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?

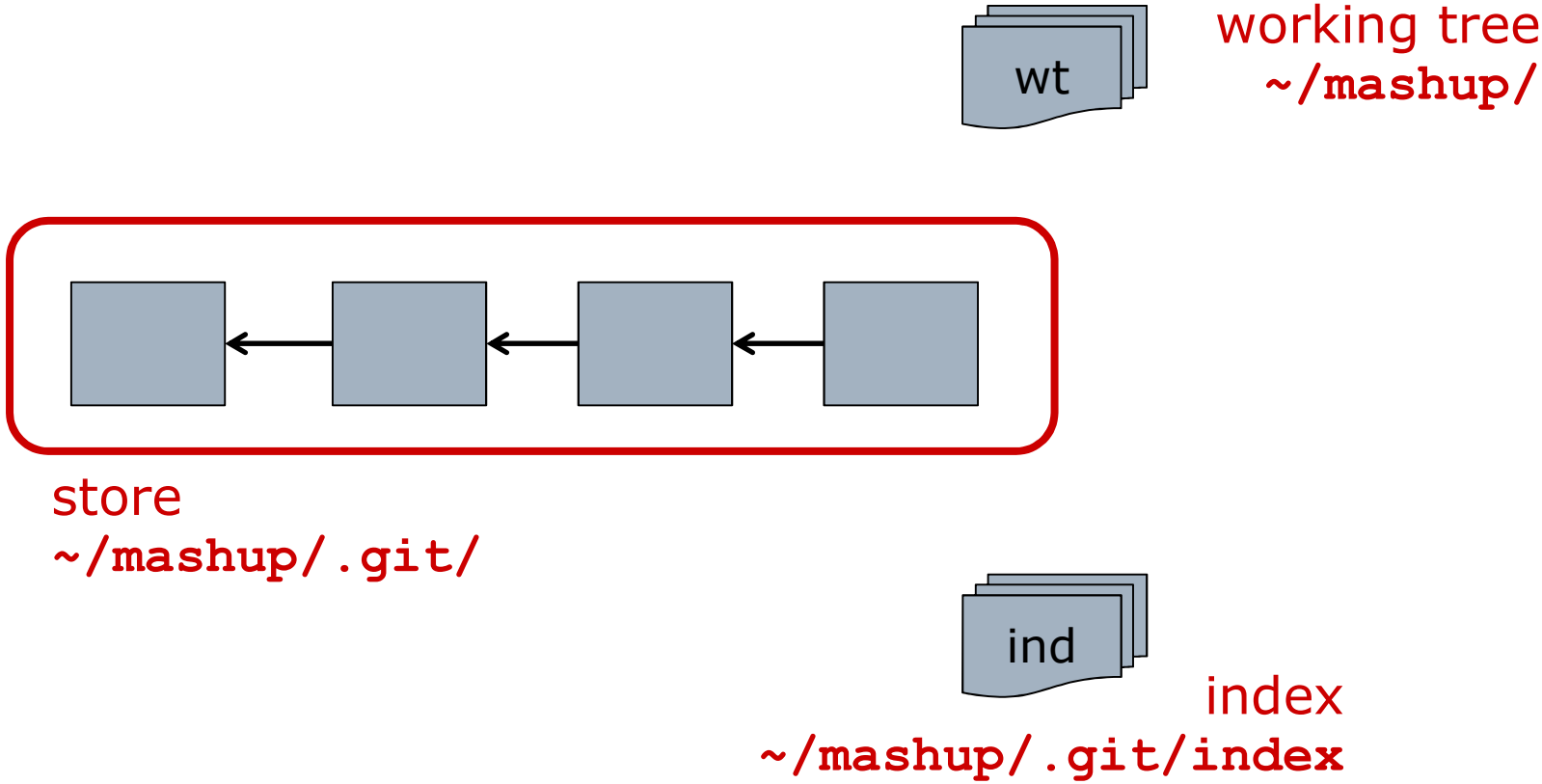
Key Idea: A Repository

- *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

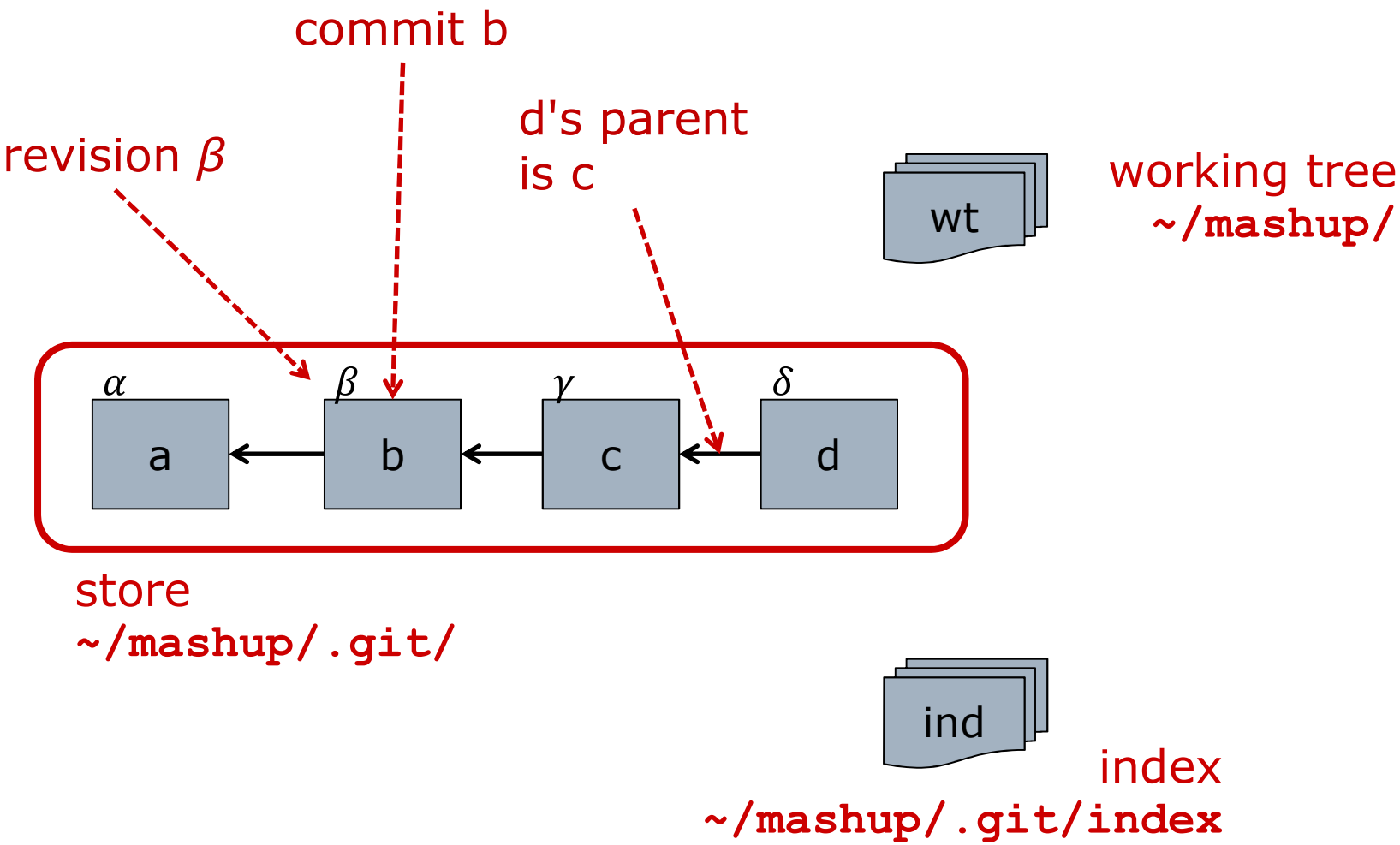
File Structure of a Repository


```
~/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

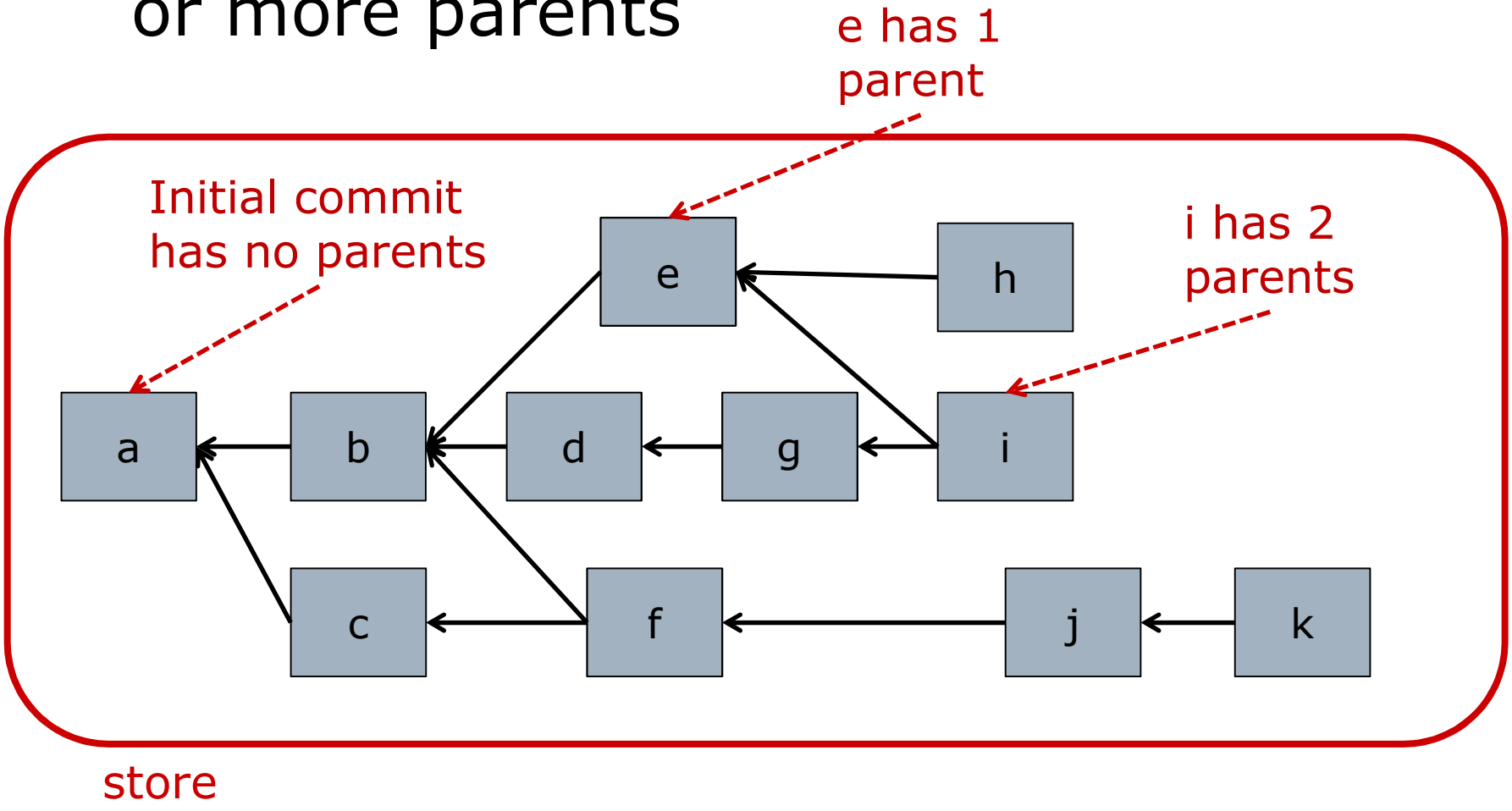


A History of Commits

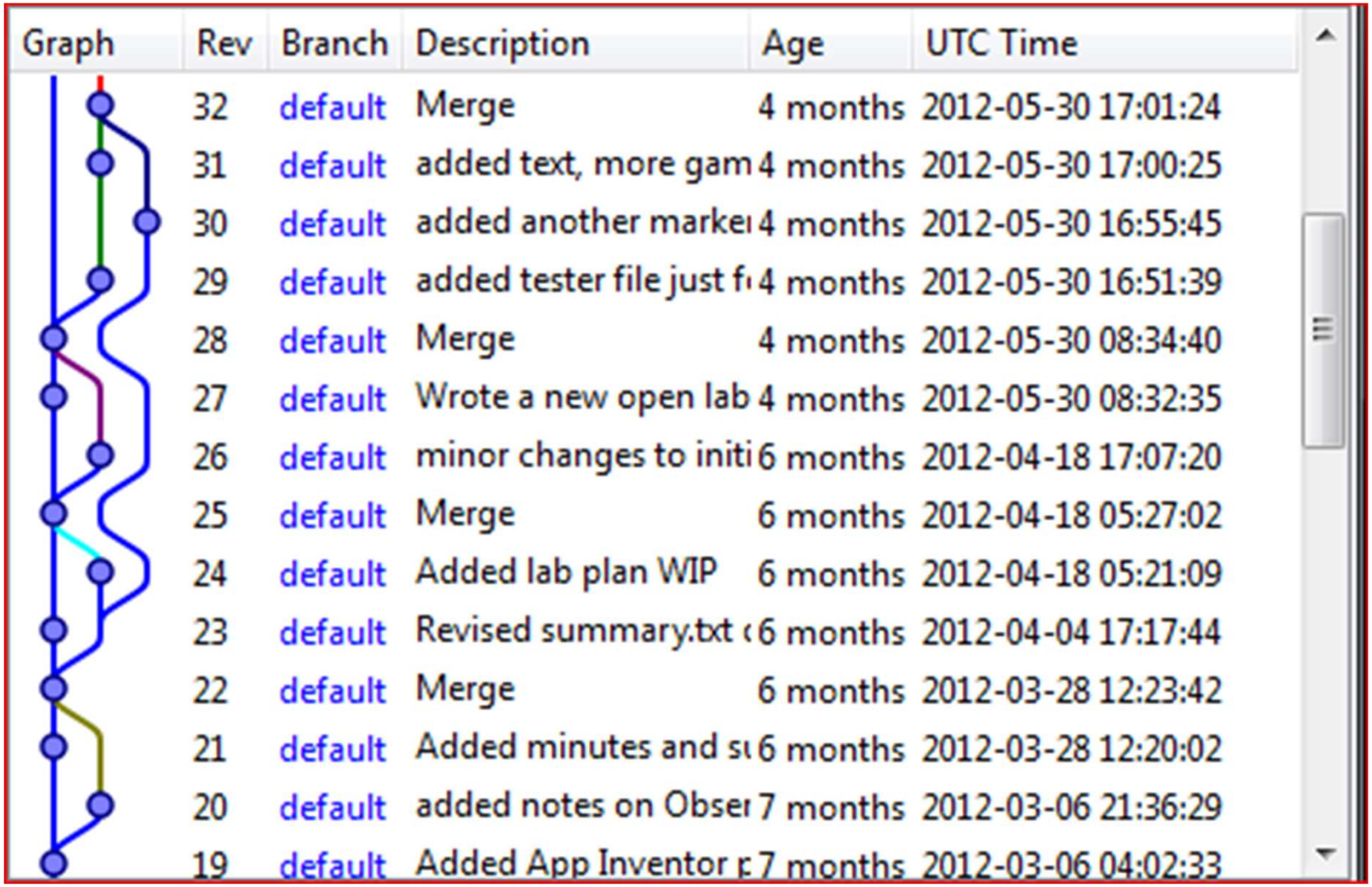


History is a DAG

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



Example View of DAG



Graph	Rev	Branch	Description	Age	UTC Time
	32	default	Merge	4 months	2012-05-30 17:01:24
	31	default	added text, more gam	4 months	2012-05-30 17:00:25
	30	default	added another marke	4 months	2012-05-30 16:55:45
	29	default	added tester file just f	4 months	2012-05-30 16:51:39
	28	default	Merge	4 months	2012-05-30 08:34:40
	27	default	Wrote a new open lab	4 months	2012-05-30 08:32:35
	26	default	minor changes to initi	6 months	2012-04-18 17:07:20
	25	default	Merge	6 months	2012-04-18 05:27:02
	24	default	Added lab plan WIP	6 months	2012-04-18 05:21:09
	23	default	Revised summary.txt (	6 months	2012-04-04 17:17:44
	22	default	Merge	6 months	2012-03-28 12:23:42
	21	default	Added minutes and s	6 months	2012-03-28 12:20:02
	20	default	added notes on Obse	7 months	2012-03-06 21:36:29
	19	default	Added App Inventor p	7 months	2012-03-06 04:02:33

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

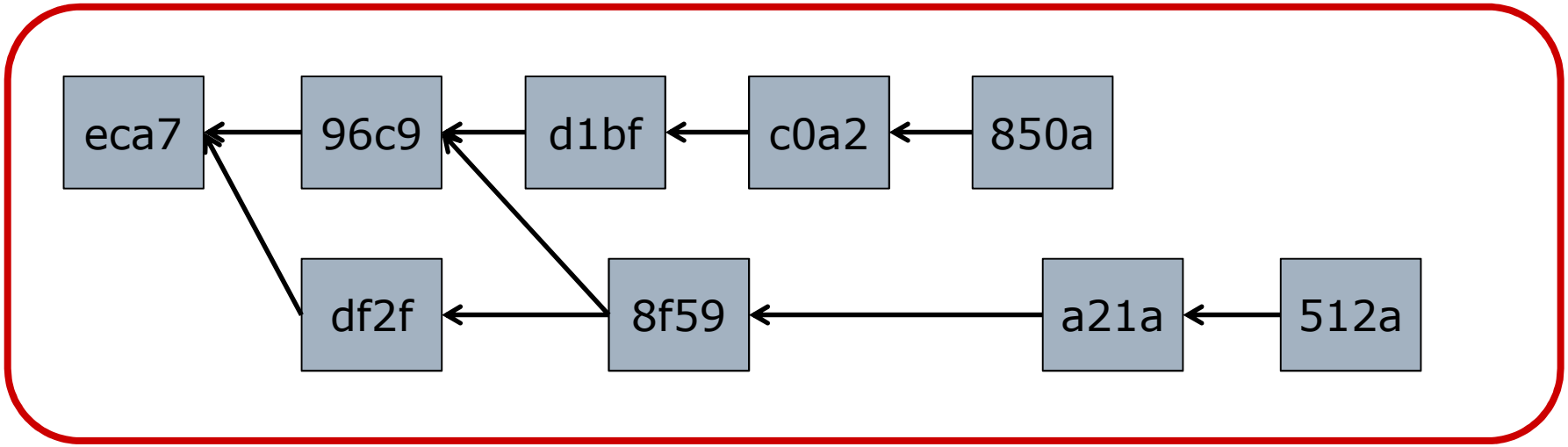
```
$ git show --name-only --no-decorate
commit 16188493c252f6924baa17c9b84a4c1baaed438b
Author: Brutus Buckeye <brutus@users.noreply.github.com>
Date:   Mon Mar 29 15:30:50 2021 +0200

