corresponds to a complete snapshot of the entire project

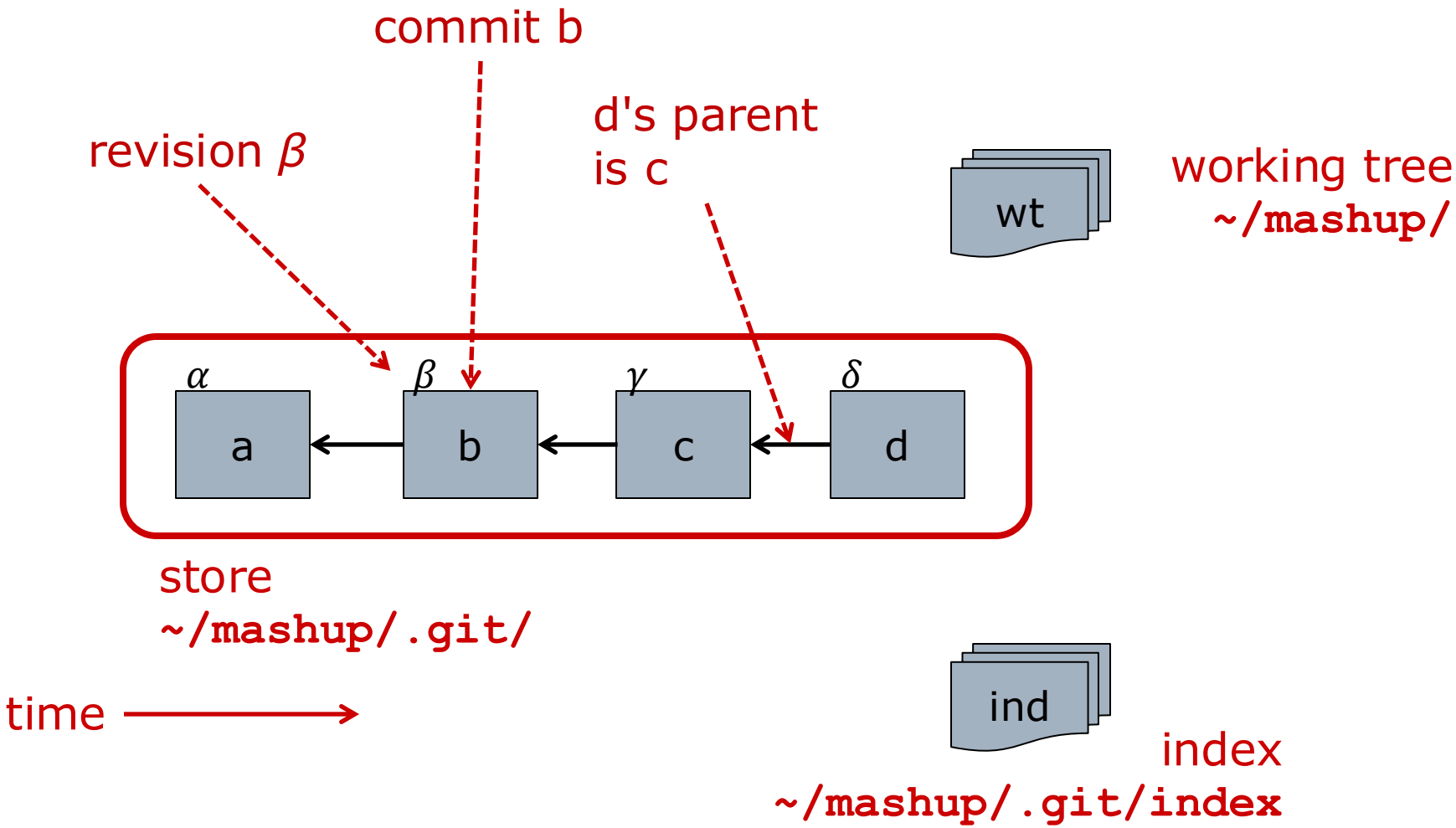
File Structure of a Repository


```
~/mashup/
├── css/
│   ├── buckeye-alert-resp.css
│   └── demo.css
├── demo-js.html
├── Gemfile
├── Gemfile.lock
├── .git/
│   ├── HEAD
│   ├── index
│   └── ...etc...
├── .gitignore
├── Rakefile
├── README.md
└── ...etc...
```

Conceptual Structure



A History of Commits

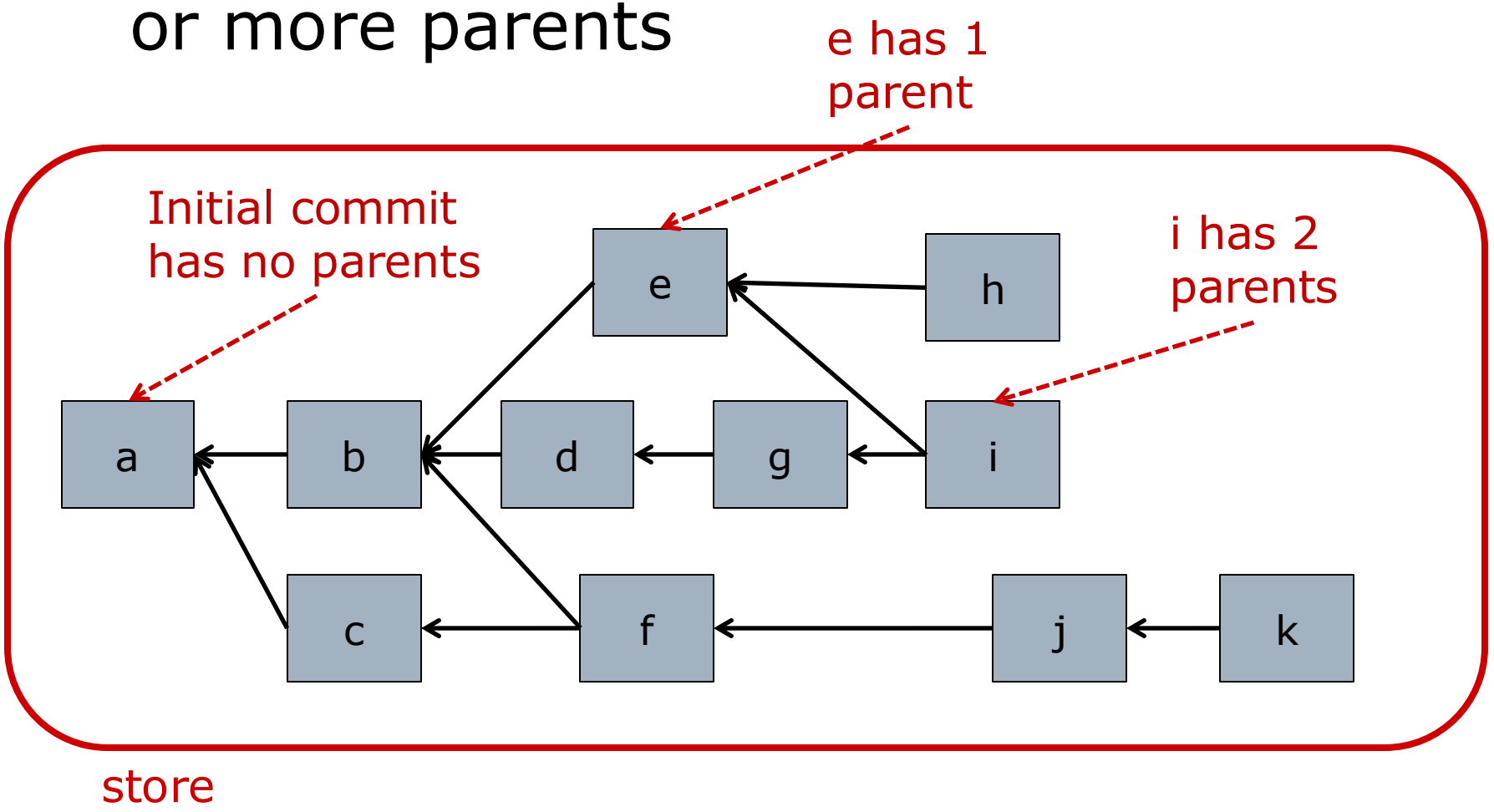


Commit: Snapshot or Delta

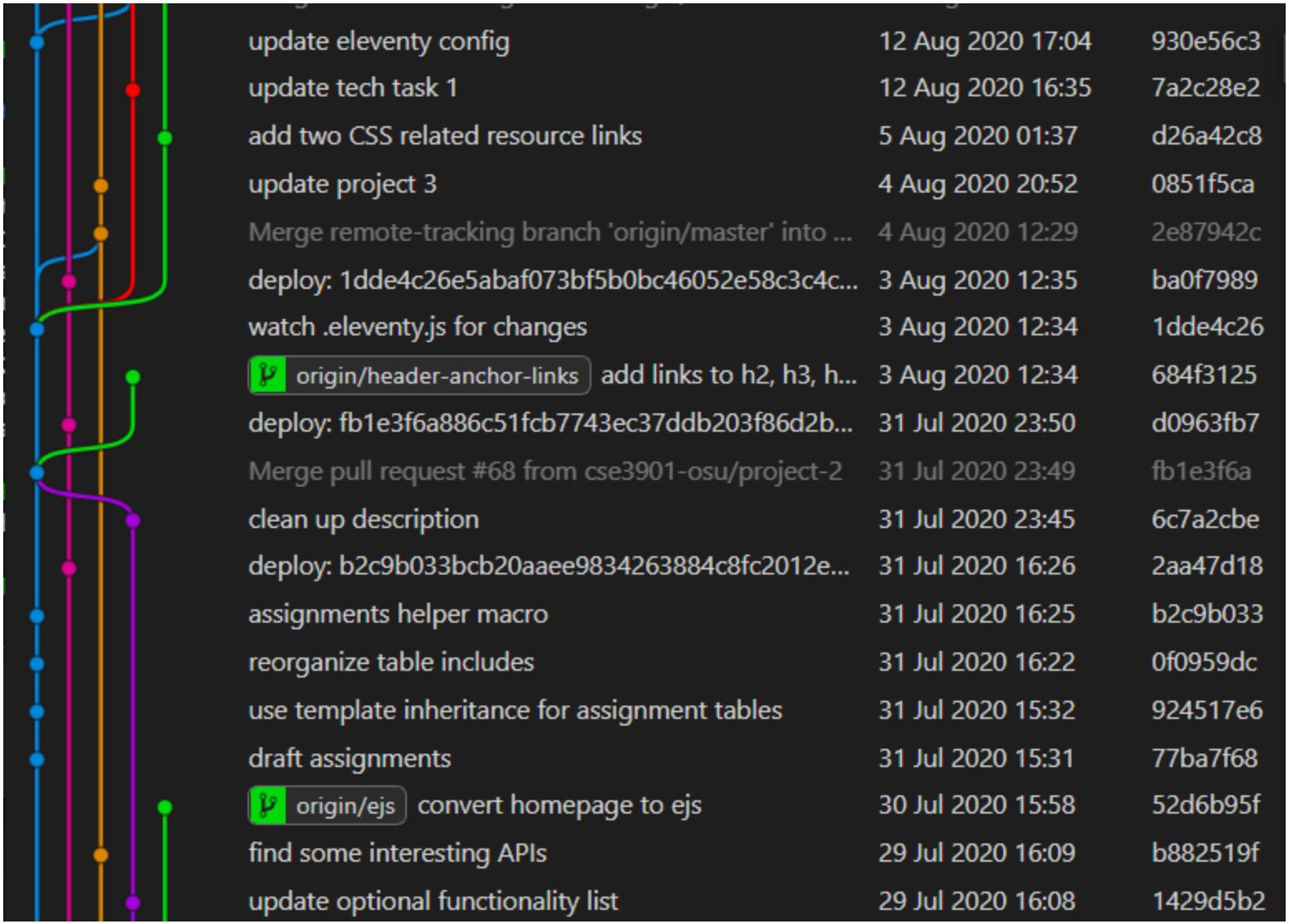
- Each commit represents *both*
  1. A complete snapshot of the project at that point in time (a revision), and
  2. A delta of the changes made (a patch); that is, a diff between snapshots
- Different git actions use different views of a commit
  - View as a snapshot is common
  - Both are useful

History is a DAG

- Every commit (except the first) has 1 or more parents



Example View of DAG



Example View of DAG


```
$ git log --oneline --no-decorate --graph

* 1618849 clean up css
*   d579fa2 merge in improvements from master
|\
| * 0f10869 replace image-url helper in css
* | b595b10 add buckeye alert notes
* | a6e8eb3 add raw buckeye alert download
|/
* b4e201c wrap osu layout around content
* e9d3686 add Rakefile and refactor schedule loop
* 515aaa3 create README.md
* eb26605 initial commit
```

Commit

- Each commit is identified by a hash
  - 160 bits (i.e., 40 hex digits)
  - Practically guaranteed to be unique
  - Can use short prefix of hash if unique

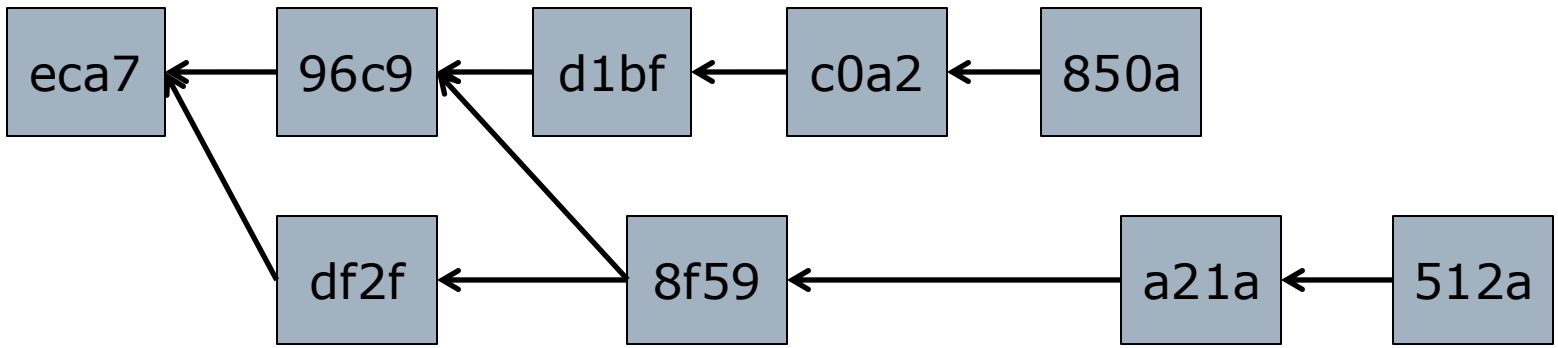
```
$ git show --name-only --no-decorate
commit 16188493c252f6924baa17c9b84a4c1baaed438b
Author: Brutus Buckeye <brutus@users.noreply.github.com>
Date:   Mon Mar 29 15:30:50 2021 +0200

    clean up css

source/stylesheets/_site.css
```

History is a DAG

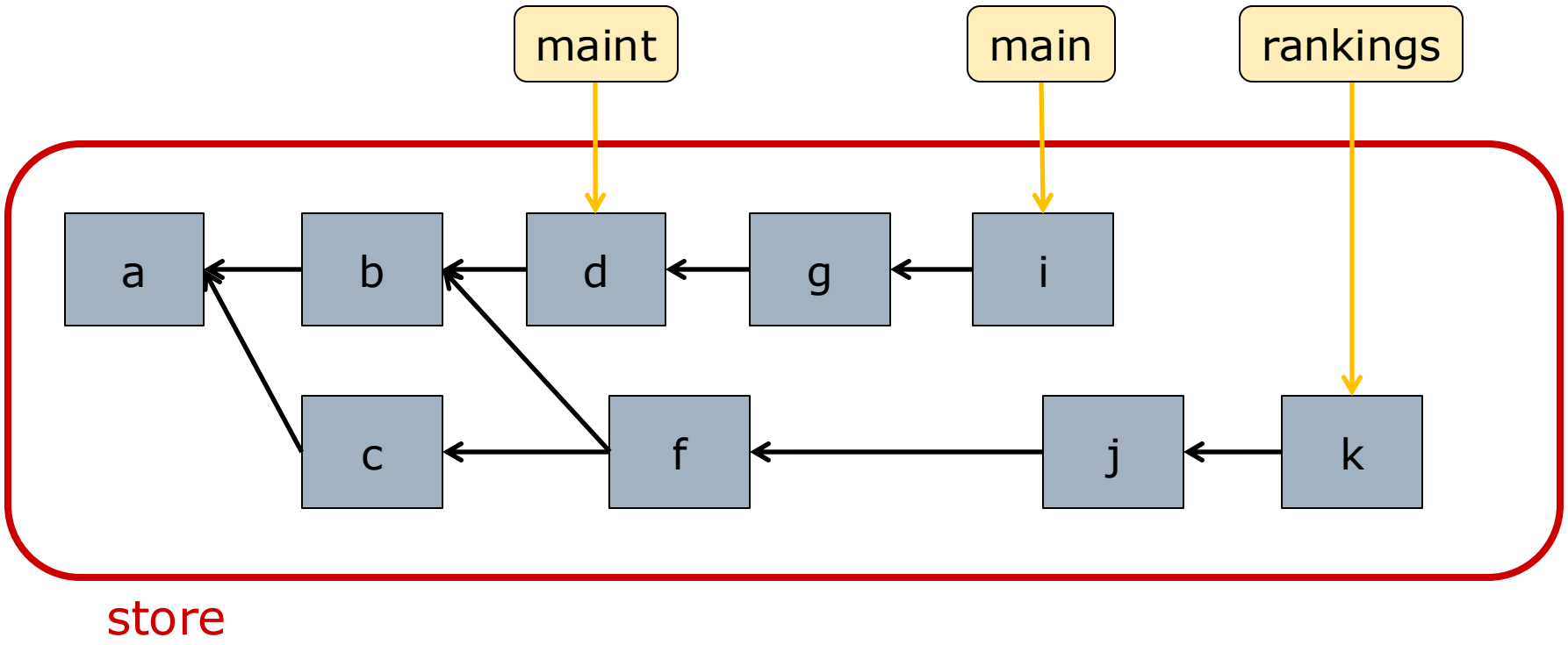
- A better picture would label each commit with its hash (prefix)



- But in these slides, we abbreviate the hash id's as just: 'a', 'b', 'c'…

Nomenclature: Branch

- *Branch*: a pointer to a commit
- Different from “branch” in DAG's shape

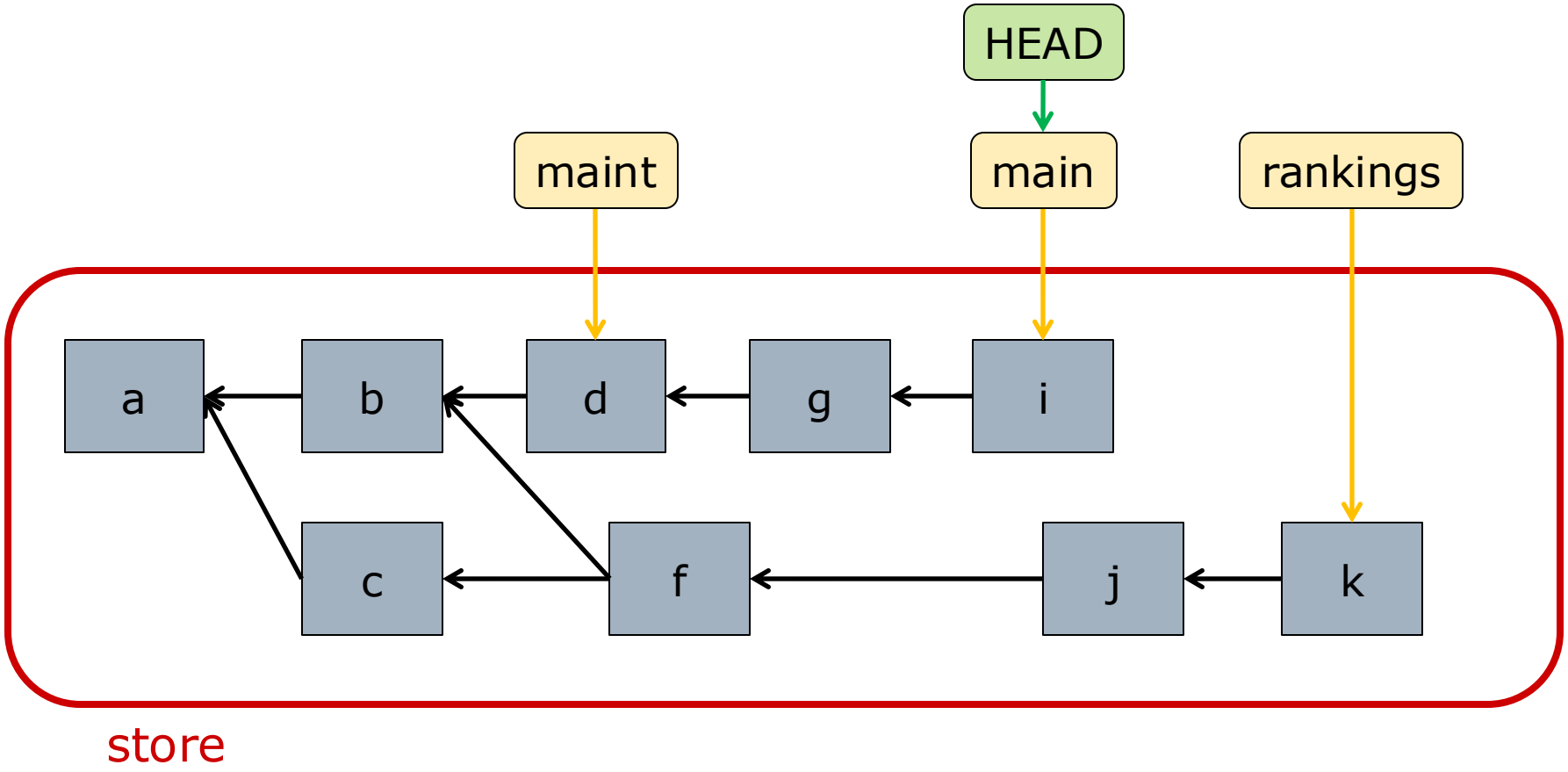


A Note on Changing Defaults

- Any name can be used for a branch
  - Typically short, but hopefully descriptive
  - Many branches, each with a unique name
- Initially, a repo has a single branch
  - Repos created on GitHub use “main” as the initial branch name
  - Repos created locally (git 2.42) still use old initial branch name (“master”)
    - This default is user-configurable! (as of git 2.28, 7/27/20)

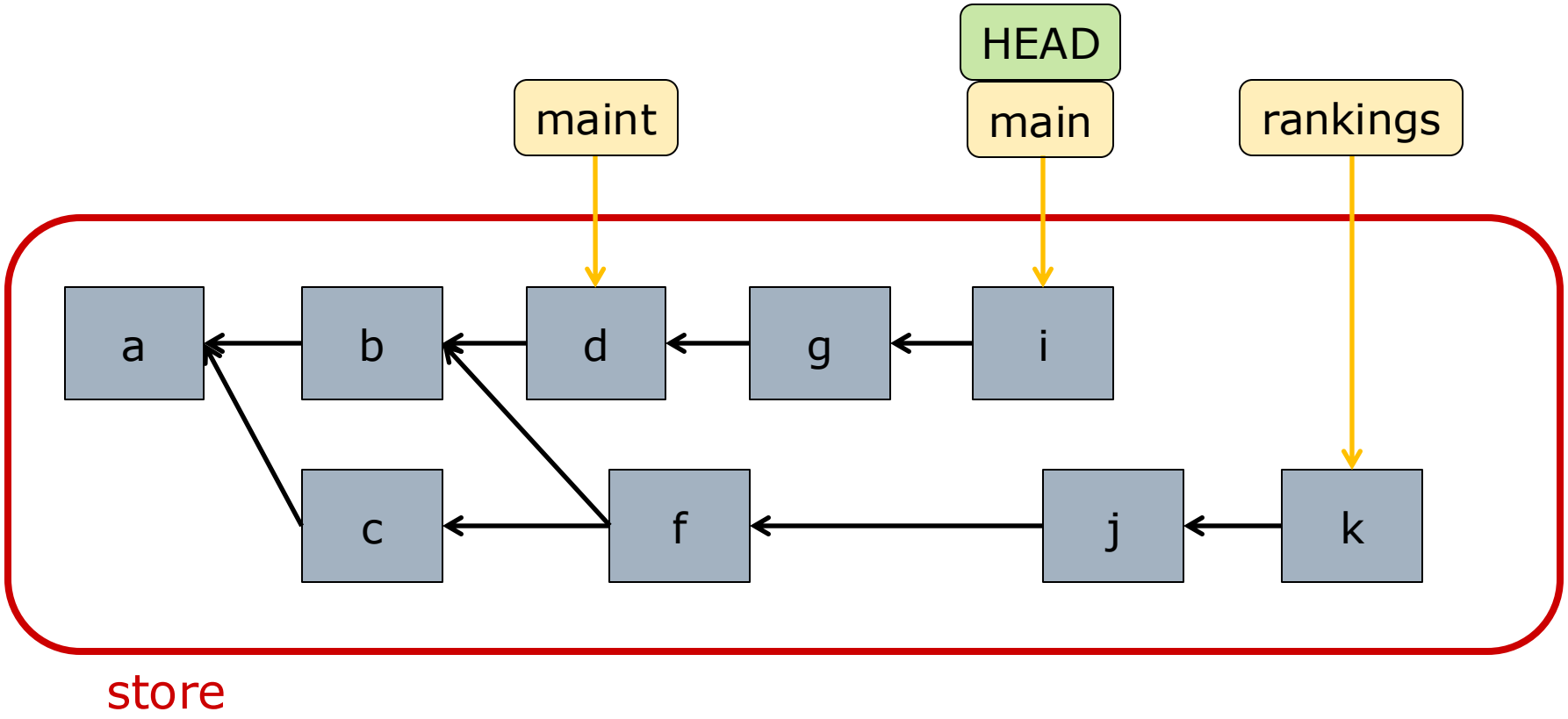
Nomenclature: HEAD

- *HEAD*: a special reference, (usually) points to a branch



Nomenclature: HEAD

- Useful to think of HEAD as being "attached" to a particular branch

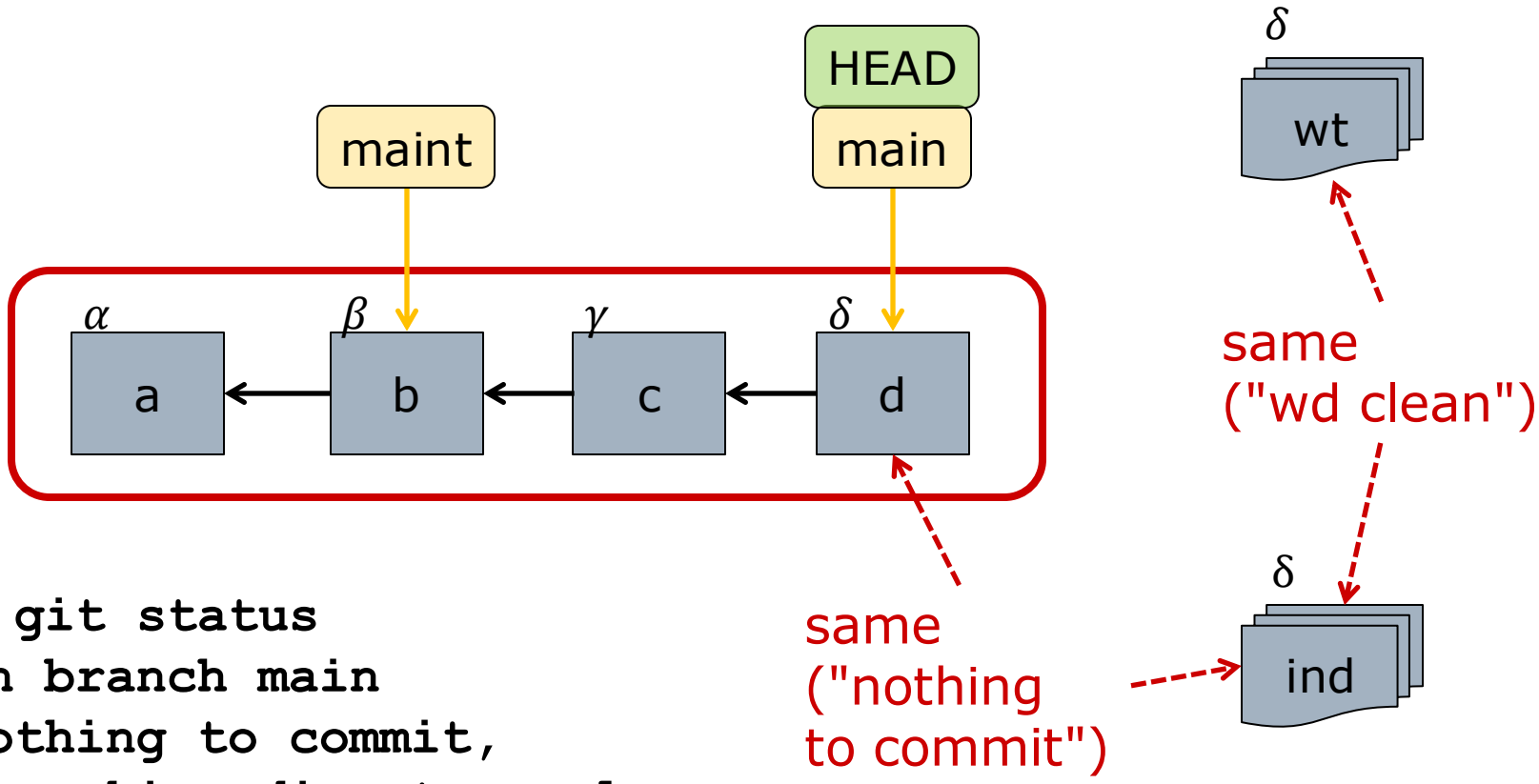


View of DAG with Branches


```
$ git log --oneline --graph

* 1618849 (HEAD -> main) clean up css
*   d579fa2 (alert) merge in improvements from master
|\
| * 0f10869 replace image-url helper in css
* | b595b10 add buckeye alert notes
* | a6e8eb3 add raw buckeye alert download
|/
* b4e201c wrap osu layout around content
* e9d3686 add Rakefile and refactor schedule loop
* 515aaa3 create README.md
* eb26605 initial commit
```