    clean up css

source/stylesheets/_site.css
```

History is a DAG

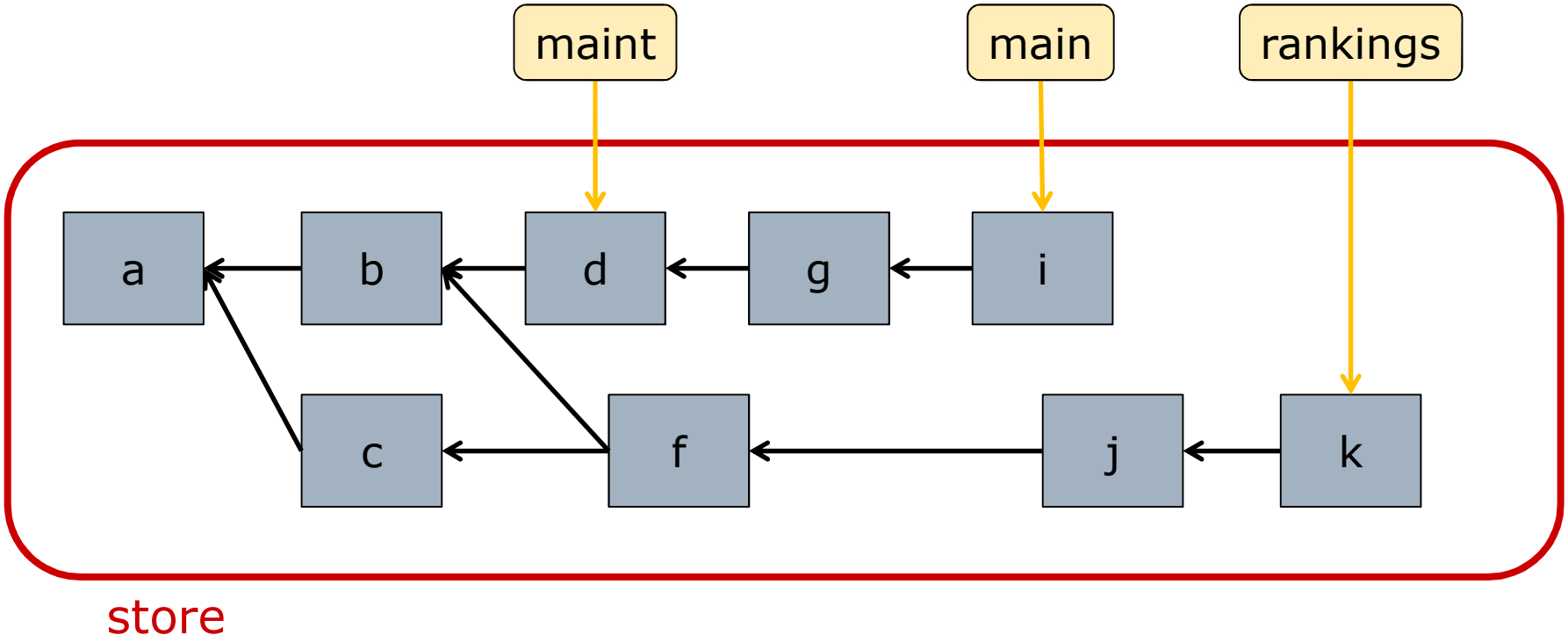
- A better picture would label each commit with its hash (prefix)



- 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

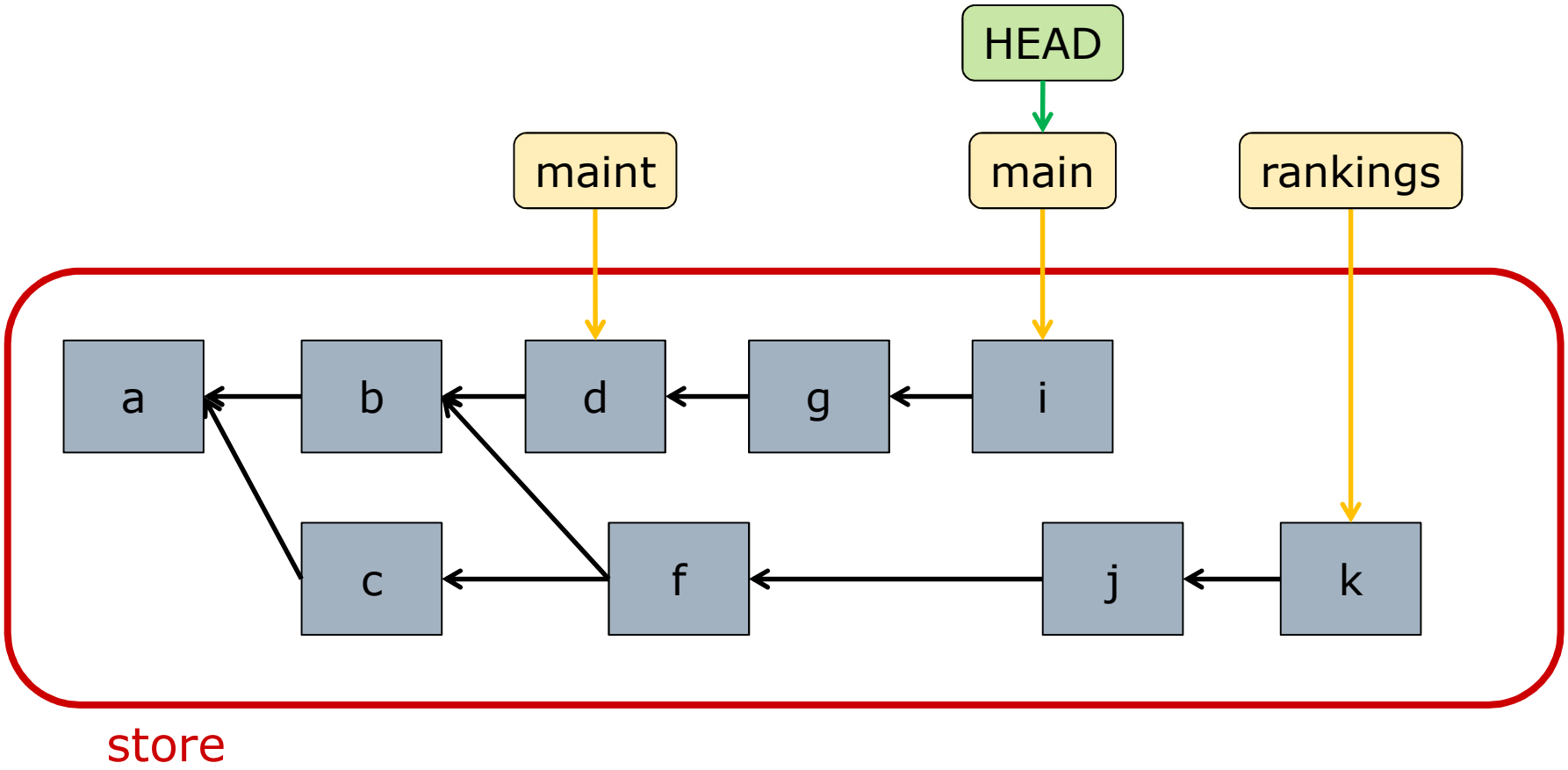


A Note on Changing Defaults

- 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
  - Recently this default became user-configurable! (git 2.28, 7/27/20)
  - Repos created on GitHub use “main” as the default name (as of 10/1/20)
  - Default name had been “master”

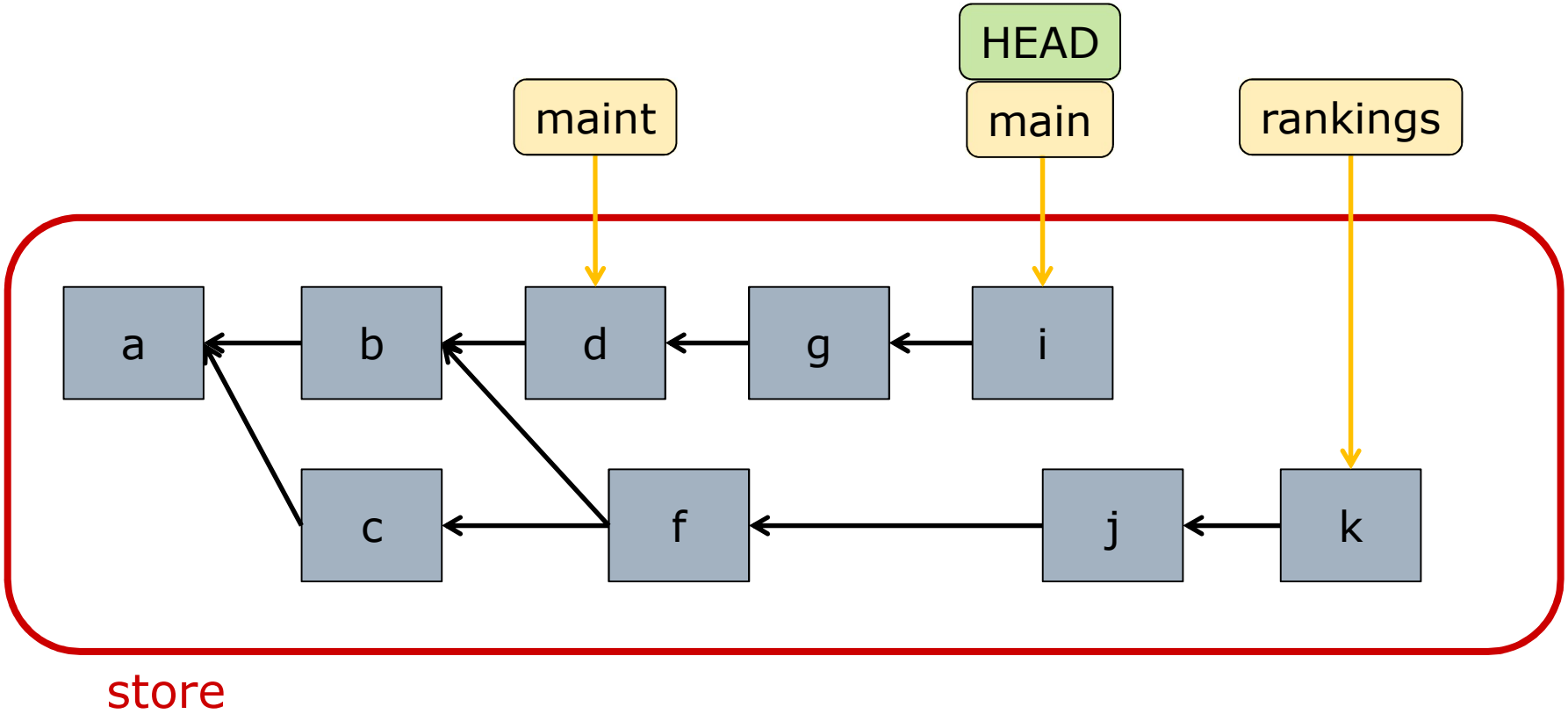
Nomenclature: HEAD

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



Nomenclature: HEAD

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

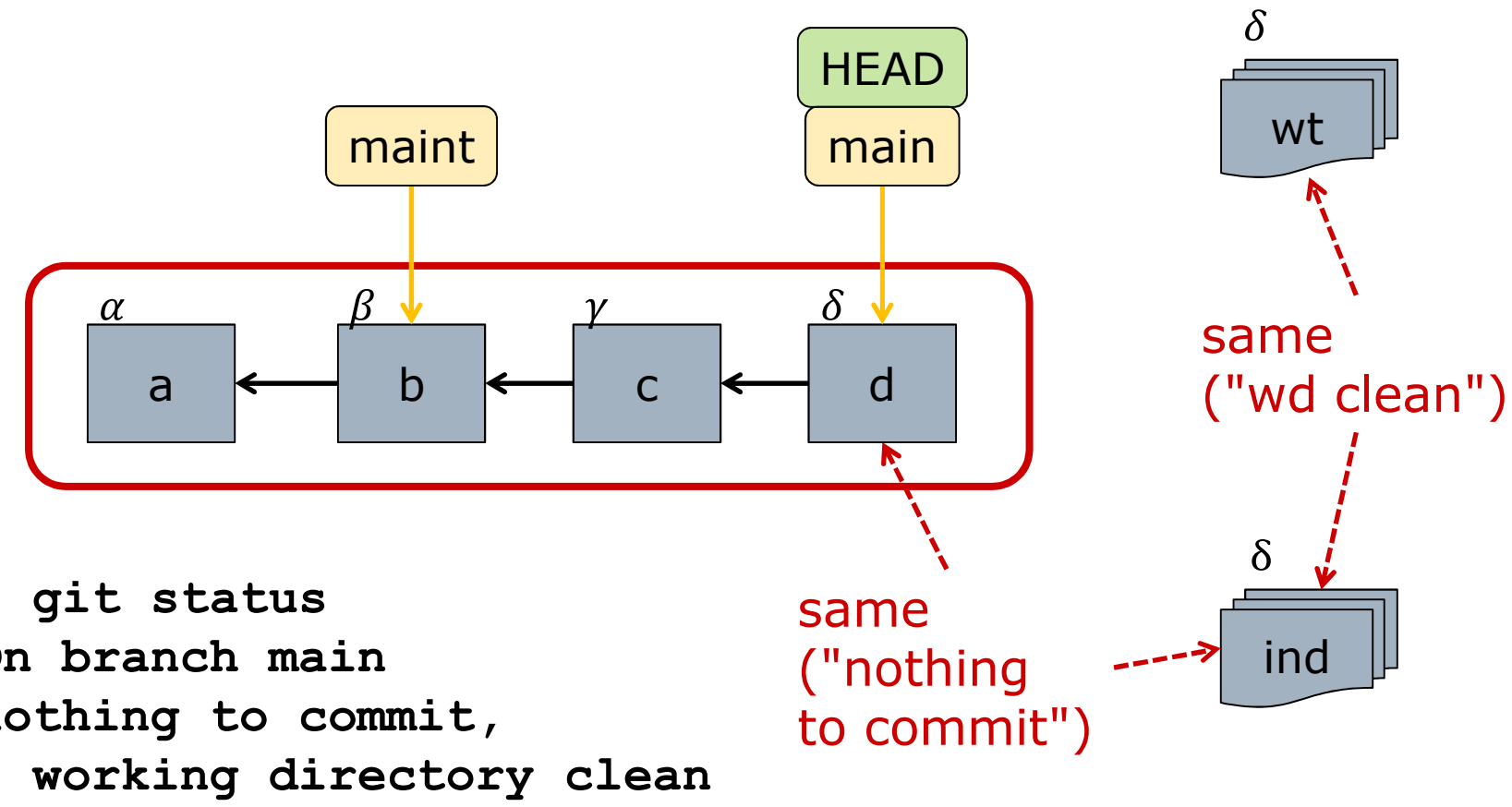


View of DAG with Branches


```
$ 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
```

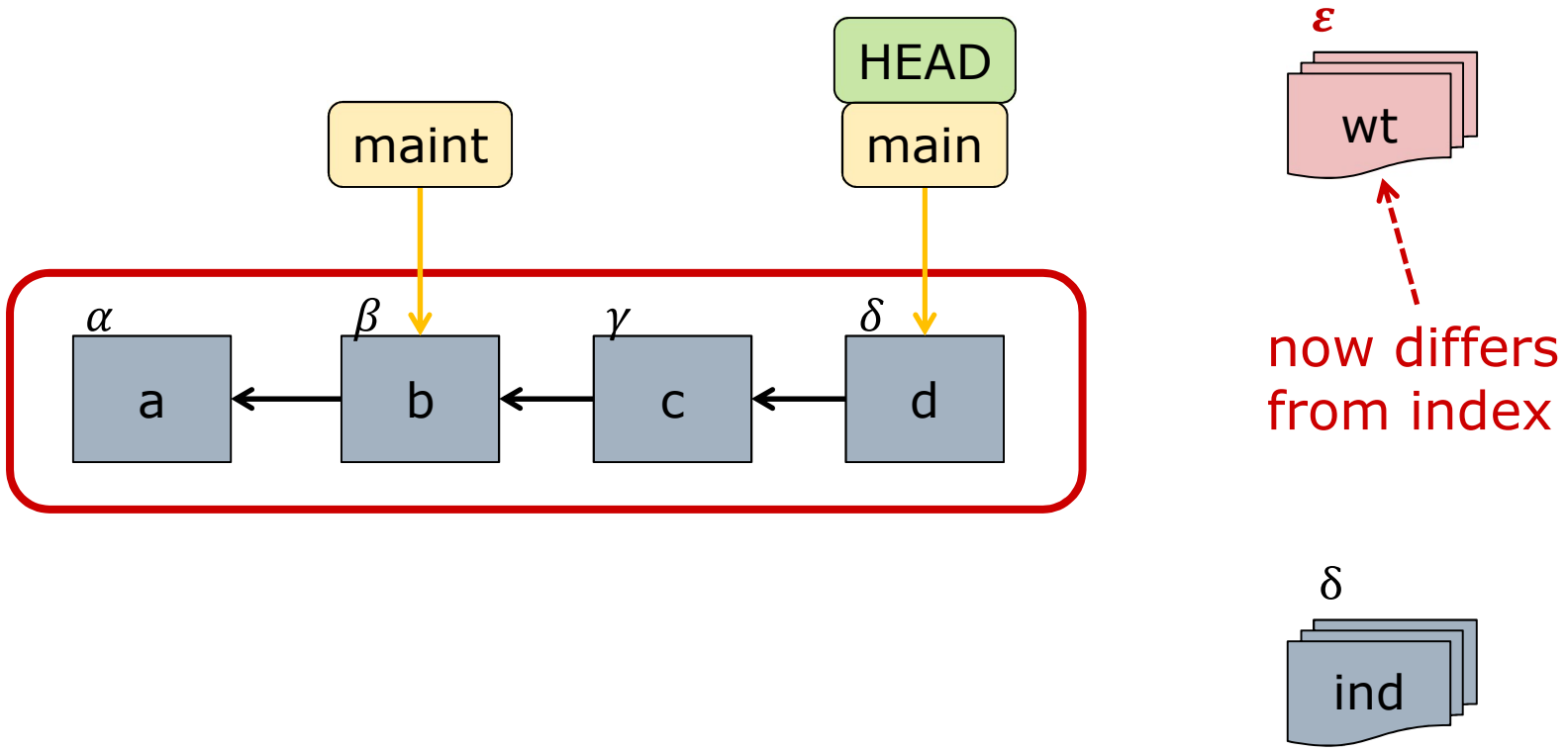
A "Clean" Repository




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

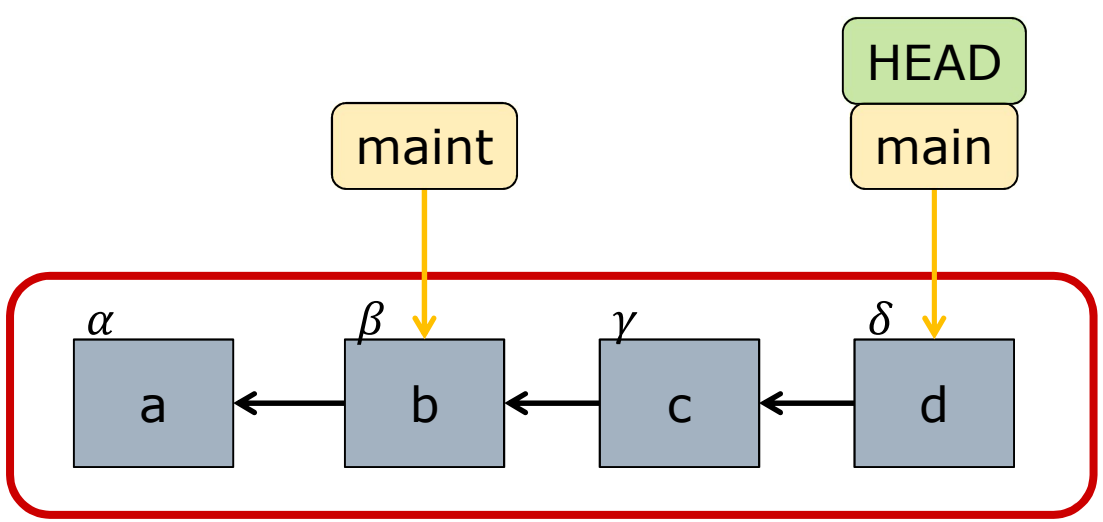
Edit Files in Working Tree

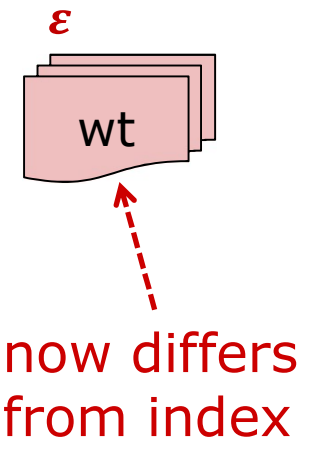
- Add files, remove files, edit files…

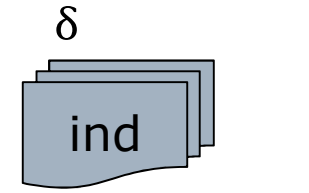


Edit Files in Working Tree

- Add files, remove files, edit files…



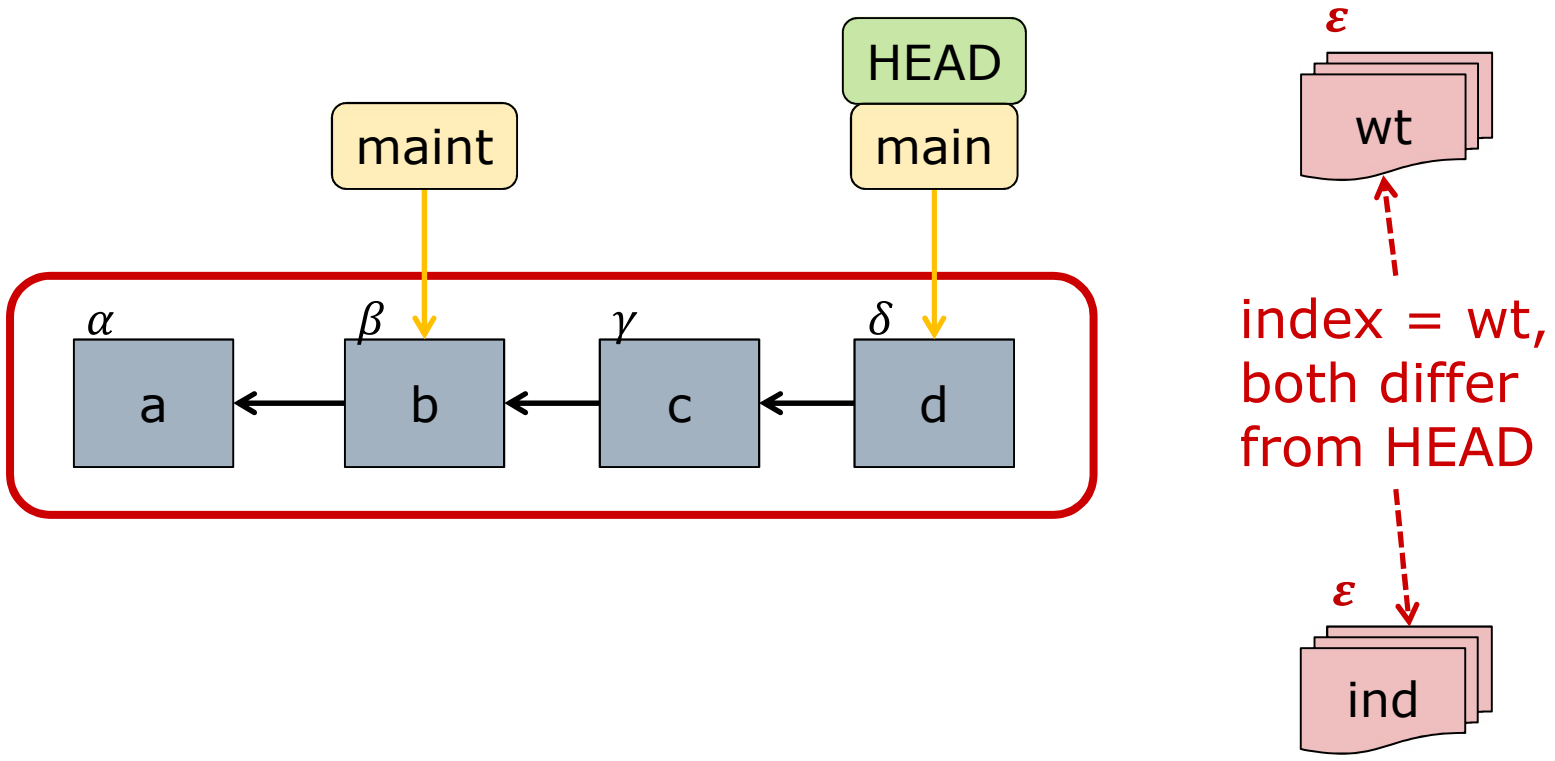





```
$ git status
On branch main
Changes not staged for commit:
    modified: css/demo.css
```

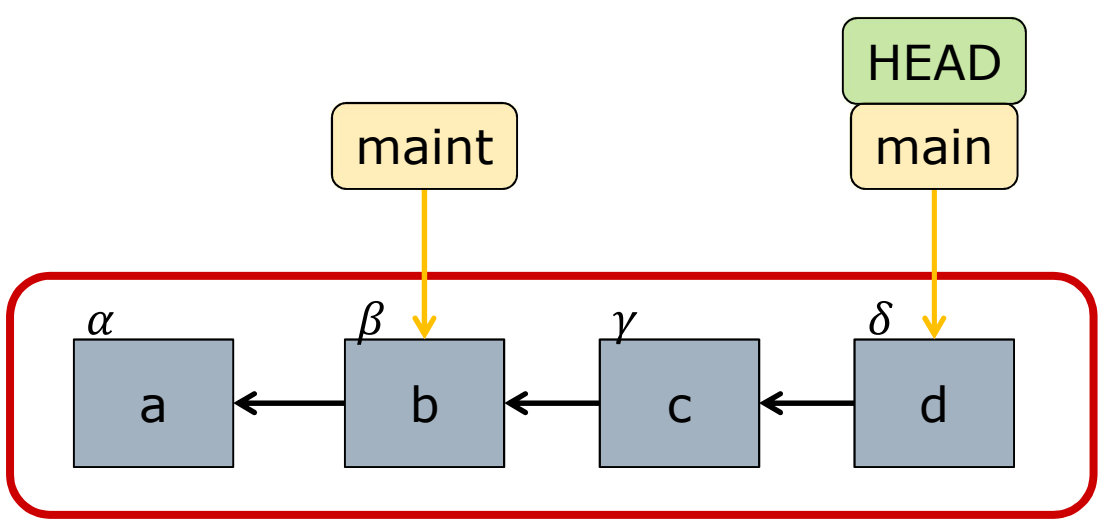
Add: Working Tree → Index

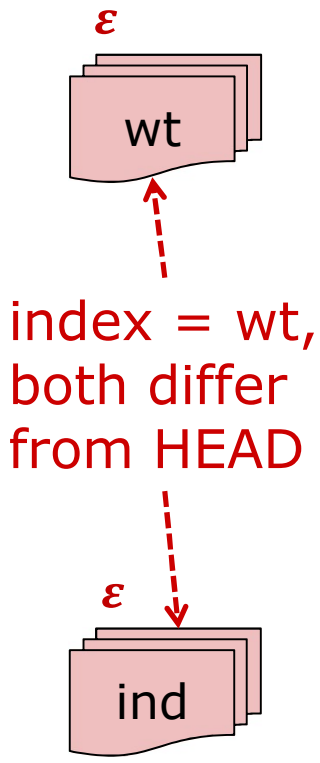

```
$ git add --all .
```



Add: Working Tree → Index


```
$ git add --all .
```

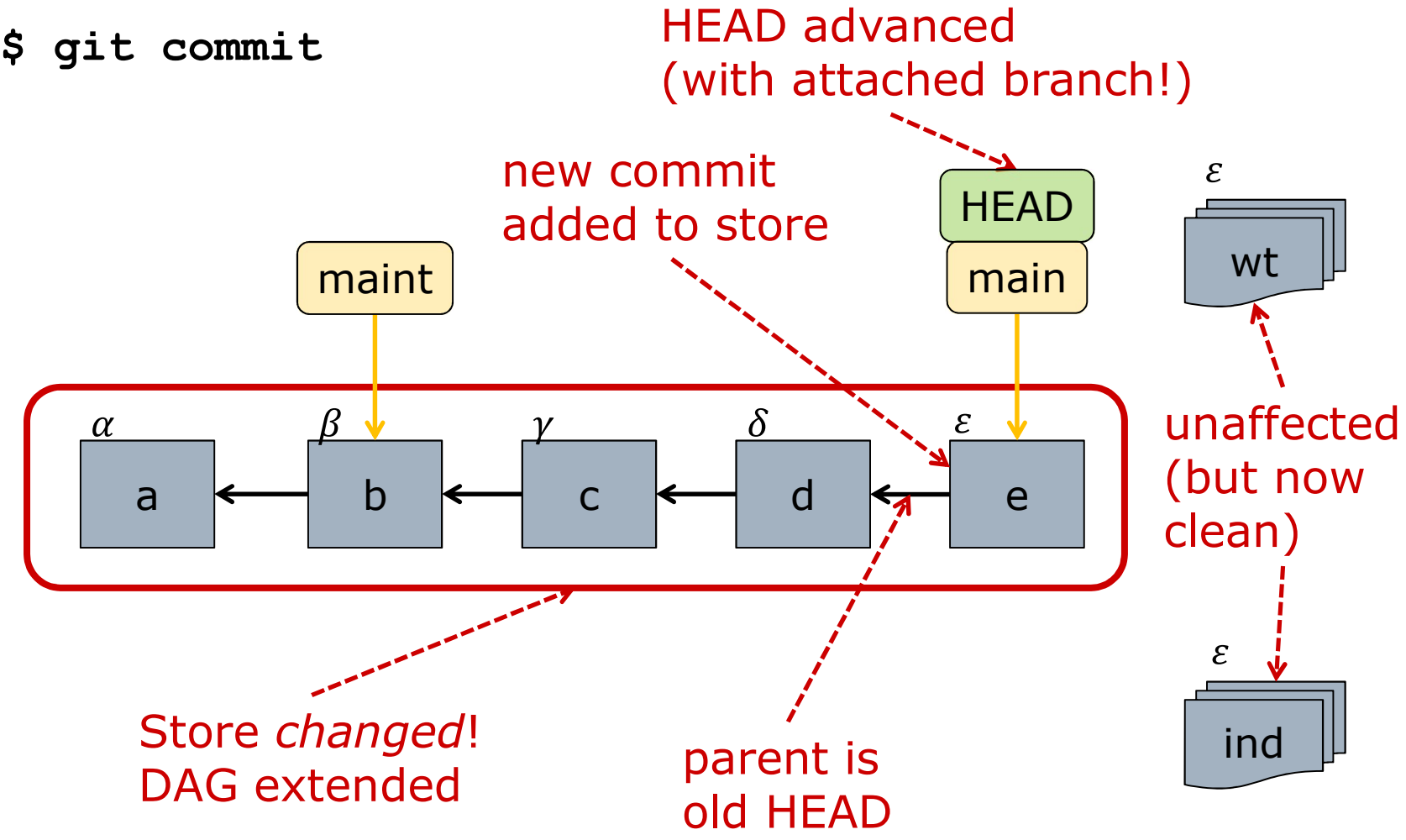




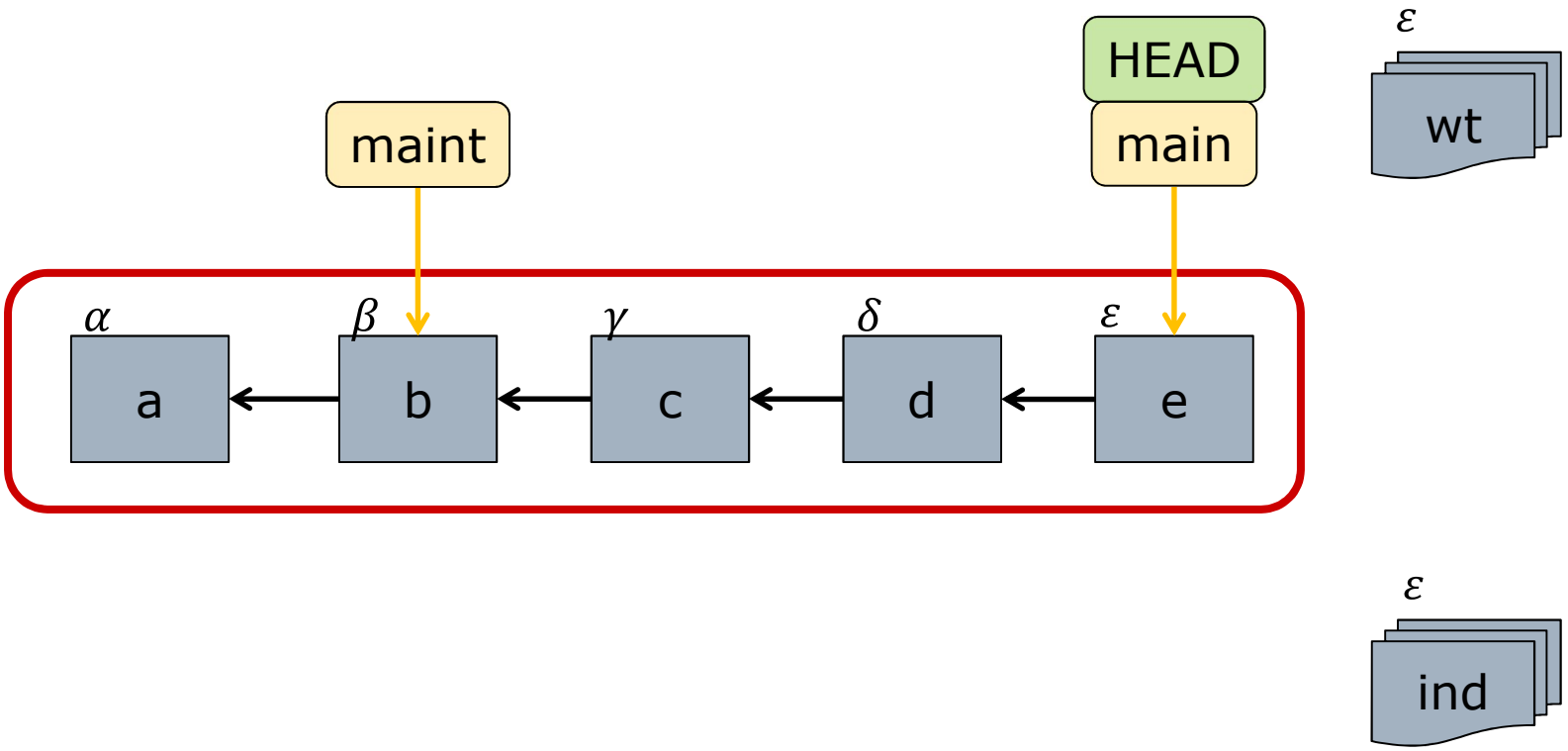

```
$ git status
On branch main
Changes to be committed:
    modified: css/demo.css
```

Commit: Index → Store


```
$ git commit
```

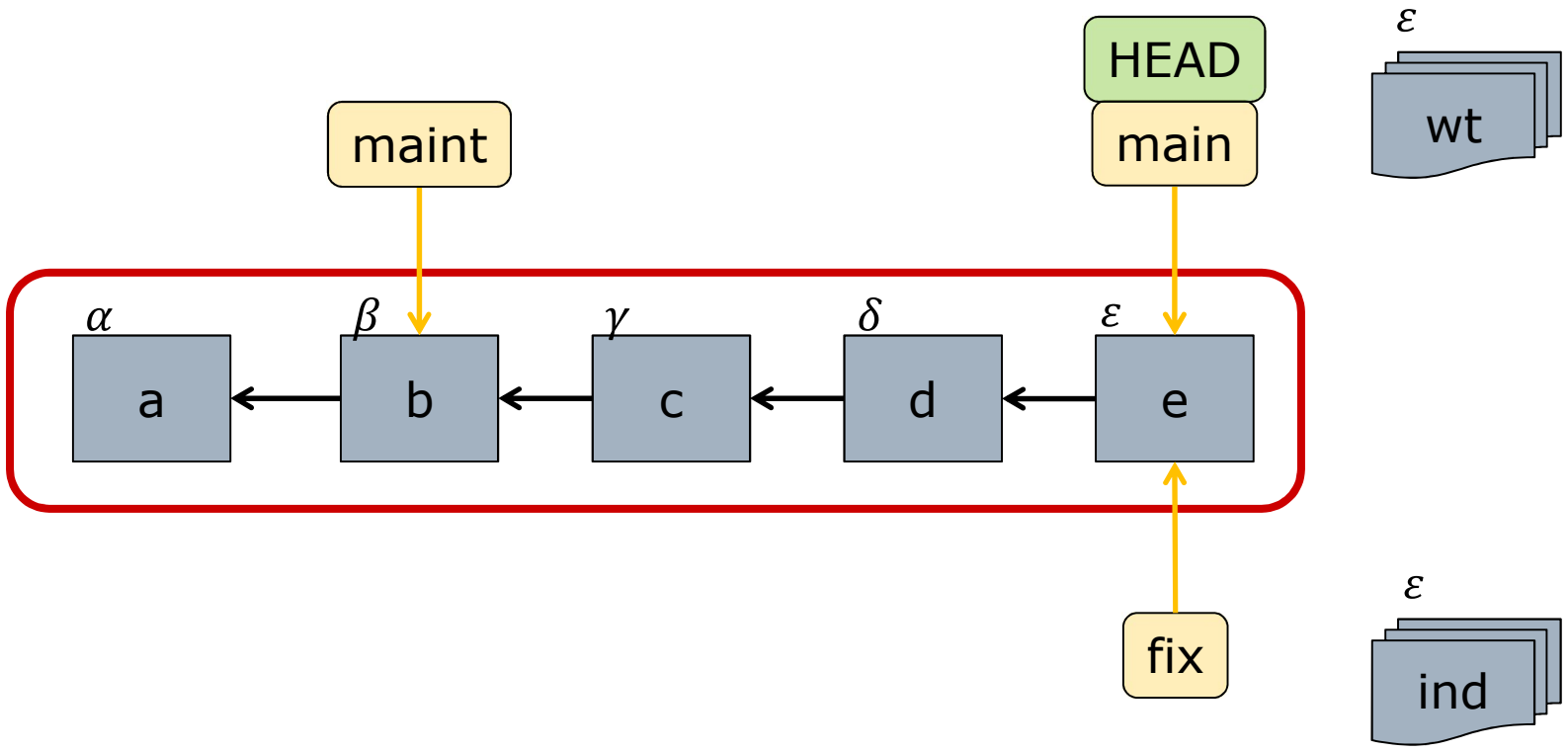


The (New) State of Repository



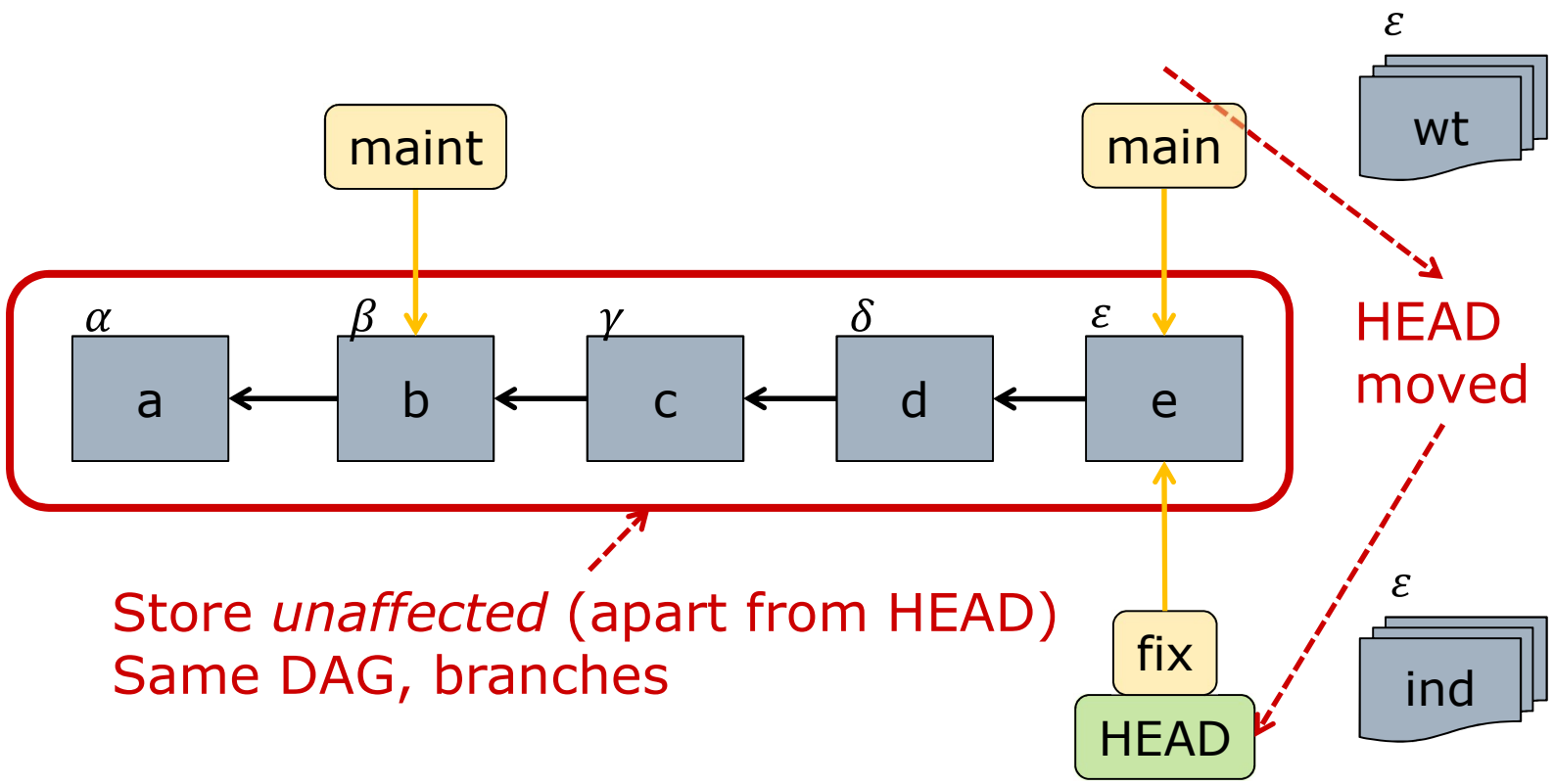
Creating a New Branch