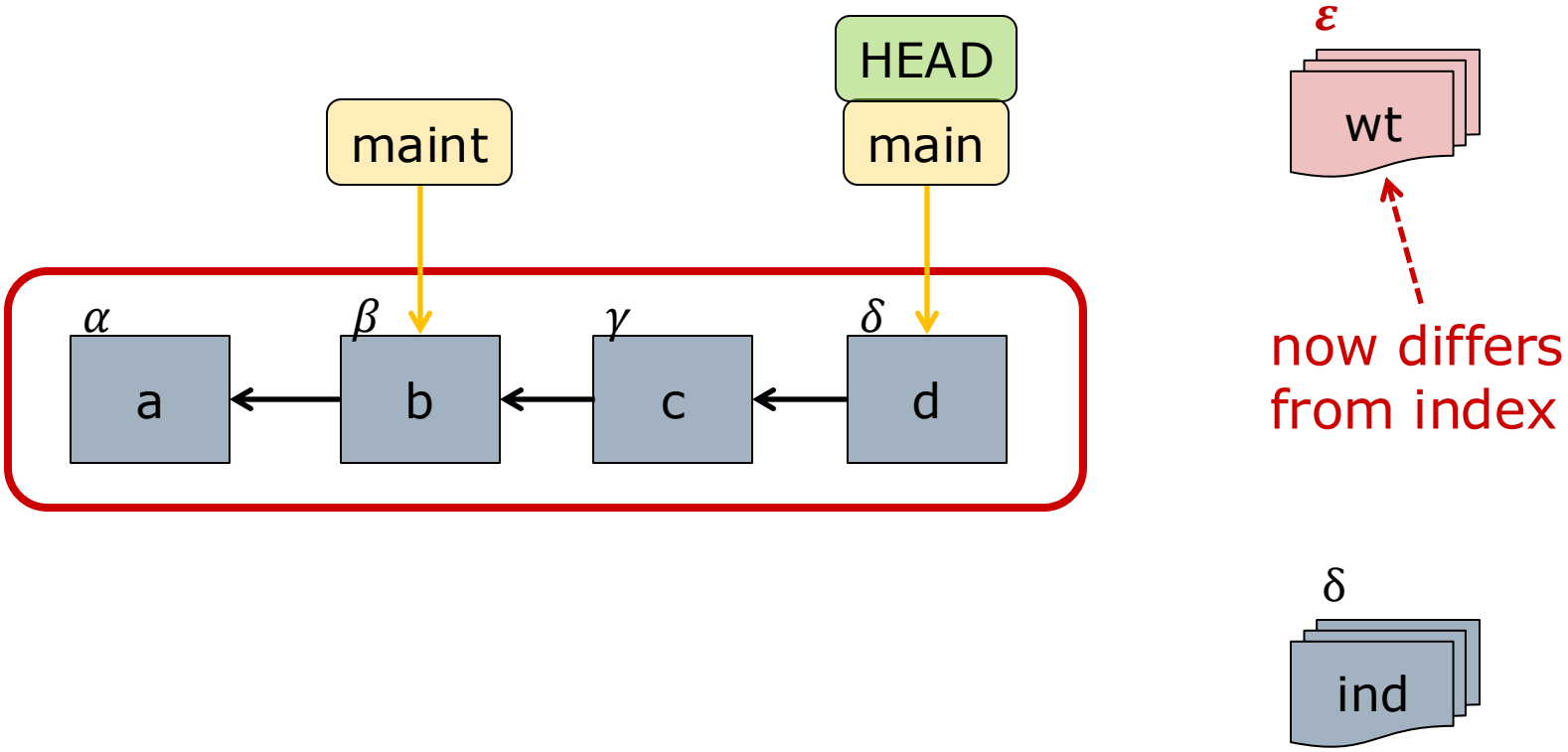
A "Clean" Repository




```
$ git status
On branch main
nothing to commit,
  working directory clean
```

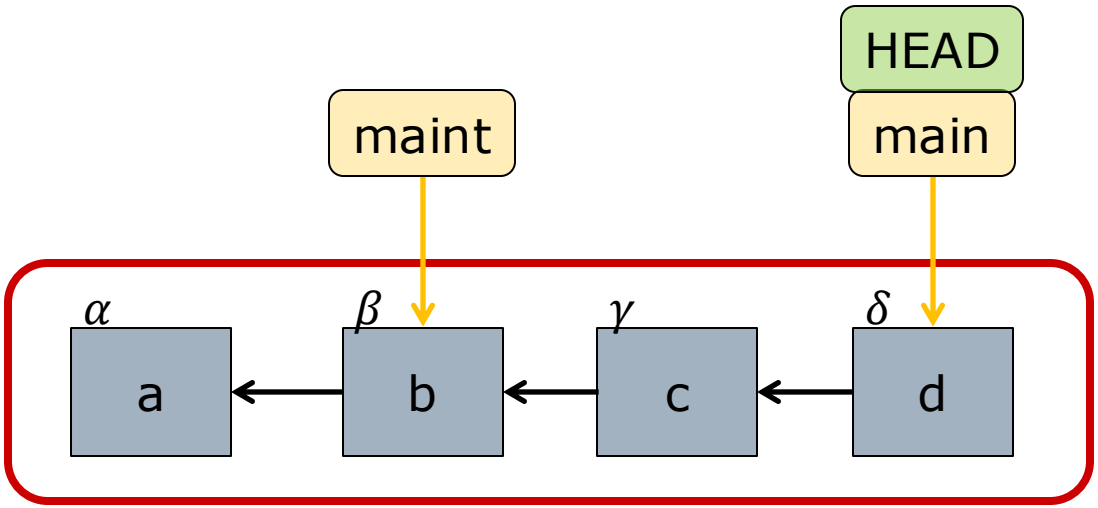
Edit Files in Working Tree

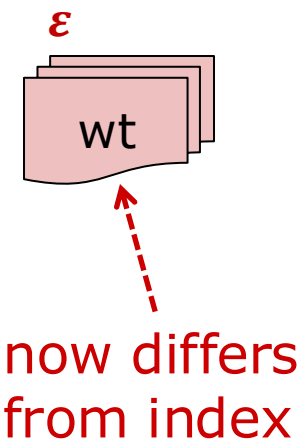
- Add files, remove files, edit files…

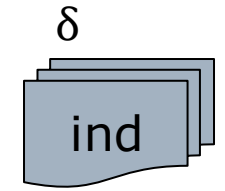


Edit Files in Working Tree

- Add files, remove files, edit files…



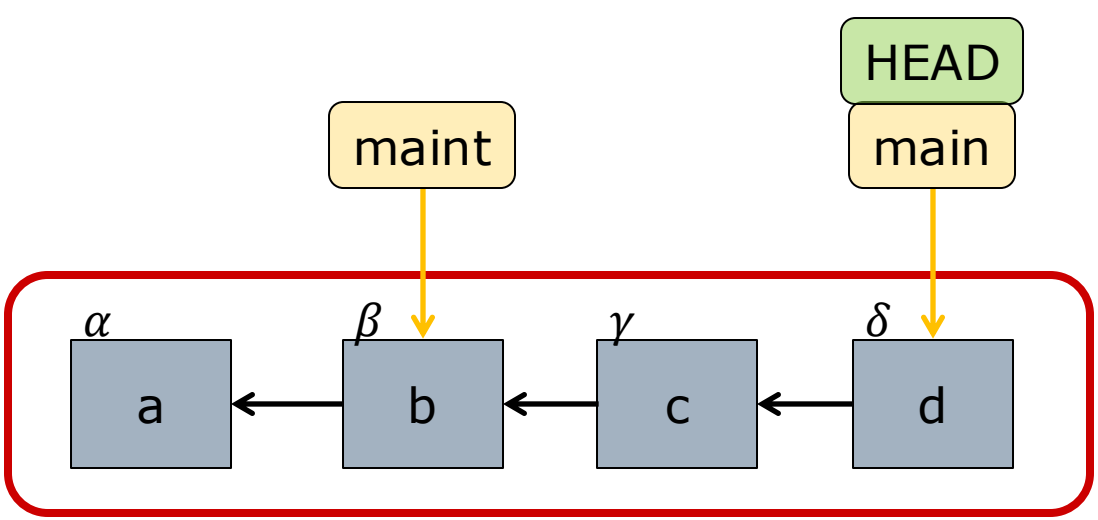


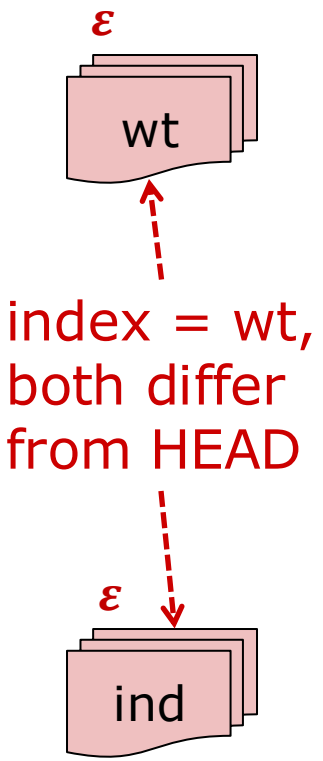



```
$ git status
On branch main
Changes not staged for commit:
     modified: css/demo.css
```

Add: Working Tree → Index

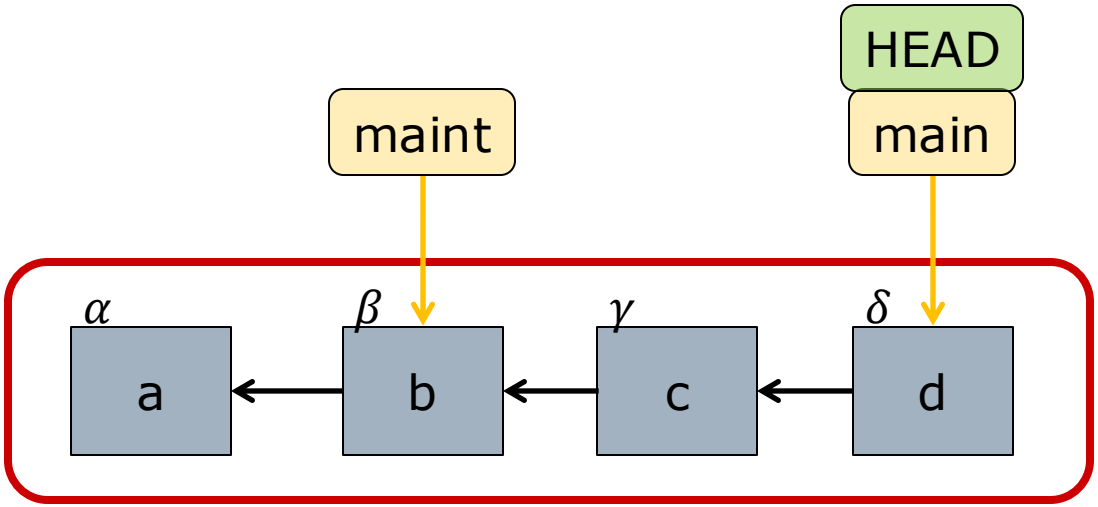

```
$ git add . # current directory, and below
```

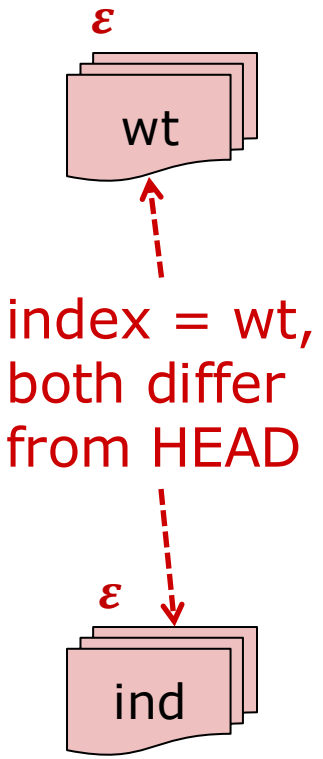




Add: Working Tree → Index


```
$ git add . # current directory, and below
```

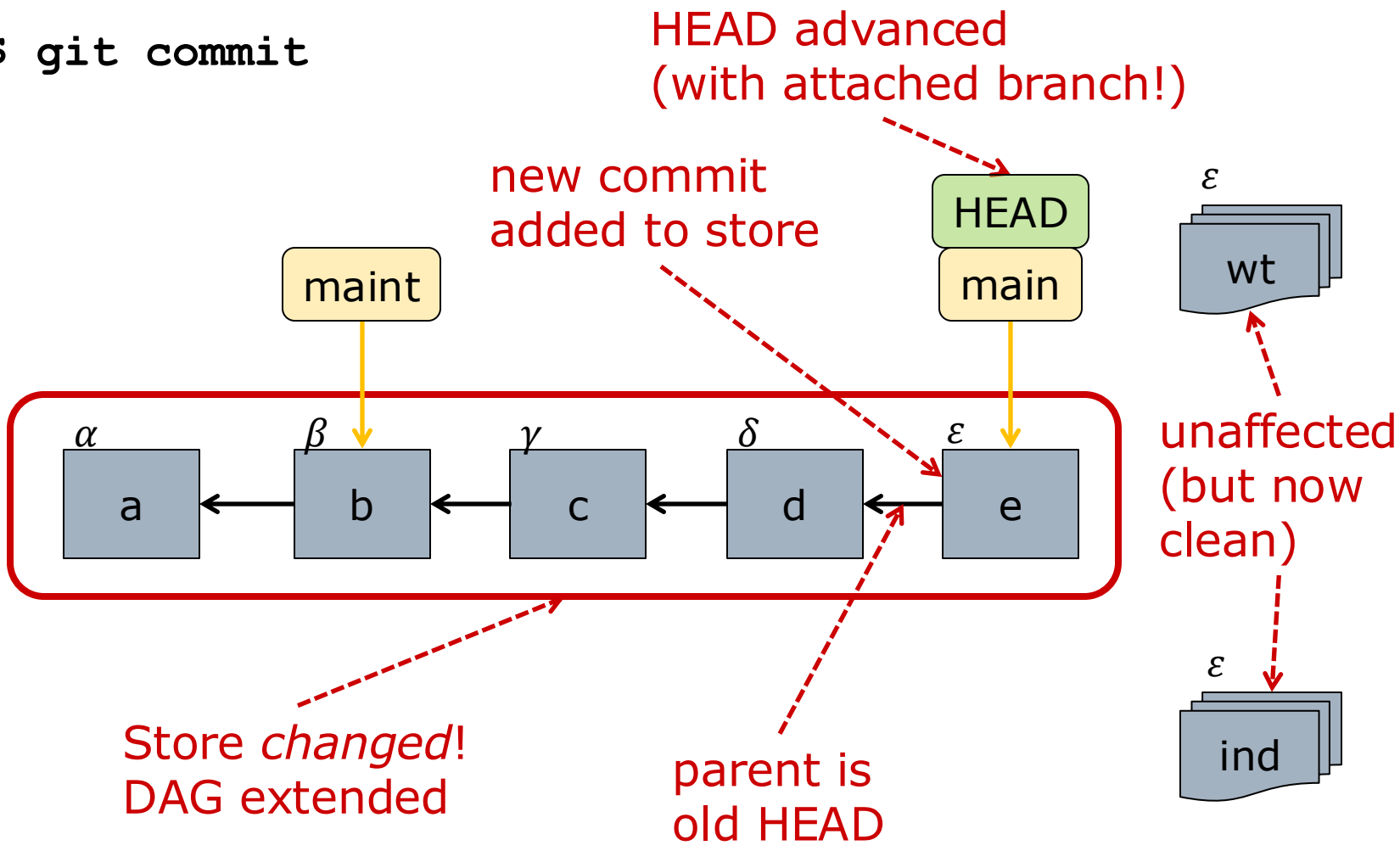




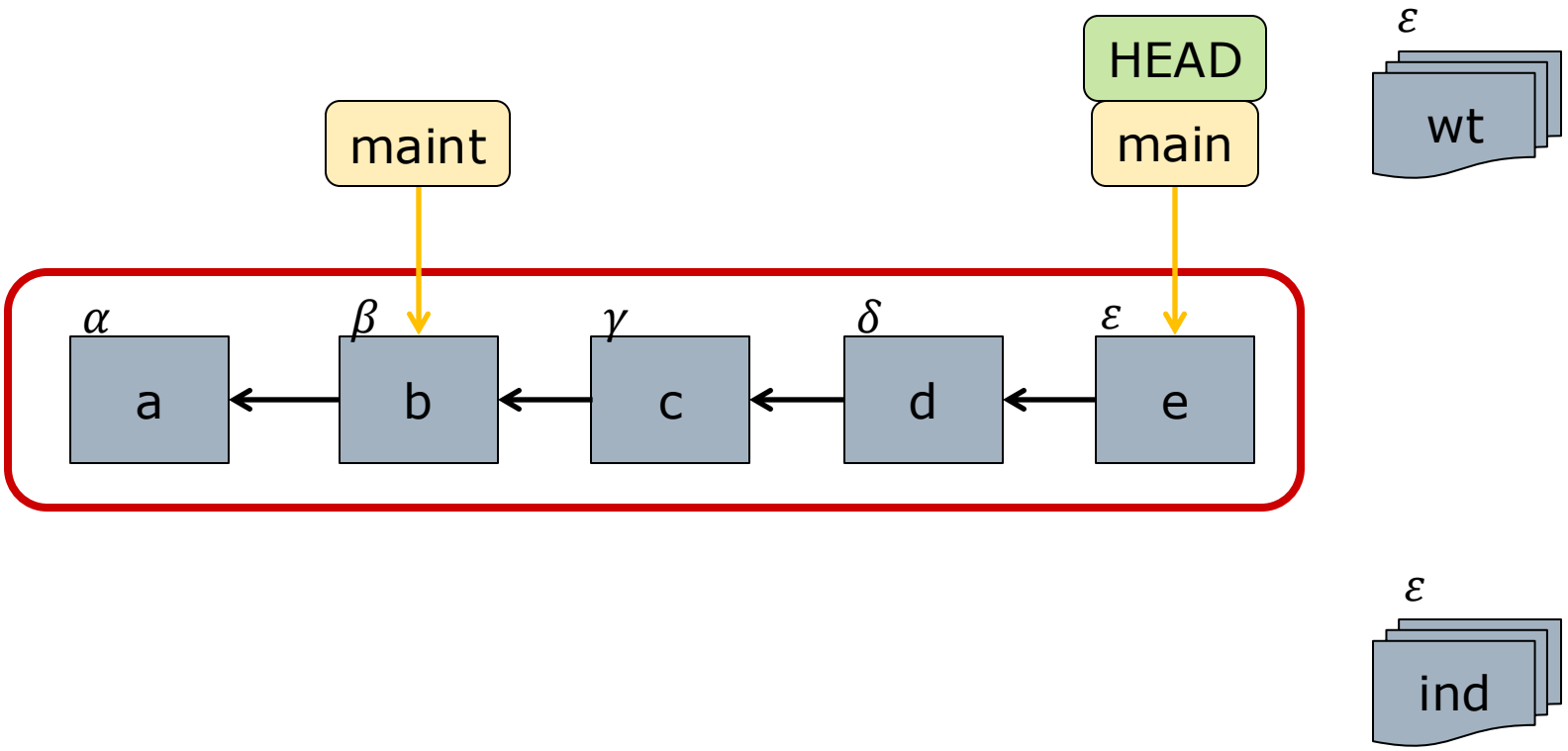

```
$ git status
On branch main
Changes to be committed:
    modified: css/demo.css
```

Commit: Index → Store


```
$ git commit
```

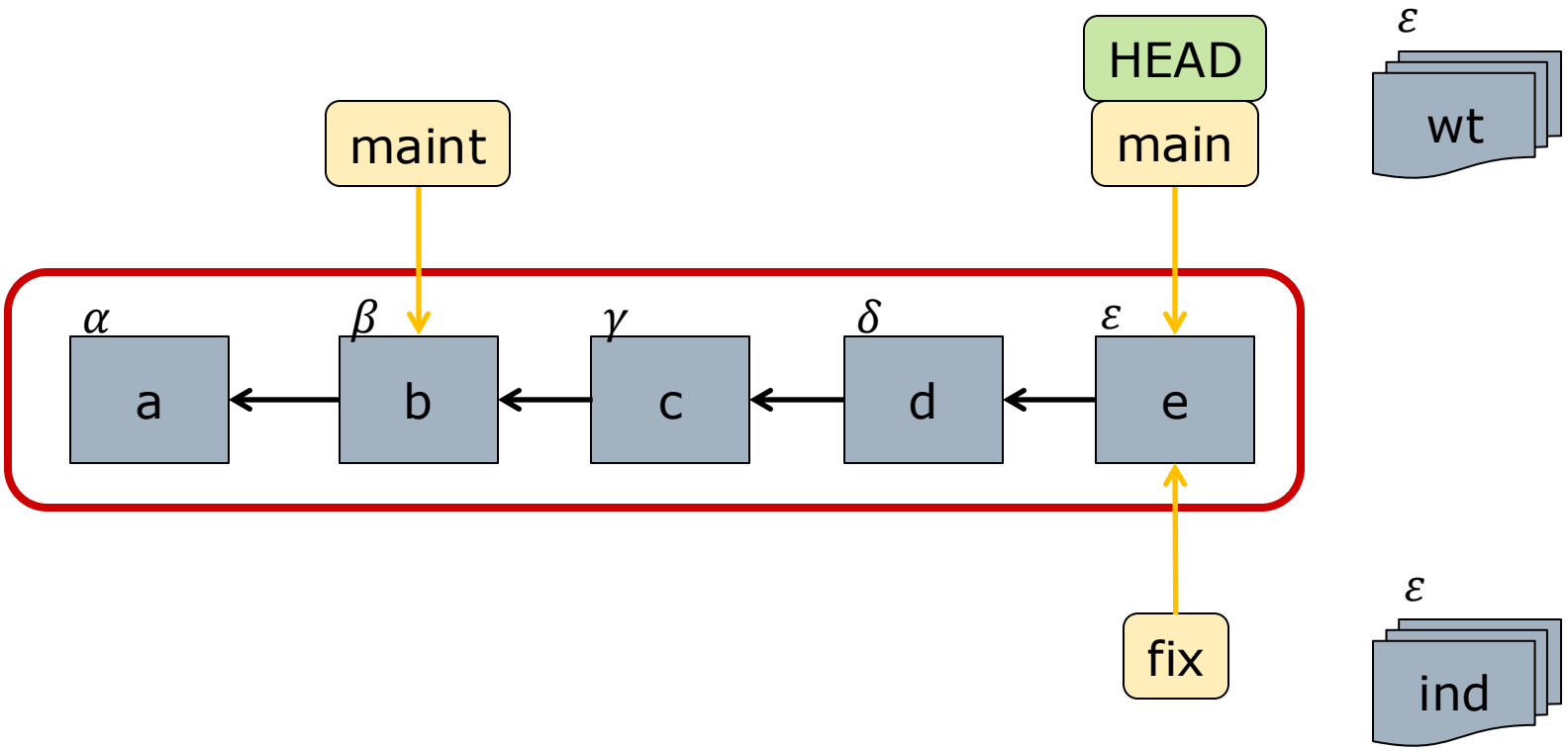


The (New) State of Repository



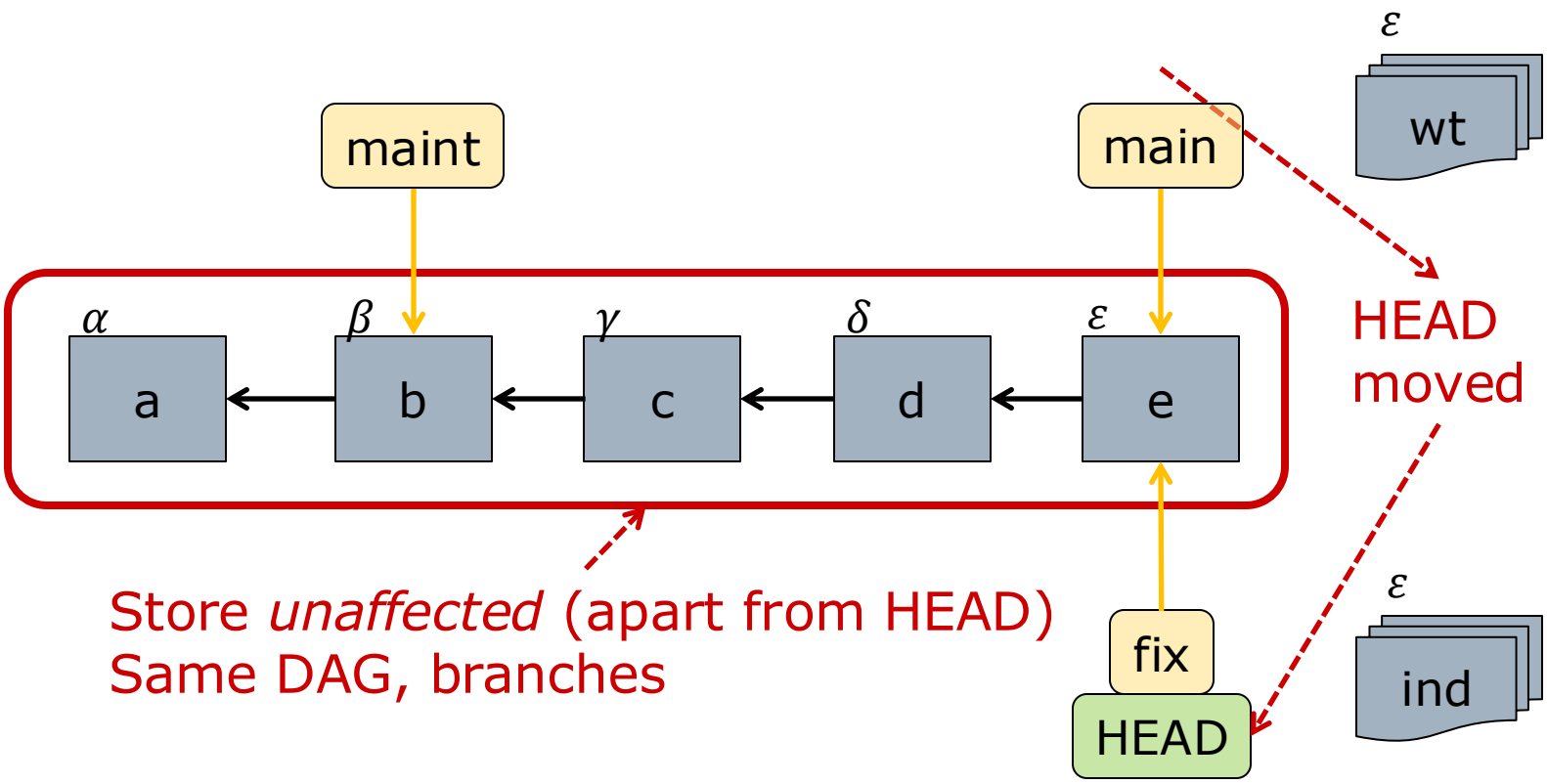
Creating a New Branch


```
$ git branch fix
```



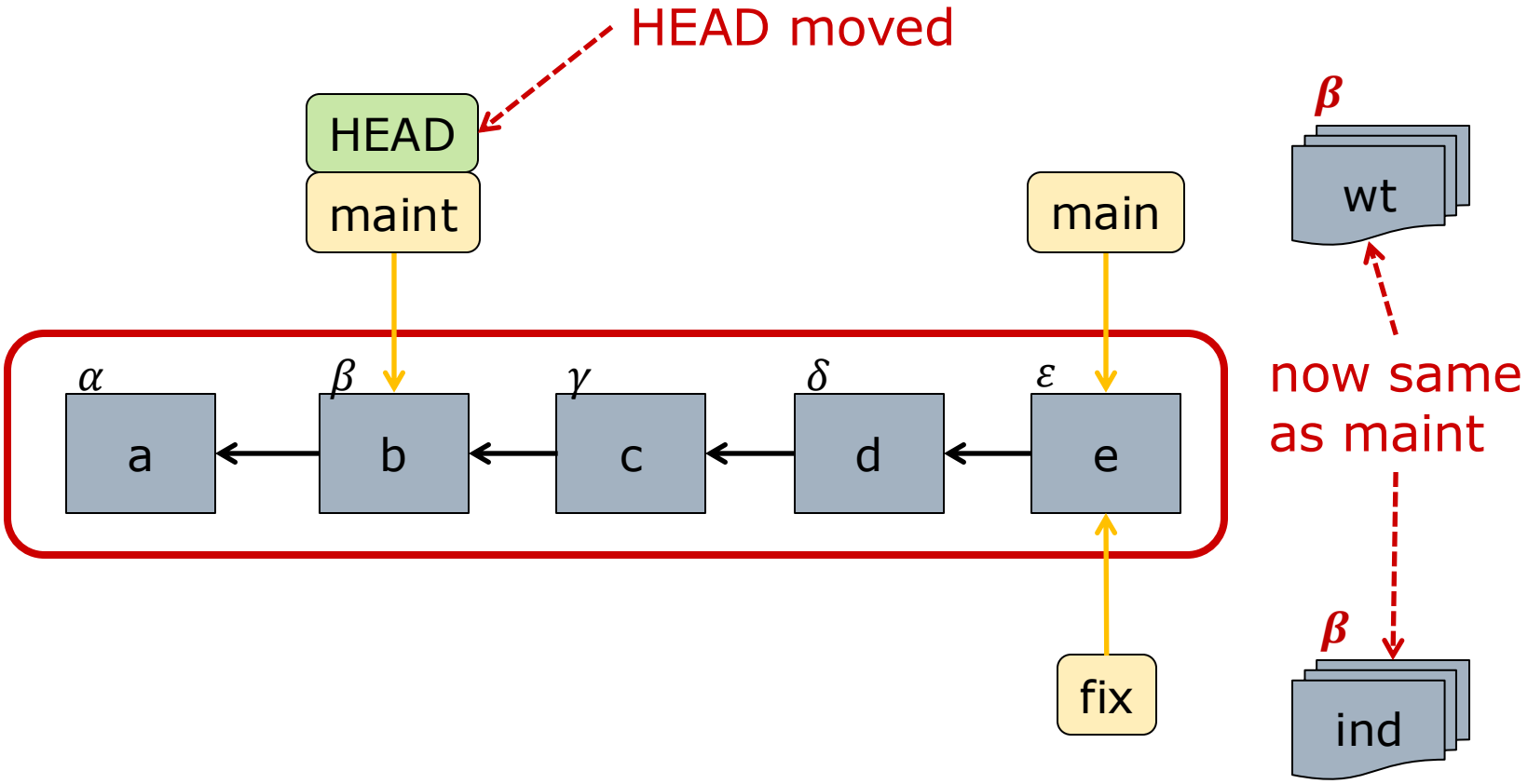
Checkout: Changing Branch


```
$ git checkout fix
```