```
$ git branch fix
```



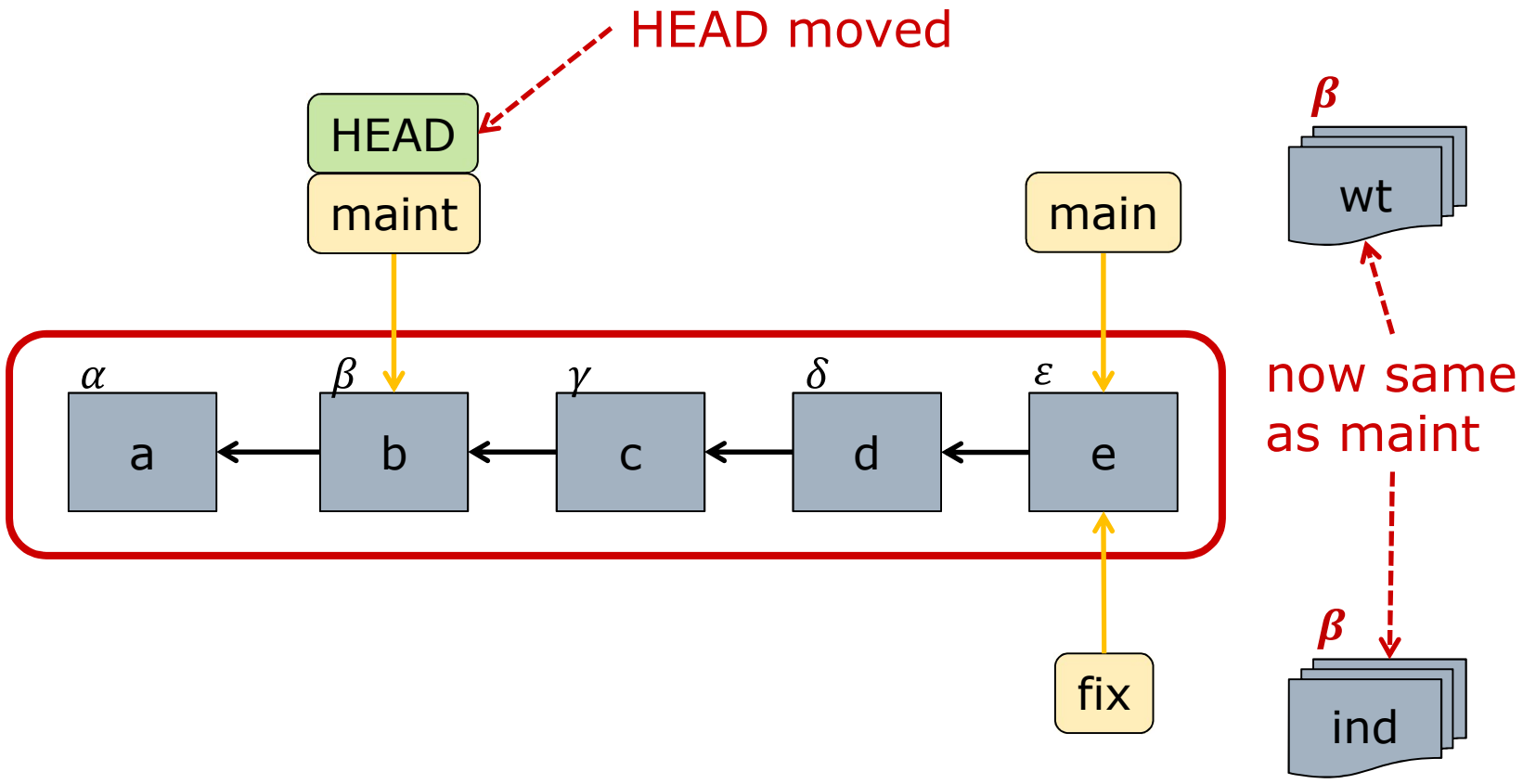
Checkout: Changing Branch


```
$ git checkout fix
```



Checkout: Changing Branch

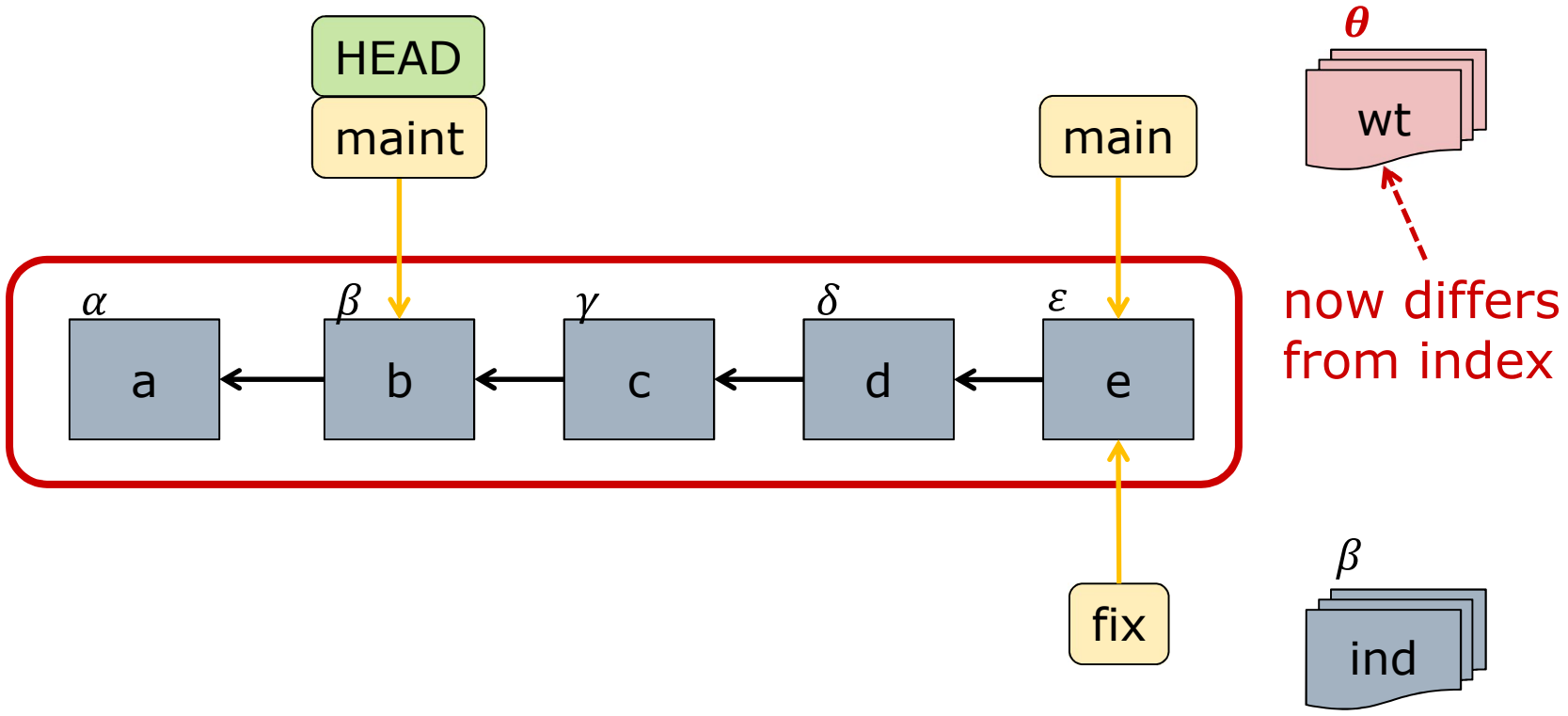

```
$ git checkout maint
```



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

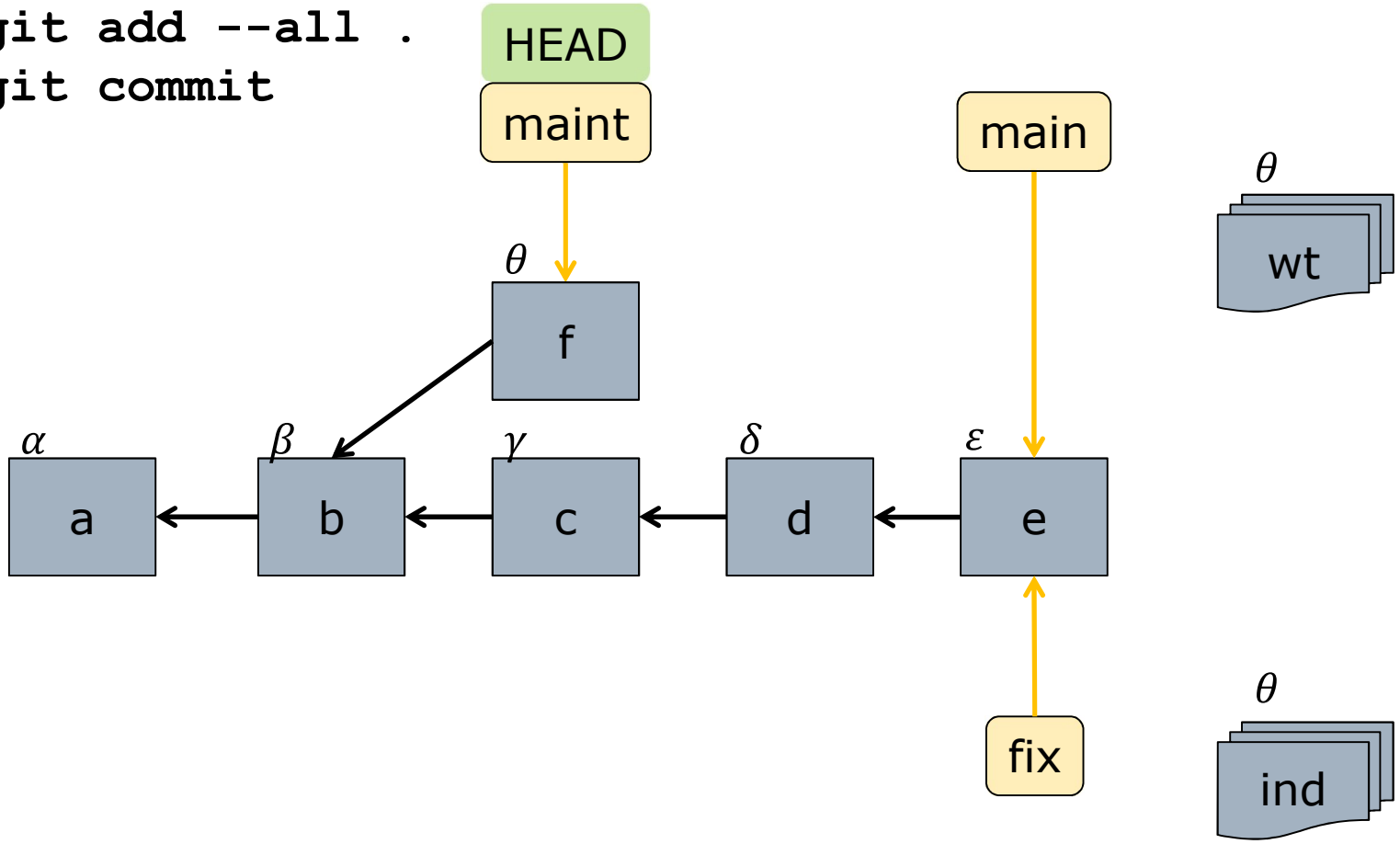
Edit Files in Working Tree

- Add files, remove files, edit files…



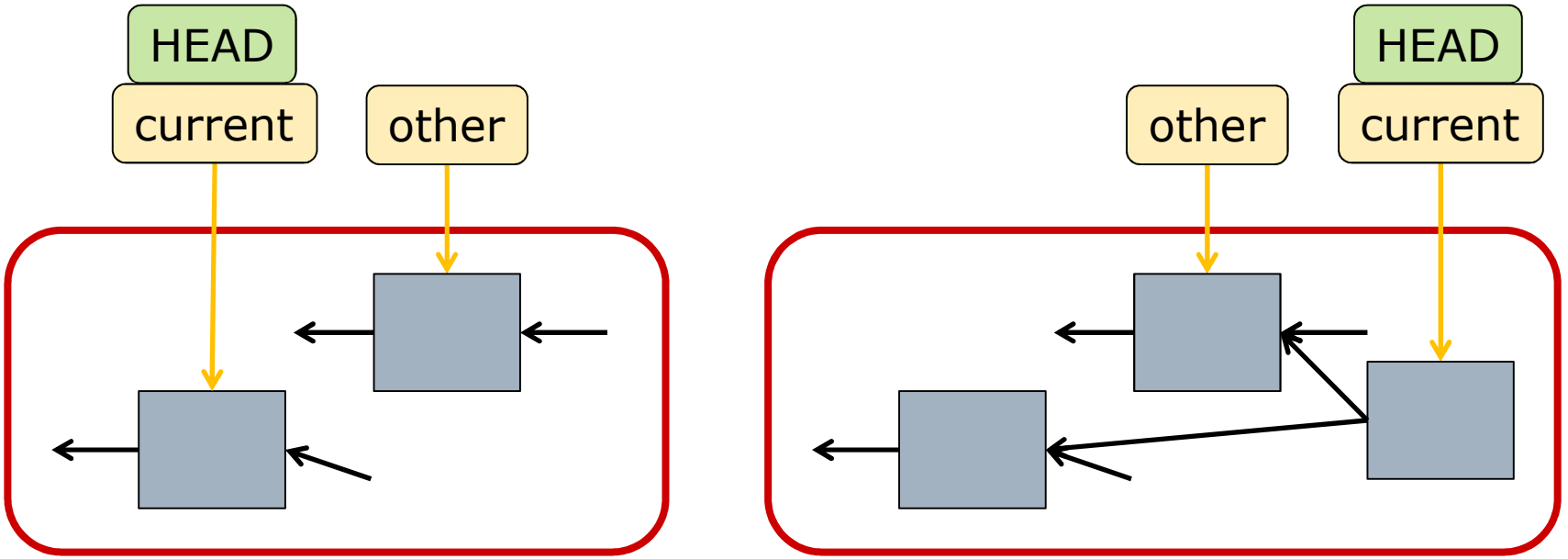
Add & Commit: Update Store