Checkout: Changing Branch

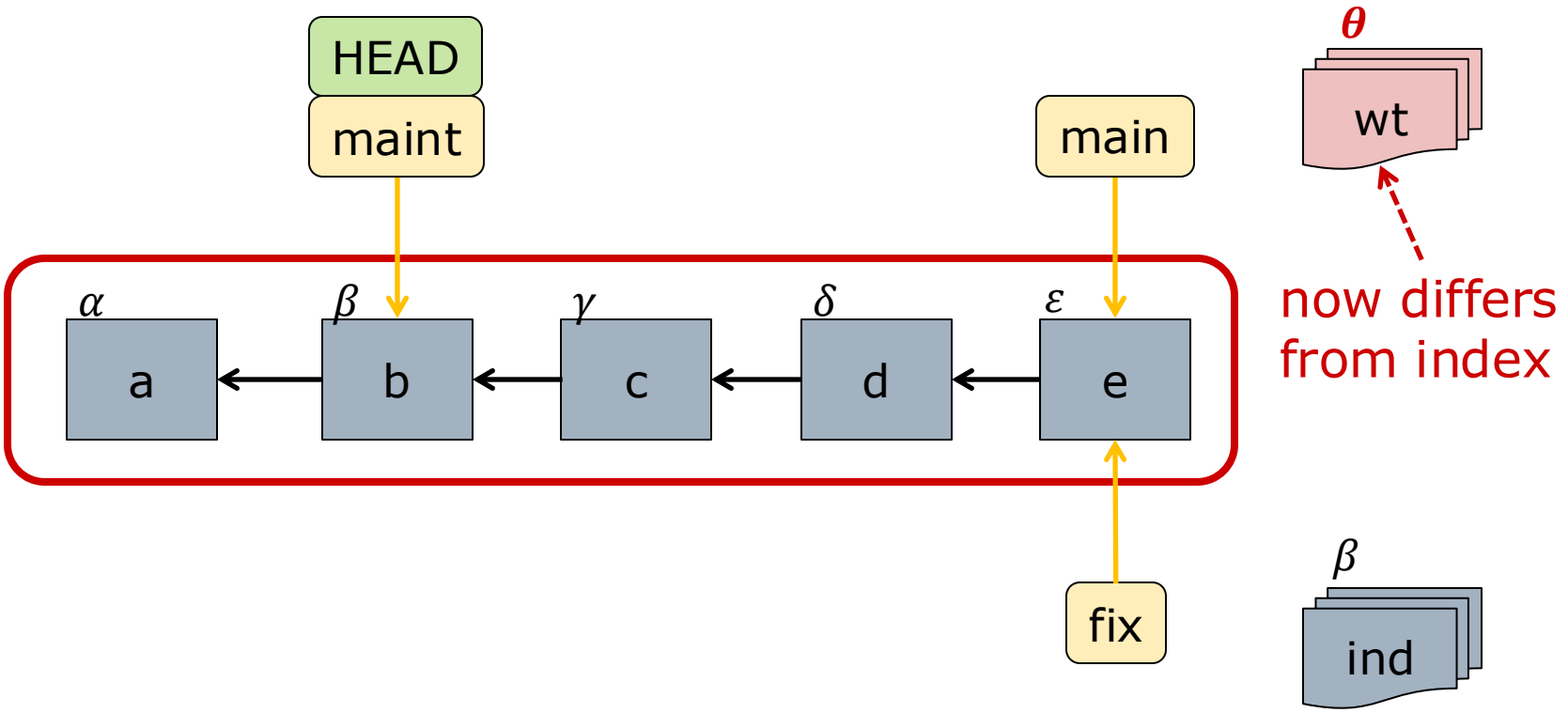

```
$ git checkout maint
```



- Advice: checkout <branch> *only* when wt is clean

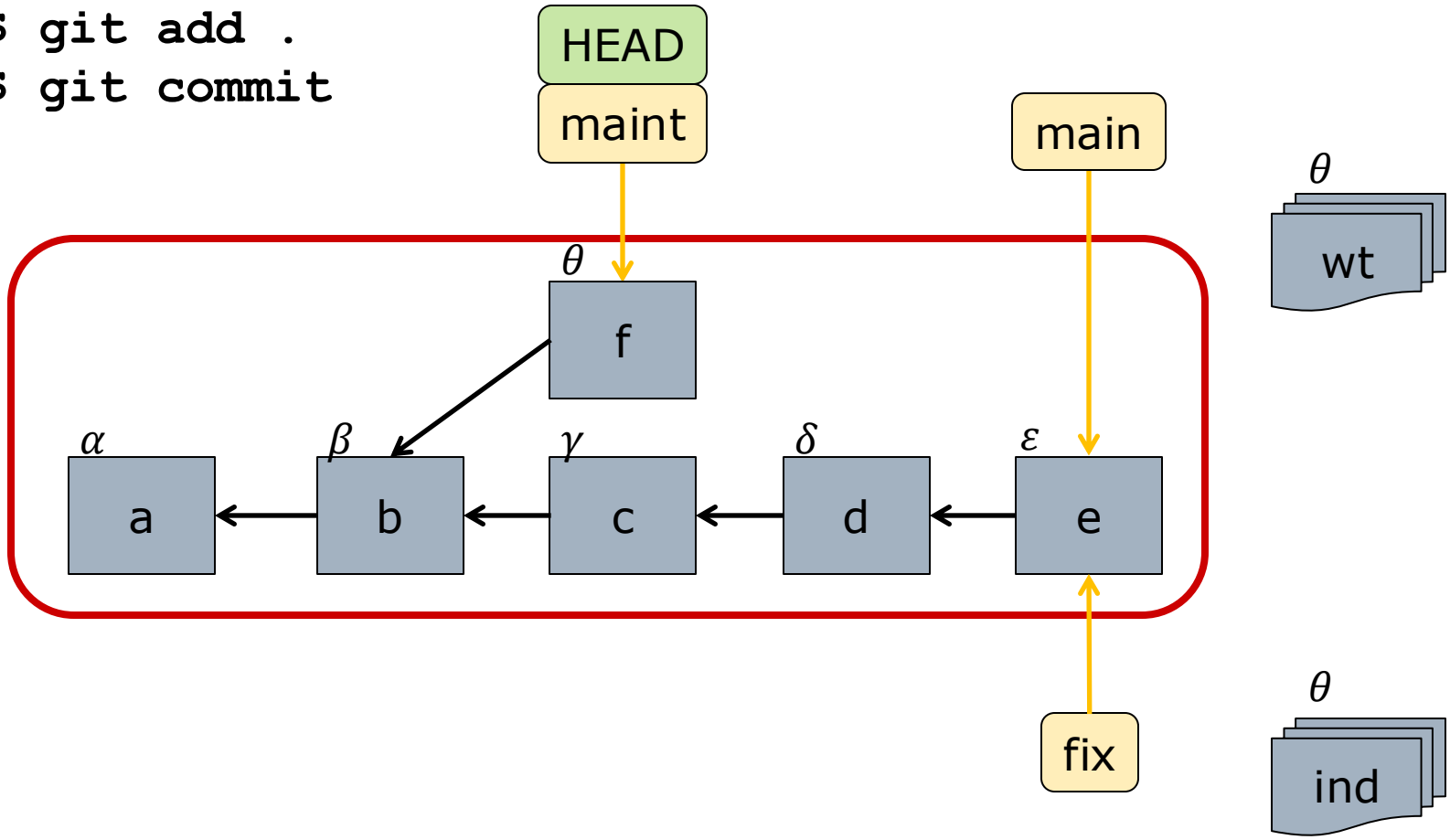
Edit Files in Working Tree

- Add files, remove files, edit files…



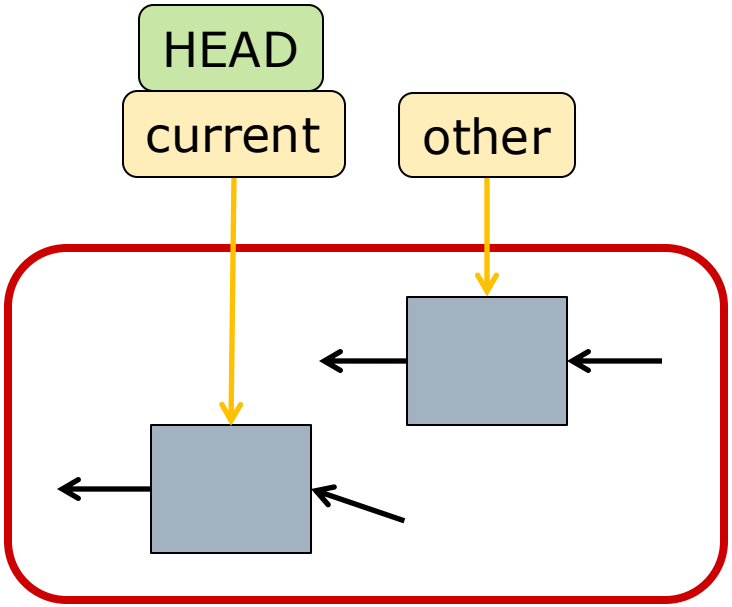
Add & Commit: Update Store

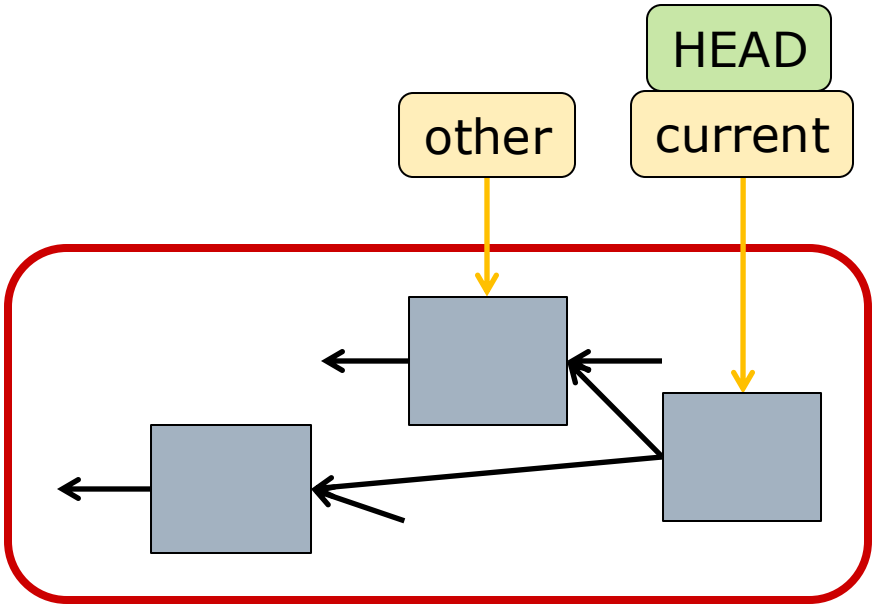

```
$ git add .
$ git commit
```



Merge: Bringing History together

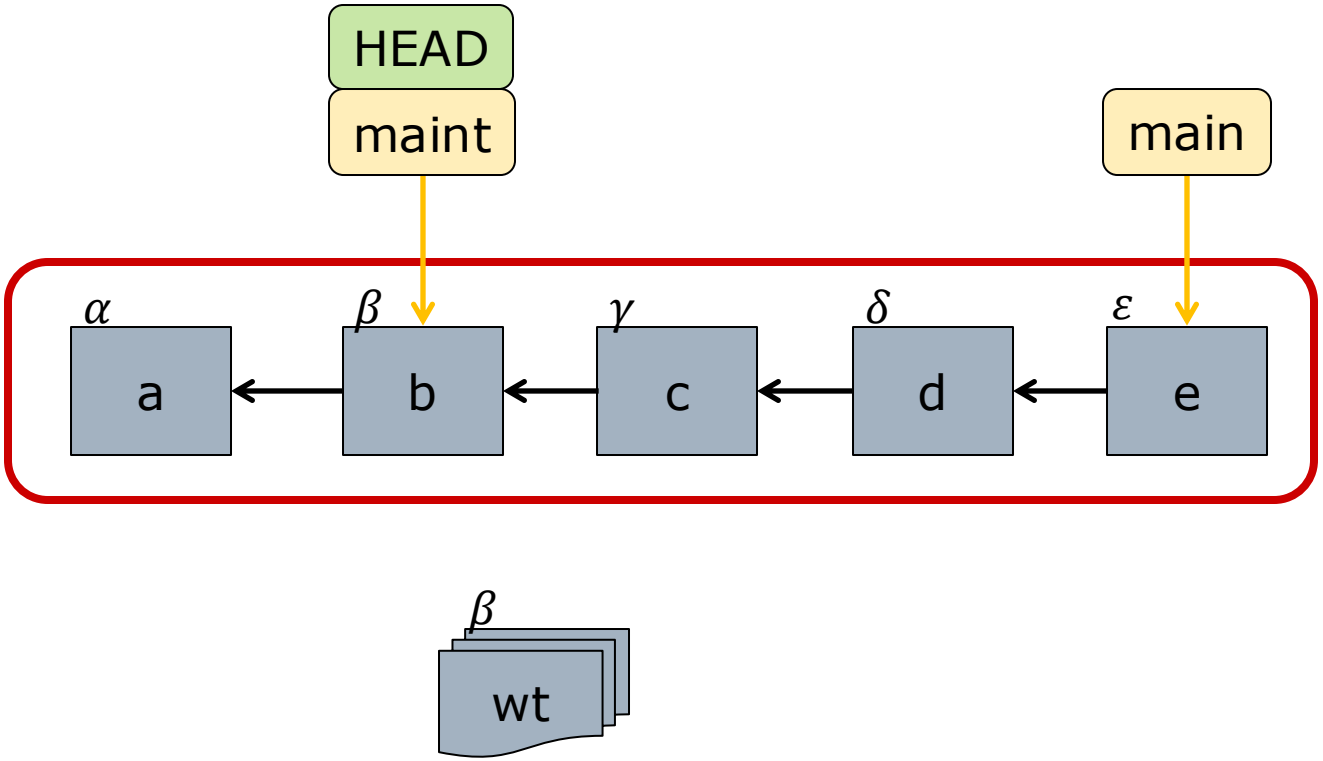
- Bring work from another branch into current branch
  - Implemented features, fixed bugs, etc.
- Updates current branch, not other





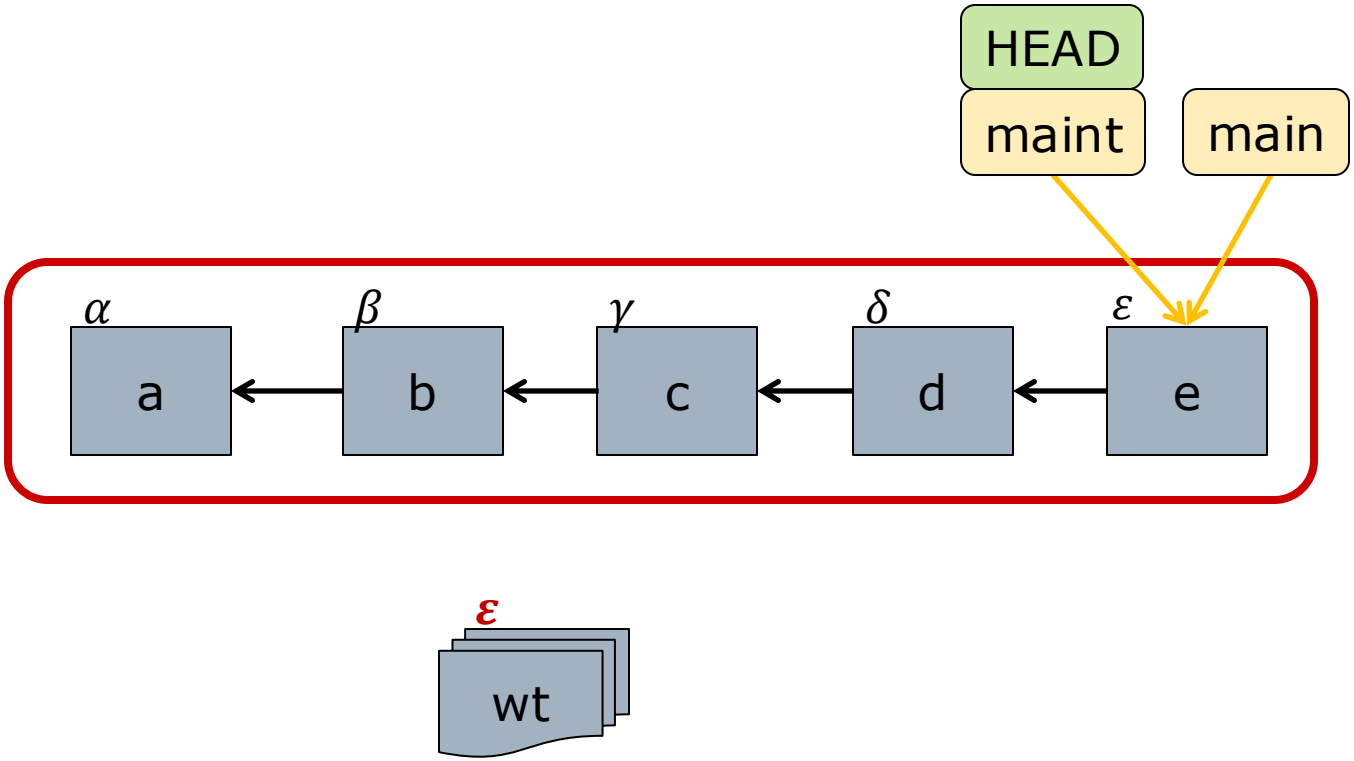
Merge – Case 1: Ancestor

- HEAD is an ancestor of other branch

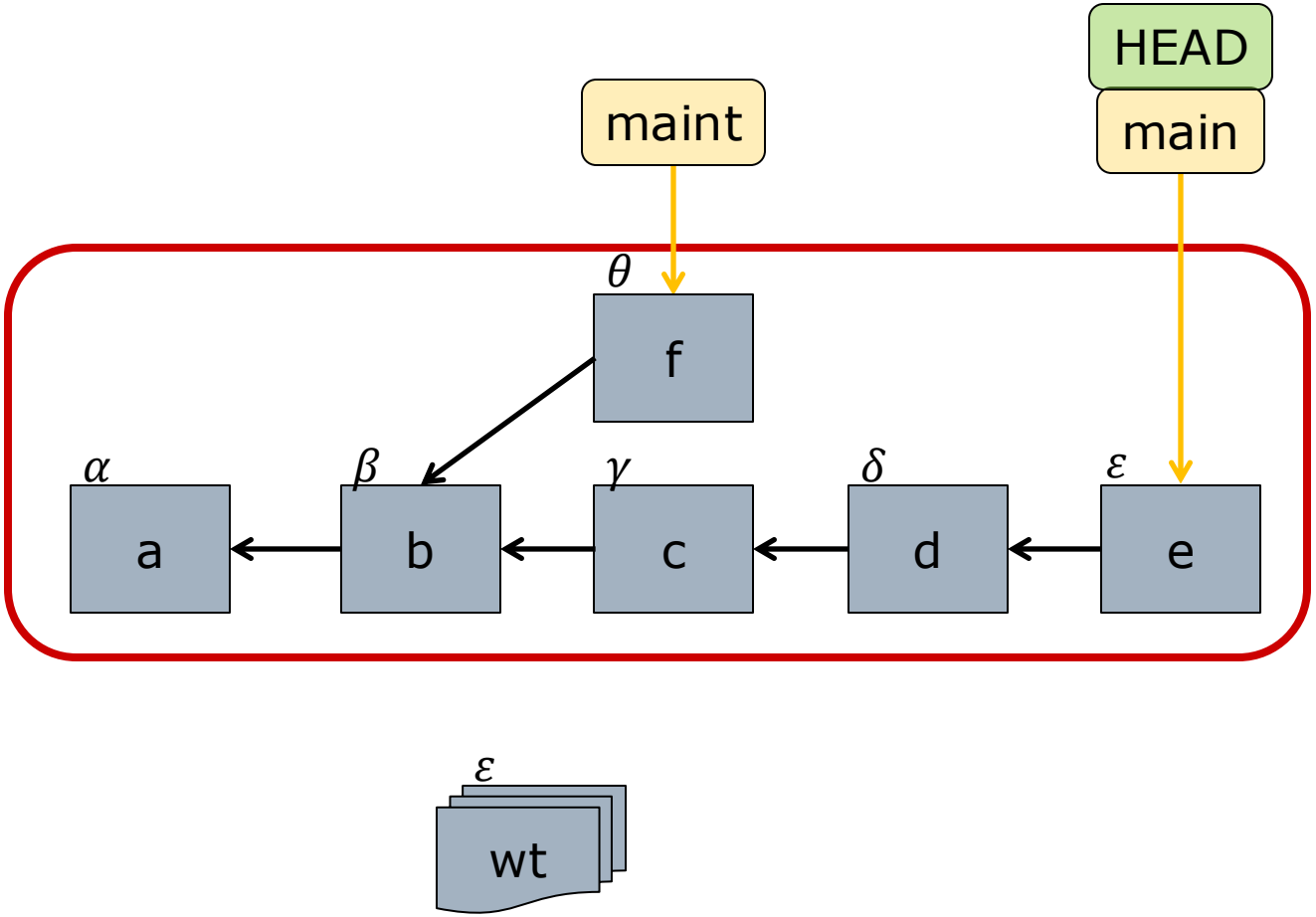


Fast-Forward Merge


```
$ git merge main
```

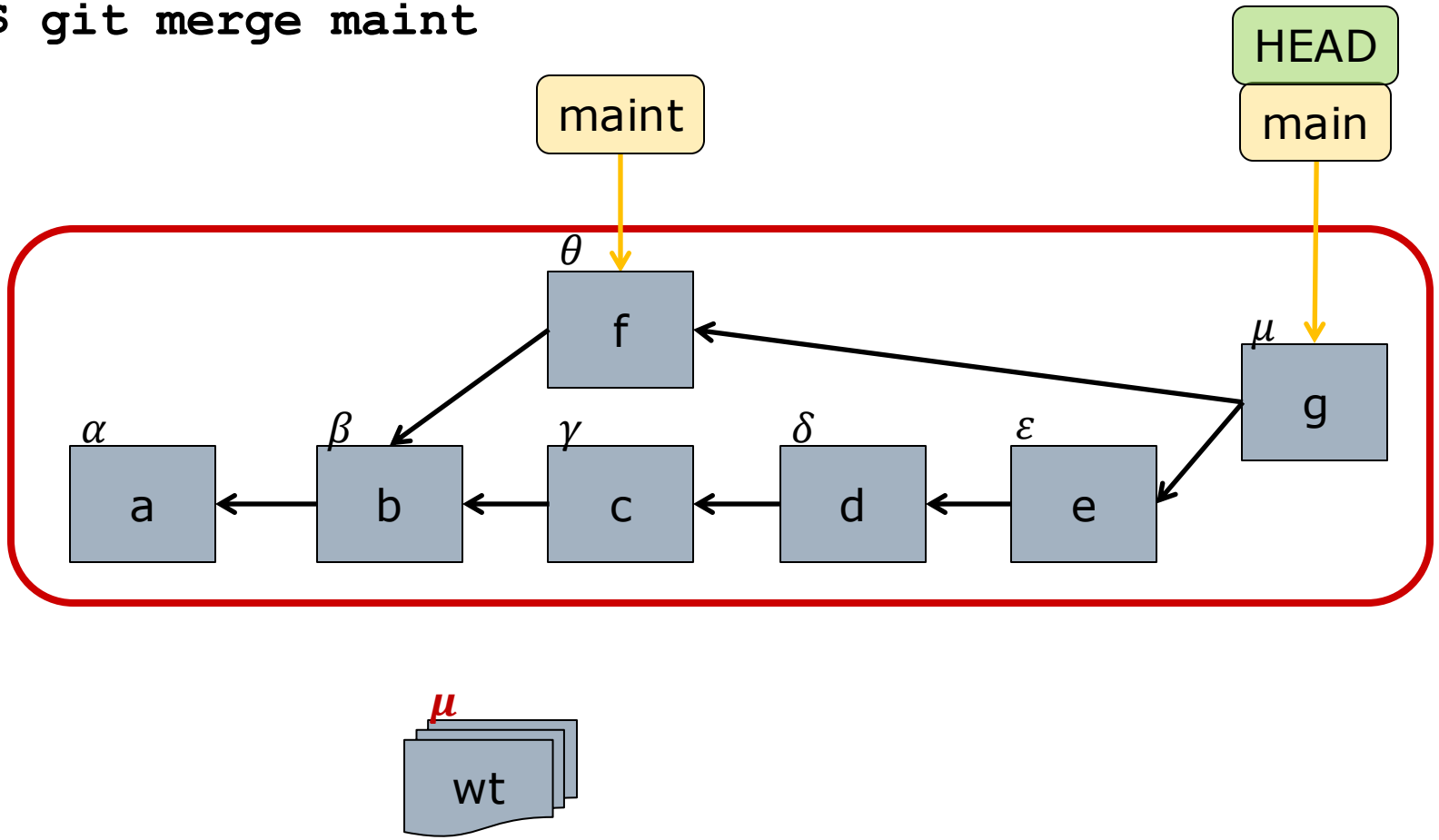


Merge – Case 2: No Conflicts



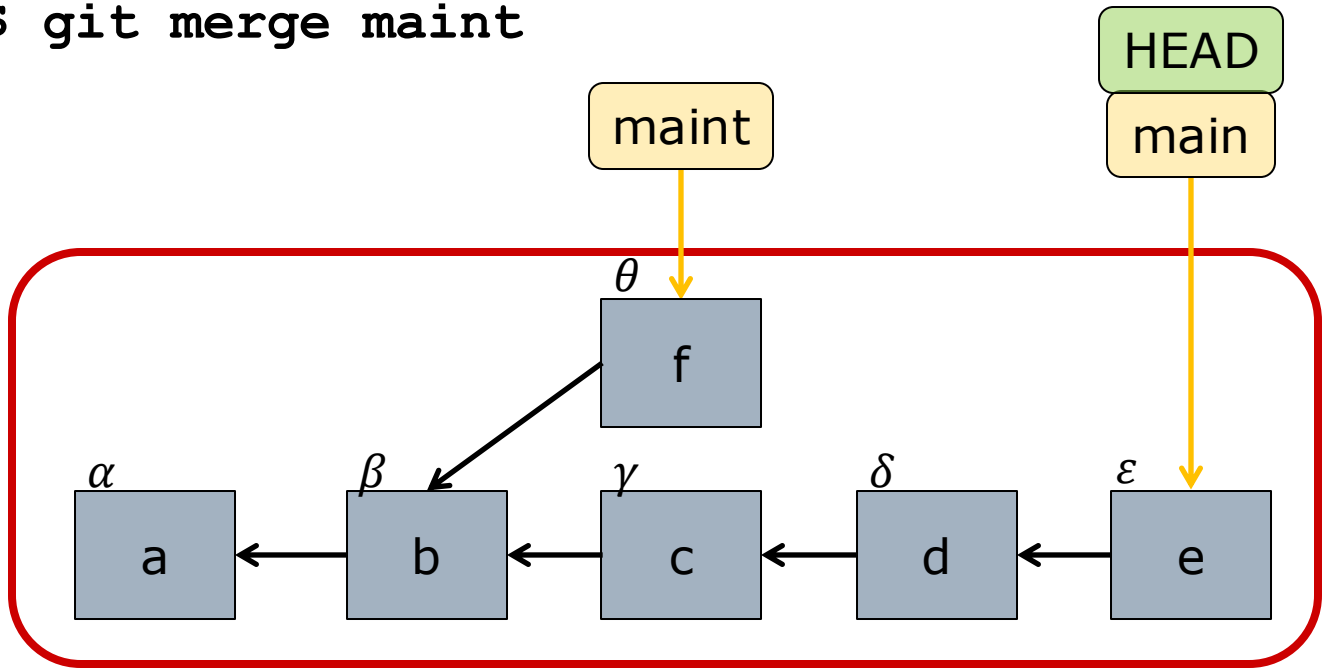
Merge Automatically Commits


```
$ git merge maint
```

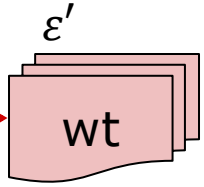


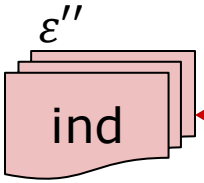
Merge – Case 3: Conflicts Exist


```
$ git merge maint
```



files with conflicts marked

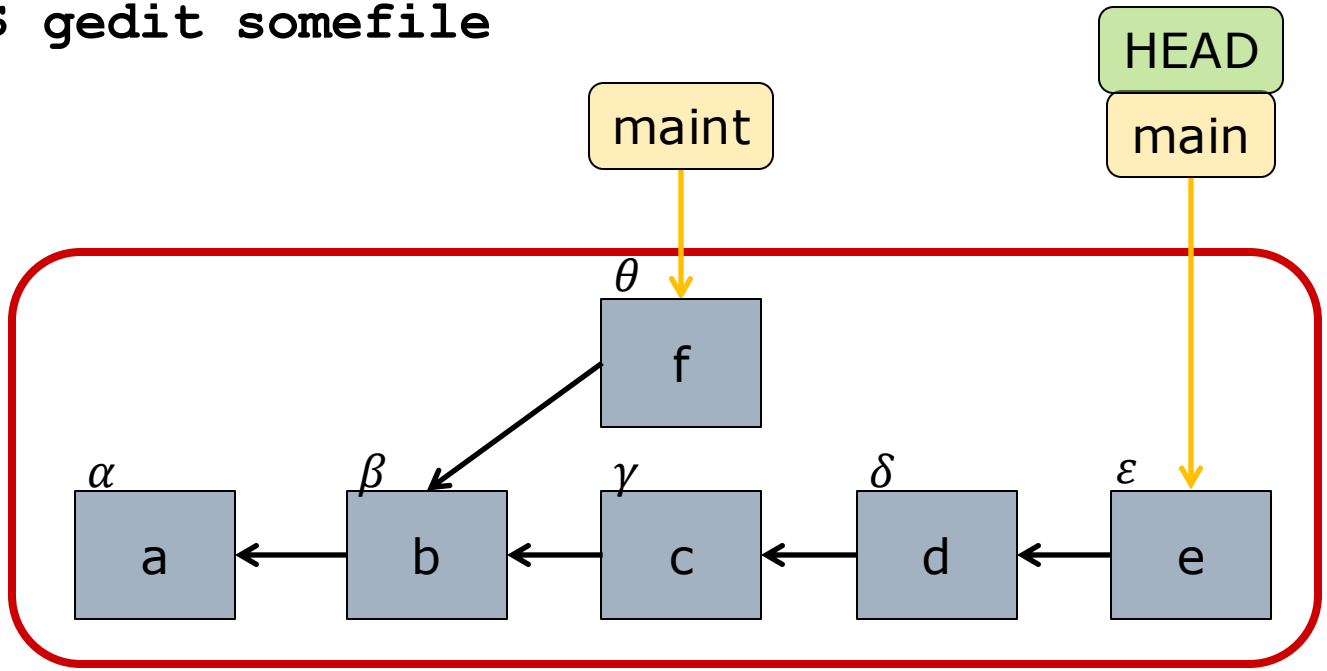




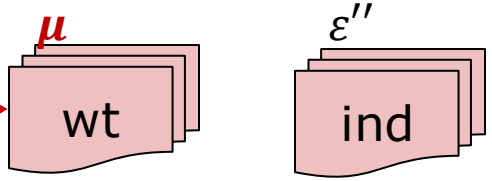
files that could be merged automatically