```
$ git add --all .
$ git commit
```



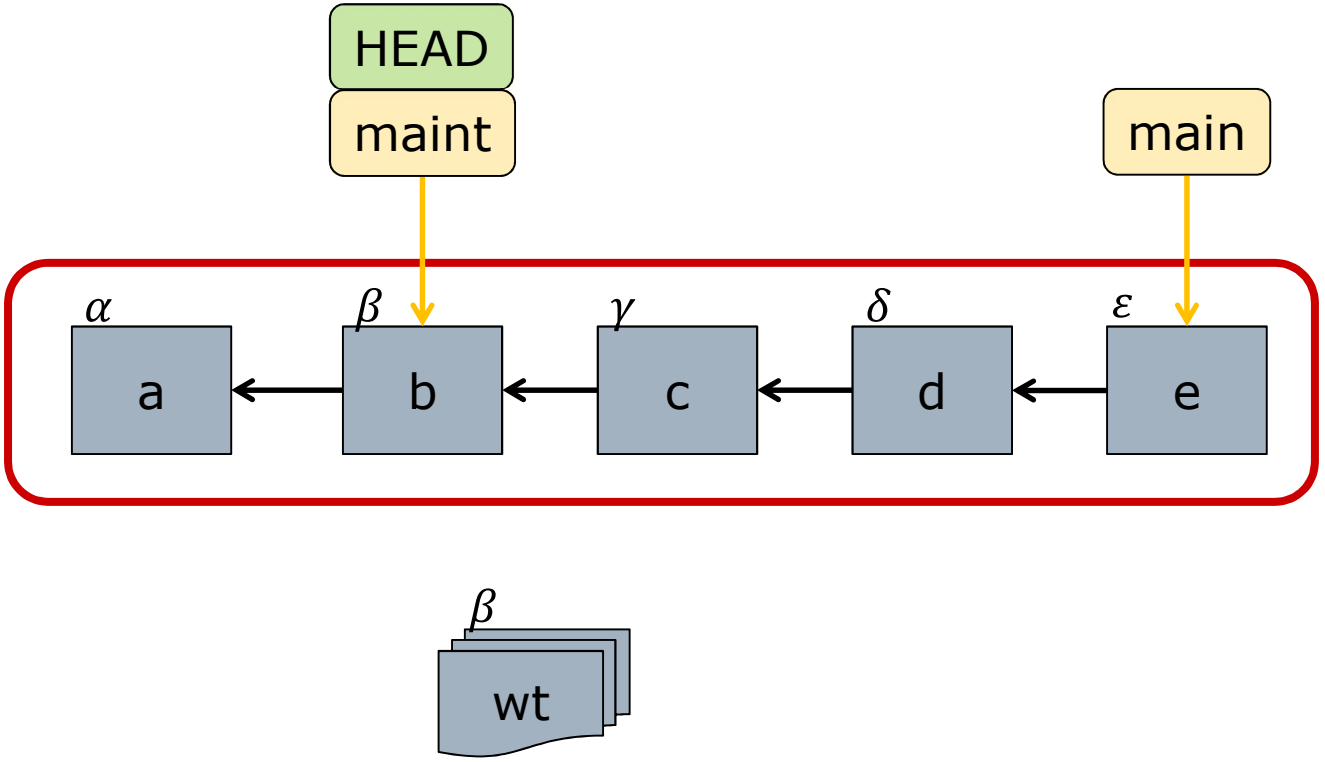
Merge: Bringing History together

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



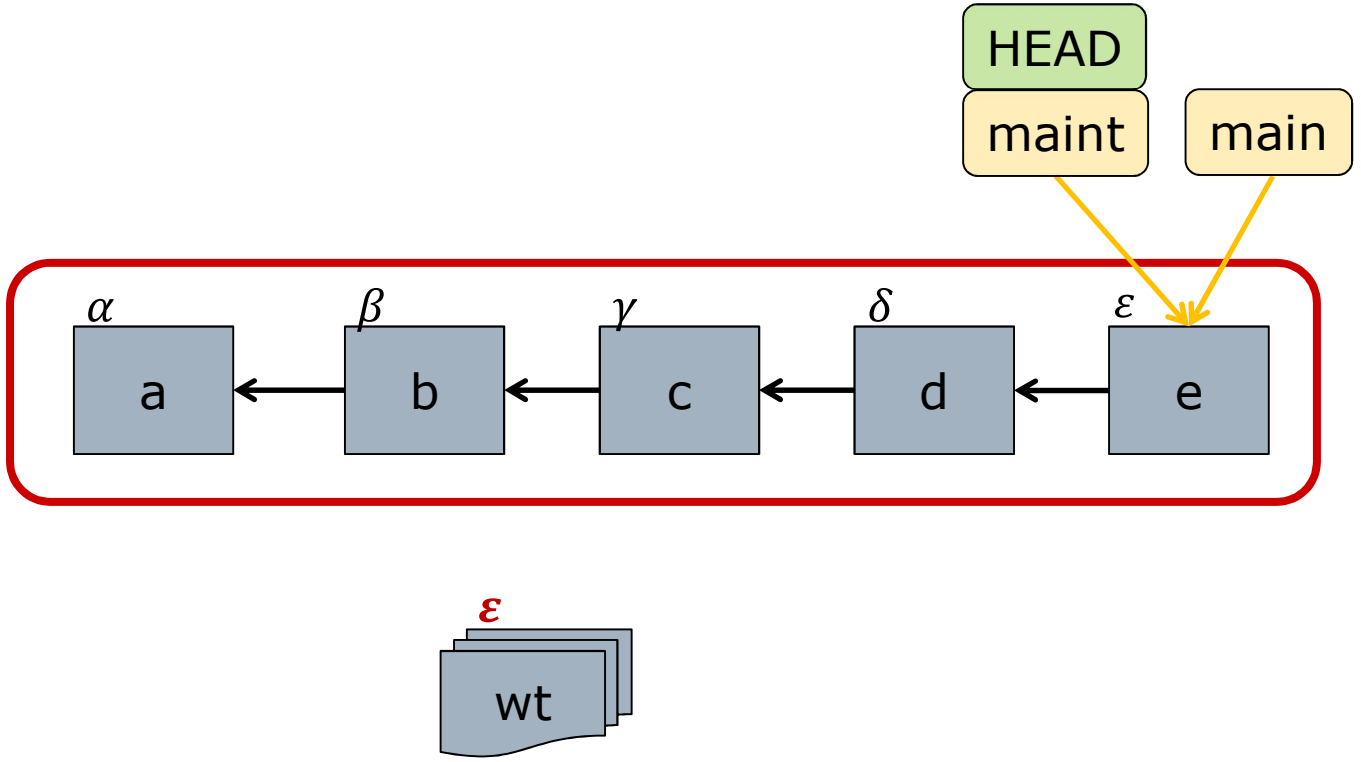
Merge – Case 1: Ancestor

- HEAD is an ancestor of other branch

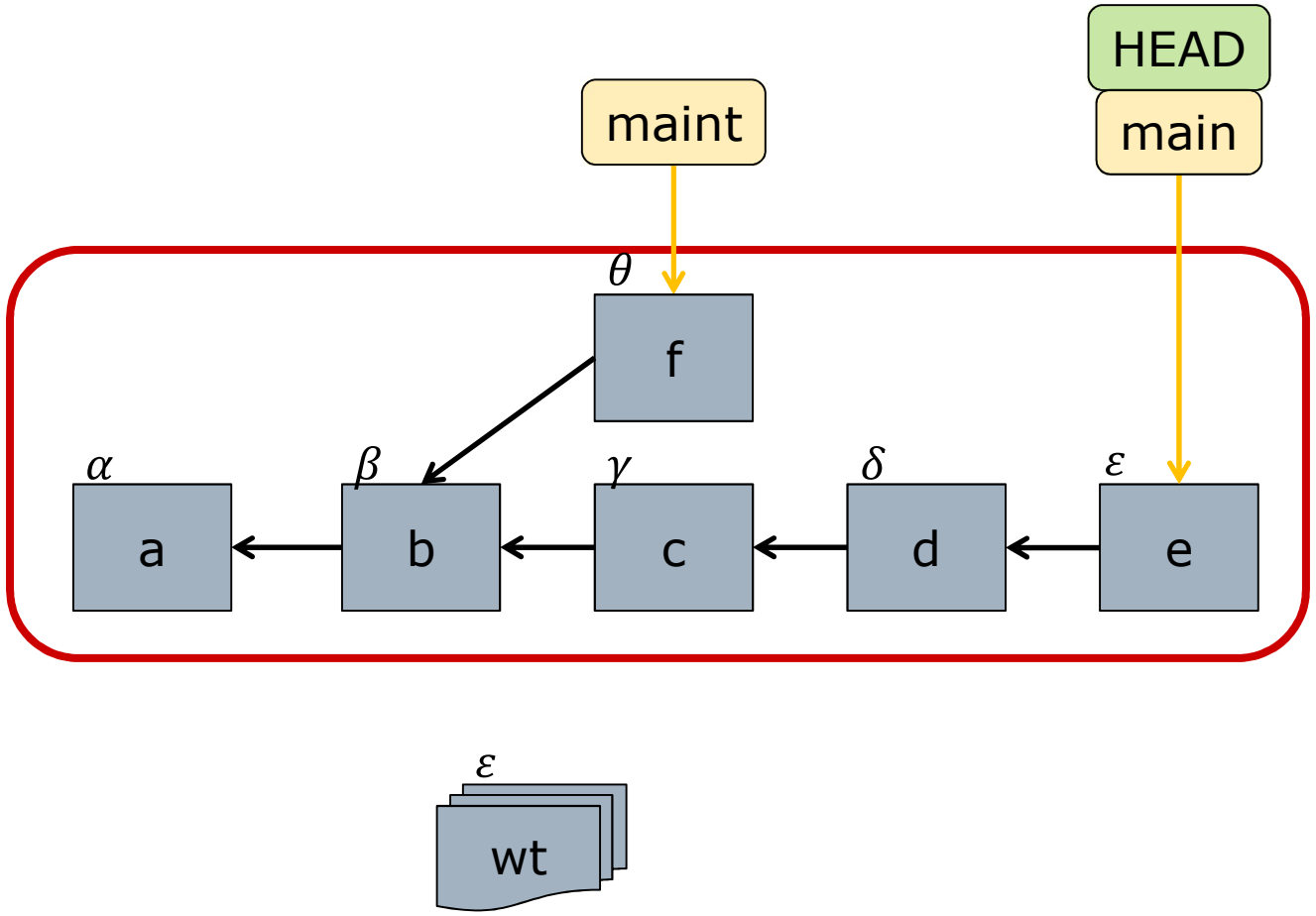


Fast-Forward Merge


```
$ git merge main
```

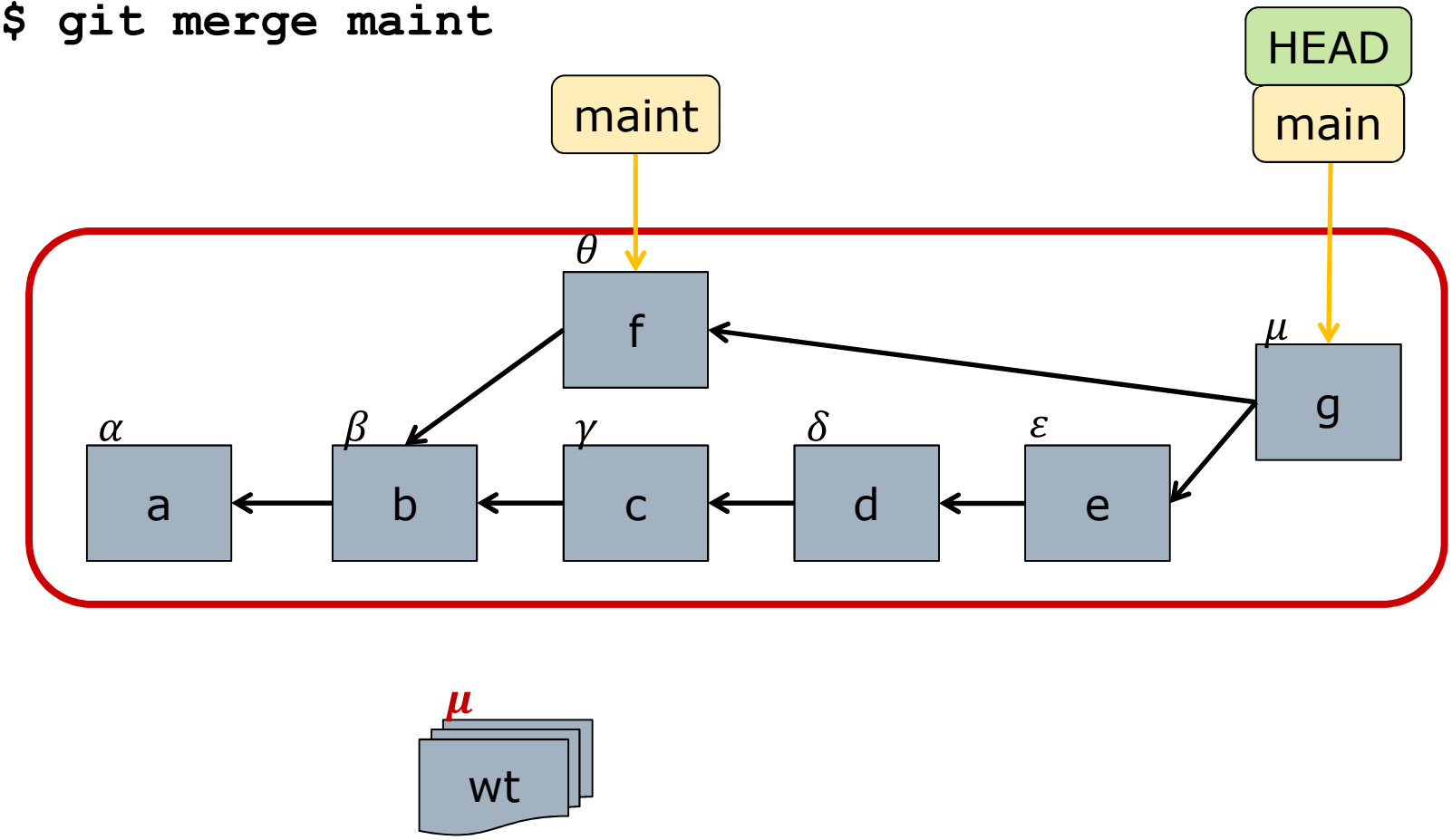


Merge – Case 2: No Conflicts



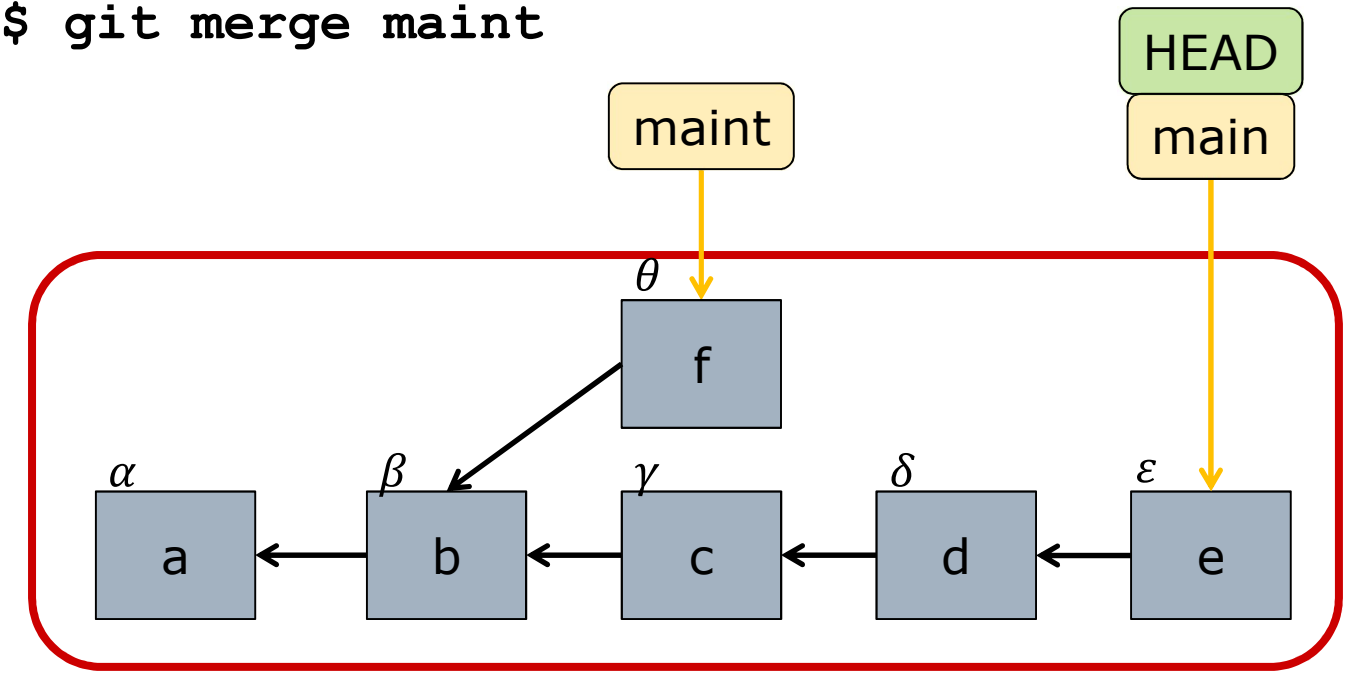
Merge Automatically Commits


```
$ git merge maint
```

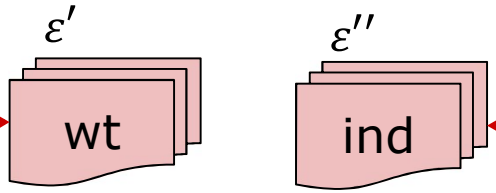


Merge – Case 3: Conflicts Exist


```
$ git merge maint
```



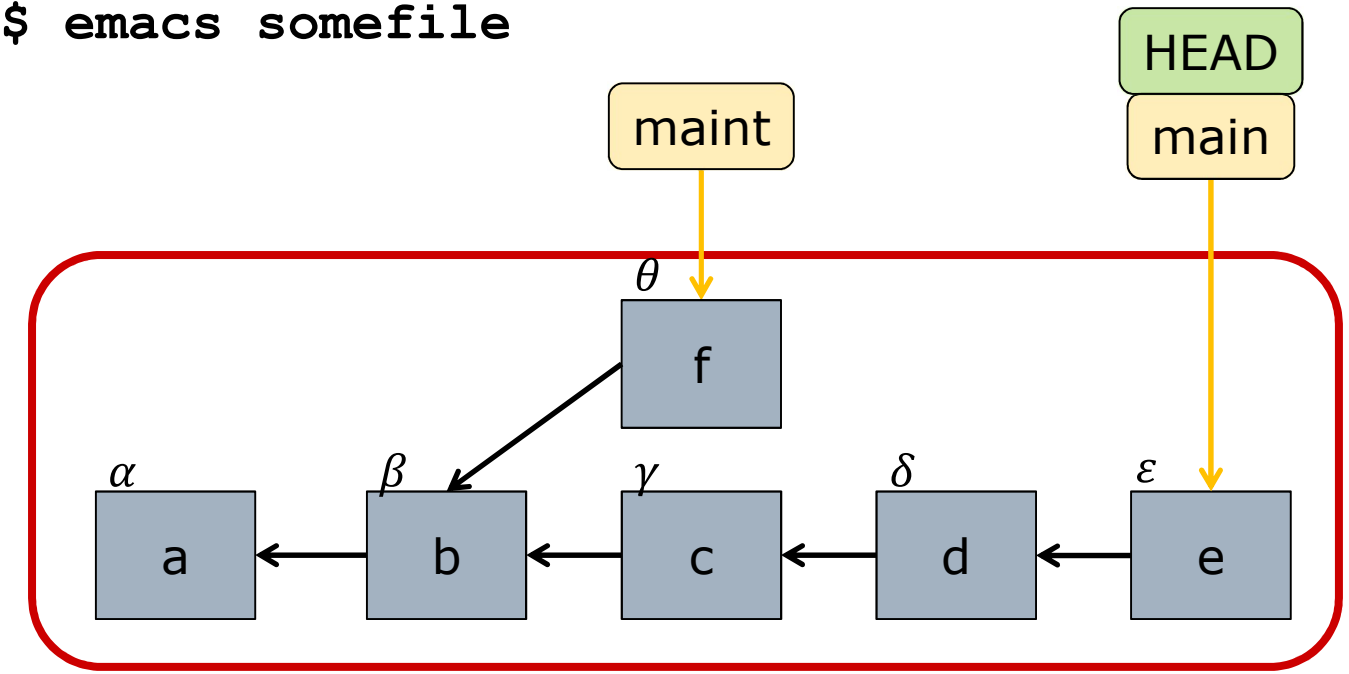
files with conflicts marked



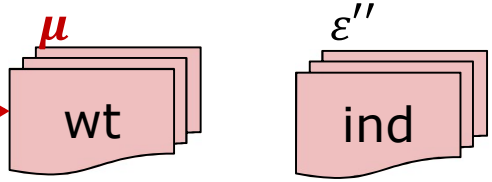
files that could be merged automatically

Merge: Resolve Conflicts


```
$ emacs somefile
```

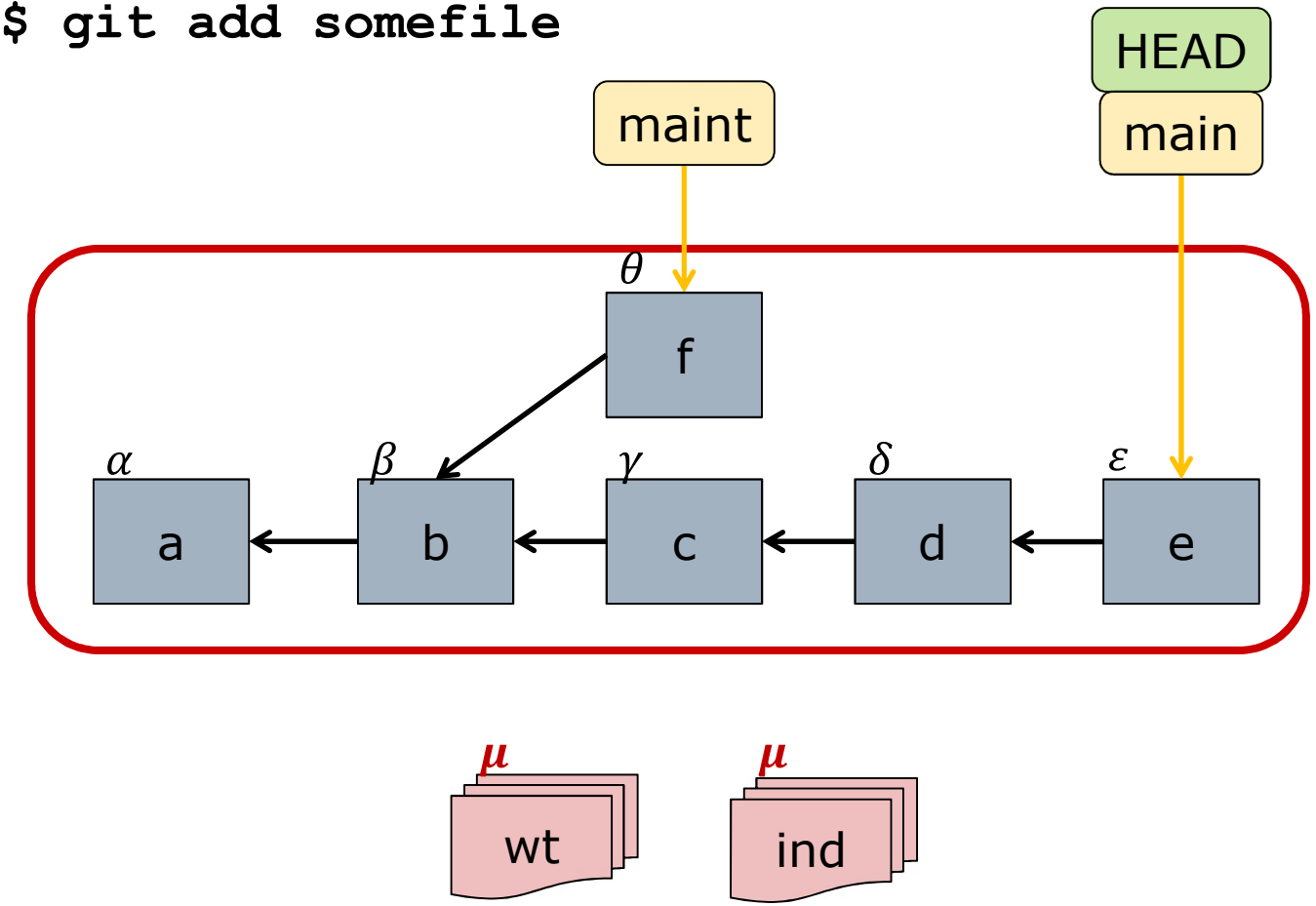


files with conflicts resolved



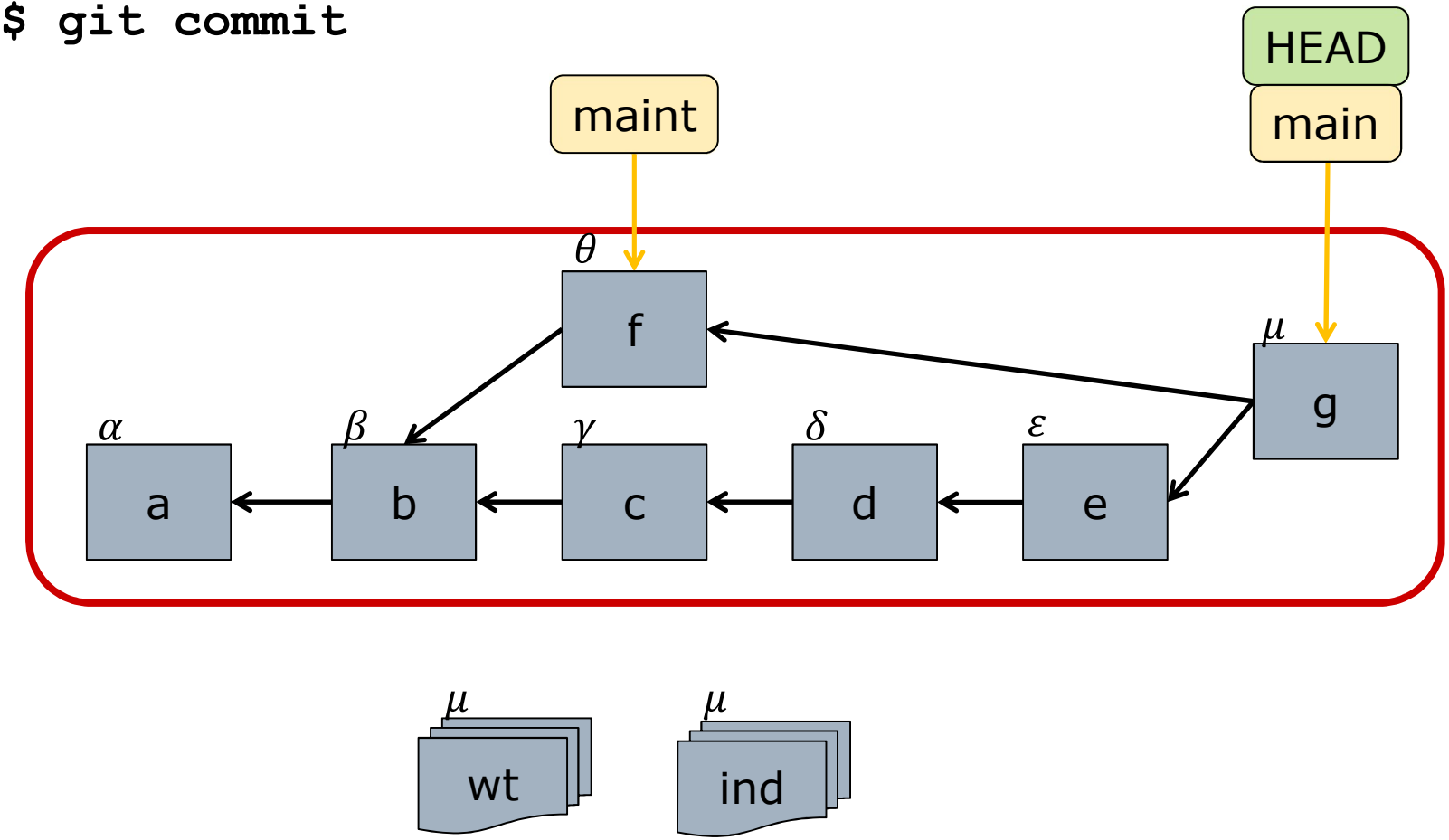
Merge with Conflicts: Add