Merge: Resolve Conflicts


```
$ gedit somefile
```

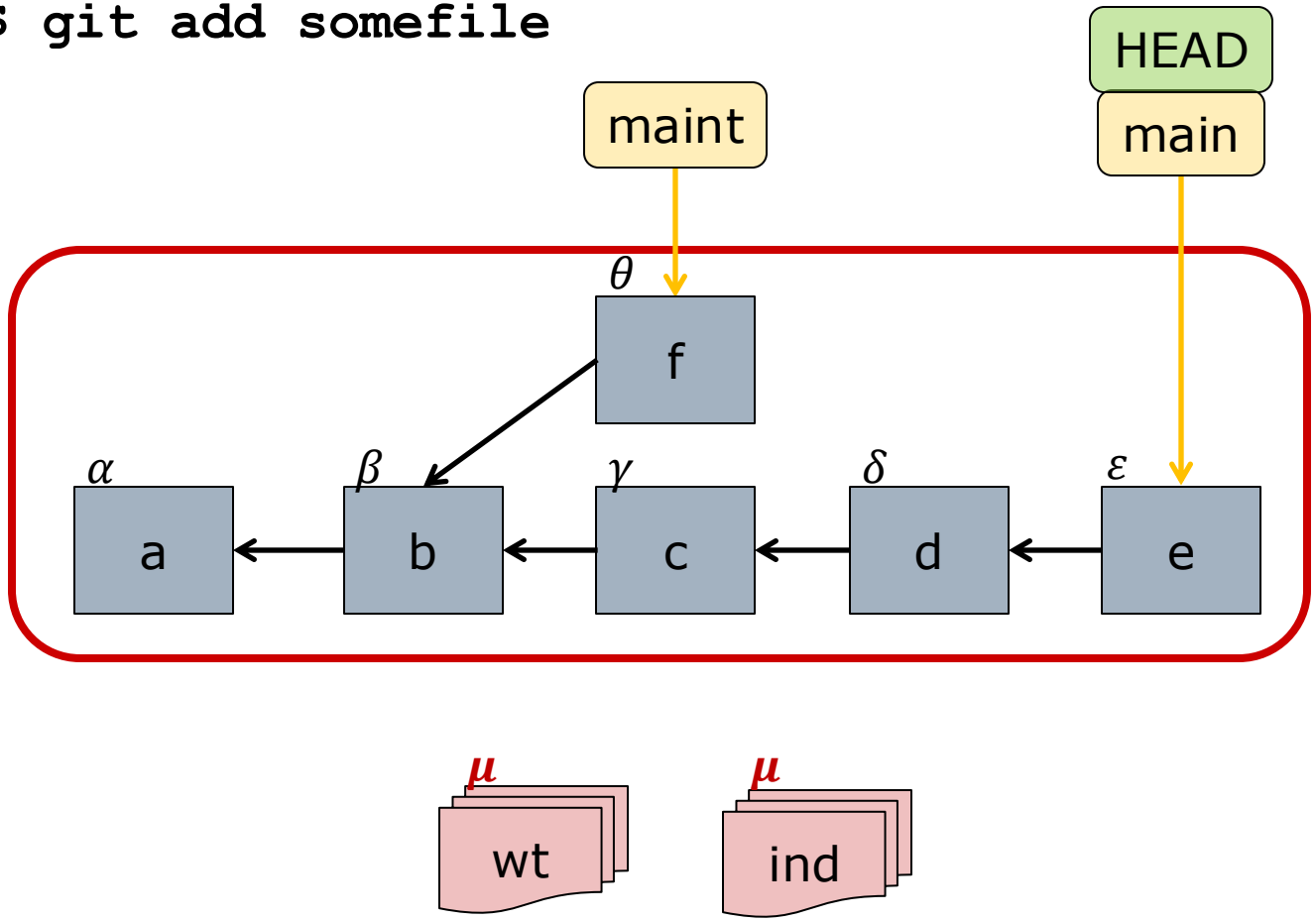


files with conflicts resolved



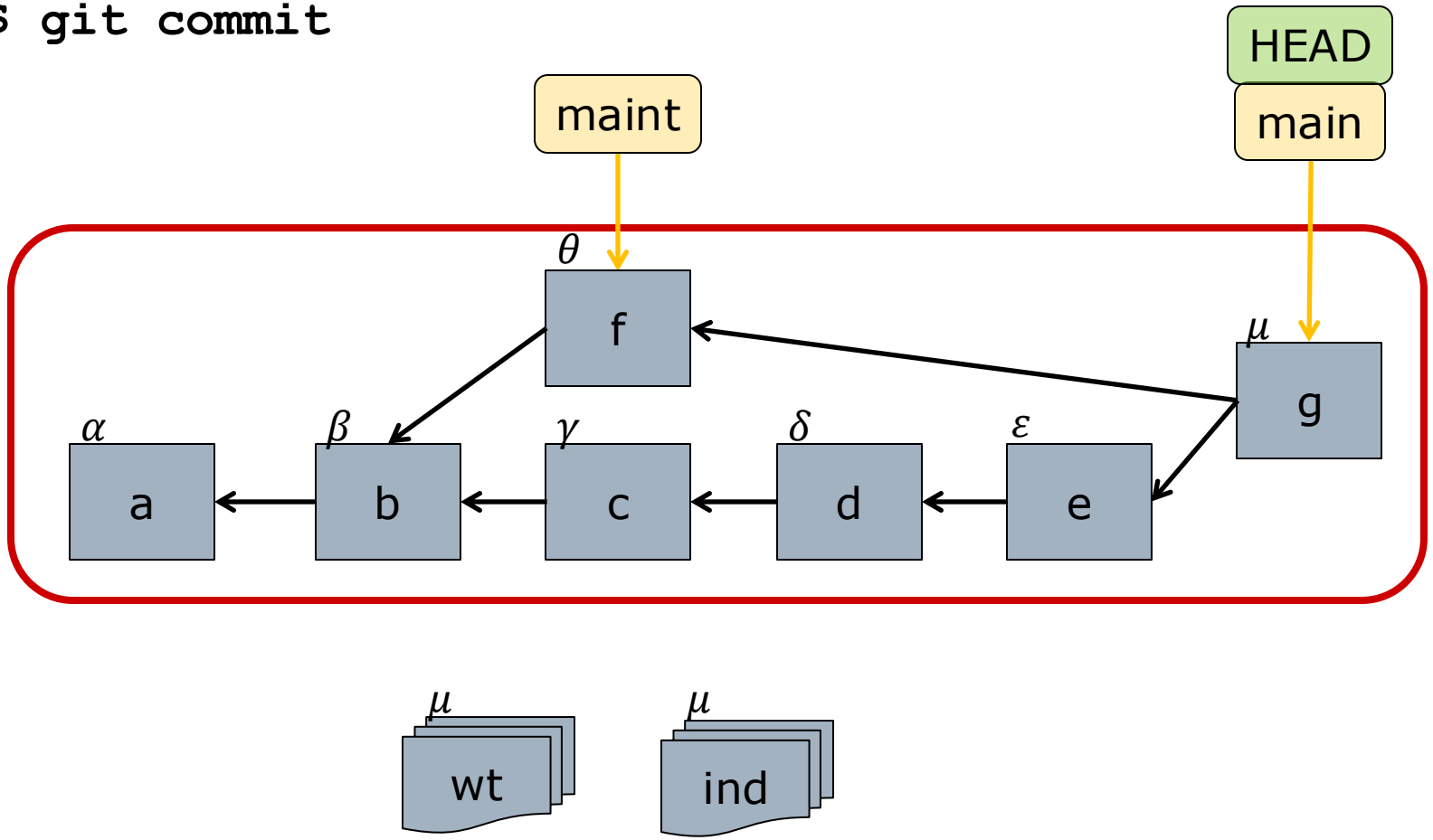
Merge with Conflicts: Add


```
$ git add somefile
```

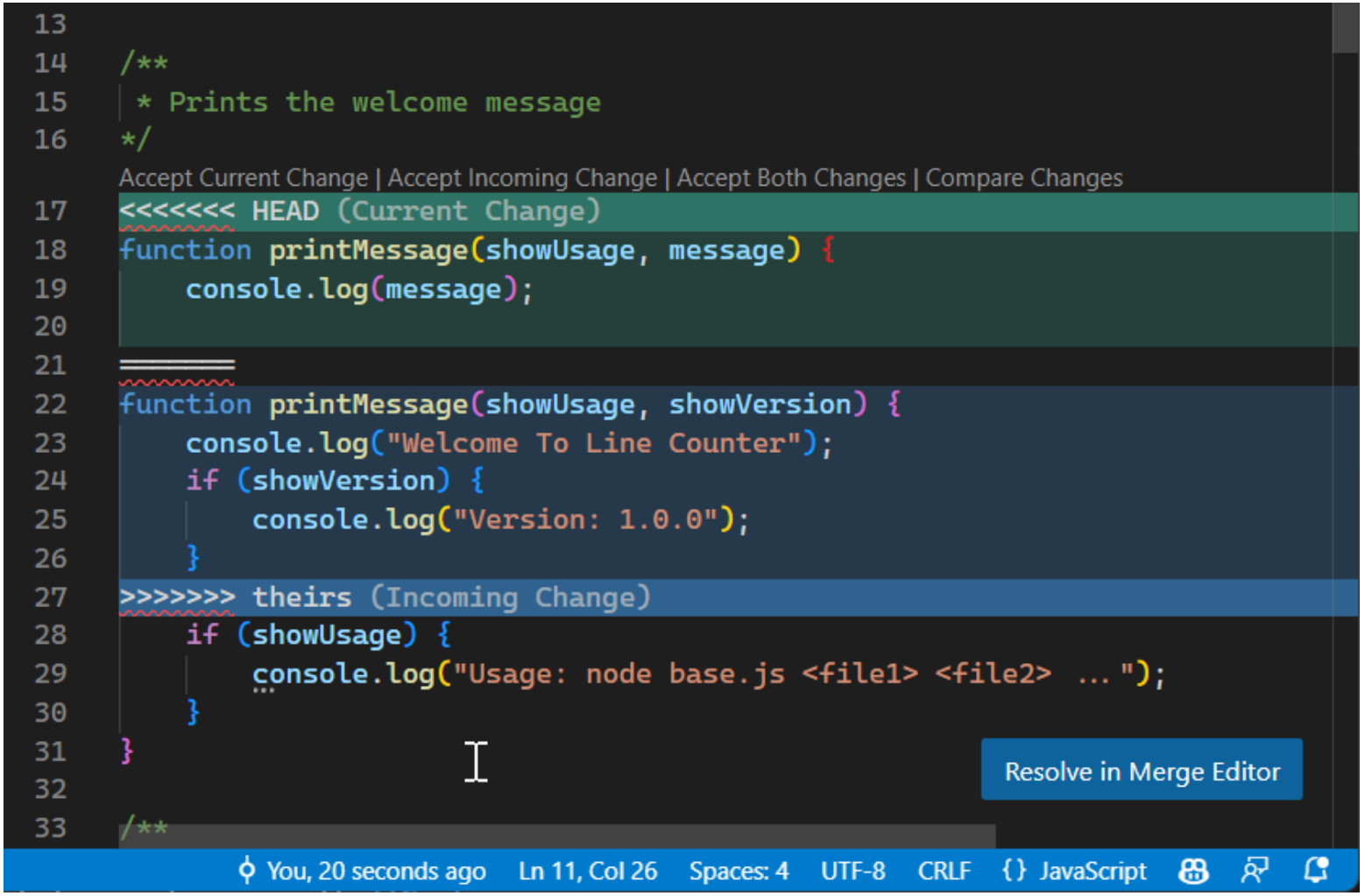


Merge with Conflicts: Commit


```
$ git commit
```

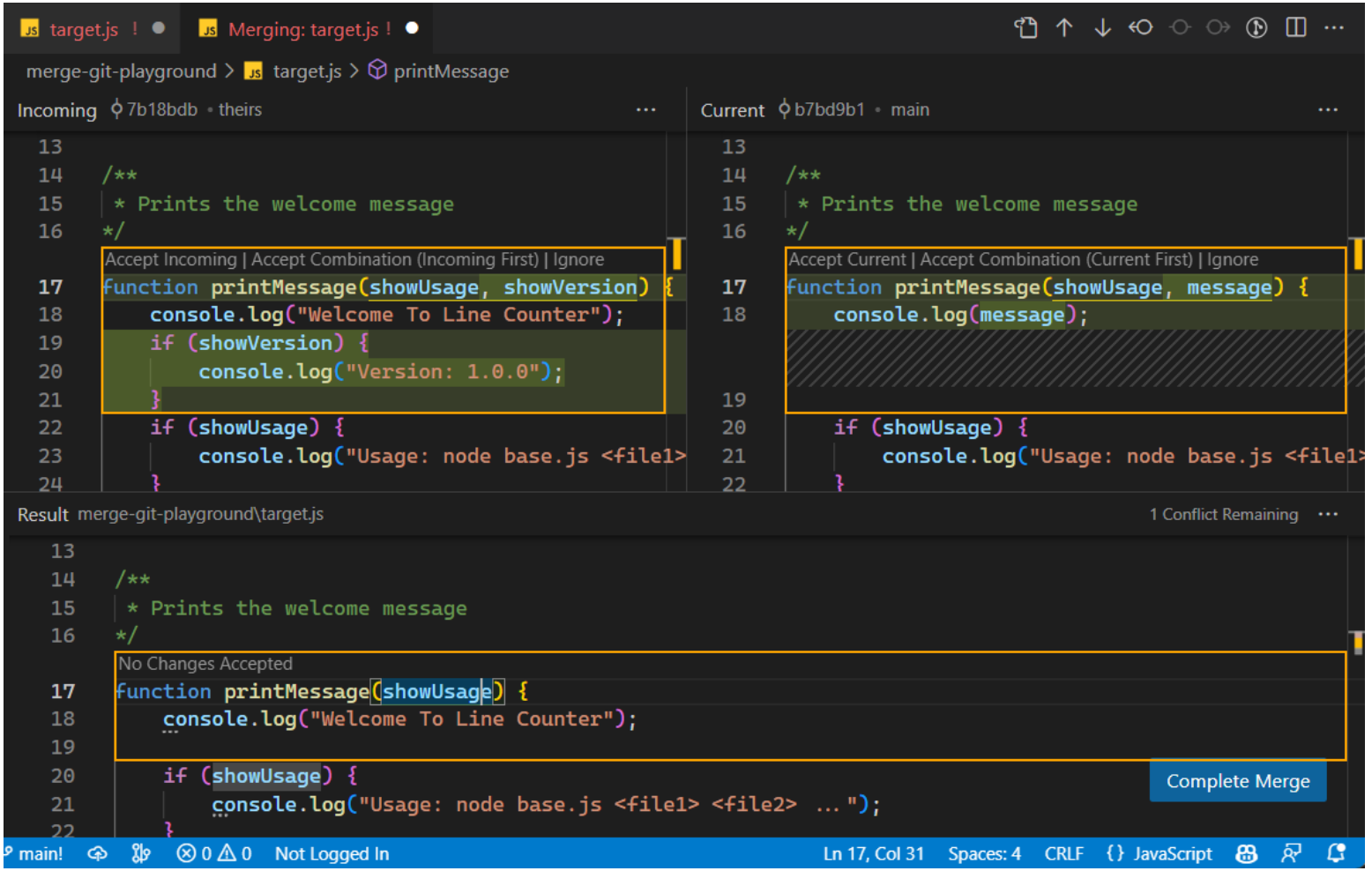


Merge: Edit to Resolve Conflicts



https://code.visualstudio.com/docs/sourcecontrol/overview

Merge: 3-way Merge Editor



https://code.visualstudio.com/docs/sourcecontrol/overview

Summary

- Repository = working tree + store
  - Store contains history
  - History is a DAG of commits
  - References, tags, and HEAD
- Commit/checkout are local operations
  - Former changes store, latter working tree
- Merge
  - Directional (merge other “into” HEAD)

Git: *Distributed* Version Control

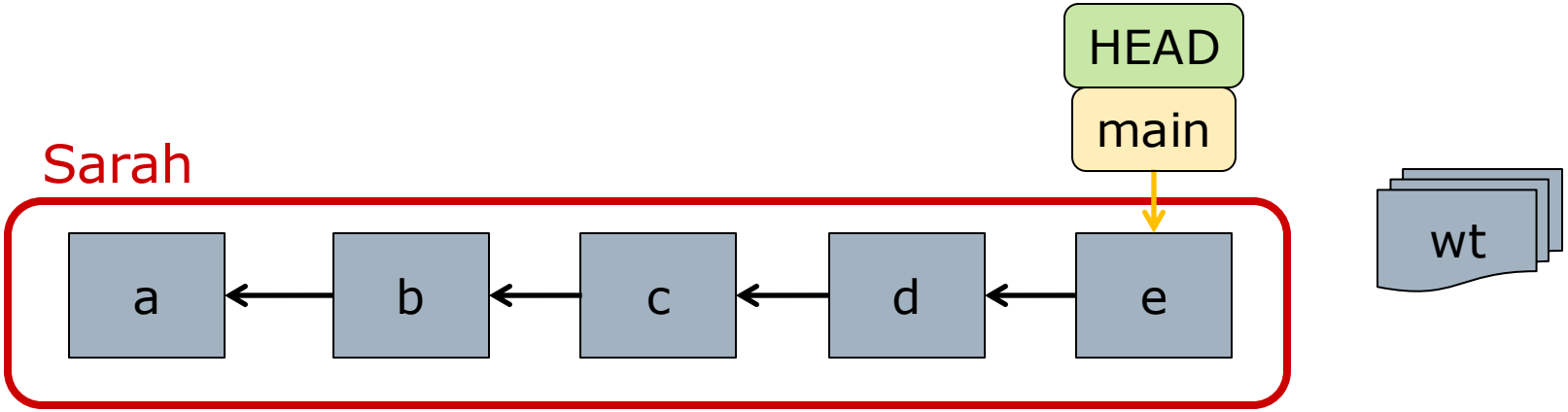
Demo

- Prep: Empty (but initialized) repo
- Linear development:
  - Create, edit, rename, ls -la files
  - Git: add, status, commit, log
- Checkout (time travel, detach HEAD)
- Branch (re-attach HEAD)
- More commits, see split in history
- Merge
  - No conflict
  - Fast-forward
- Play: git-school.github.io/visualizing-git

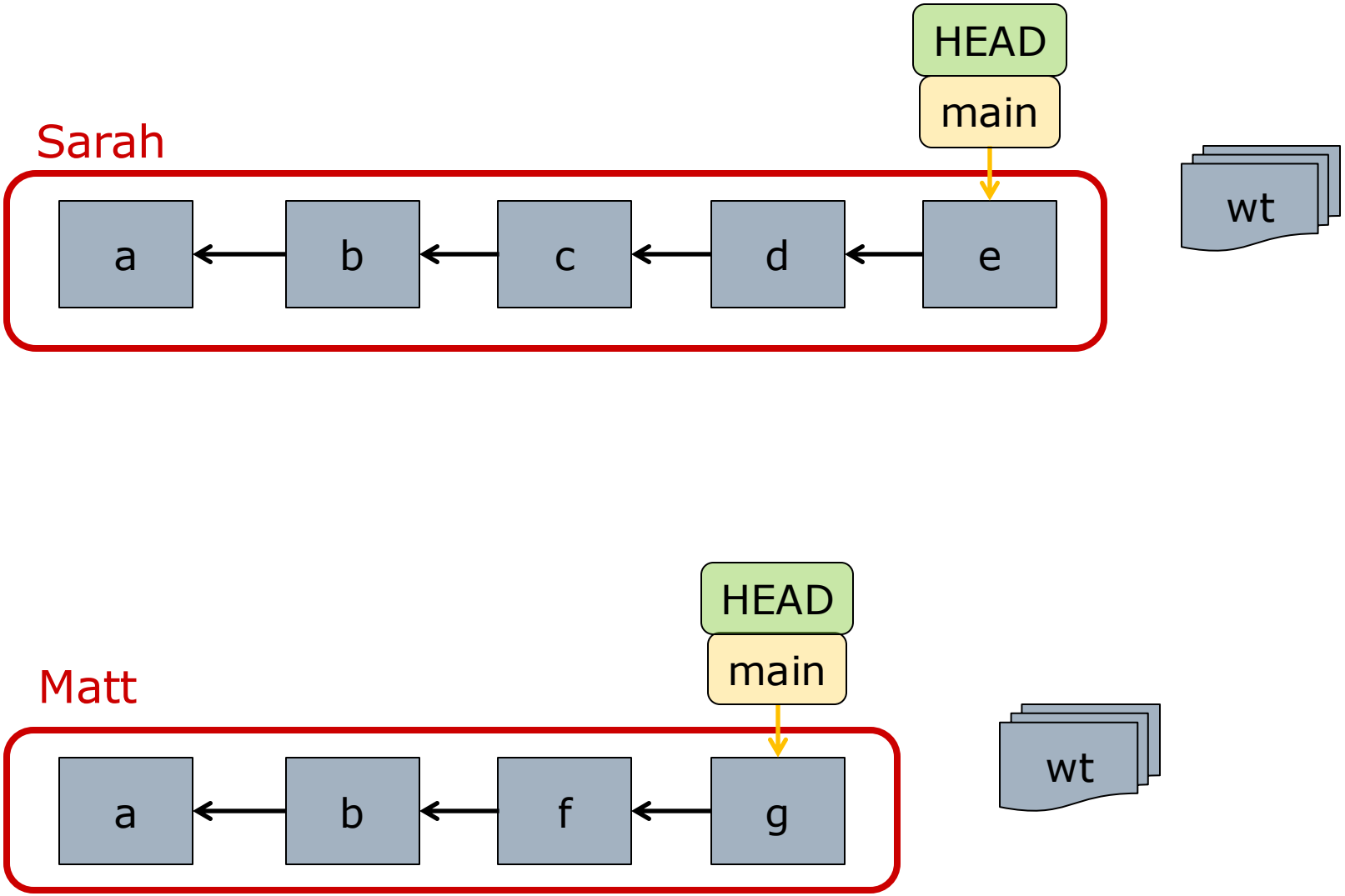
What Does "D" Stand For?

- *Distributed* version control
  - Multiple people, distributed across network
- Each person has their own repository!
  - Everyone has their own store (history)!
  - Big difference with older VCS (eg SVN)
- Units of data movement: changeset
  - Communication between teammates is to bring *stores* in sync
  - Basic operators: fetch and push

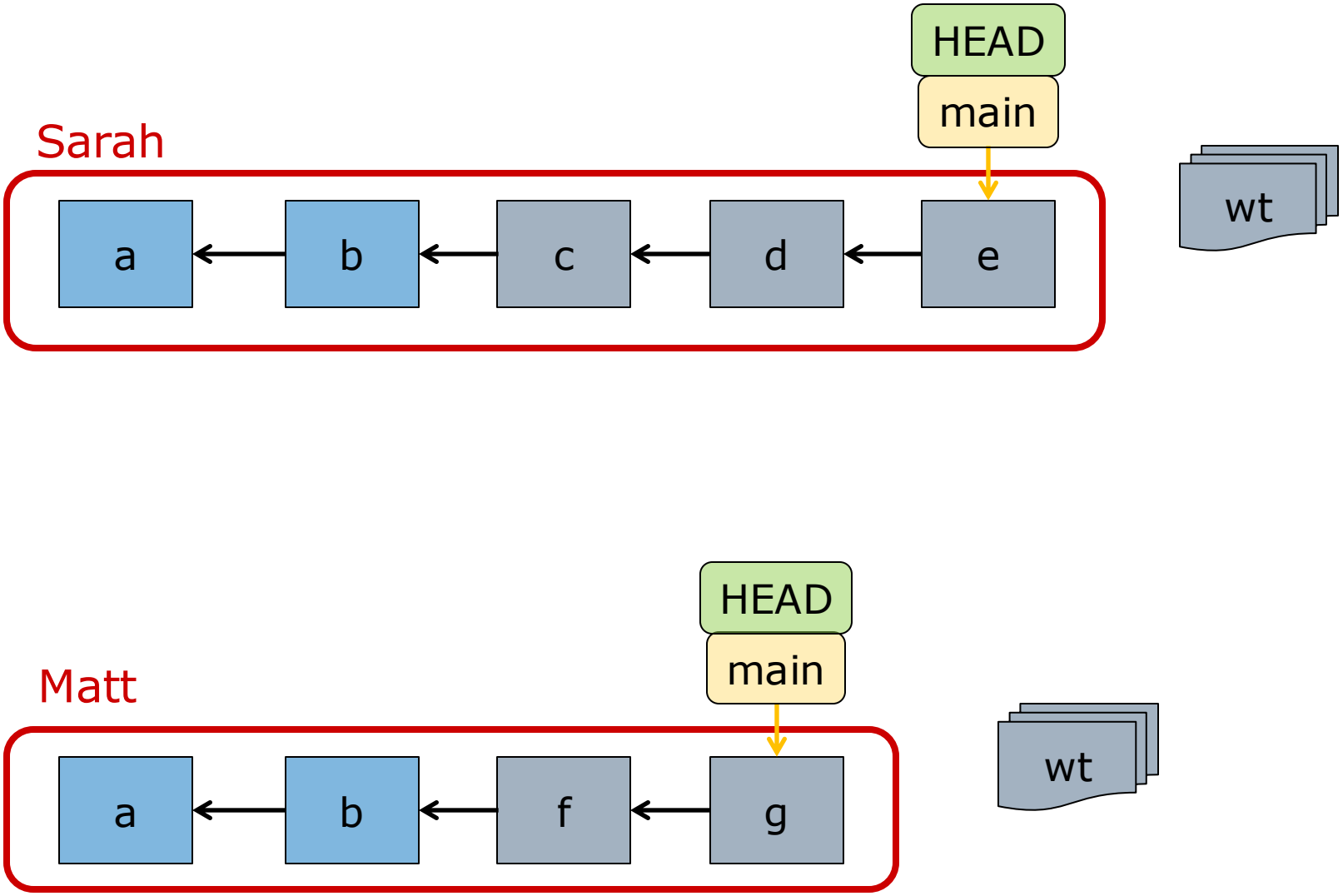
Sarah's Repository



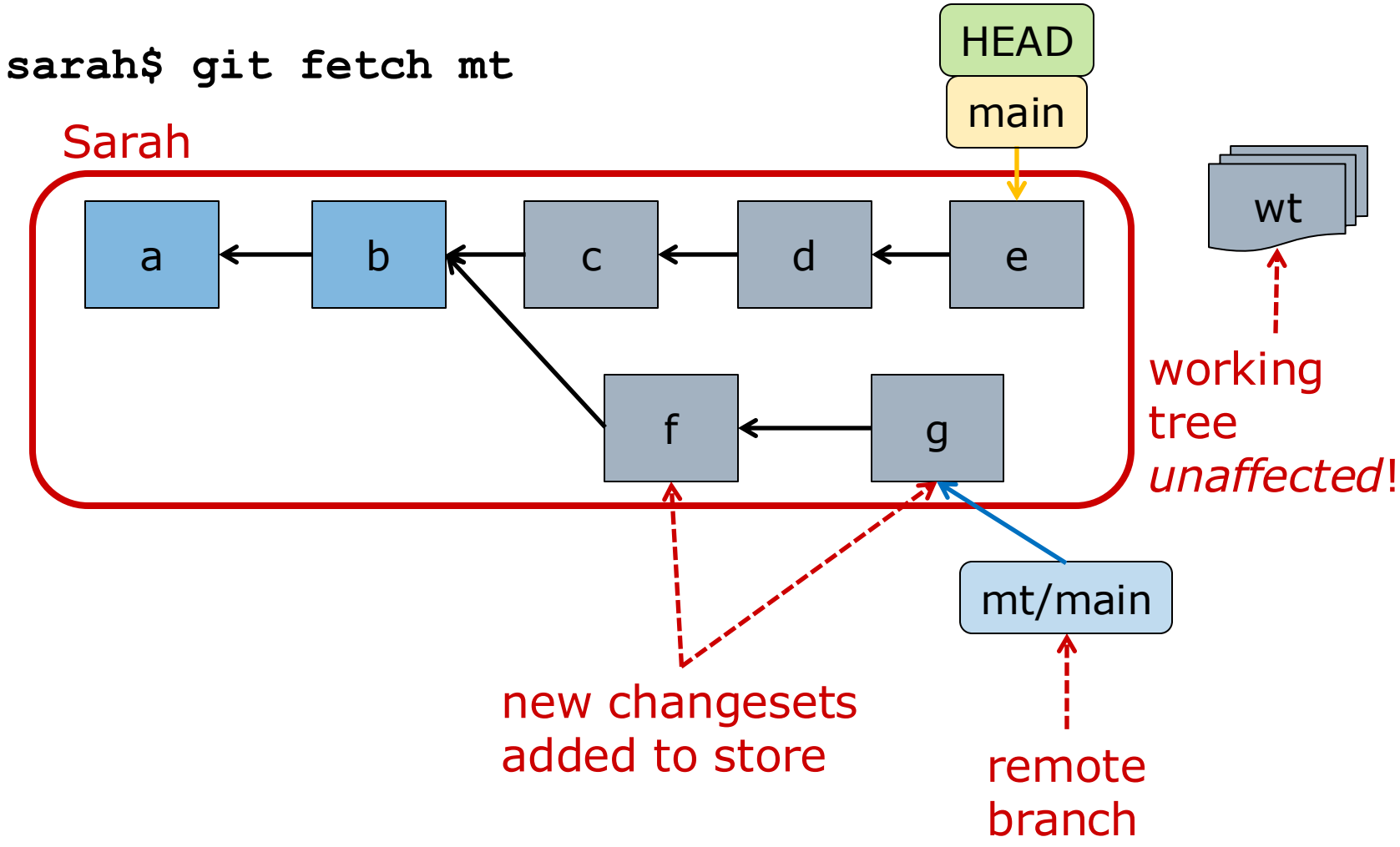
And Matt's Repository



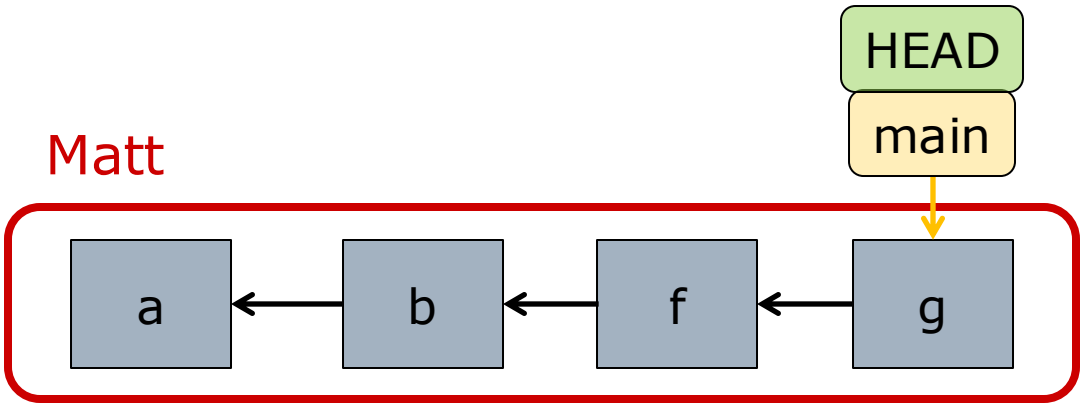
Some Shared History



Fetch: Remote Store → Local

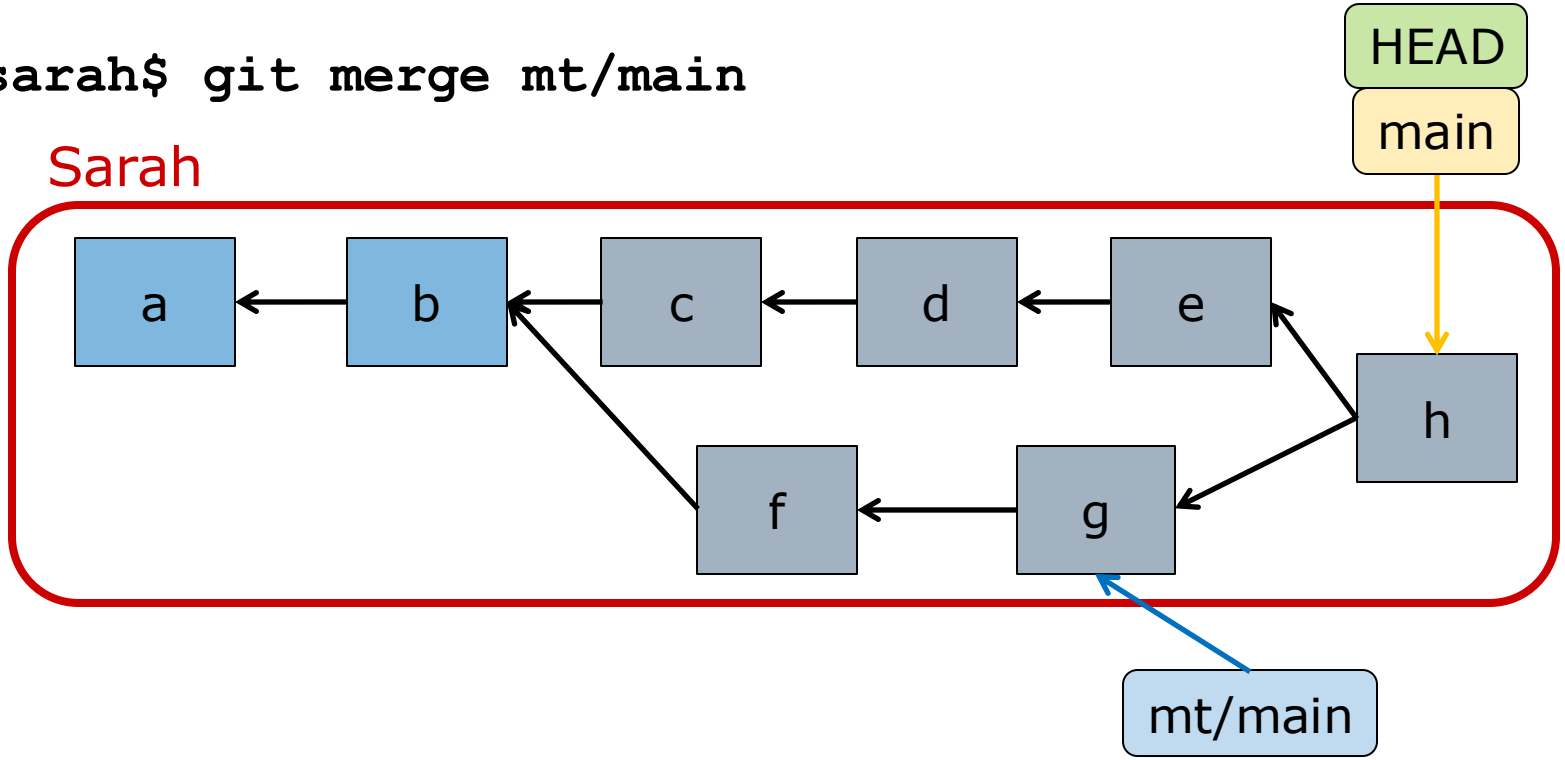


Remote Repository Unchanged

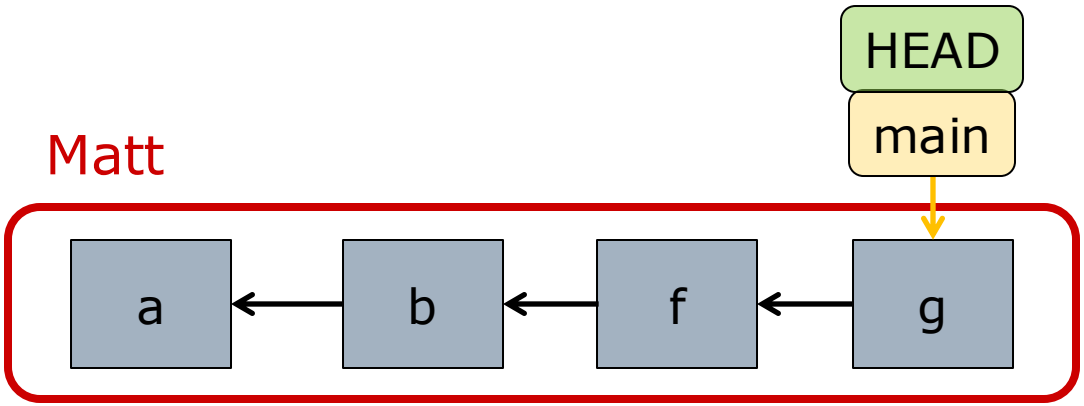


Workflow: Merge After Fetch


```
sarah$ git merge mt/main
```



Remote Repository Unchanged



View of DAG with All Branches

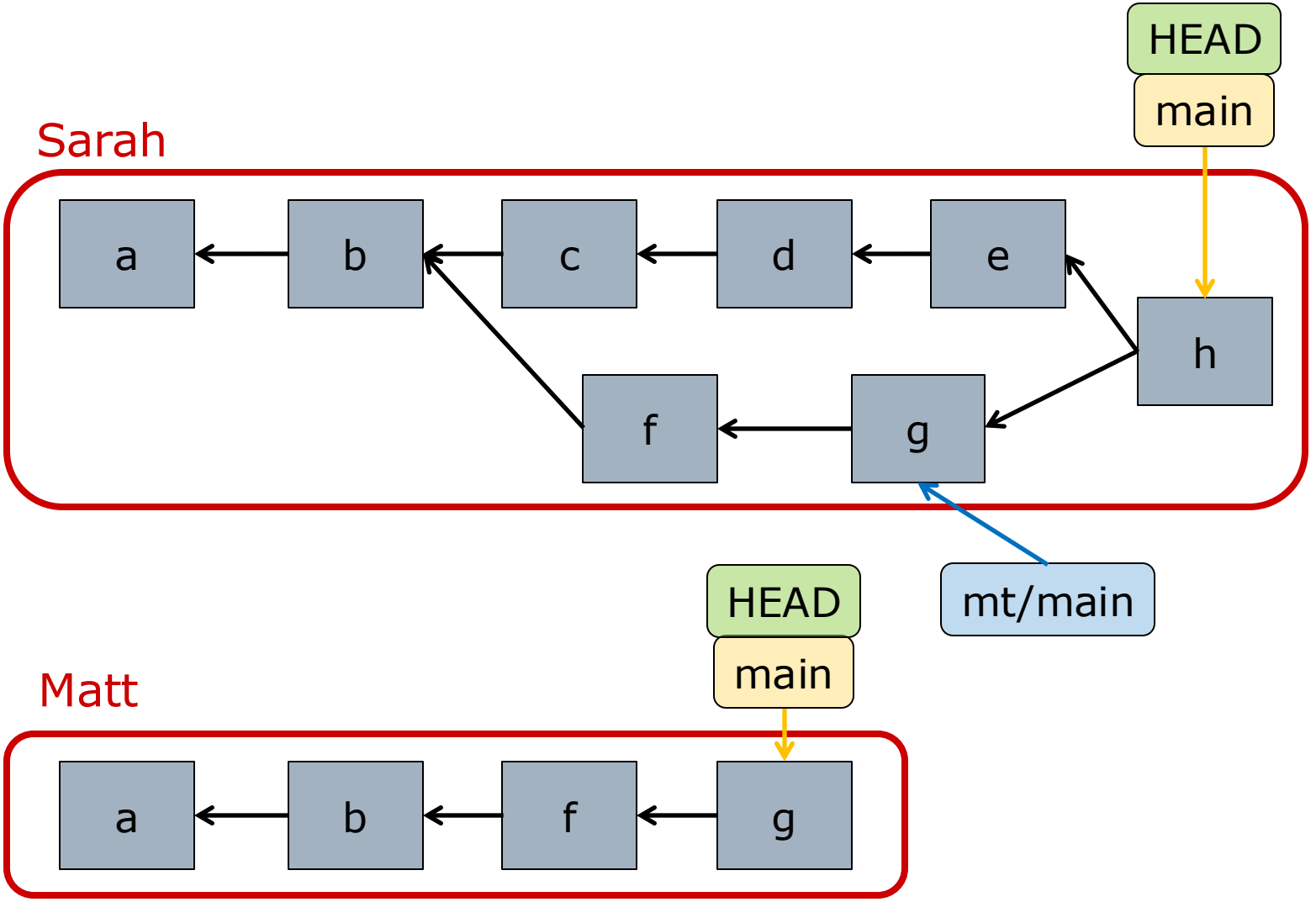

```
$ git log --oneline --graph # shows local & remote

* 1618849 (HEAD -> main, origin/main) clean up css
*   d579fa2 (alert) merge in improvements from master
|\
| * 0f10869 replace image-url helper in css
* | b595b10 (origin/alert) add buckeye alert notes
* | a6e8eb3 add raw buckeye alert download
|/
* b4e201c wrap osu layout around content
* e9d3686 add Rakefile and refactor schedule loop
* 515aaa3 create README.md
* eb26605 initial commit
```

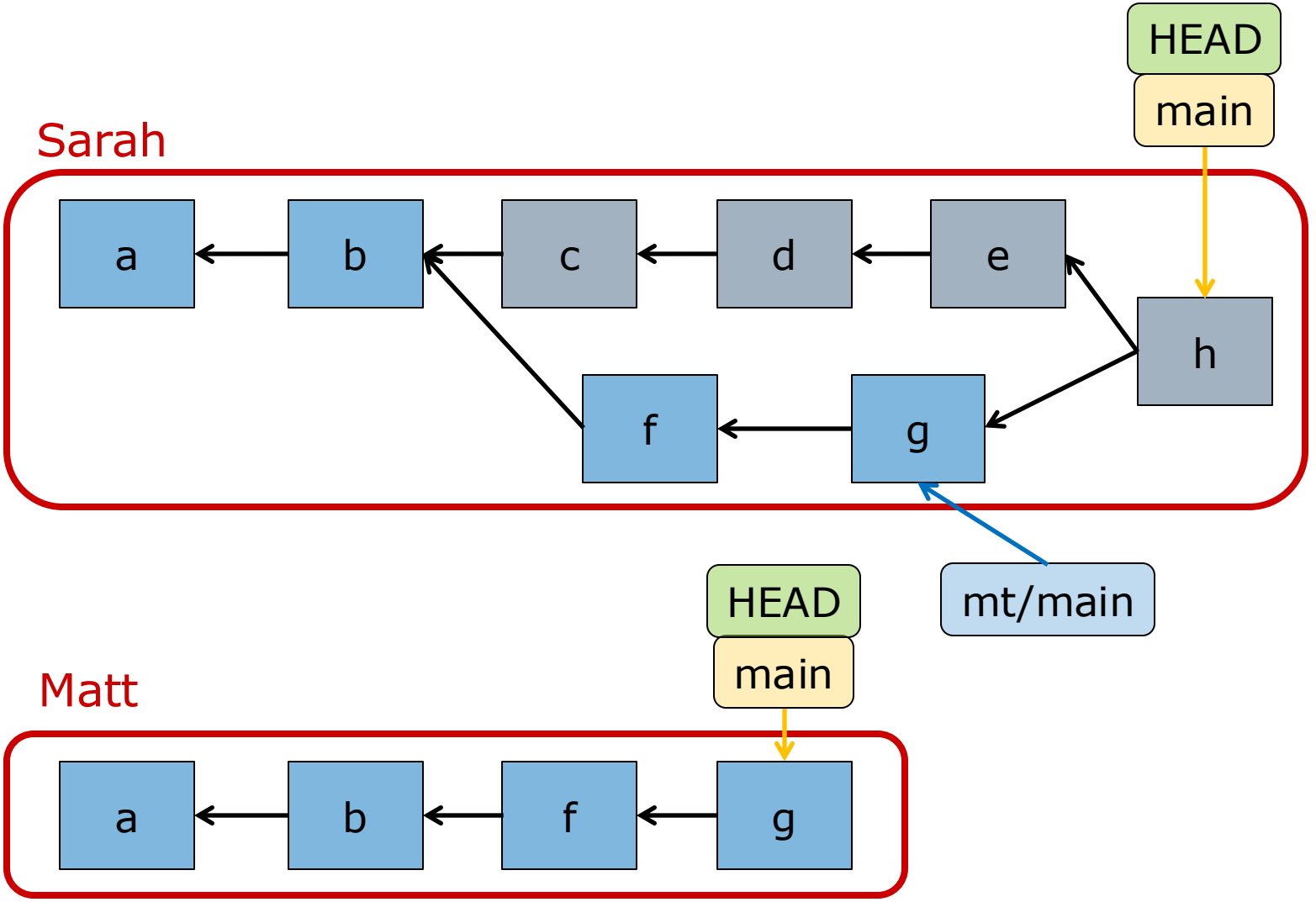
Your Turn

- Show the state of Matt's repository after each of the following steps
  - Fetch (from Sarah)
  - Merge

Sarah and Matt's Repositories



Some Shared History

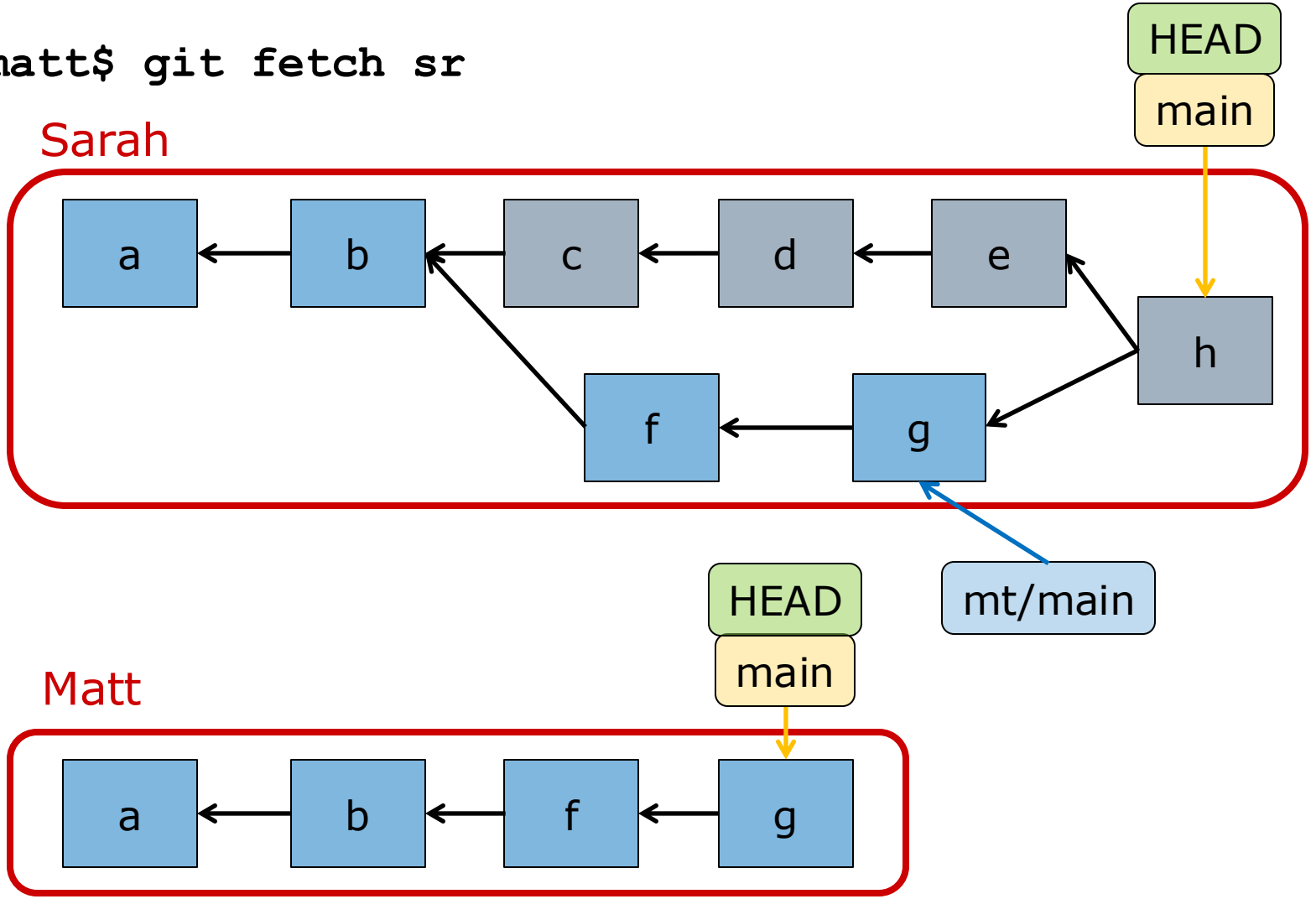


Your Turn: Fetch


```
matt$ git fetch sr
```

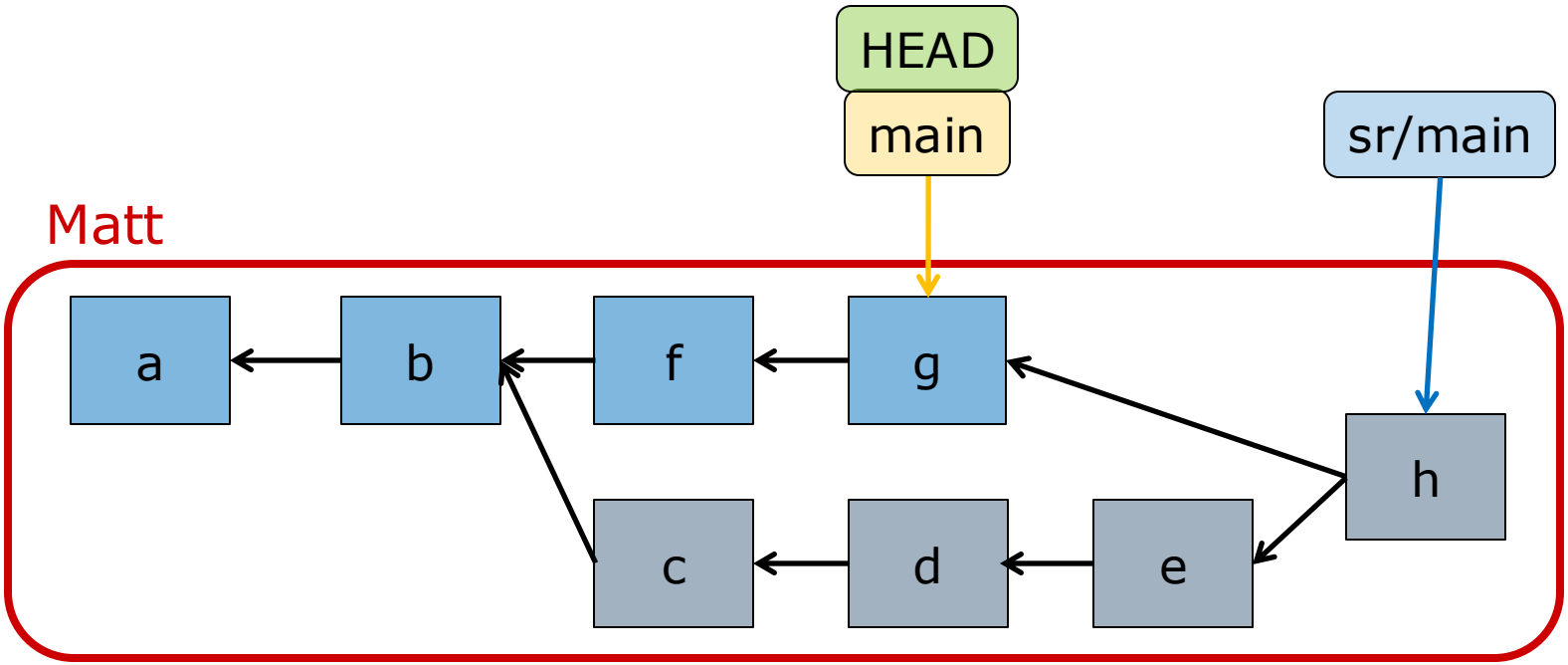
Your Turn: Fetch


```
matt$ git fetch sr
```



Fetch: After


```
matt$ git fetch sr
```

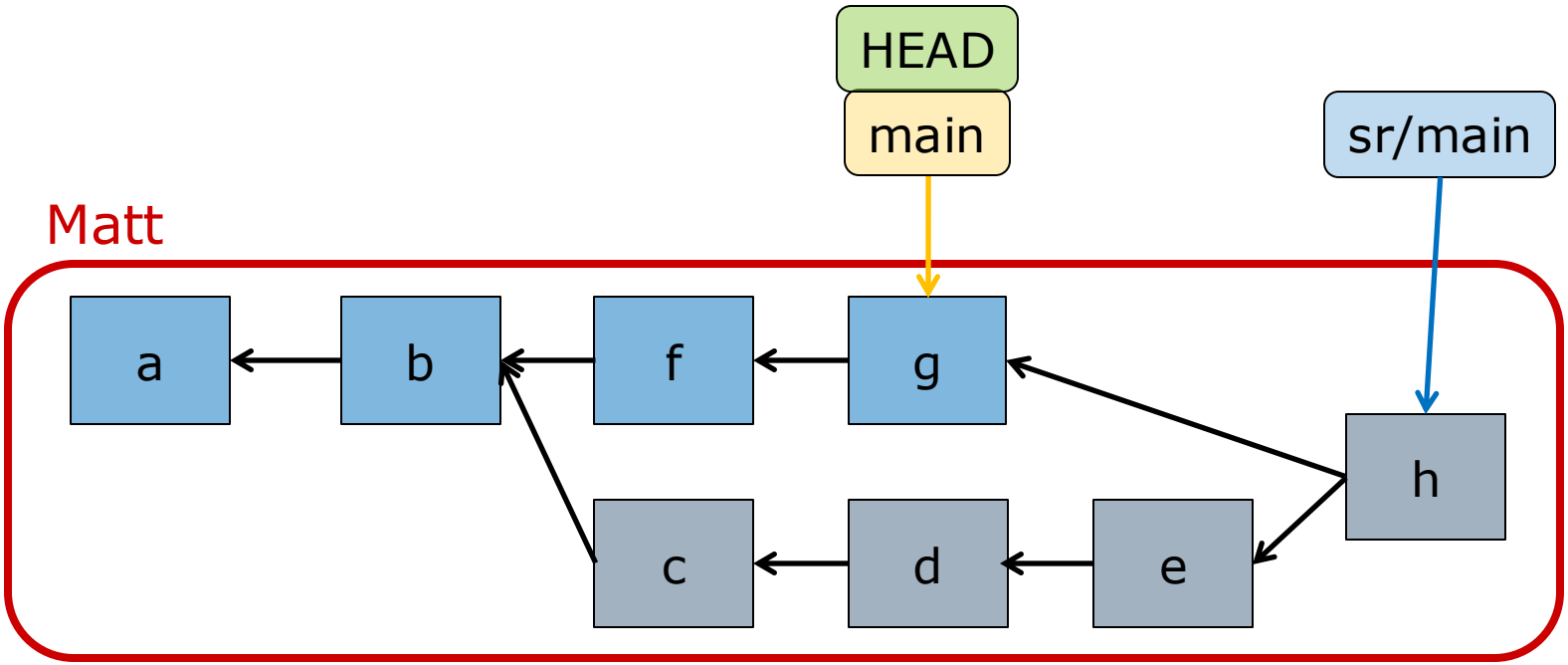


Your Turn: Merge


```
matt$ git merge sr/main
```

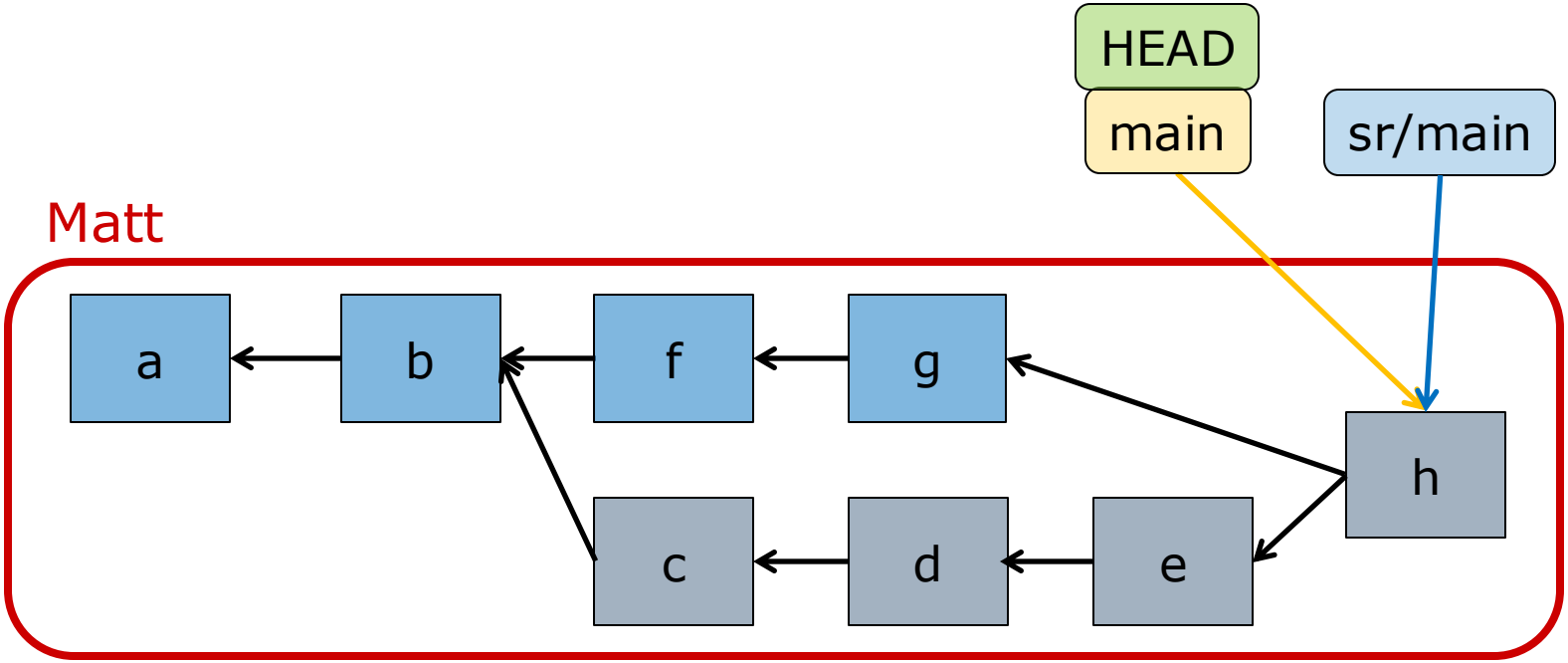
Your Turn: Merge


```
matt$ git merge sr/main
```



Merge: After


```
matt$ git merge sr/main
```



Demo

- https://git-school.github.io/visualizing-git/#upstream-changes
- Try:

```
git commit
git fetch origin # see origin/feature
git merge origin/feature # see feature
```

Pull: Fetch then Merge

- A *pull* combines both fetch & merge

```
matt$ git pull sr
```

- Advice: Prefer explicit fetch, merge
  - After fetch, examine new work

```
$ git log       # see commit messages
$ git checkout  # see work
$ git diff      # compare
```

  - Then merge
  - Easier to adopt more complex workflows (*e.g.*, rebasing instead of merging)

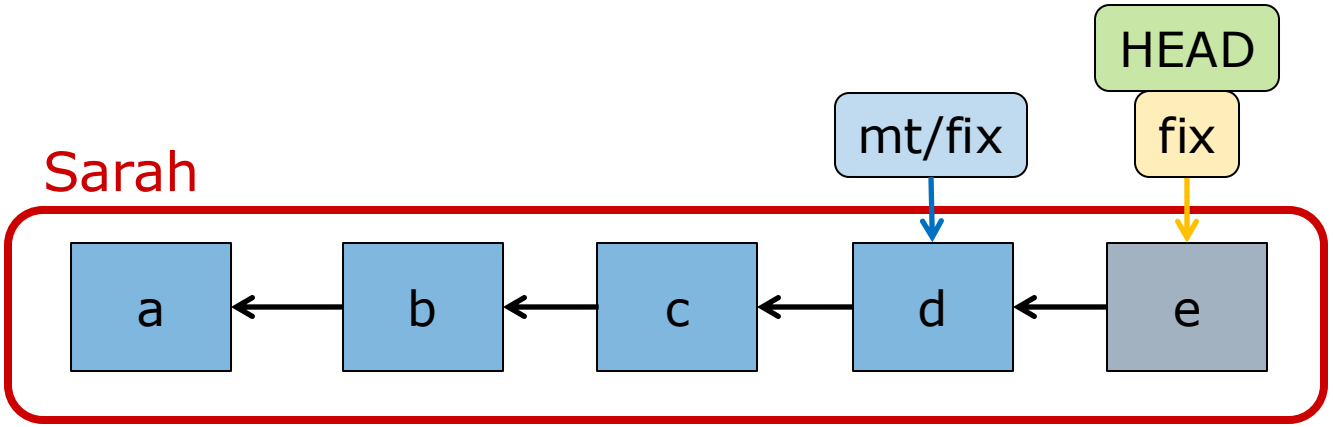
Push: Local Store → Remote

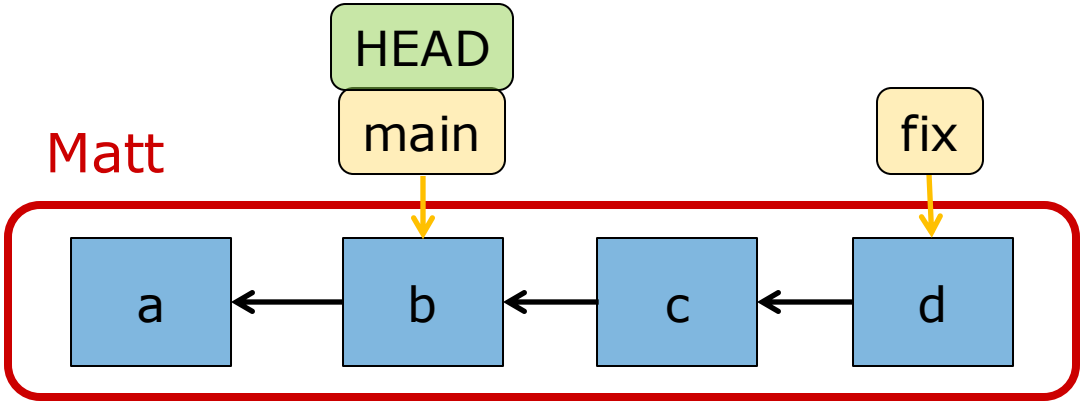
- Push sends local commits to remote store
- Usually push one branch (at a time)

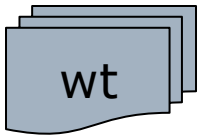
```
sarah$ git push mt fix
```

  - Advances Matt's fix branch
  - Advances Sarah's mt/fix remote branch
- Requires:
  1. Matt's fix branch *must not* be his HEAD
  2. Matt's fix branch *must be* ancestor of Sarah's
- Common practices:
  1. Only push to *bare* repositories (bare means no working tree, ie no HEAD)
  2. Get remote store's branch into local DAG (ie fetch, merge, commit) *before* pushing