```
$ git add somefile
```



Merge with Conflicts: Commit


```
$ git commit
```



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

Lecture 3

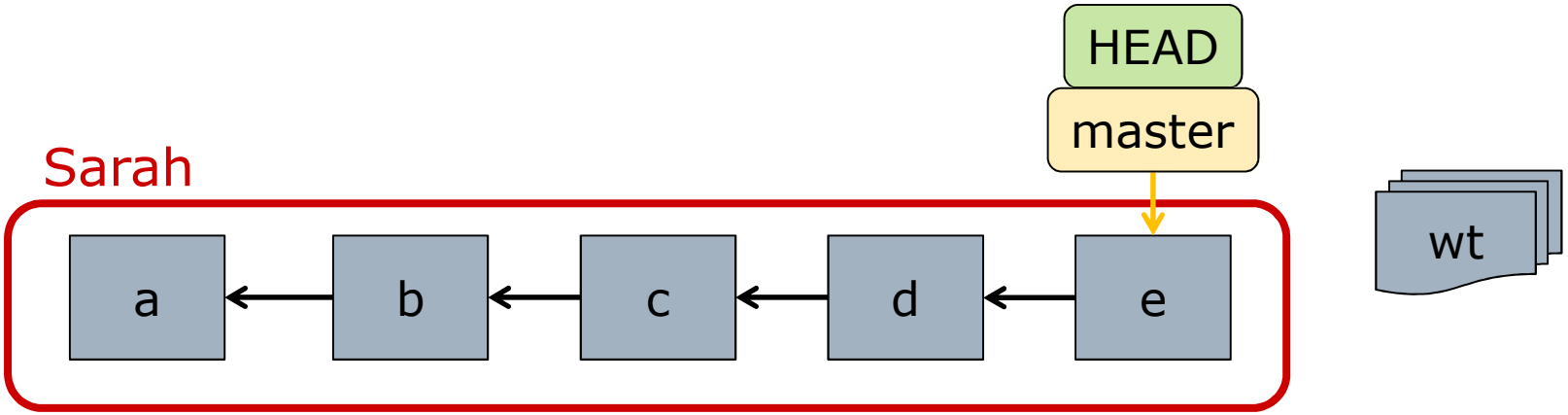
Demo

- 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

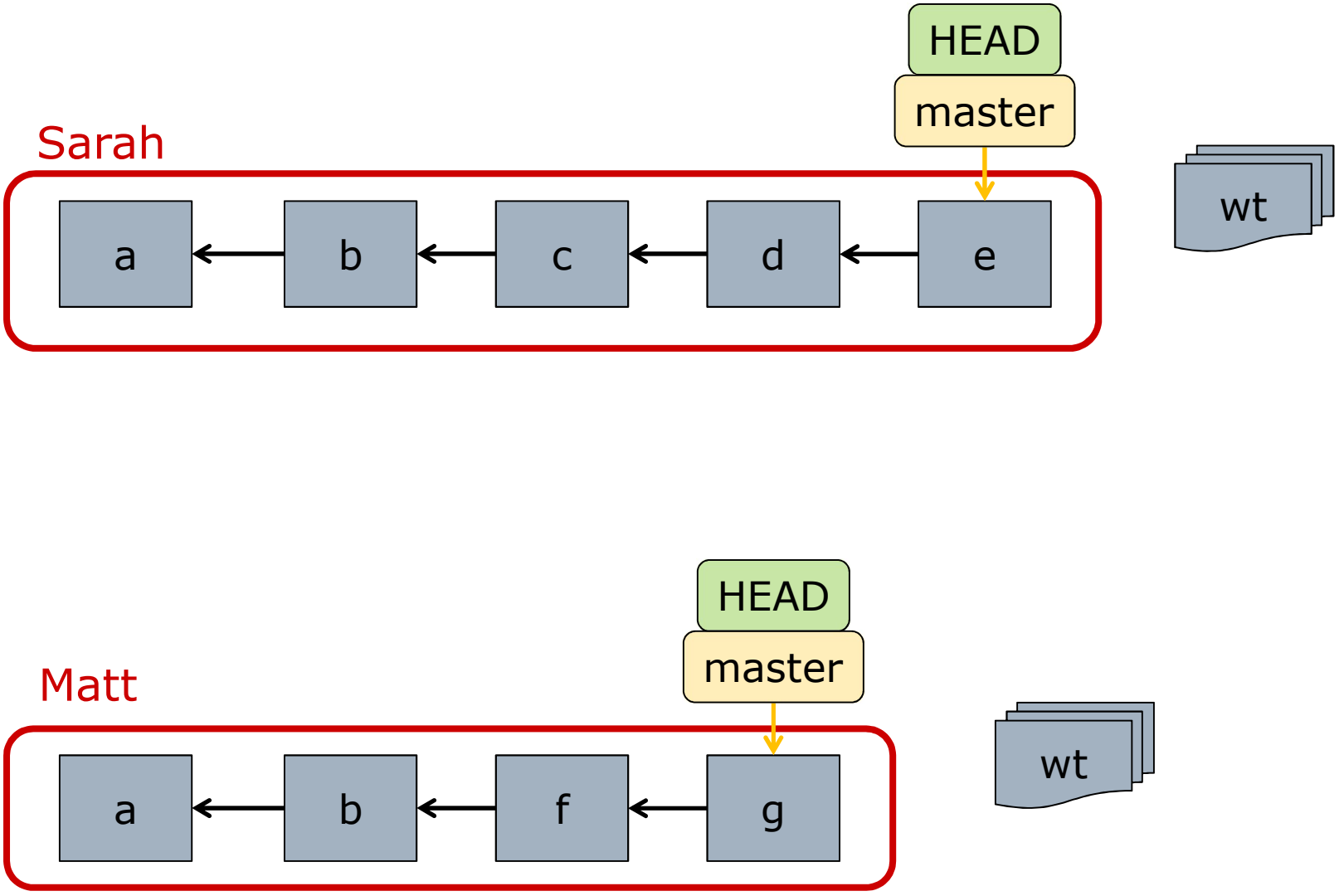
What Does "D" Stand For?

- *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

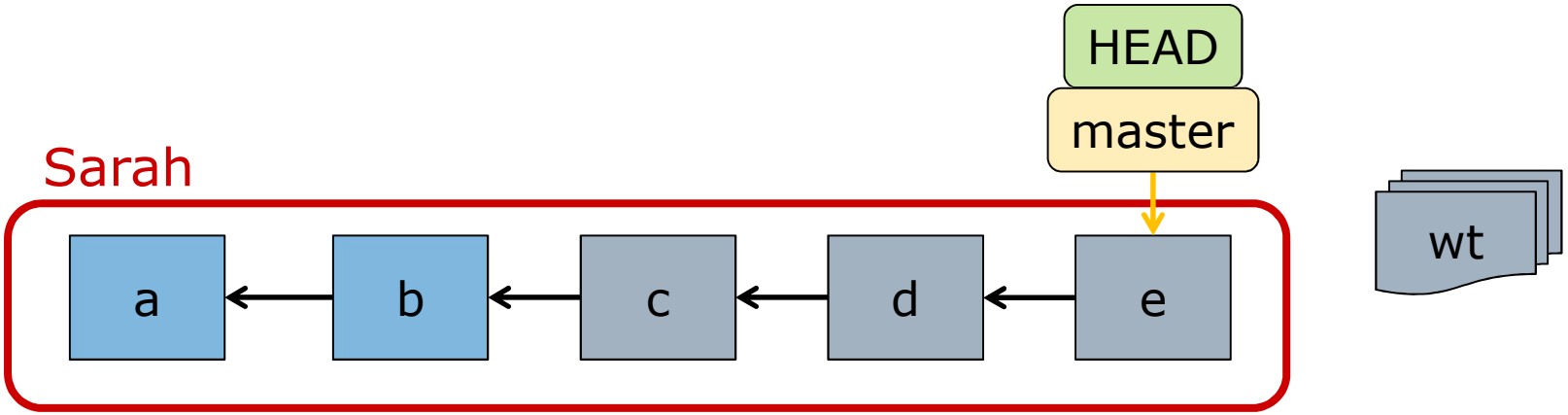
Sarah's Repository

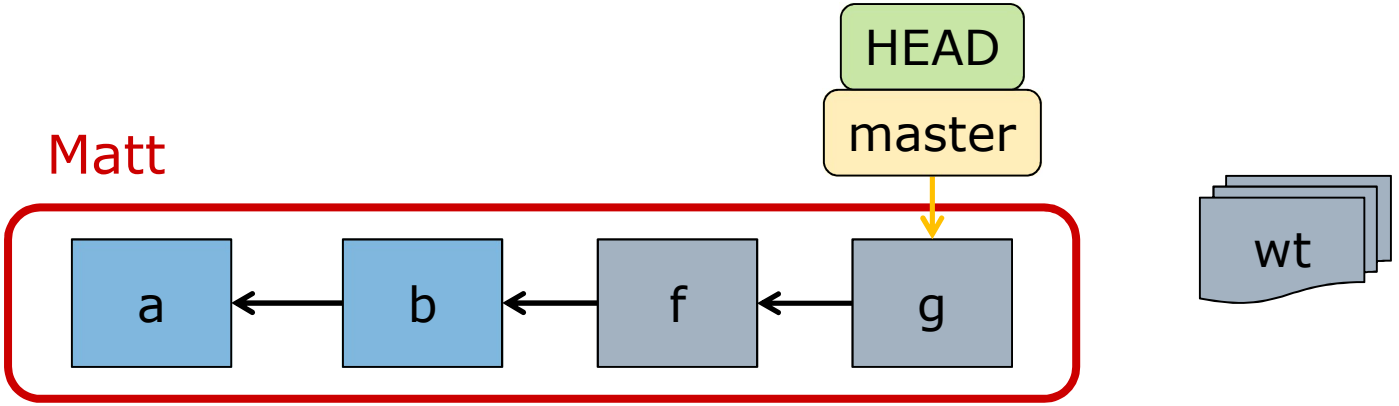


And Matt's Repository

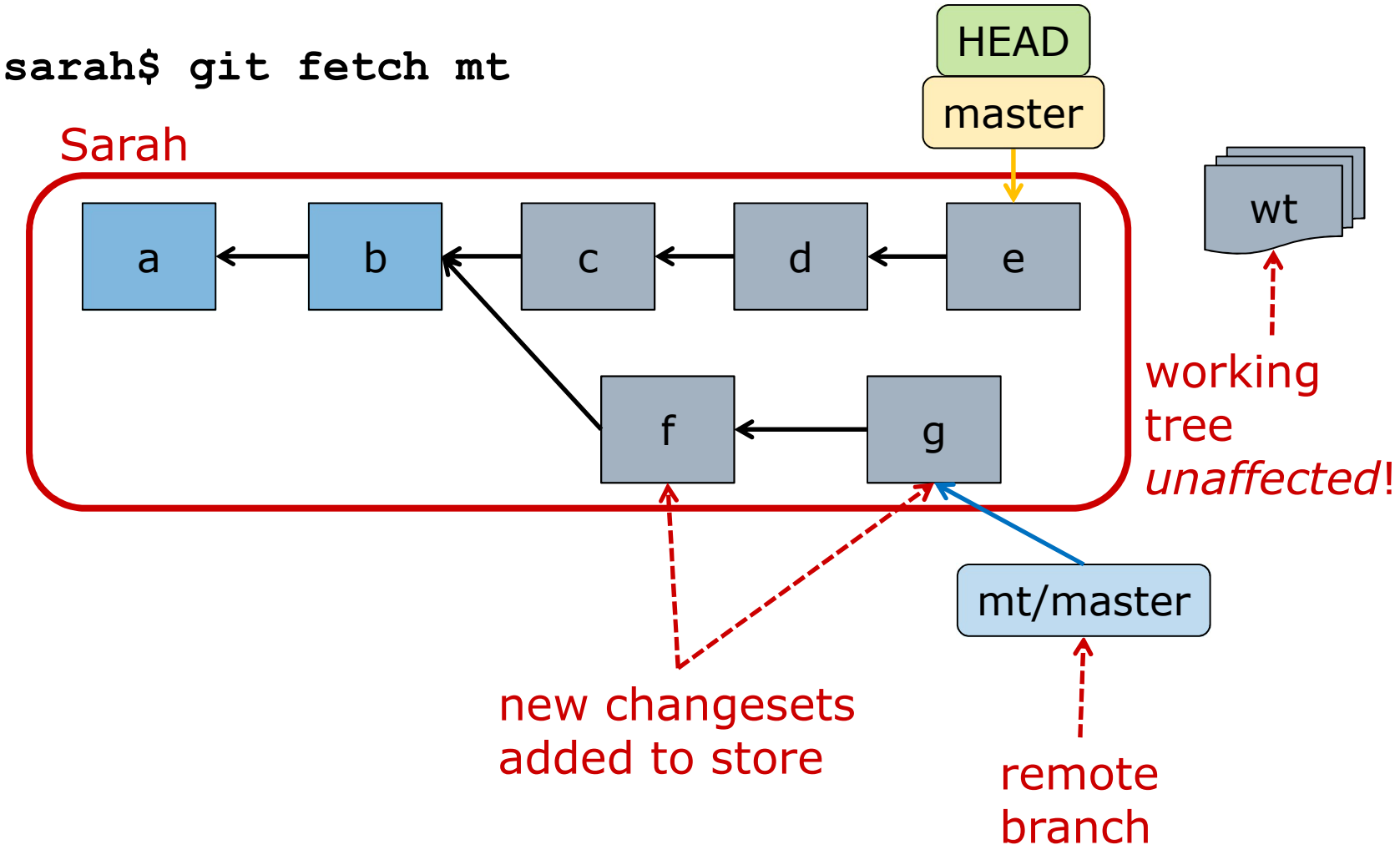


Some Shared History

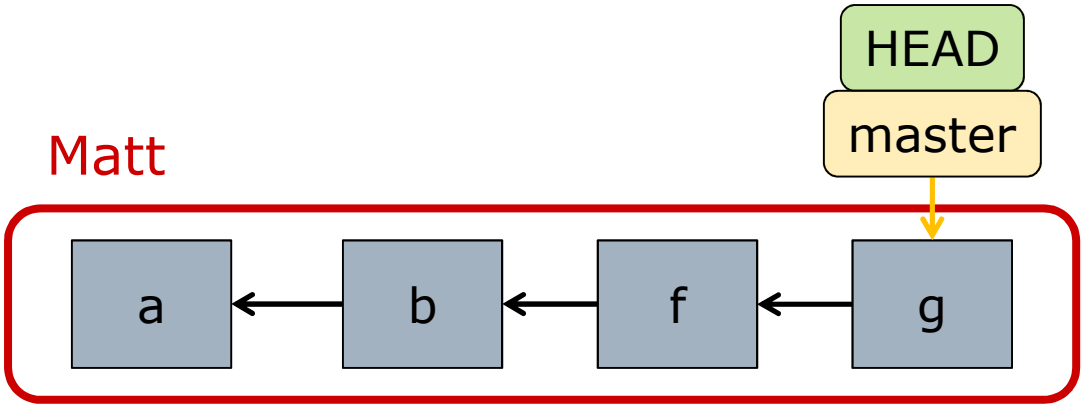




Fetch: Remote Store → Local

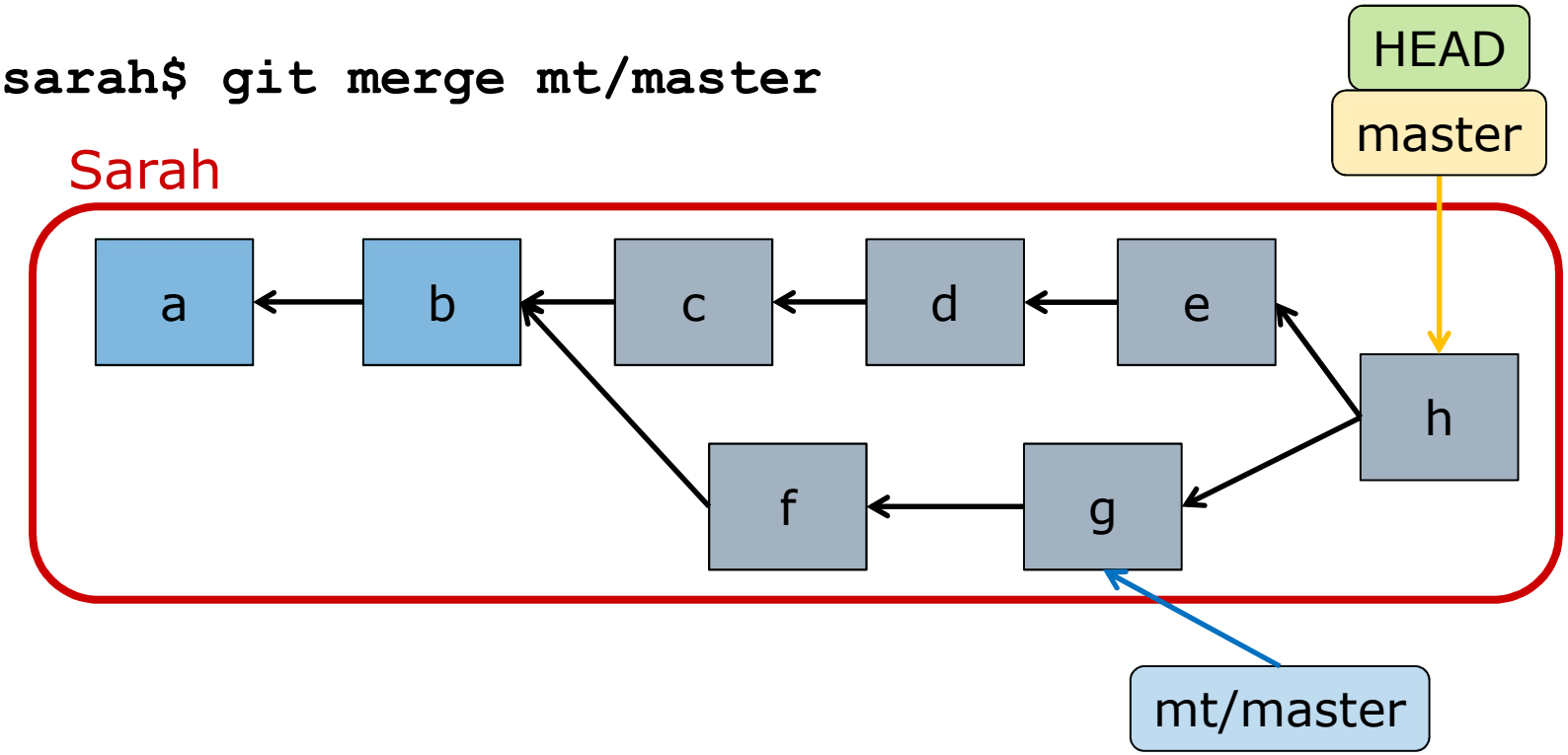


Remote Repository Unchanged

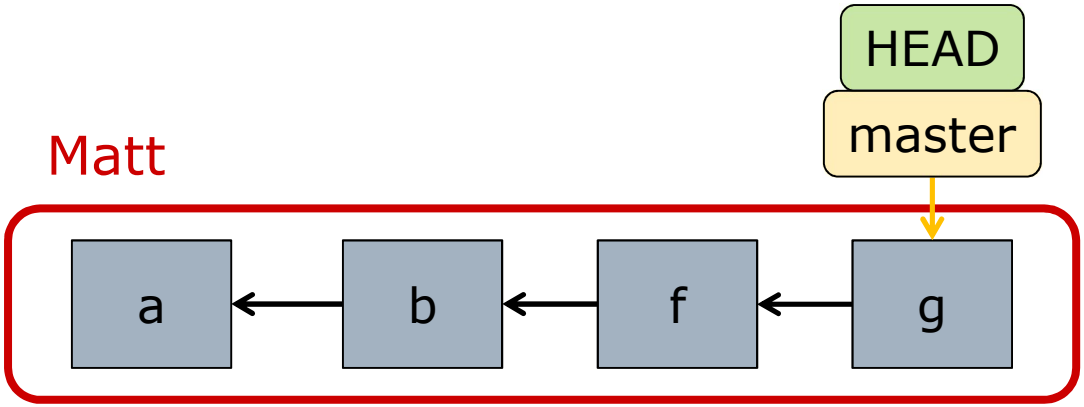


Workflow: Merge After Fetch