Remote's Branch is Ancestor

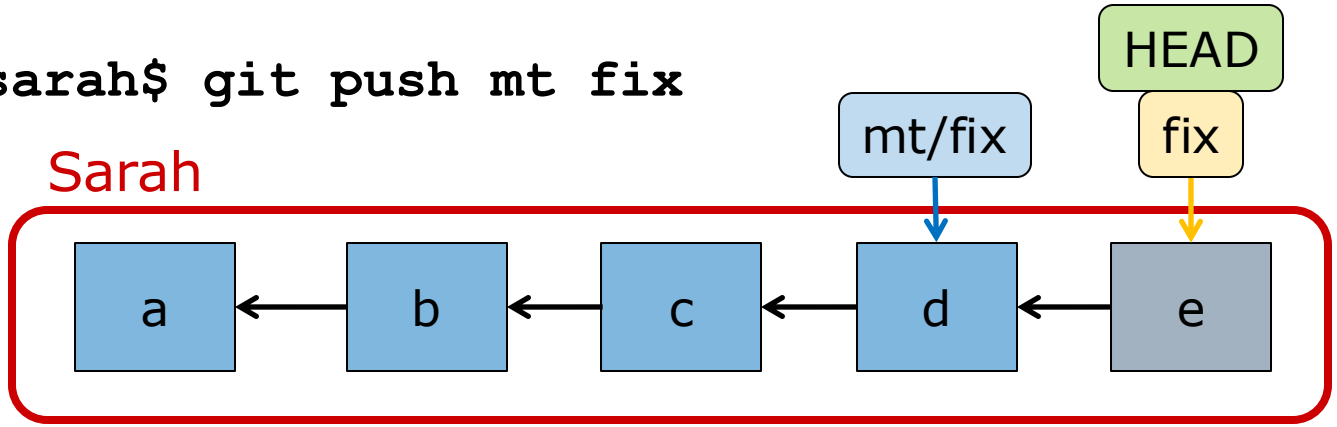


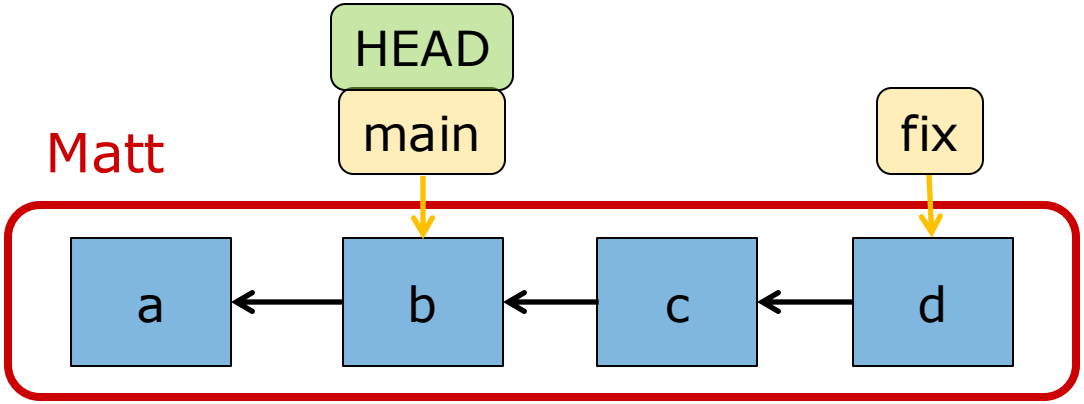


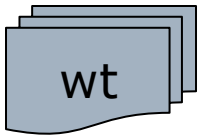


Push: Local Store → Remote


```
sarah$ git push mt fix
```

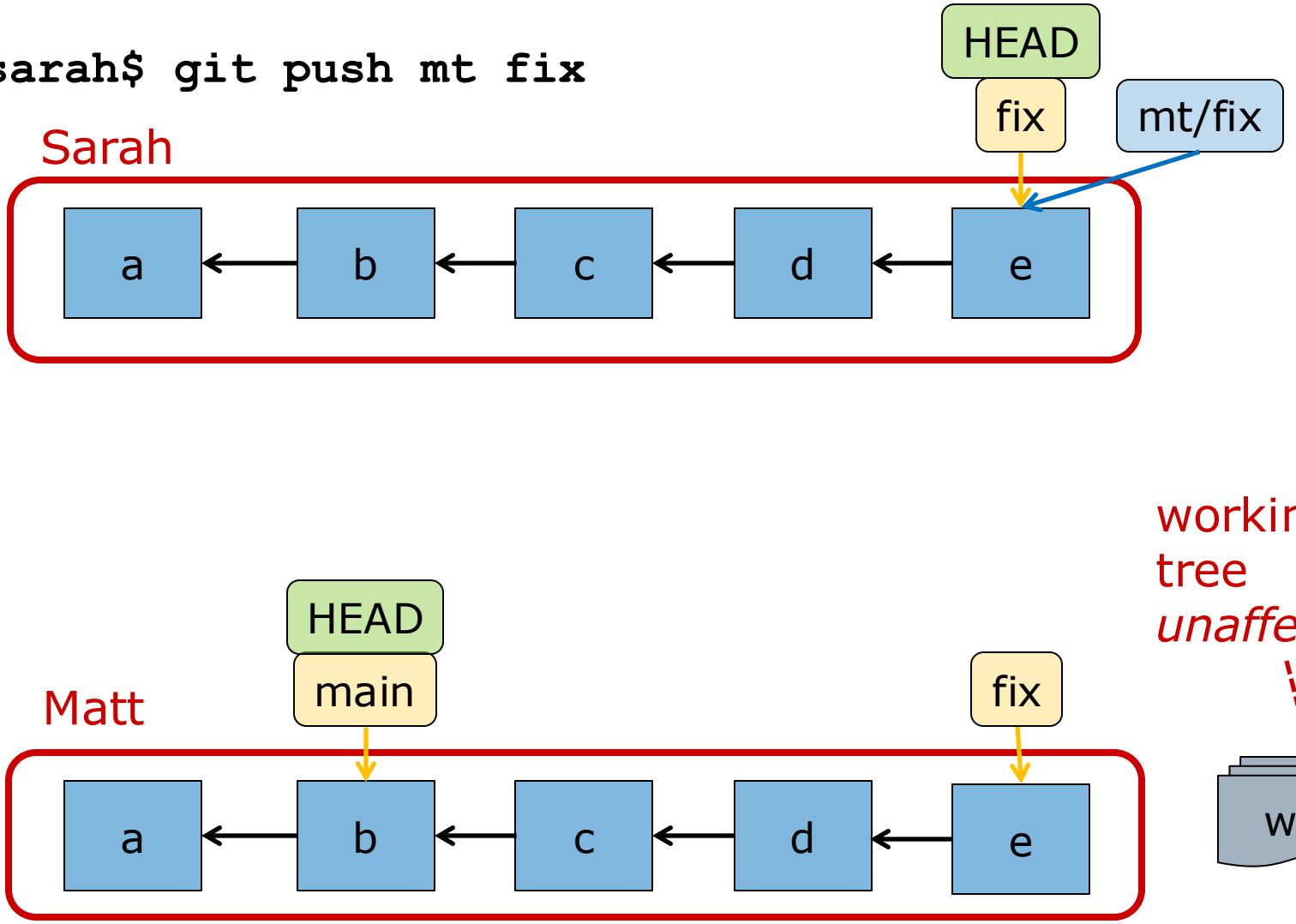




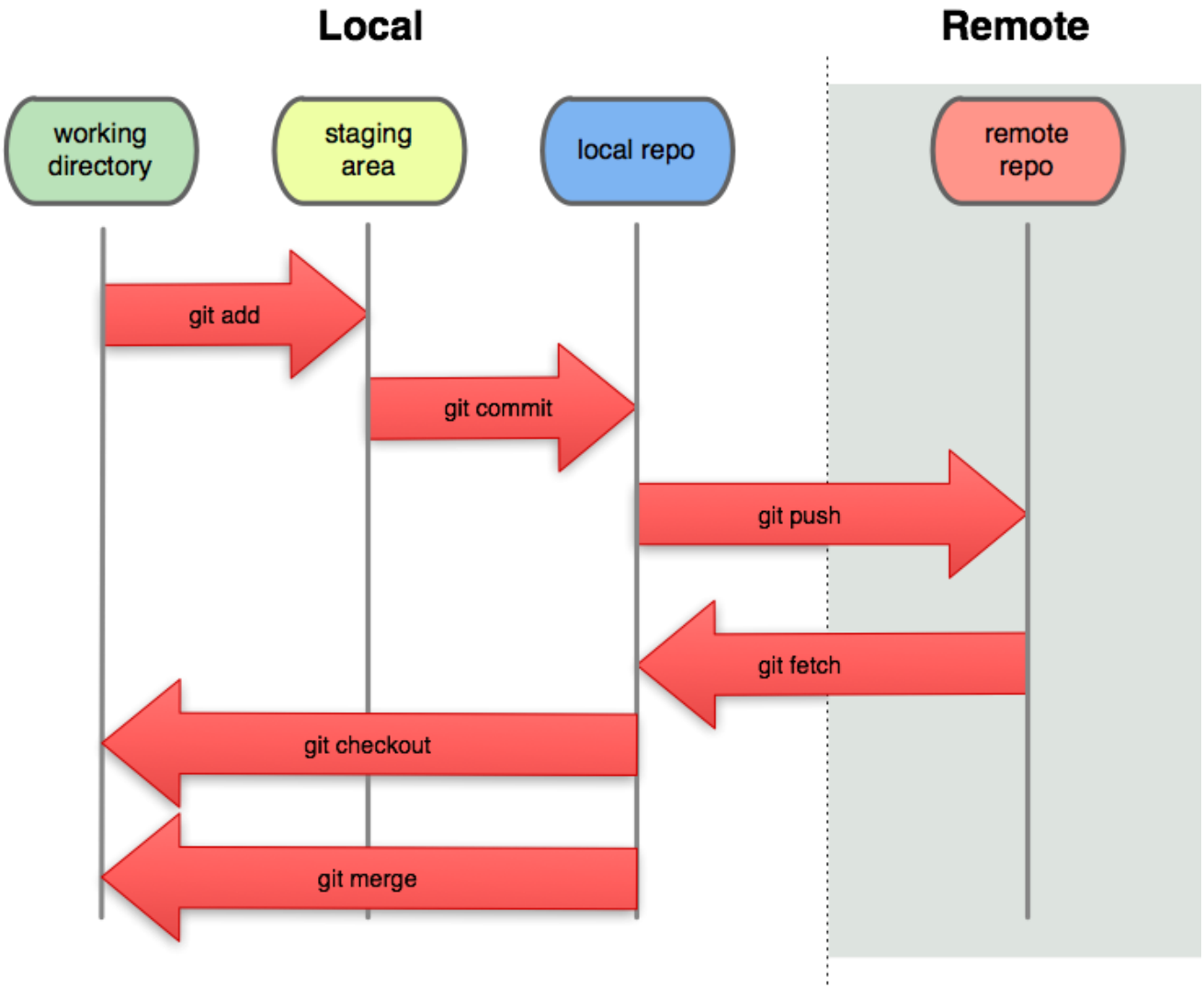


Push: After


```
sarah$ git push mt fix
```



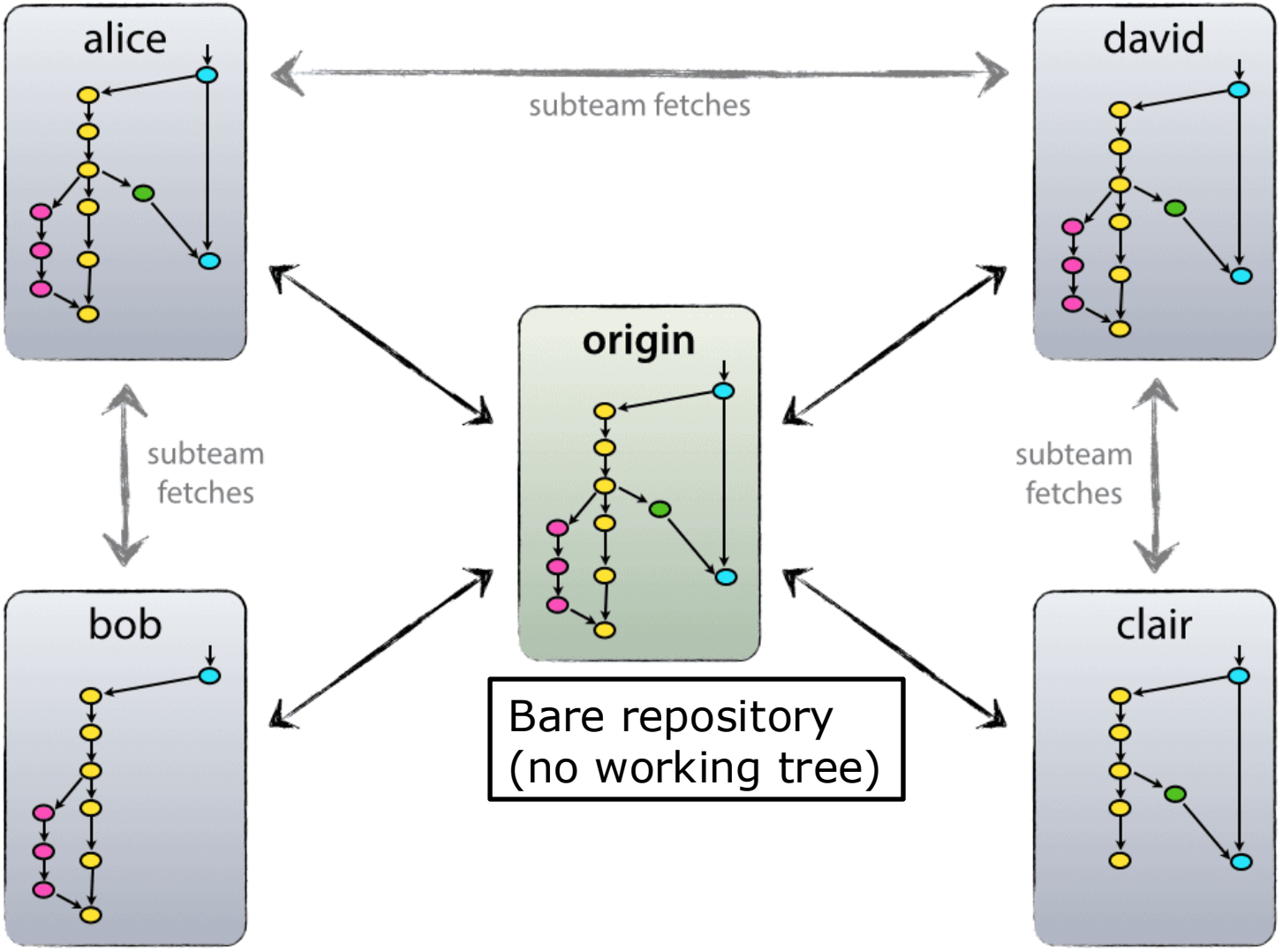
Commit/Checkout vs Push/Fetch



Common Topology: Star

- *n*-person team has *n+1* repositories
  - 1 shared central repository (bare!)
  - 1 local repository / developer
- Each developer *clones* central repository
  - Creates (local) copy of (entire) central repo
  - Local repo has a remote called "origin"
  - Default source/destination for fetch/push
- Variations for central repository:
  - Everyone can read and write (ie push)
  - Everyone can read, but only 1 person can write (responsible for pulling and merging)

Common Topology: Star



Source: http://nvie.com/posts/a-successful-git-branching-model/

Summary

- Push/fetch to share your store with remote repositories
  - Neither working tree is affected
- Branches in history are easy to form
  - Committing when HEAD is not a leaf
  - Fetching work based on earlier commit
- Team coordination
  - One single, central repo
  - Every developer pushes/fetches from their (local) repo to this central (remote) repo

Project Groups: To Do

1. Find your group on Carmen (People)
2. Exchange contact information
   - Phone, discord
   - Schedules
3. Choose a group name

Git:
Advanced Topics

Basic Workflow: Overview

1. Configure git locally (everyone)
2. Create central repo (1 person)
3. Create local repo (everyone)
4. Local development (everyone):
   - Commit locally
   - Fetch/merge as appropriate
   - Push to share

Step 1: Configure Git Locally

- Each team member, in their own VM
  - Req'd: Set identity for authoring commits

```
$ git config --global user.name "Brutus Buckeye"
$ git config --global user.email bb@osu.edu
```

  - Rec'd: set default initial branch name (2.28+)

```
$ git config --global init.defaultBranch main
```

  - Tips
    - Add email to GitHub account (Settings > Email)
    - Alternative: use GitHub-generated fake address:
      - Settings > Email > Keep my address private
      - Find *ID+USERNAME*`@users.noreply.github.com`
    - Add your SSH key to your GitHub account

Step 2: Initialize Central Rep

- One person, once per project
- Hosting services (GitHub, GitLab, BitBucket…) use a web interface for this step
- Alternative: a location that the group has access to (*e.g.* stdlinux):
  - Create central repository in group's project directory (/project/c3901aa03)

```
$ cd /project/c3901aa03
$ mkdir proj1  # an ordinary directory
```

  - Initialize this directory as a *bare* git repository, with group permissions

```
$ git init --bare --shared proj1
```

Step 3: Create Local Repository

- Each team member, once, in their VM
  - Create local repo by *cloning* the central one

```
$ git clone git@github.com:bb/proj1.git
```

  - Copies entire repo, including store, and sets a remote called "origin"

```
$ cd proj1
proj1$ git remote -v # display info
   origin git@github.com:bb/proj1.git (fetch)
   origin git@github.com:bb/proj1.git (push)
```

- Different ways to clone
  - SSH: Add your SSH key to the remote host, then it is easy to fetch/push
  - Git Credential Manager

Step 4: Local Development

- Each team member repeats:
  - Edit and commit (to local repository) often

```
$ git status/add/rm/commit
```

  - Pull others' work when you can benefit

```
$ git fetch origin # bring in changes
$ git log/checkout # examine new work
$ git merge, commit # merge work
```

  - Push to central repository when confident

```
$ git push origin main # share
```

Demo

- https://git-school.github.io/visualizing-git/#upstream-changes
- Try:

```
git commit
git fetch origin # see origin/feature
git merge origin/feature # see feature
git push origin feature # see remote
```

Your Turn: Playing with Git

- Navigate to class org on GH and find the repo called *first-commits*
- Clone the repo to your VM
- Do some development!
  - Edit
  - Inspect the store's DAG

    ```
    $ git log --graph --oneline --all
    ```
  - Commit, fetch, merge, push…
  - Rinse, repeat

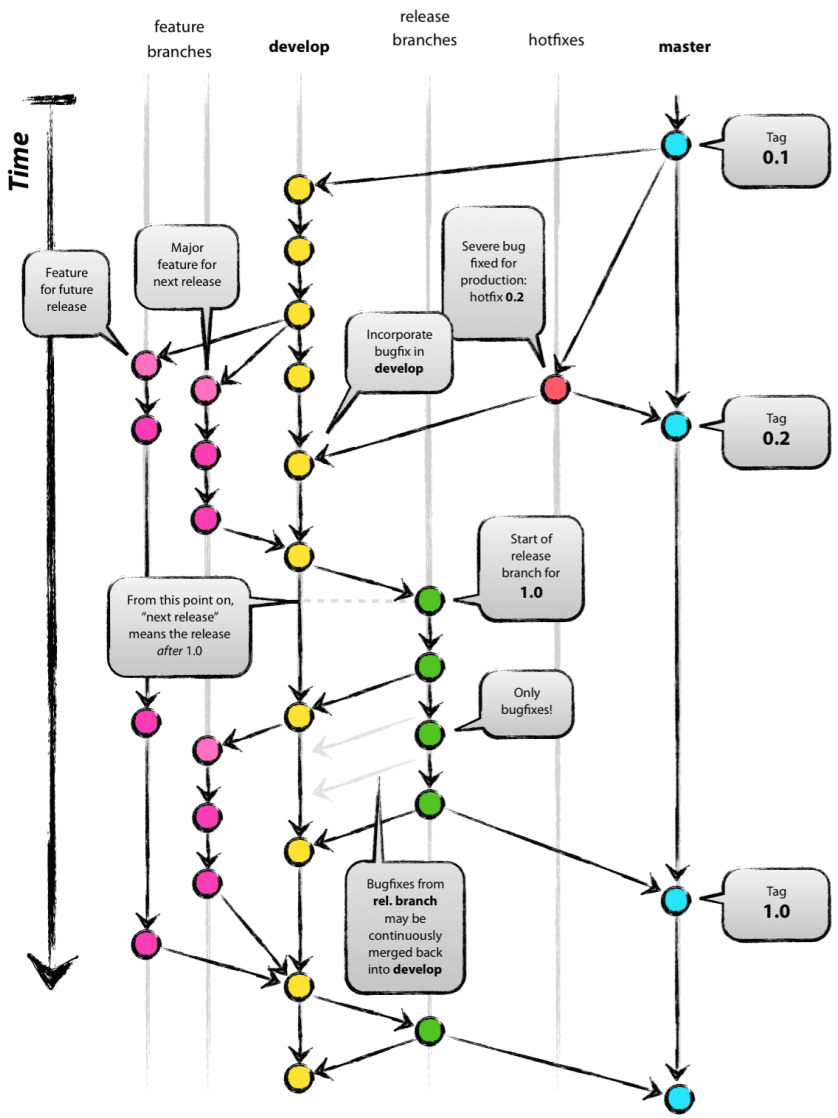
Professional Git

- Commit/branch conventions
- Deciding what goes in, and what stays out of the store
  - Share all the things that should be shared
  - Only share things that should be shared
- Normalizing contents of the store
  - Windows vs linux line endings

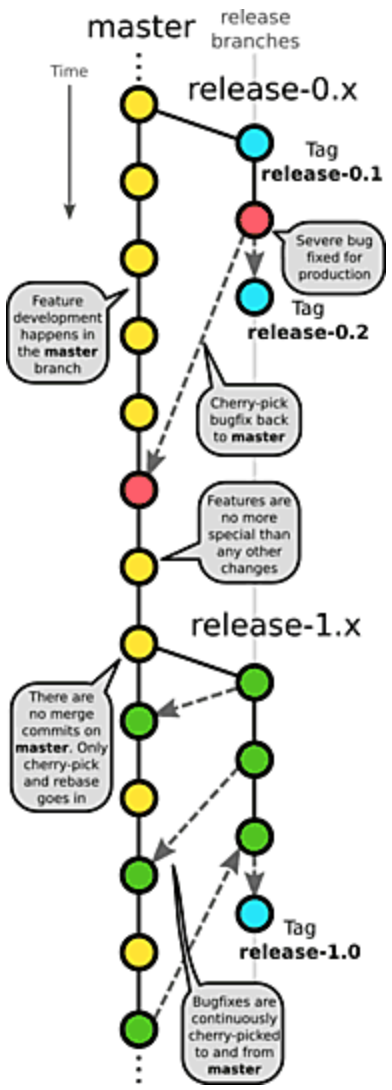
Commit/Branch Conventions

- Team strategy for managing the structure of the DAG (ie the store)
- Examples:
  - "Main is always deployable"
    - All work is done on other branches, merged with main only when result is executable
  - "Feature branches", "developer branches"
    - Each feature developed on its own branch vs. each developer works on their own branch
  - "Favor rebase over merge"
    - Always append to latest origin/branch

Example: Branch-Based Dev



Example: Trunk-Based Dev



What Goes Into Central Repo?

- Avoid developer-specific environment settings
  - Hard-coded file/directory paths from local machine
  - OK to include a sample config (each developer customizes but keeps their version out of store)
- Avoid living binaries (docx, pdf)
  - Meaningless diffs
- Avoid generated files
  - compiled files, the build
- Avoid IDE-specific files (.settings)
  - Some generic ones are OK so it is easier to get started by cloning, especially if the team uses the same IDE
- Avoid private information
  - Passwords, secret tokens
  - Better: Use *environment variables* instead
- Agree on code formatting
  - Auto-format is good, but only if everyone uses the same format settings!
  - Spaces vs tabs, brace position, etc

Ignoring Files from Working Tree

- Use a .gitignore file in root of project
  - Committed as part of the project
  - Consistent policy for everyone on team
- Examples: https://github.com/github/gitignore

```
# github:gitignore/Java.gitignore
# Compiled class file
*.class

# Log file
*.log

# Package Files #
*.jar
*.war
*.ear
*.zip
*.tar.gz
*.rar
```

Problem: End-of-line Confusion

- Differences between OS's in how a new line is encoded in a text file
  - Windows: 2 bytes, CR + LF ("\r\n", 0x0D 0x0A)
  - Unix/Mac: 1 byte, LF ("\n", 0x0A)
- Difference is hidden by most editors
  - An IDE might recognize either when opening a file, but convert all to \r\n when saving
  - Demo: hexdump (or VSCode hex editor)
- But difference matters to git when comparing files!
- Problem: OS differences within team
  - Changing 1 line causes every line to be modified
  - Flood of spurious changes masks the real edit

Solution: Normalization

- Convention: Store uses \n (ie linux)
  - Working tree uses OS's native eol
  - Convert when moving data between the two (e.g., commit, checkout)
- Note: Applies to text files only
  - A binary file, like a jpg, might contain 0x0D and/or 0x0A, but they should never be converted
- How does git know whether a file is text or binary?
  - Heuristics: auto-detect based on contents
  - Configuration: filename matches a pattern

Normalization With .gitattributes

- Use a .gitattributes file in root of project
  - Committed as part of the project
  - Consistent policy for everyone on team
- Example:

```
# Auto detect text files and perform LF normalization
* text=auto

# These files are text, should be normalized (crlf=>lf)
*.java       text
*.md         text
*.txt        text
*.classpath  text
*.project    text

# These files are binary, should be left untouched
*.class      binary
*.jar        binary
```

Ninja Git: Advanced Moves

- Temporary storage

  `stash`

- Undoing big and small mistakes in the working tree

  `reset`, **`checkout`**

- Undoing mistakes in store

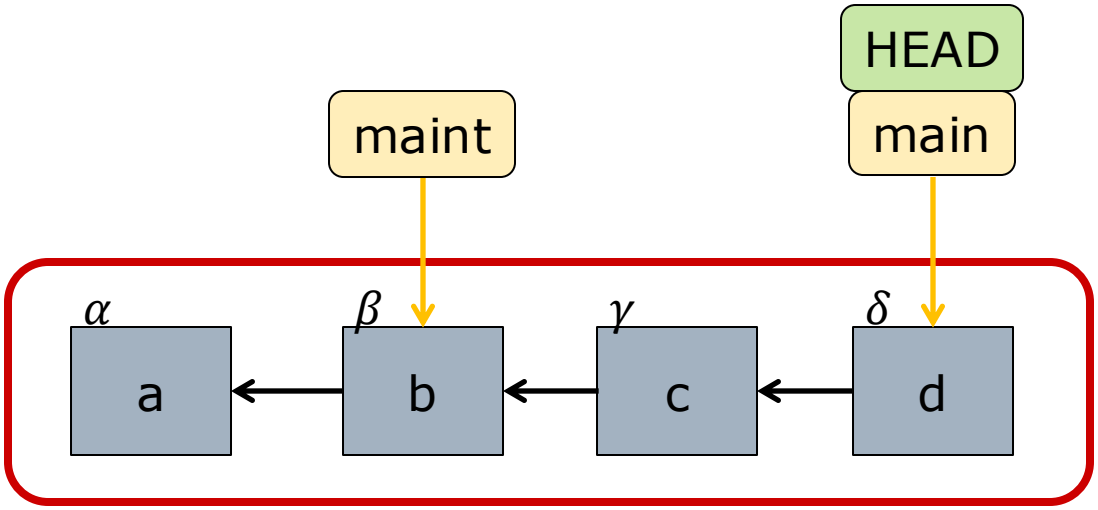
  `amend`

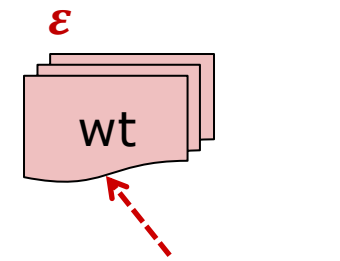
- DAG surgery

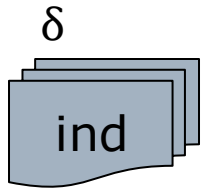
  `rebase`

Advanced: Temporary Storage

- Say you have uncommitted work and want to look at a different branch
- Checkout won't work! (Recall: "only checkout when wt is clean")

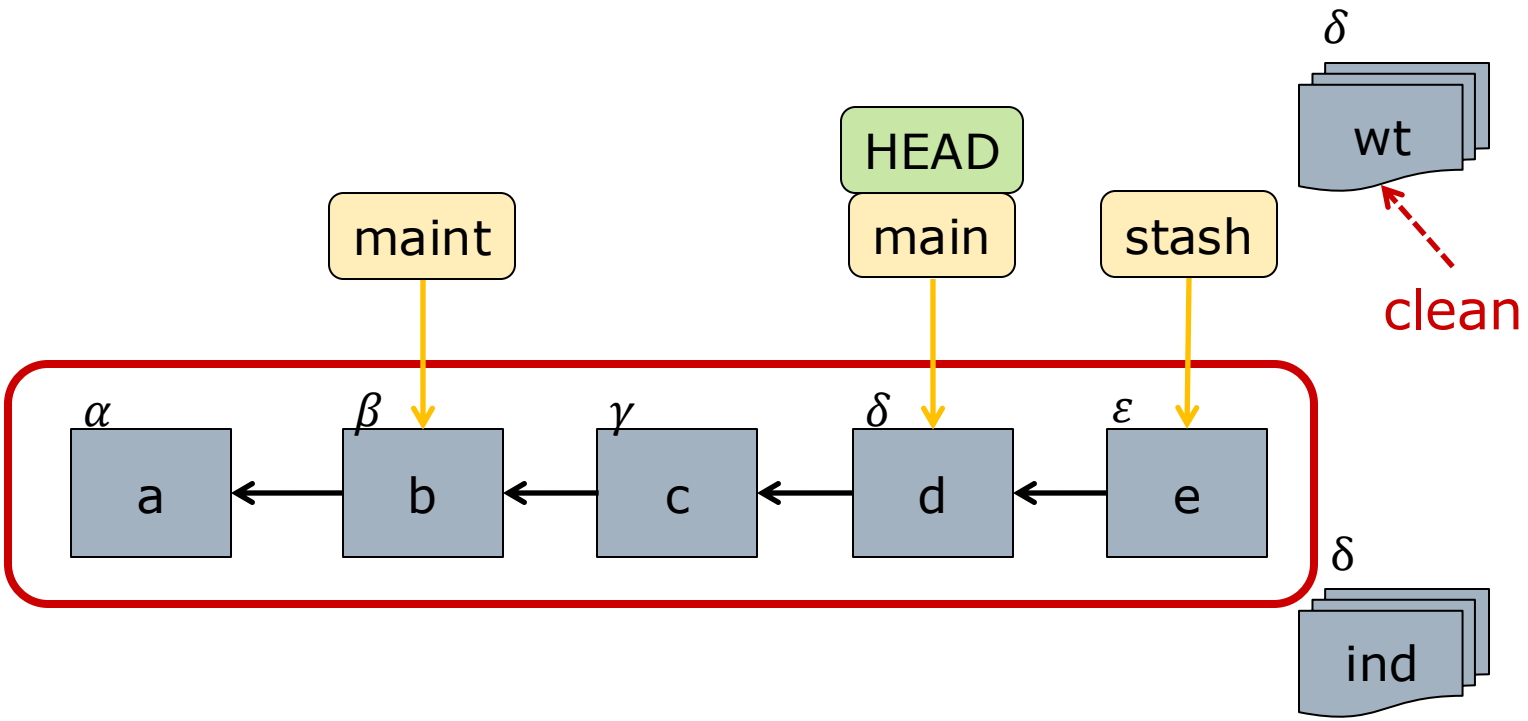






Stash: Push Work Onto a Stack

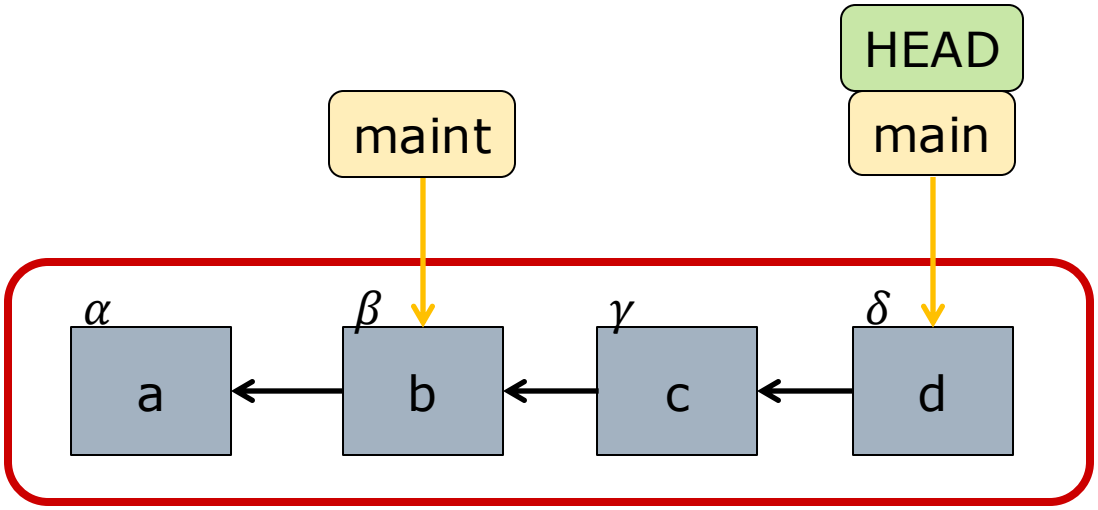

```
$ git stash # repo now clean
$ git checkout …etc… # feel free to poke around
```

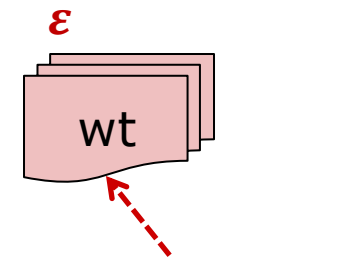


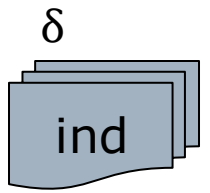
Stash: Pop Work Off the Stack


```
$ git stash pop # restores state of wt/index

# equivalent to:
$ git stash apply # restore wt and index
$ git stash drop # restore store
```

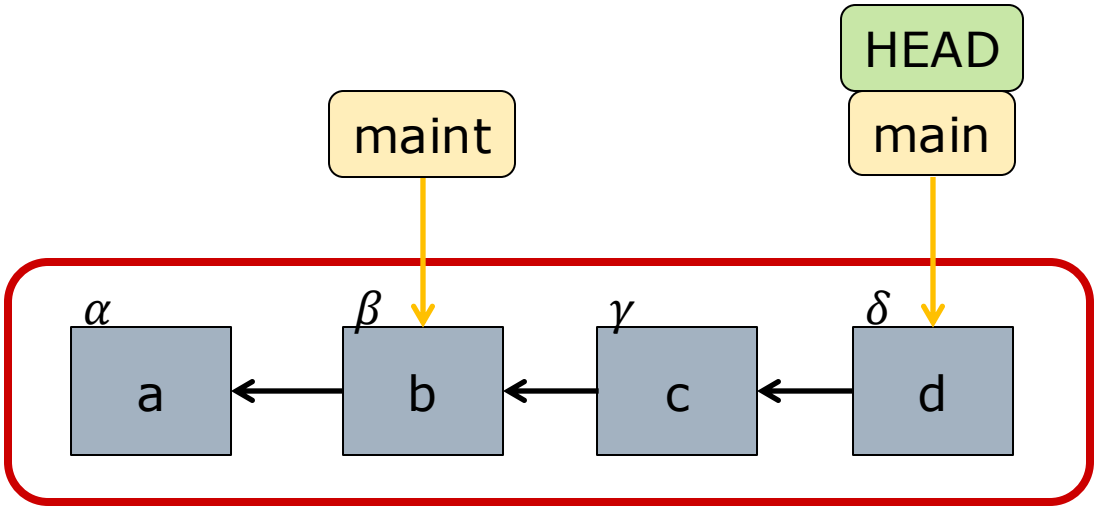






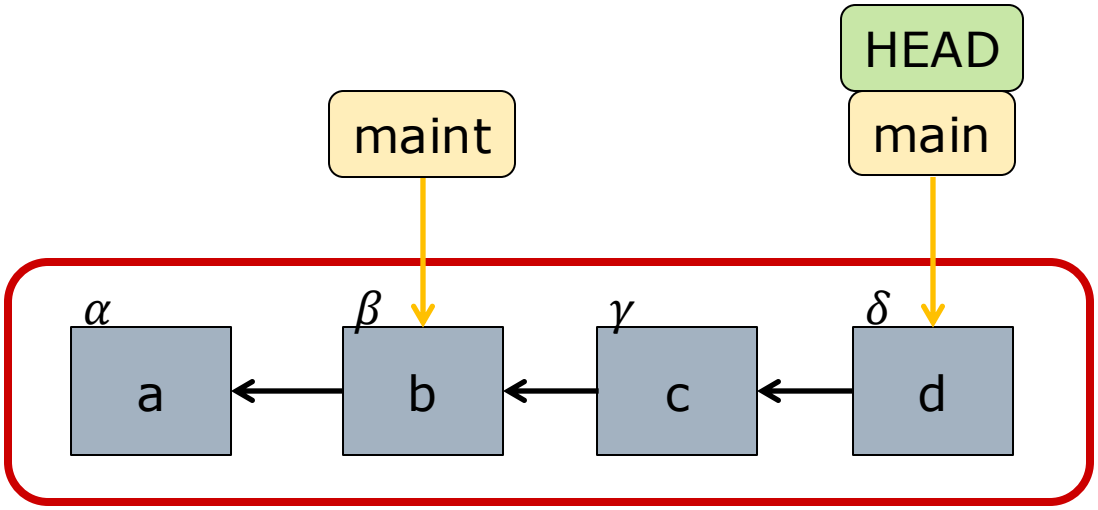
Advanced: Undoing Big Mistakes

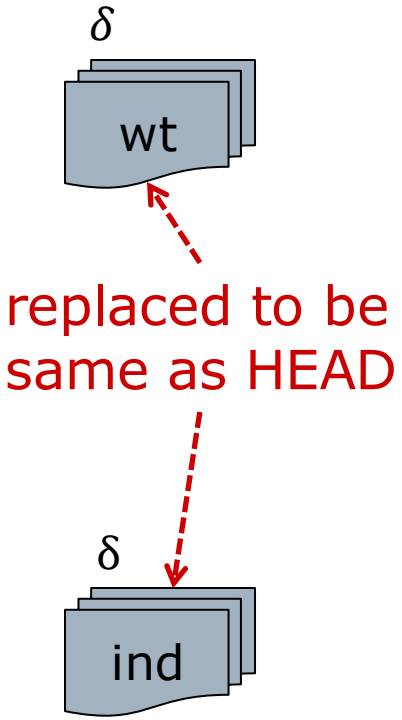
- Say you want to throw away *all* your uncommited work
  - ie "Roll back" to last commited state
- Checkout HEAD won't work!



Reset: Discarding Changes

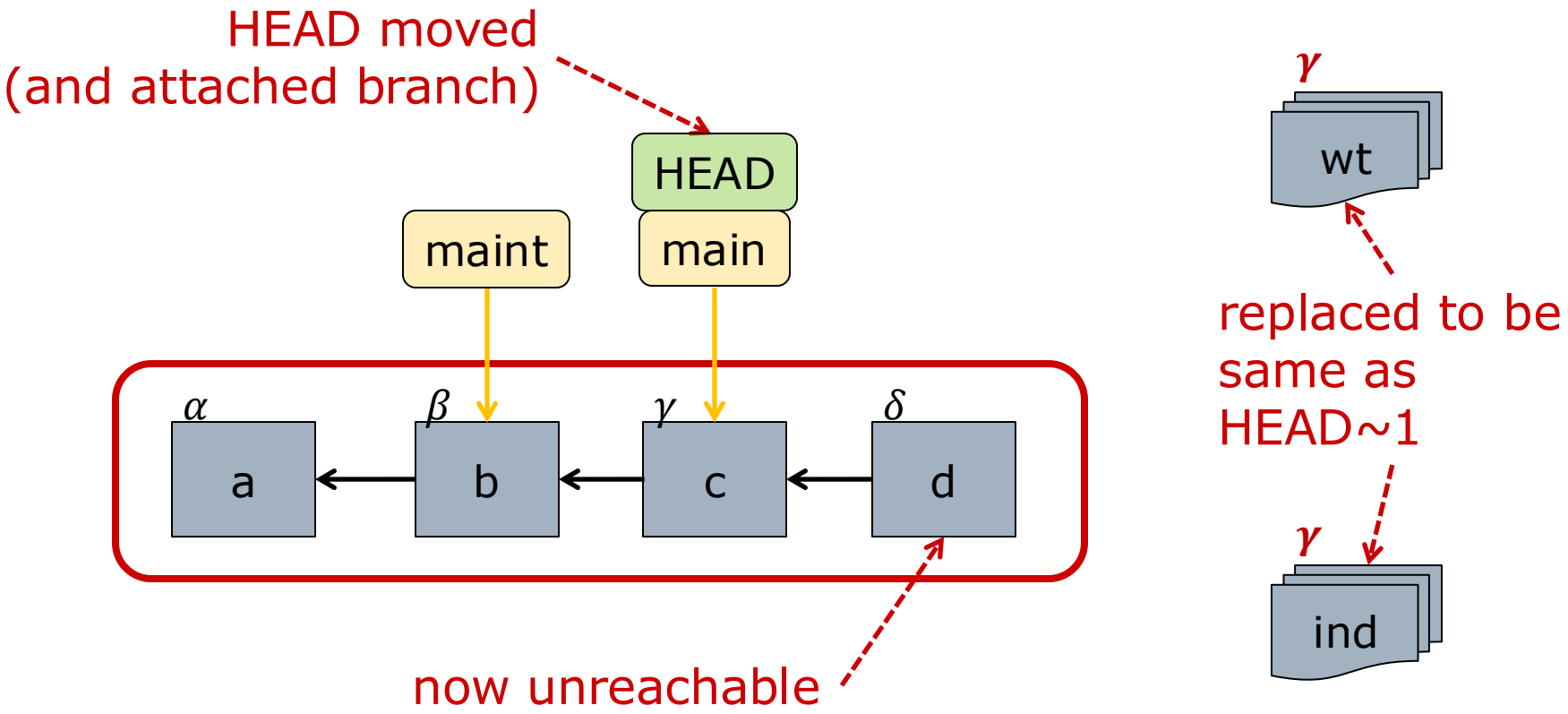

```
$ git reset --hard     # updates wt to be HEAD
$ git clean --dry-run  # list untracked files
$ git clean --force    # remove untracked files
```





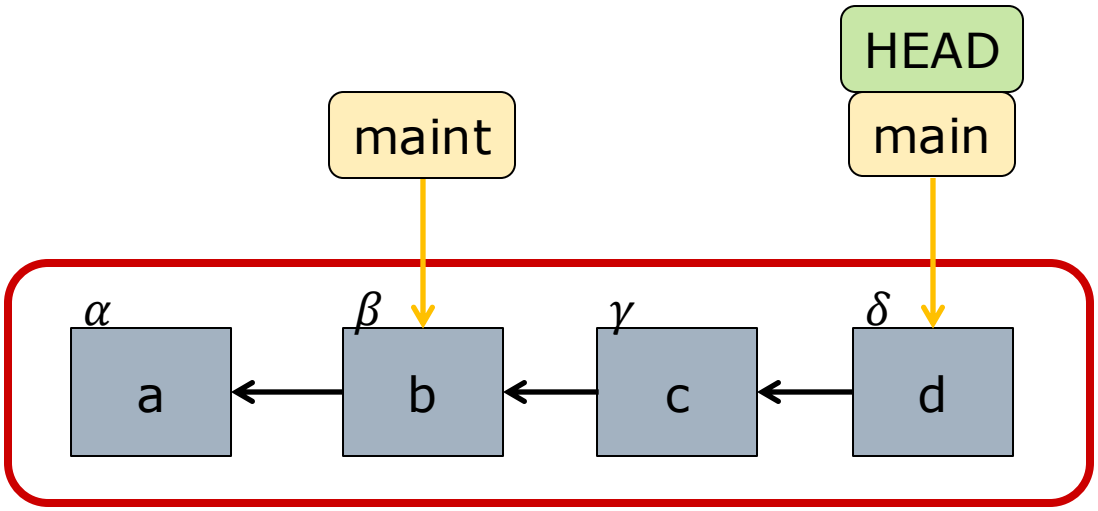
Reset: Discarding Commits


```
$ git reset --hard HEAD~1
# no need to git clean, since wt was already clean
```



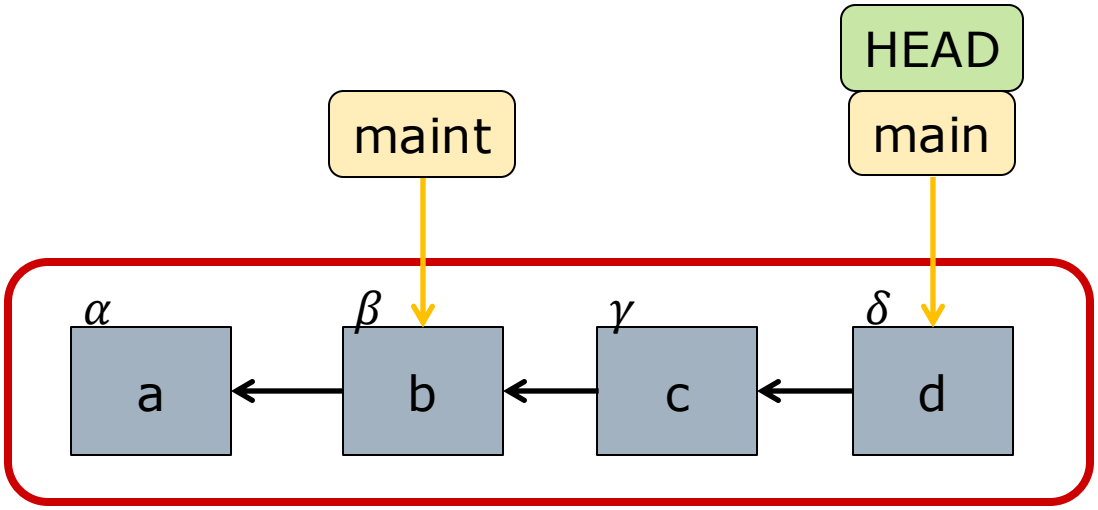
Advanced: Undo Small Mistakes

- Say you want to throw away *some of* your uncommited work
  - Restore a file to last committed version



Advanced: Undo Small Mistakes


```
$ git checkout -- README.md
# -- means: rest is file/path (not branch)
# git checkout README.md ok, if not ambiguous
```



Advanced: Rewriting History

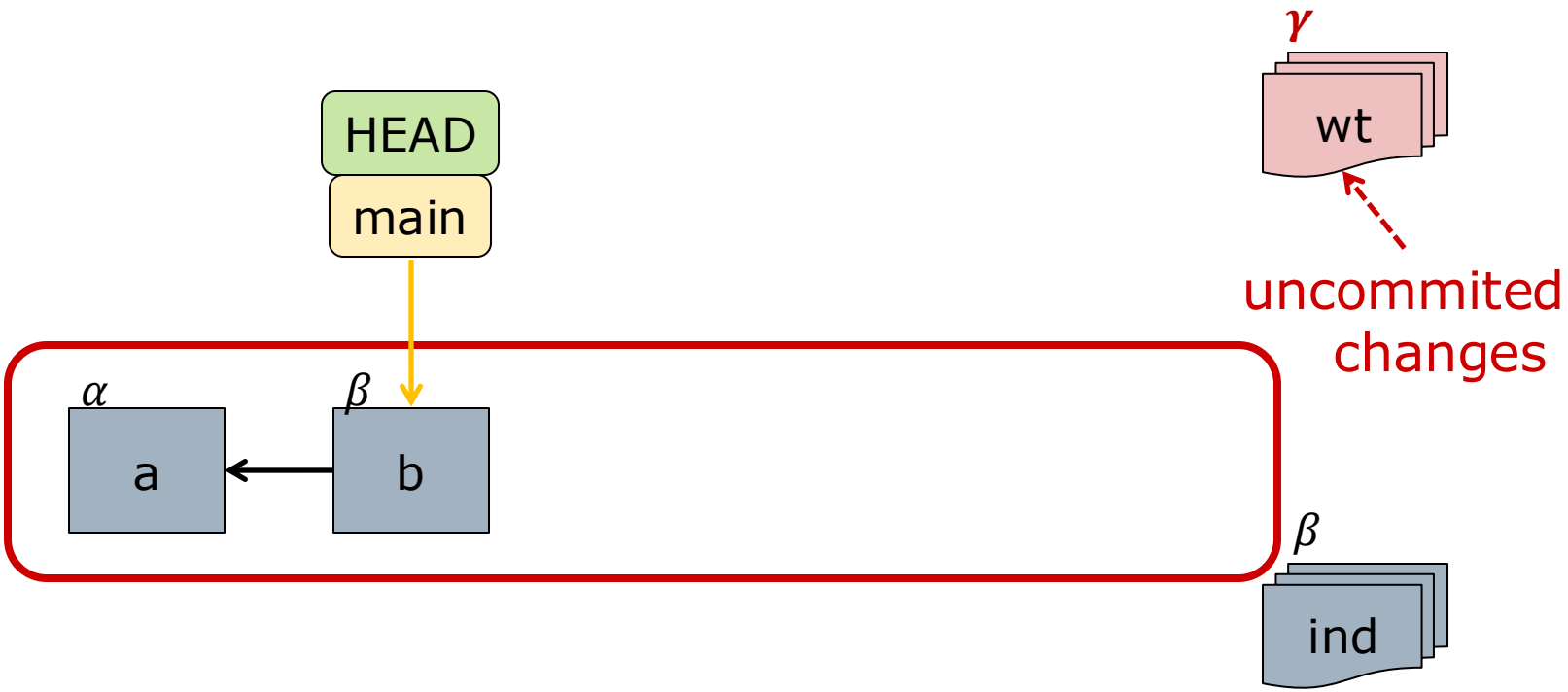
The Power to Change History

- Changing the store lets us:
  - Fix mistakes in recent commits
  - Clean up messy DAGs to make history look more linear

- Rule: Never change *shared* history
  - Once something has been pushed to a remote repo (*e.g.*, origin), do not change that part of the DAG
  - So: A *push* is really a *commit*ment!