```
sarah$ git merge mt/master
```



Remote Repository Unchanged



View of DAG with All Branches

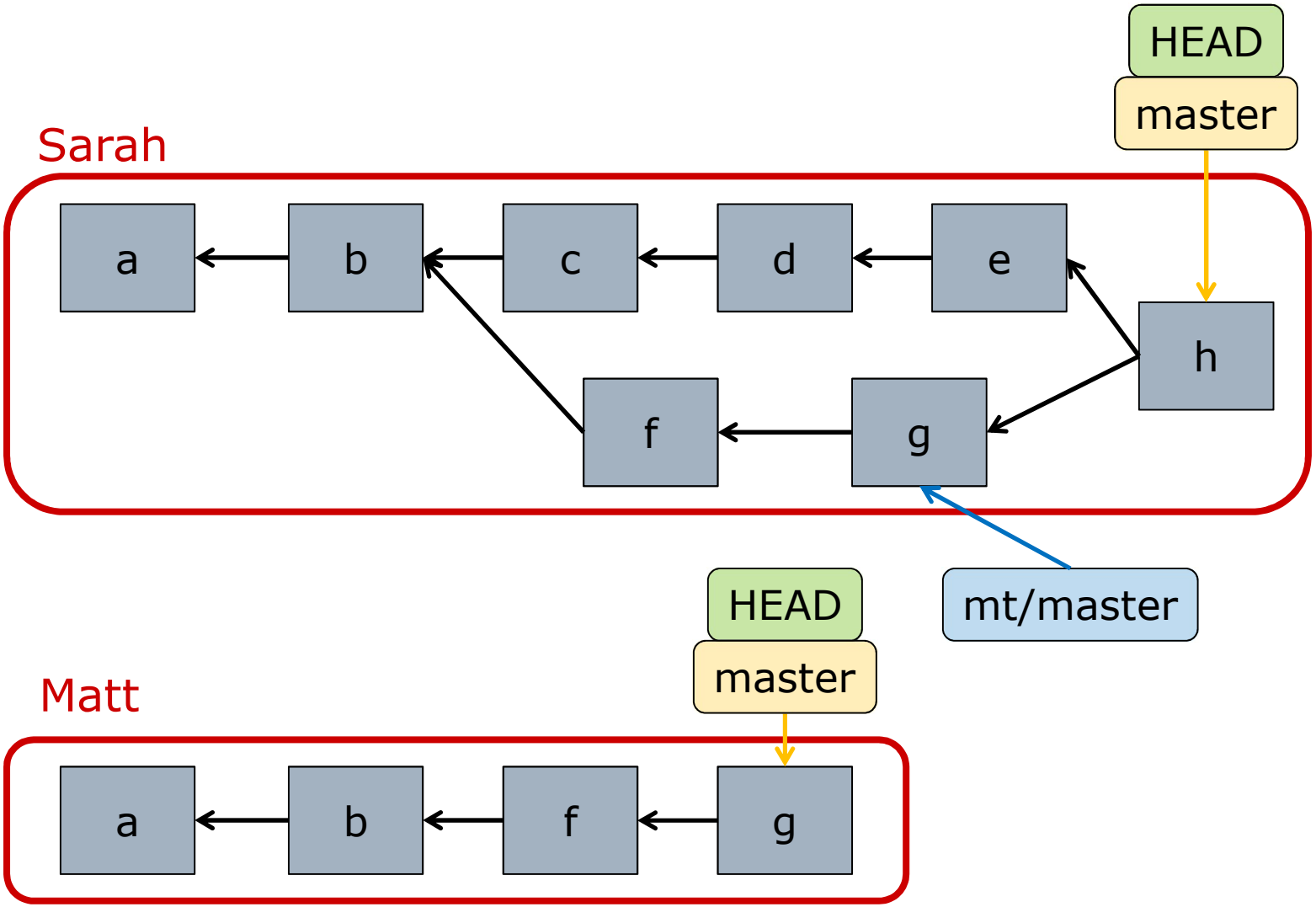

```
$ git log --oneline --graph --decorate --all

* 1618849 (HEAD-> master, origin/master) 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
```

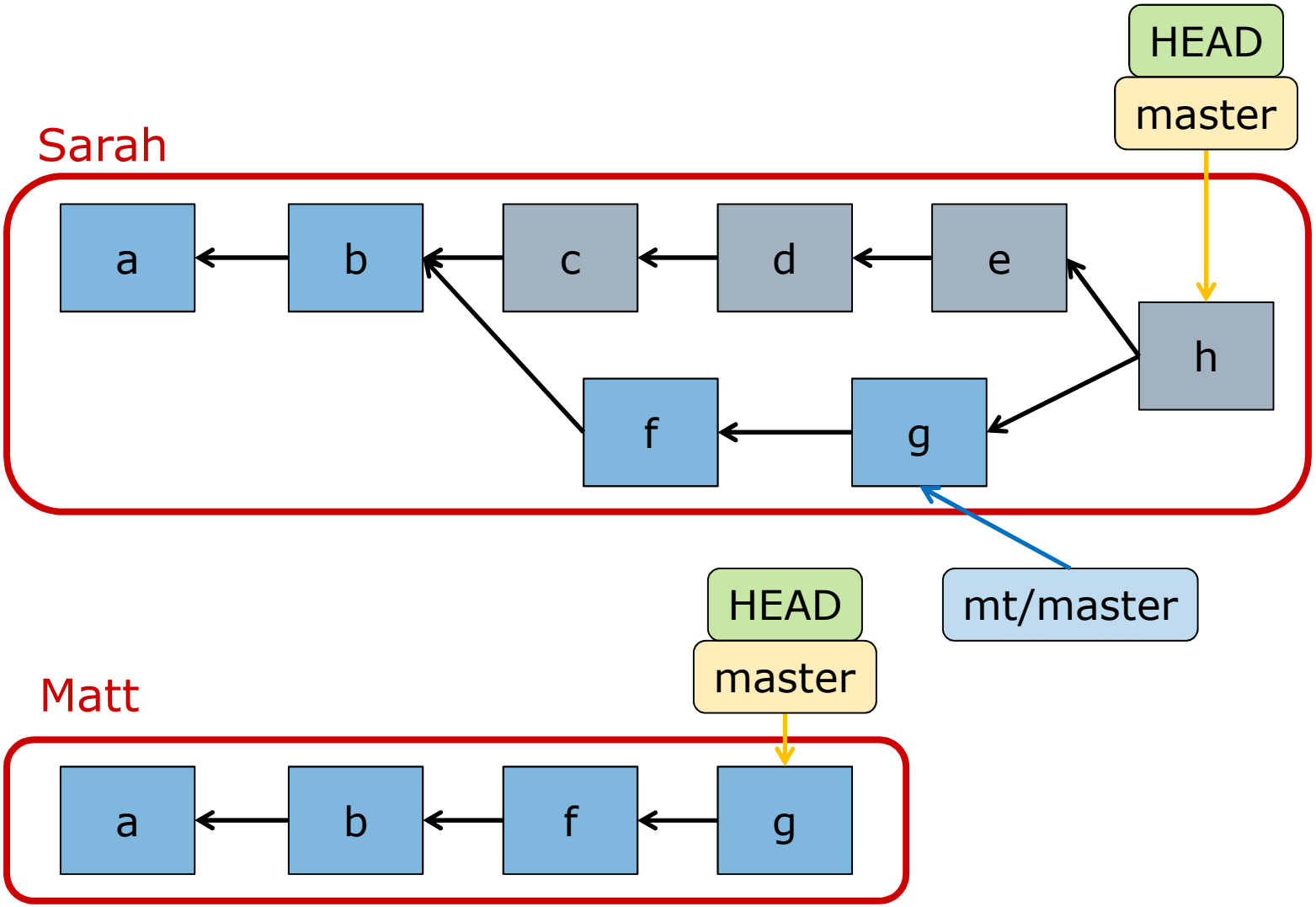
Your Turn

- 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: Merge


```
matt$ git merge sr/master
```

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 --all #see commit messages
$ git checkout #see work
$ git diff #compare
```

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

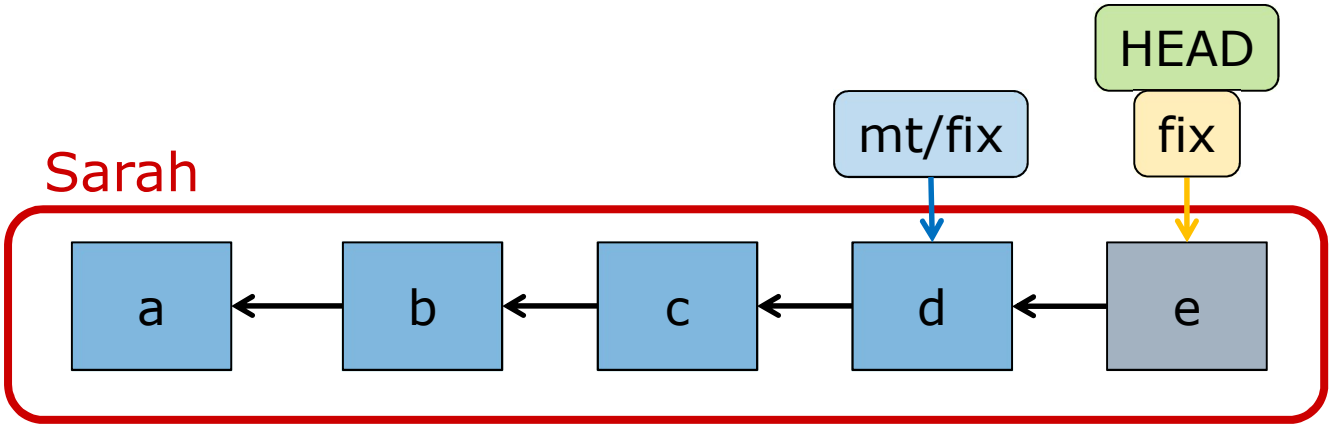
Push: Local Store → Remote

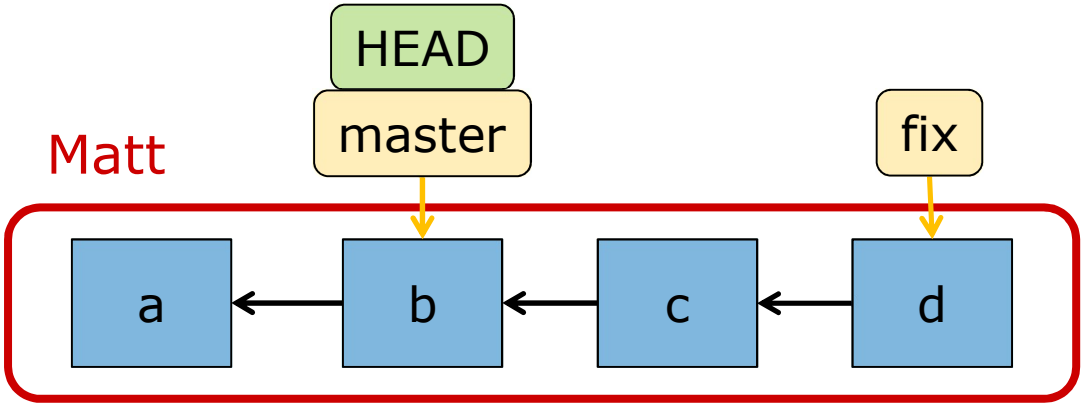
- Push sends local commits to remote store
- Usually push one branch (at a time)