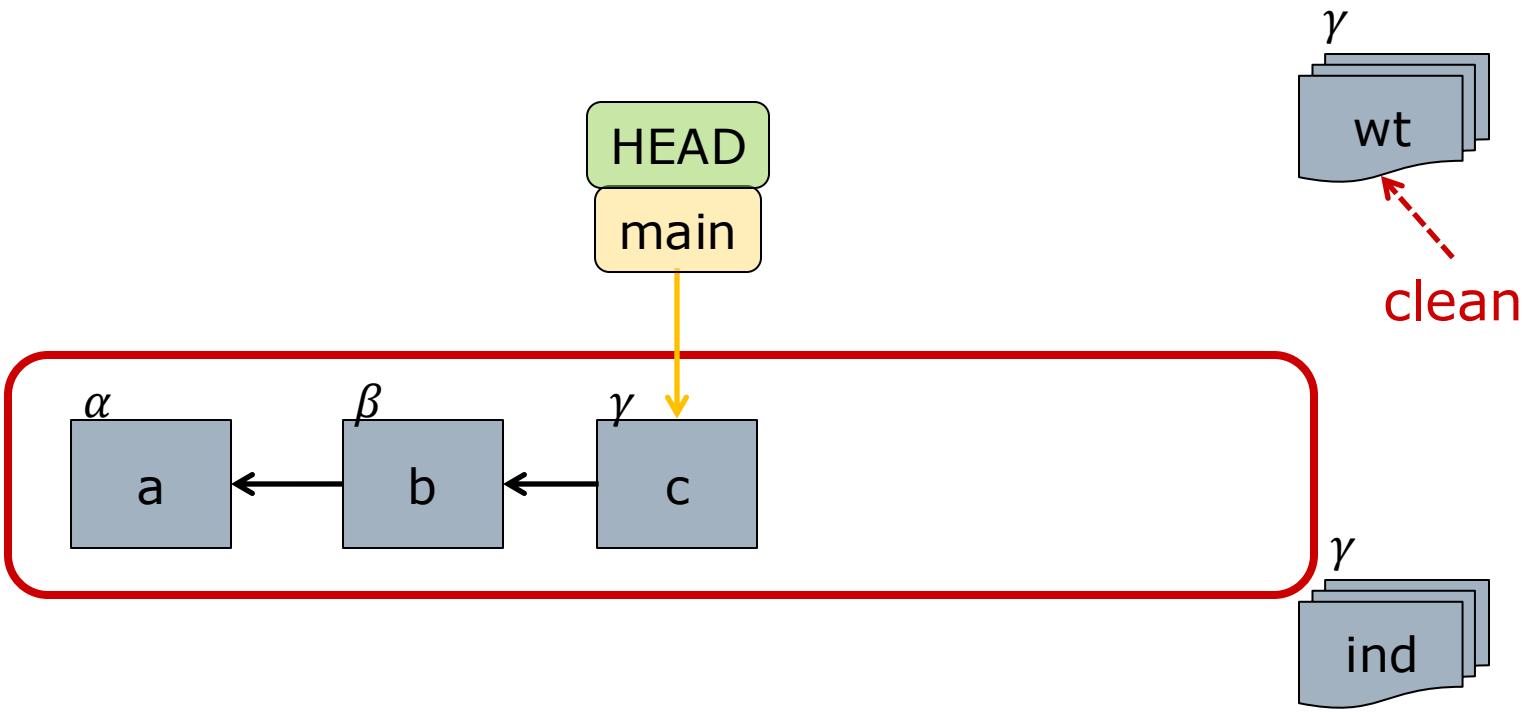
Advanced: Rewriting History

- Problem 1: Wrong or incomplete commit



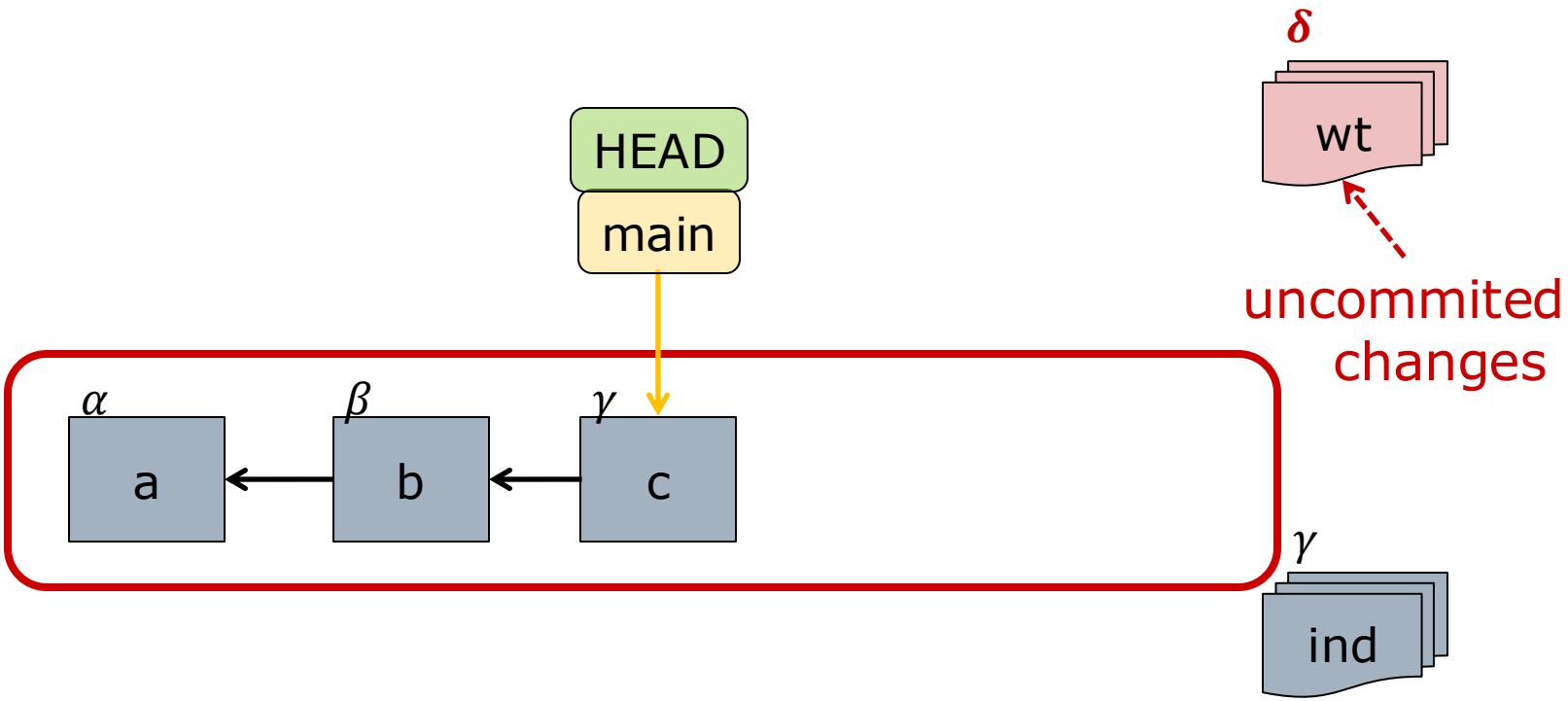
Advanced: Rewriting History

- Problem 1: Wrong or incomplete commit



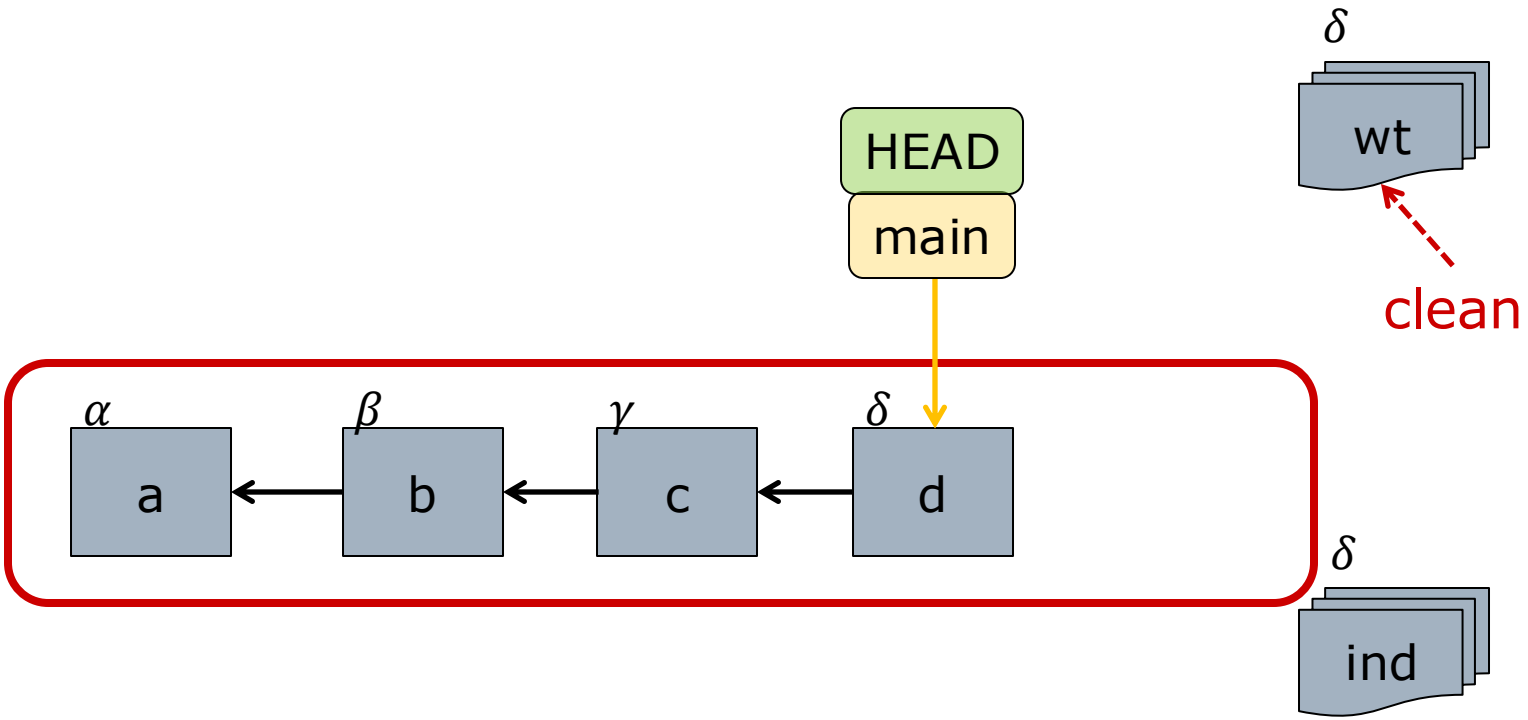
Advanced: Rewriting History

- Problem 1: Wrong or incomplete commit
  - Oops! That wasn't quite right…



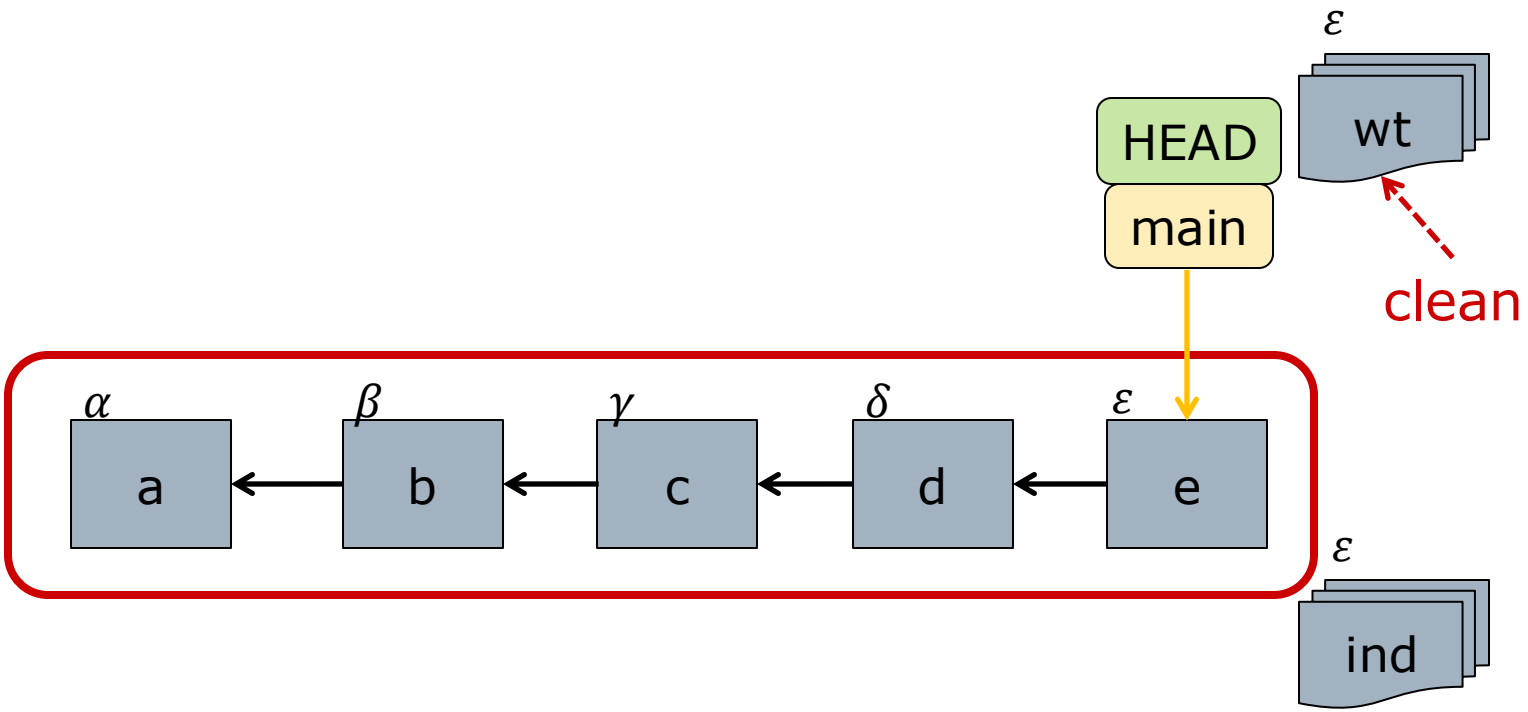
Advanced: Rewriting History

- Problem 1: Wrong or incomplete commit
  - Oops! That wasn't quite right…



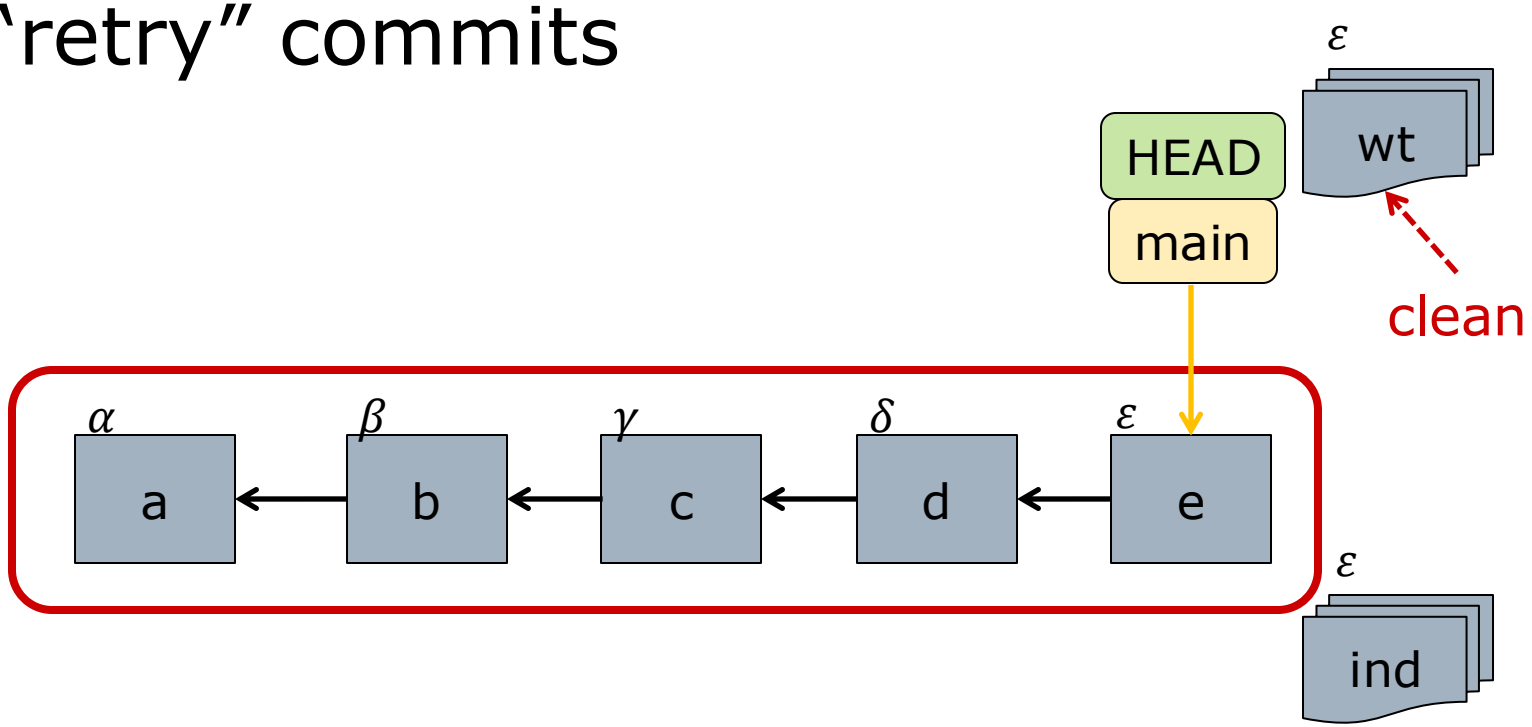
Advanced: Rewriting History

- Problem 1: Wrong or incomplete commit
  - Oops! That wasn't quite right…



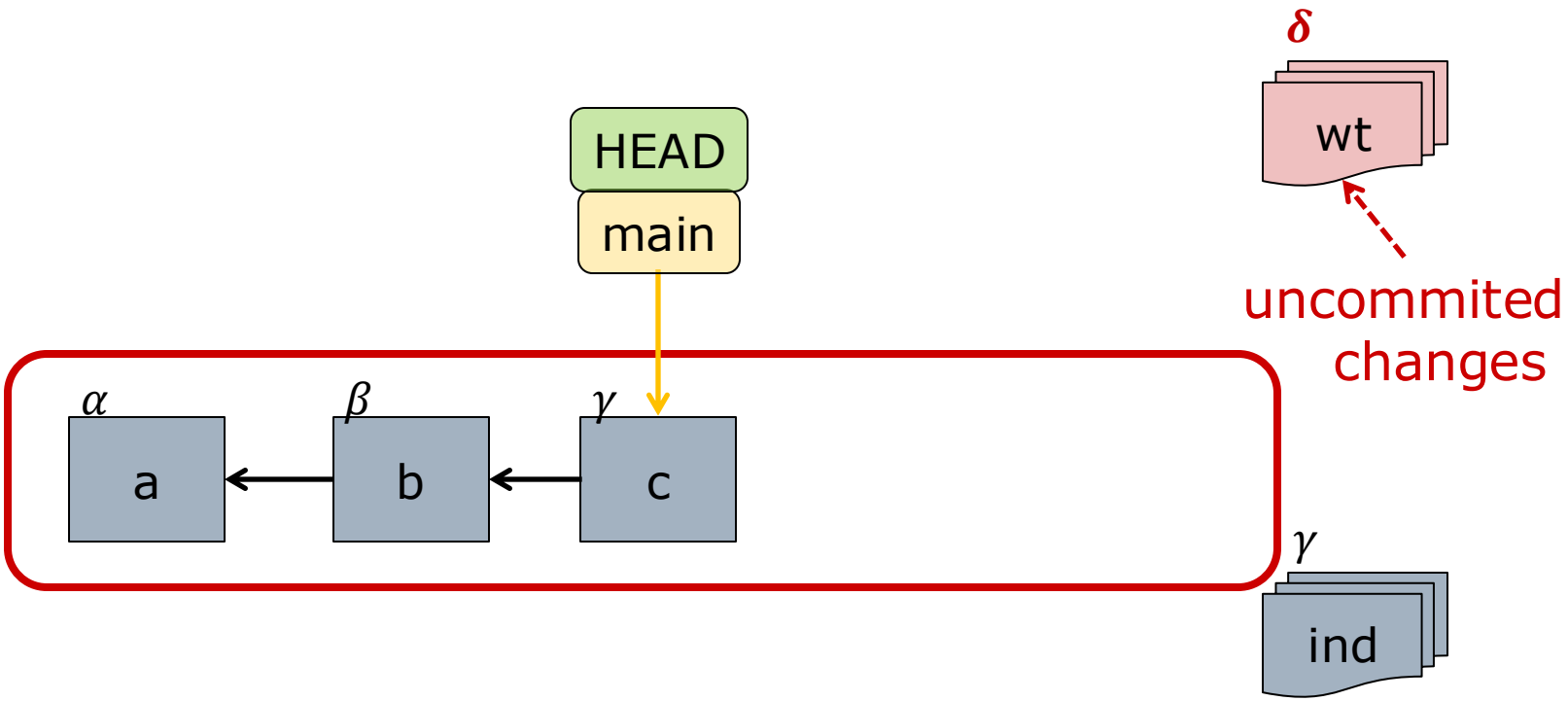
Advanced: Rewriting History

- Problem 1: Wrong or incomplete commit
- Result: Lots of tiny “fix it”, “oops”, “retry” commits



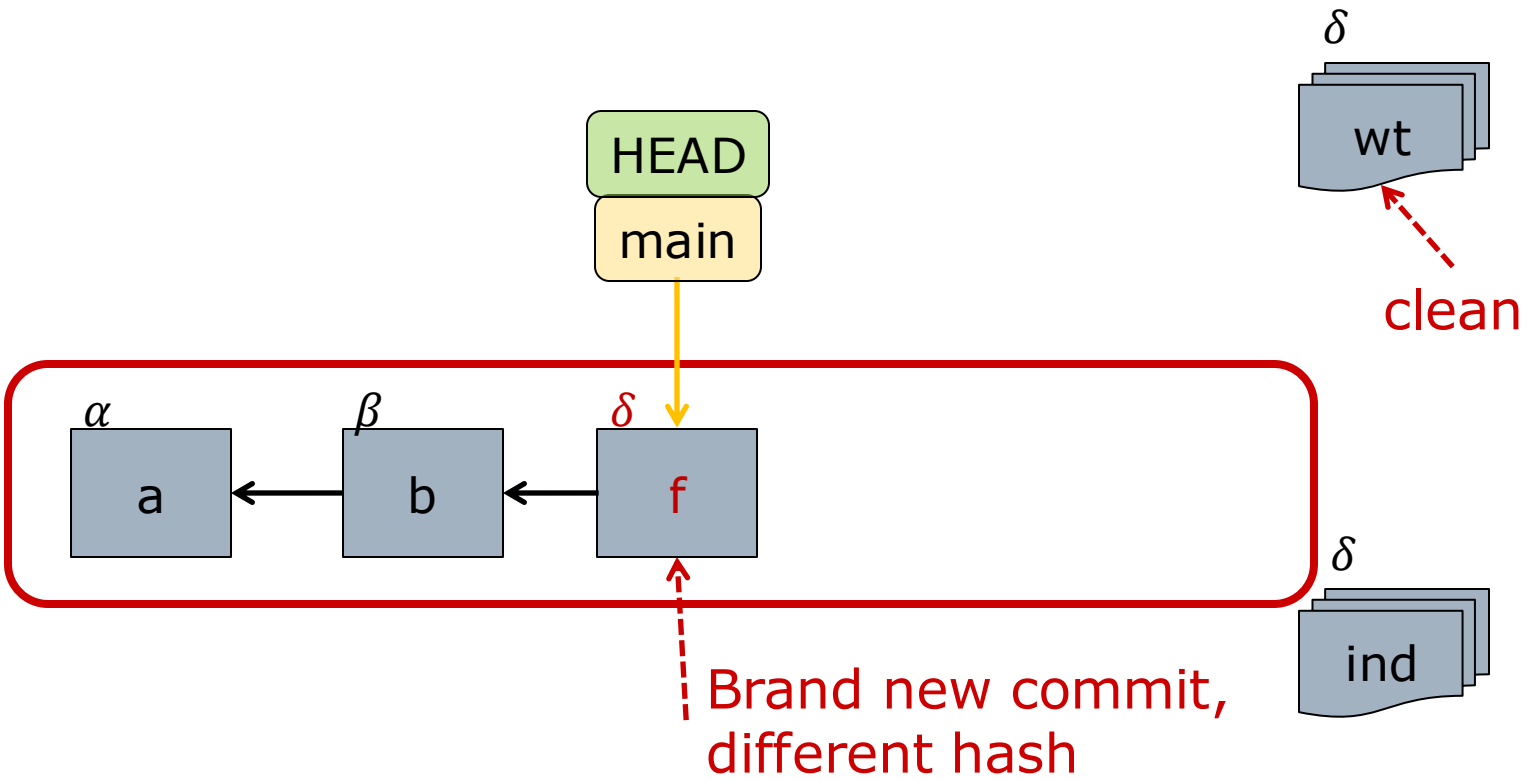
Commit --amend: Tip Repair

- Alternative: Change most recent commit(s)



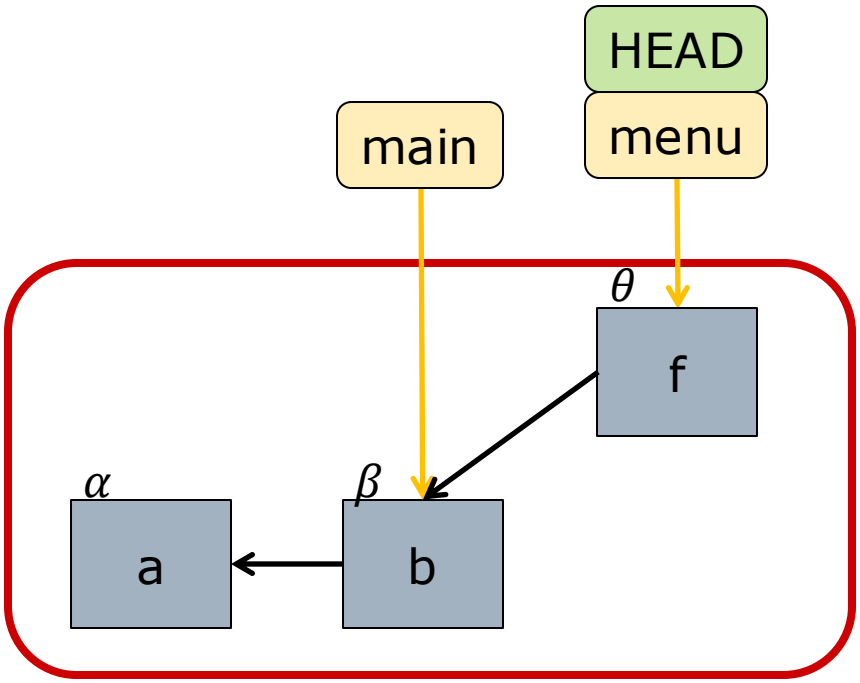
Commit --amend: Tip Repair


```
$ git add .
$ git commit --amend --no-edit
# no-edit keeps the same commit message
```



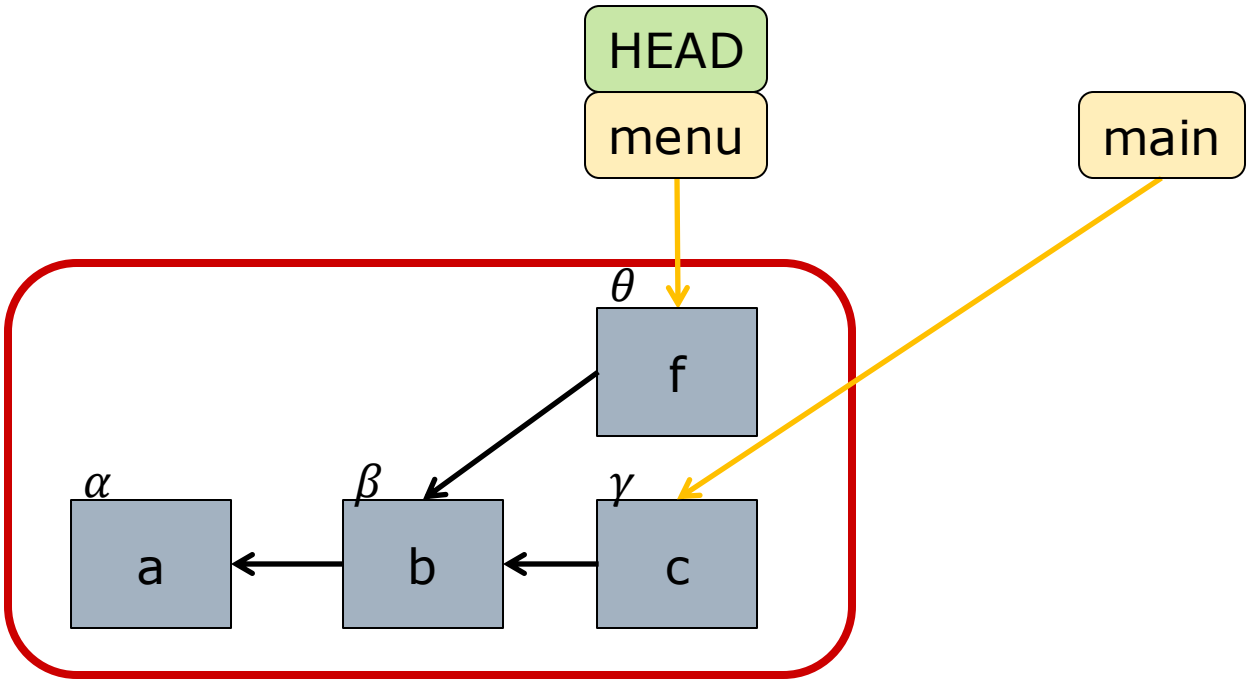
Advanced: Rewriting History

- Problem 2: As an independent branch is being developed, main also evolves



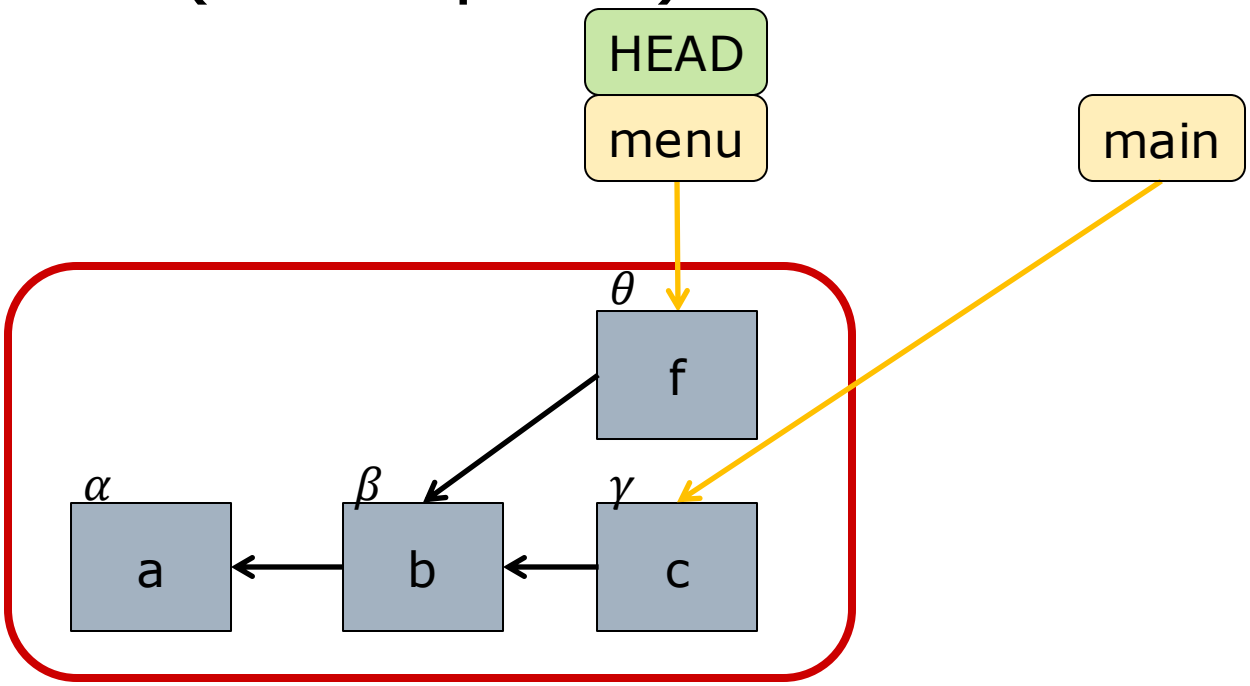
Advanced: Rewriting History

- Problem 2: As an independent branch is being developed, main also evolves



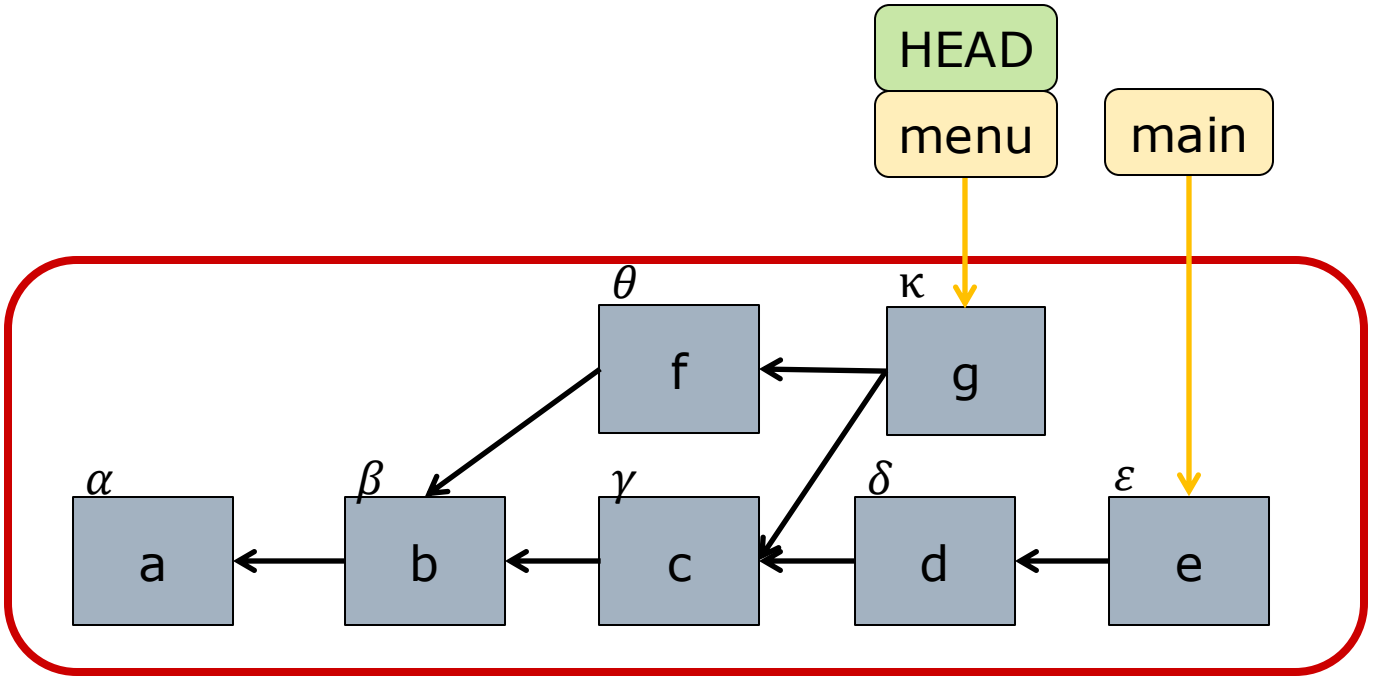
Advanced: Rewriting History

- Problem 2: As an independent branch is being developed, main also evolves
- Result: Need periodic merges of main with (incomplete) branch



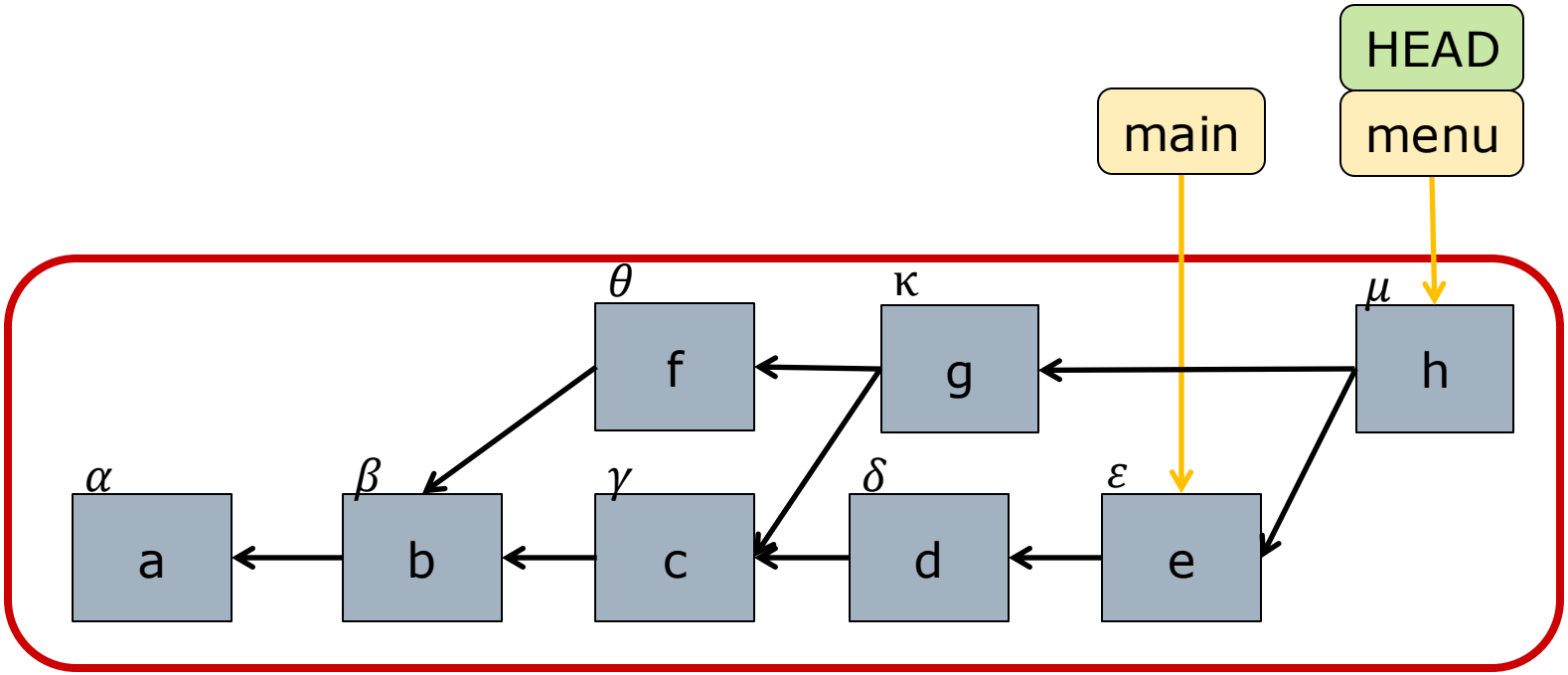
Advanced: Rewriting History

- Problem 2: As an independent branch is being developed, main also evolves
- Result: Need periodic merges of main with (incomplete) branch



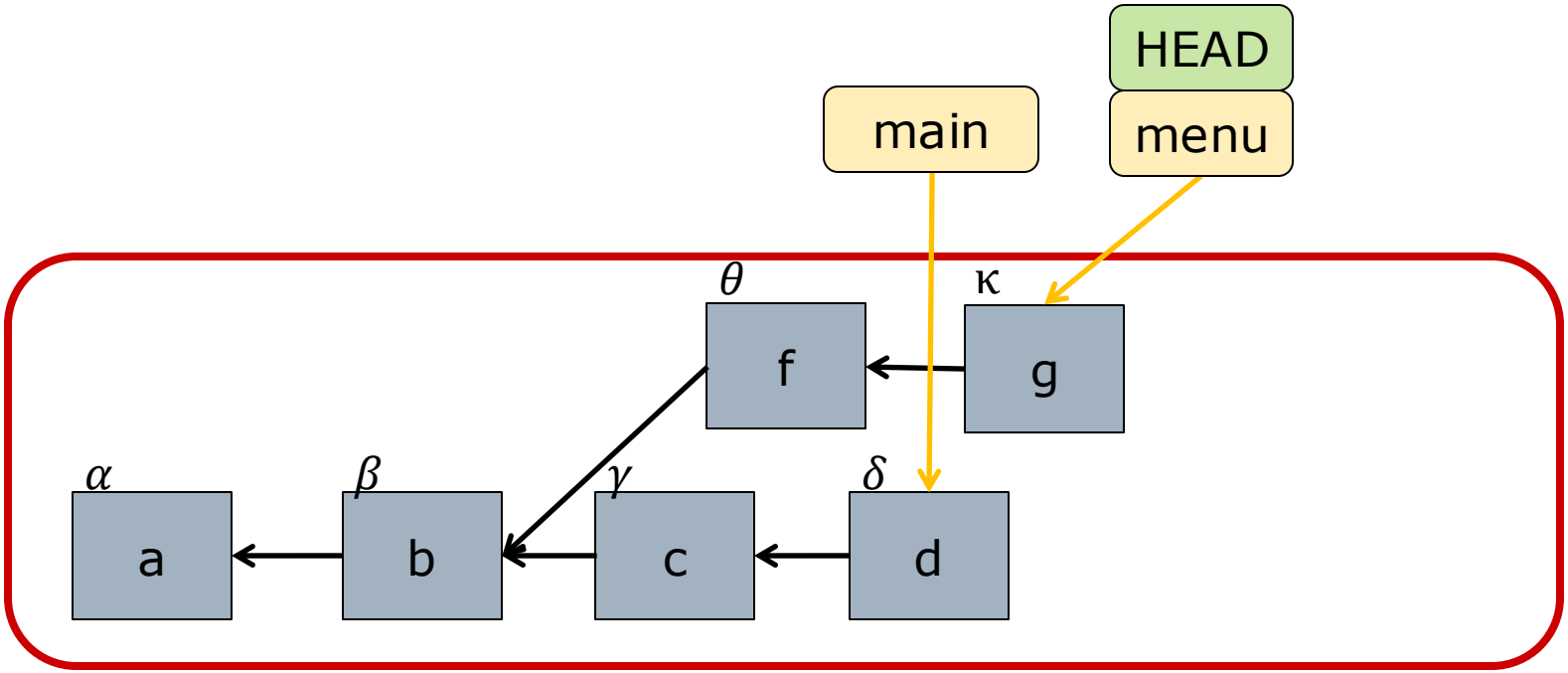
Advanced: Rewriting History

- Problem 2: As an independent branch is being developed, main also evolves
- Result: Need periodic merges of main with (incomplete) branch



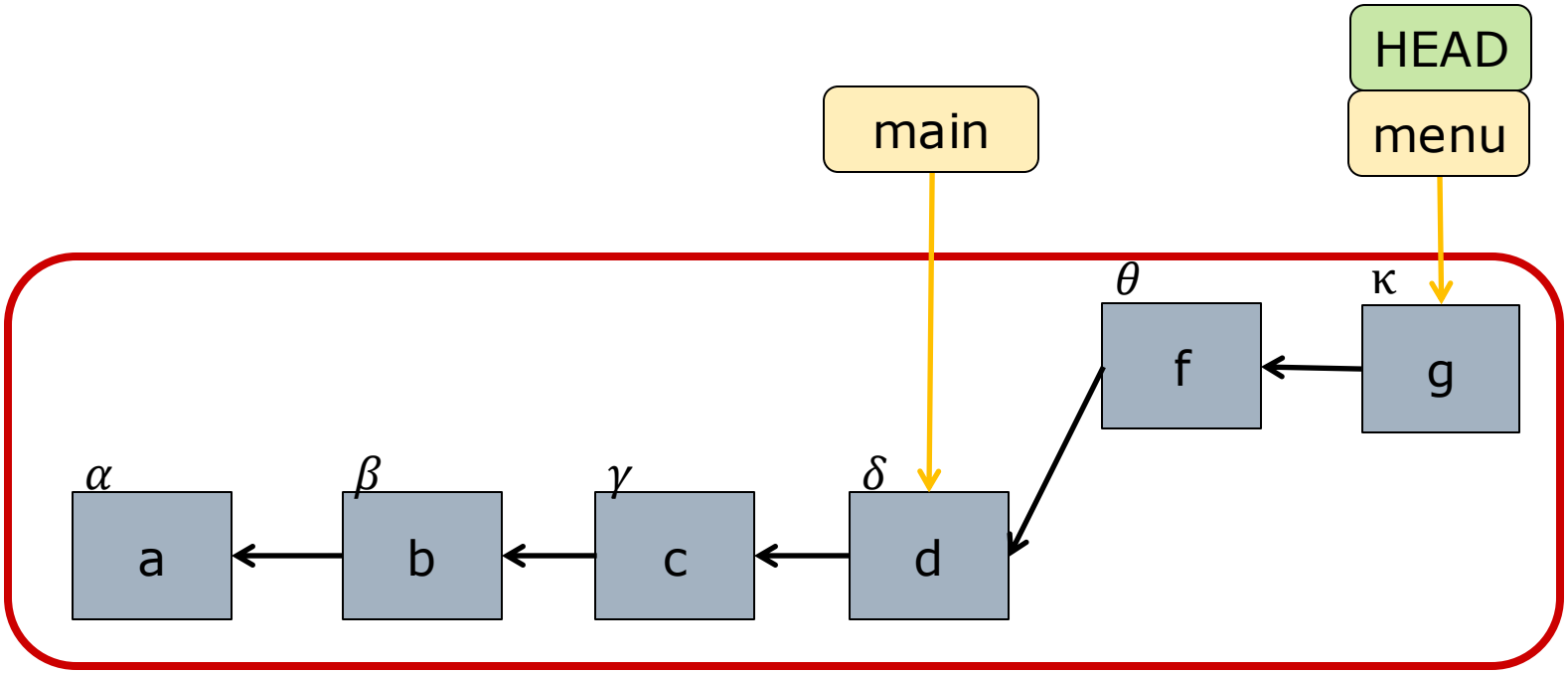
Rebase: DAG Surgery

- Alternative: Move commits to a different part of the DAG



Rebase: DAG Surgery


```
$ git rebase main
# merging main into menu is now a fast-forward
```



Git Clients and Hosting Services

- Recommend'n: Know the command line!
- IDEs are helpful too
  - VSCode, plus Git Graph extension
- Lots of sites for hosting your repos:
  - GitHub, GitLab, Bitbucket, SourceForge…
  - See: git.wiki.kernel.org/index.php/GitHosting
- These cloud services provide
  - Storage space, account/access management
  - Pretty web interface
  - Issues, bug tracking
  - Workflow (eg forks) to promote contributions from others

Clarity

git != GitHub

Warning: Academic Misconduct

- GitHub is a very popular service
  - New repos are *public* by default
  - Even free plan allows unlimited *private* repo's (and collaborators)
  - 3901 has an organization for your private repo's and team access
- Other services (*e.g.* GitLab, Bitbucket) have similar issues
- Public repo's containing coursework can create academic misconduct issues
  - Problems for poster
  - Problems for plagiarist

Summary

- Workflow
  - Fetch/push frequency
  - Respect team conventions for how/when to use different branches
- Central repo is a shared resource
  - Contains common (source) code
  - Normalize line endings and formats
- Advanced techniques
  - Stash, reset, rebase
- Advice
  - Learn by using the command line
  - Beware academic misconduct