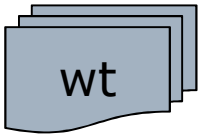
```
sarah$ git push mt fix
```

  - 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

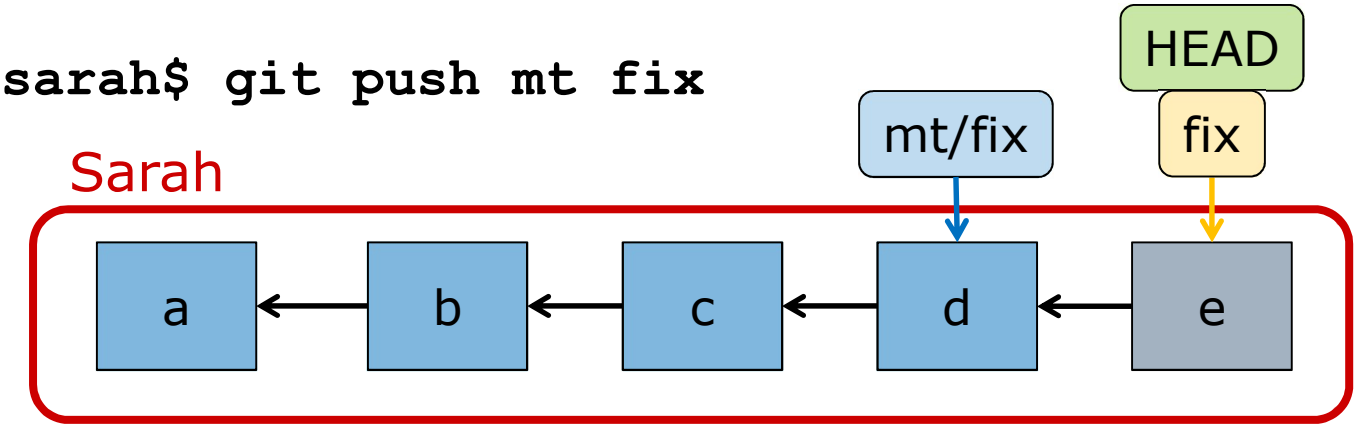


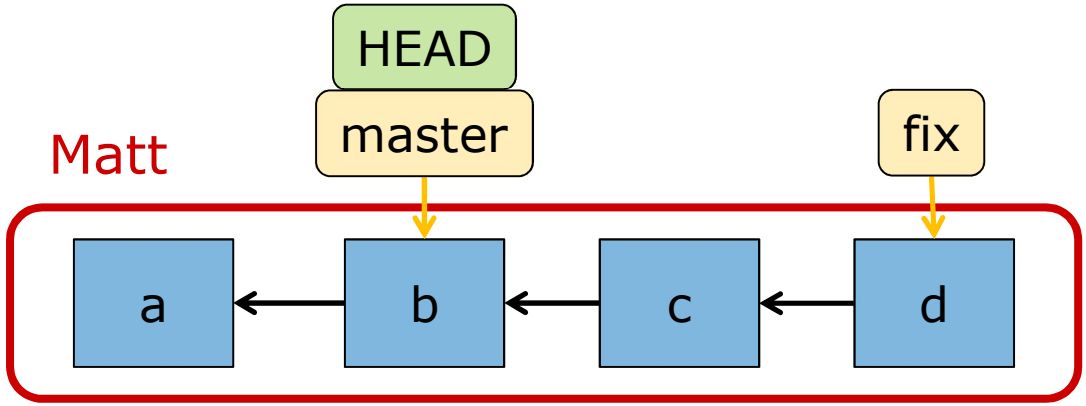


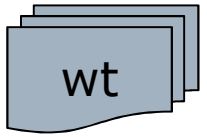


Push: Local Store → Remote


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

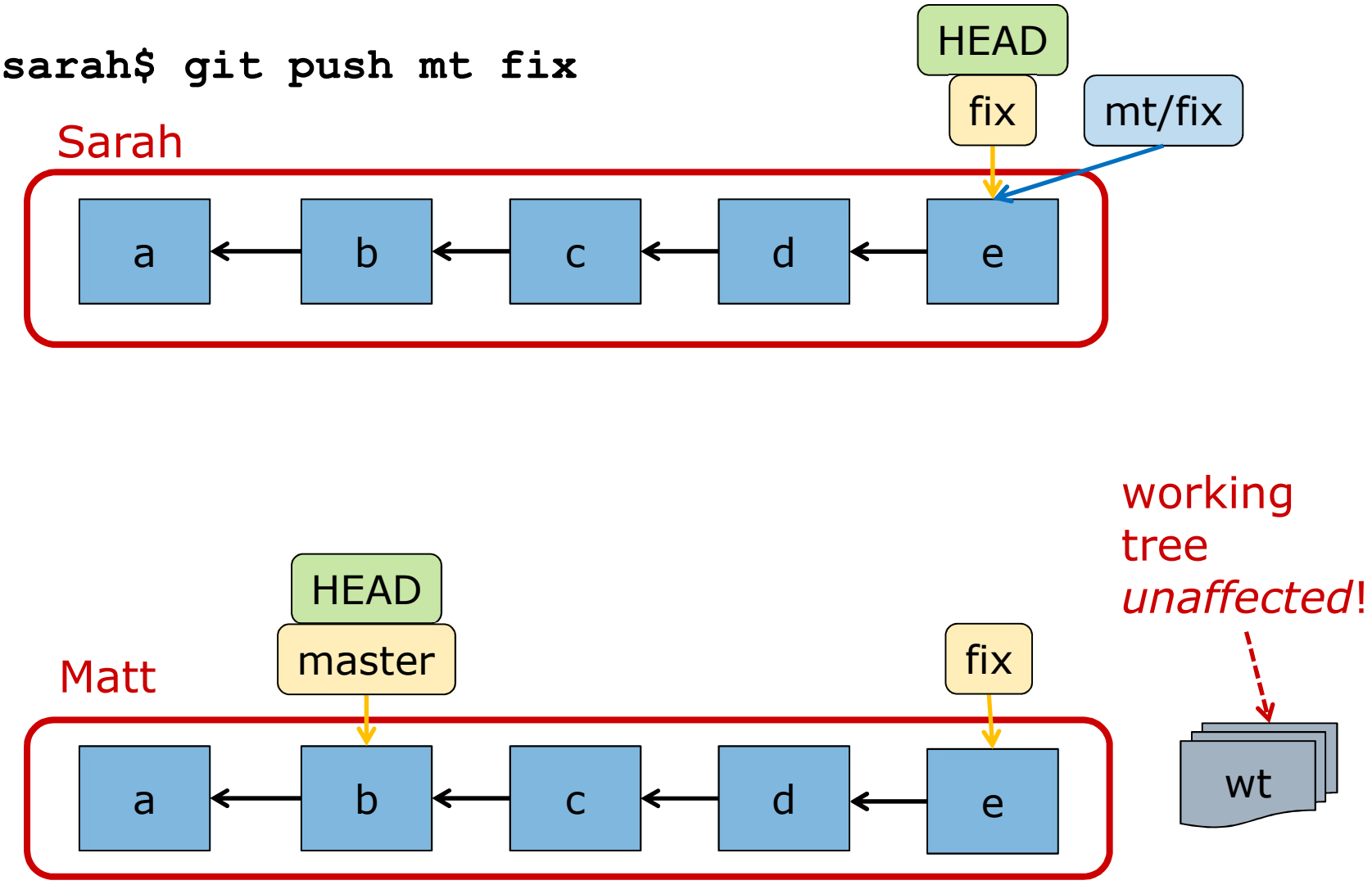




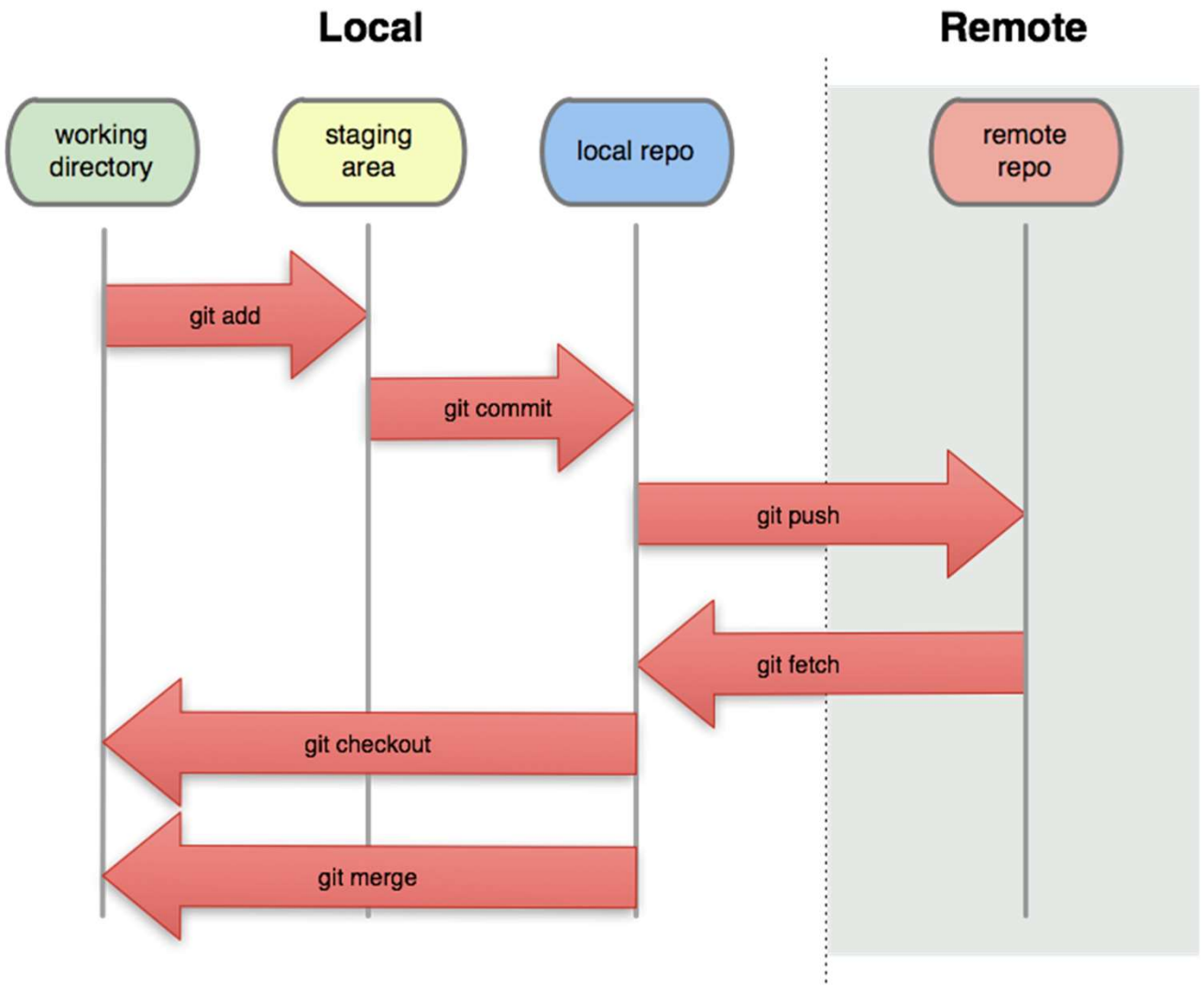


Push: After


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



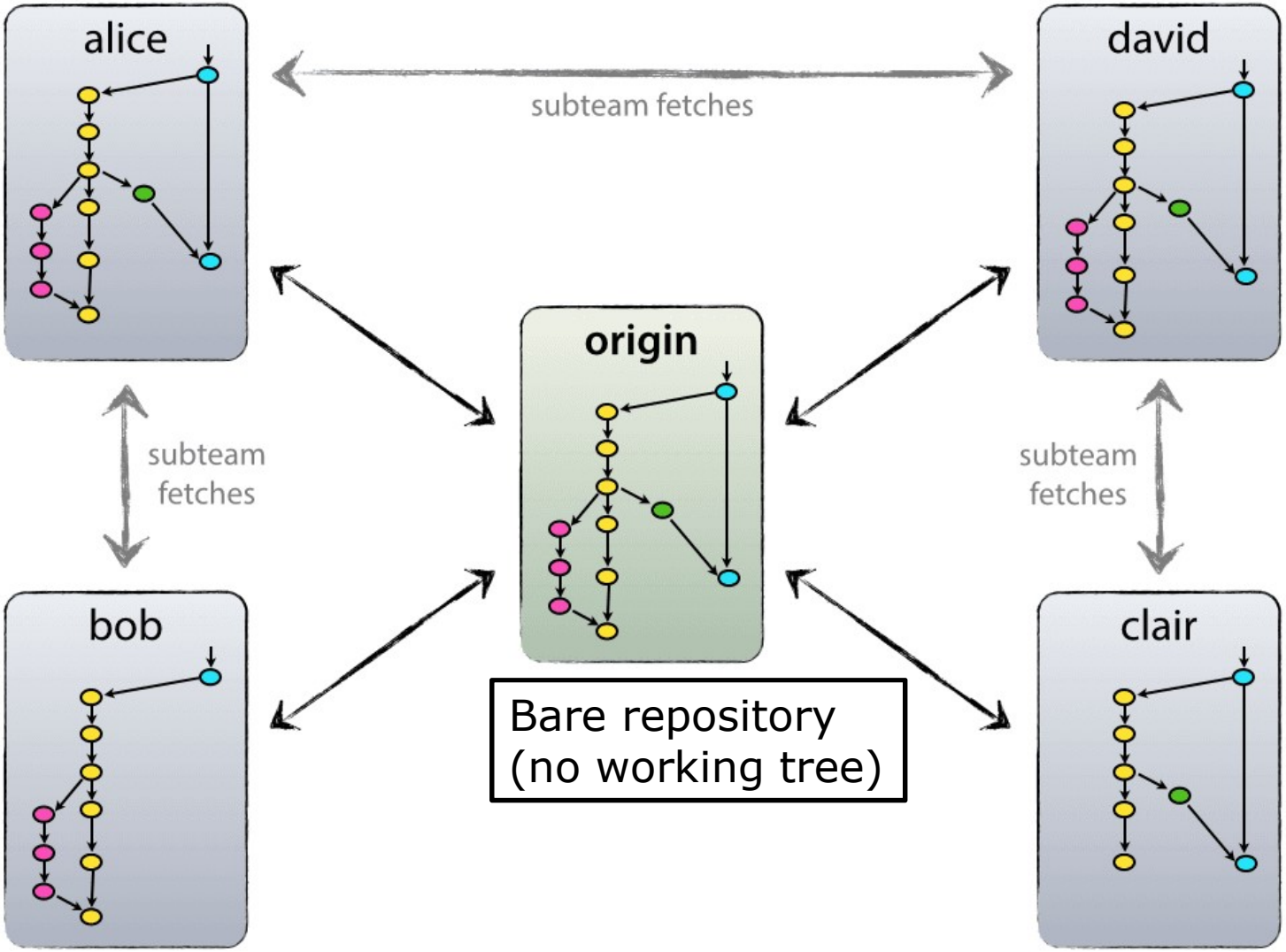
Commit/Checkout vs Push/Fetch



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. Exchange contact information
2. Choose a group name
3. Each person chooses a tech area
   - HTML/CSS, JavaScript, or Ruby
   - Group constraints on choices:
     - Each technology must be represented
     - No more than 2 people per technology
4. Also choose a backup tech area
   - "Don't Care" is fine (as primary or secondary)
5. Create a private Slack channel

Git: Miscellaneous Topics

Lecture 4

Basic Workflow: Overview

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

Step 1: Configure Git

- Each team member, in their own VM
  - Set identity for authoring commits

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

  - Optional: diff and merge tool (eg meld)

```
$ sudo apt install meld # to get tool
$ git config --global merge.tool meld
$ git config --global diff.tool meld
# example use:
$ git difftool e9d36
```

Step 2: Initialize Central Rep

- One person, once per project:
- Hosting services (GitHub, BitBucket…) use a web interface for this step
- Or, could use stdlinux instead:
  - Create central repository in group's project directory (/project/c3901aa03)

```
$ cd /project/c3901aa03
$ mkdir rep.git  # ordinary directory
```

  - Initialize central repository as bare and shared within the group

```
$ git init --bare --shared rep.git
```

Step 3: Create Local Repository

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

```
$ git clone
ssh://brut@stdlinux.cse.ohio-
state.edu//project/c3901aa03/proj1.git
mywork
```

  - You will be prompted for your (stdlinux) password (every time you fetch and push too)
  - To avoid having to enter your password each time, create an ssh key-pair (see VM setup instructions)

Step 4: Local Development

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

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

  - Pull others' work when 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 master # 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
```

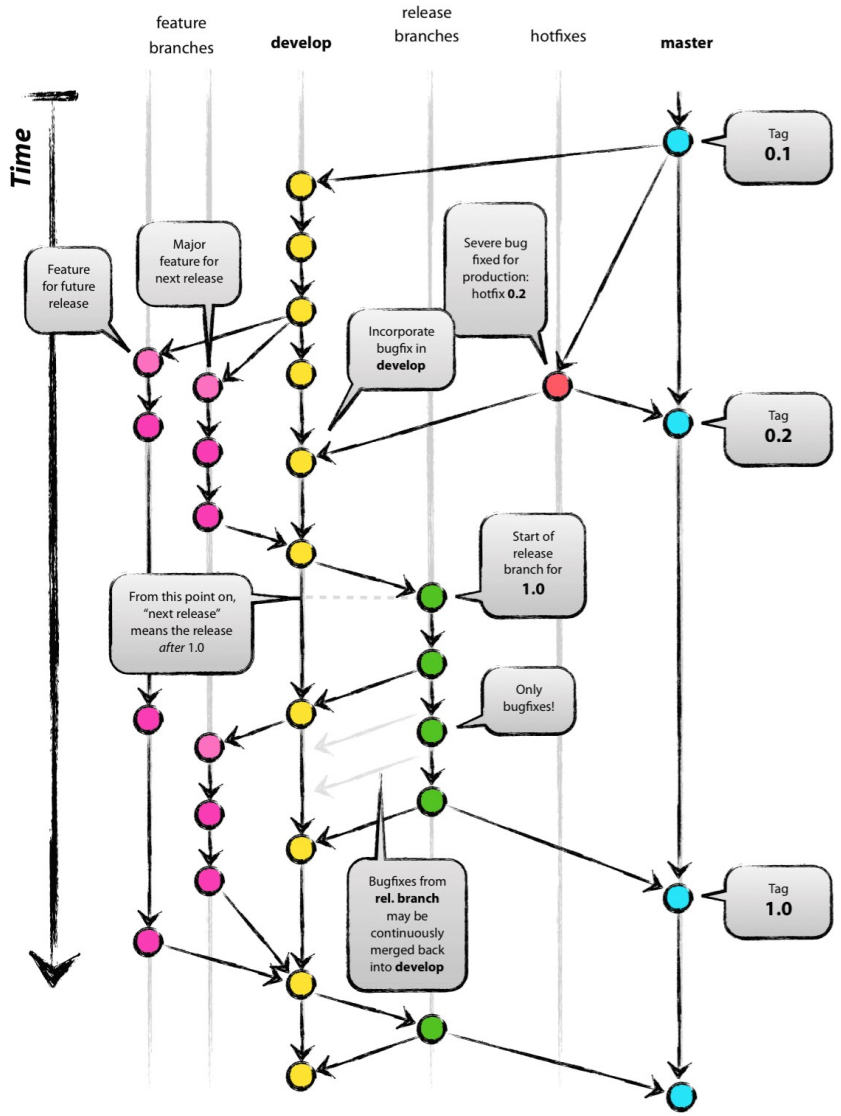
Professional Git

- 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

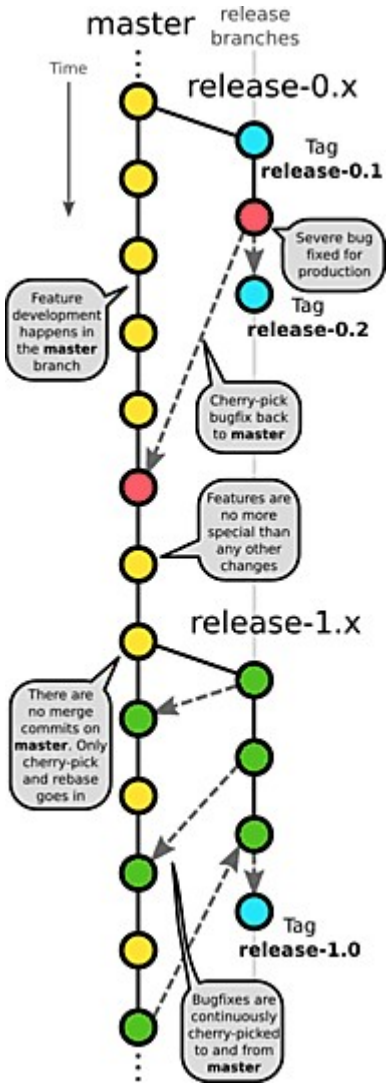
Commit/Branch Conventions

- 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/direcotry paths from local machine
  - Passwords
  - Better: Use *variables* instead
  - 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
- 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
- Example:

```
# see github:gitignore/Ruby, /Global/
# Ignore auto-saved emacs files
*~

# Ignore bundler config
/.bundle

# Ignore the default SQLite database
/db/*.sqlite3

# Ignore all logfiles and tempfiles
/log/*
/tmp/*
```

Problem: End-of-line Confusion

- Differences between OS's in how a “new line” is encoded in a text file
  - Windows: CR + LF (ie "\r\n", 0x0D 0x0A)
  - Unix/Mac: LF (ie "\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

- Git convention: use \n in the store
  - 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 these bytes (0x0D and/or 0x0A), but they should not 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

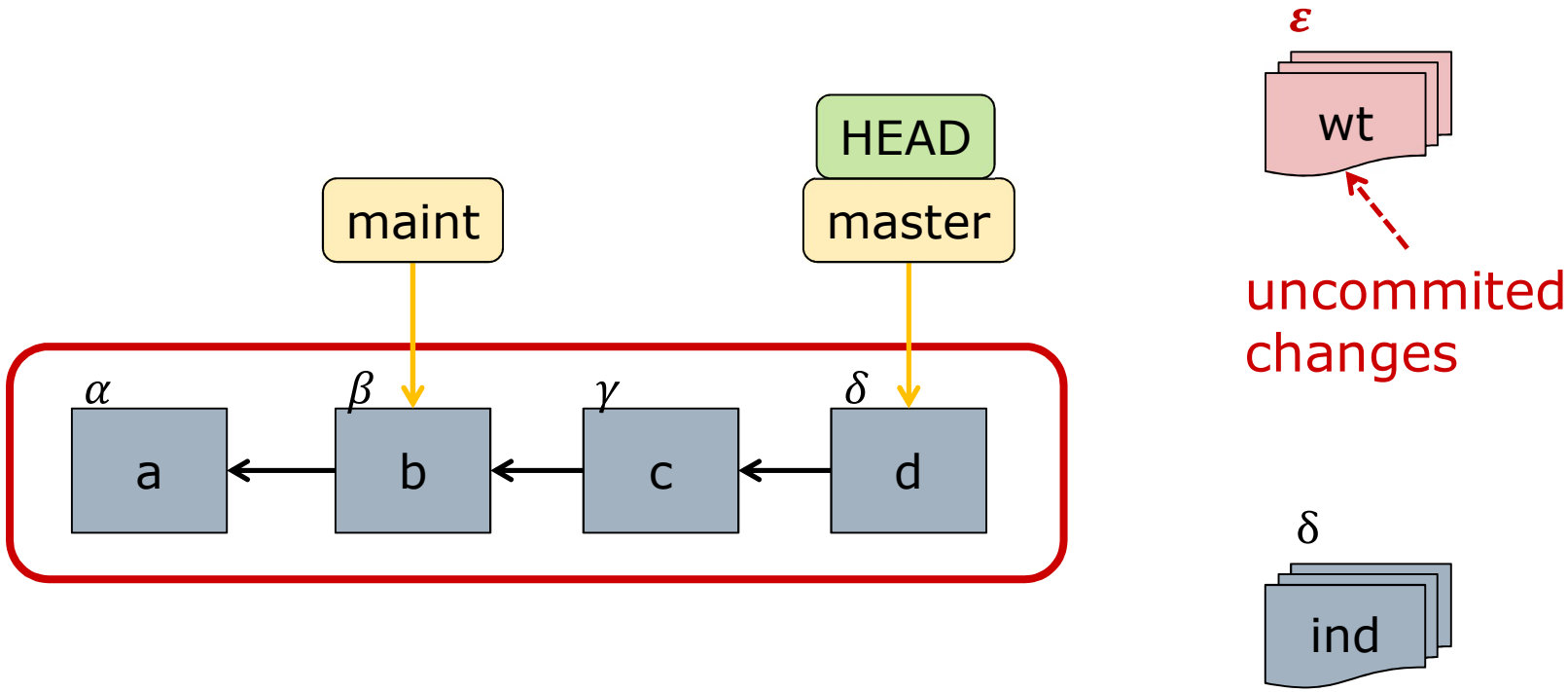
  `amend`

- DAG surgery

  `rebase`

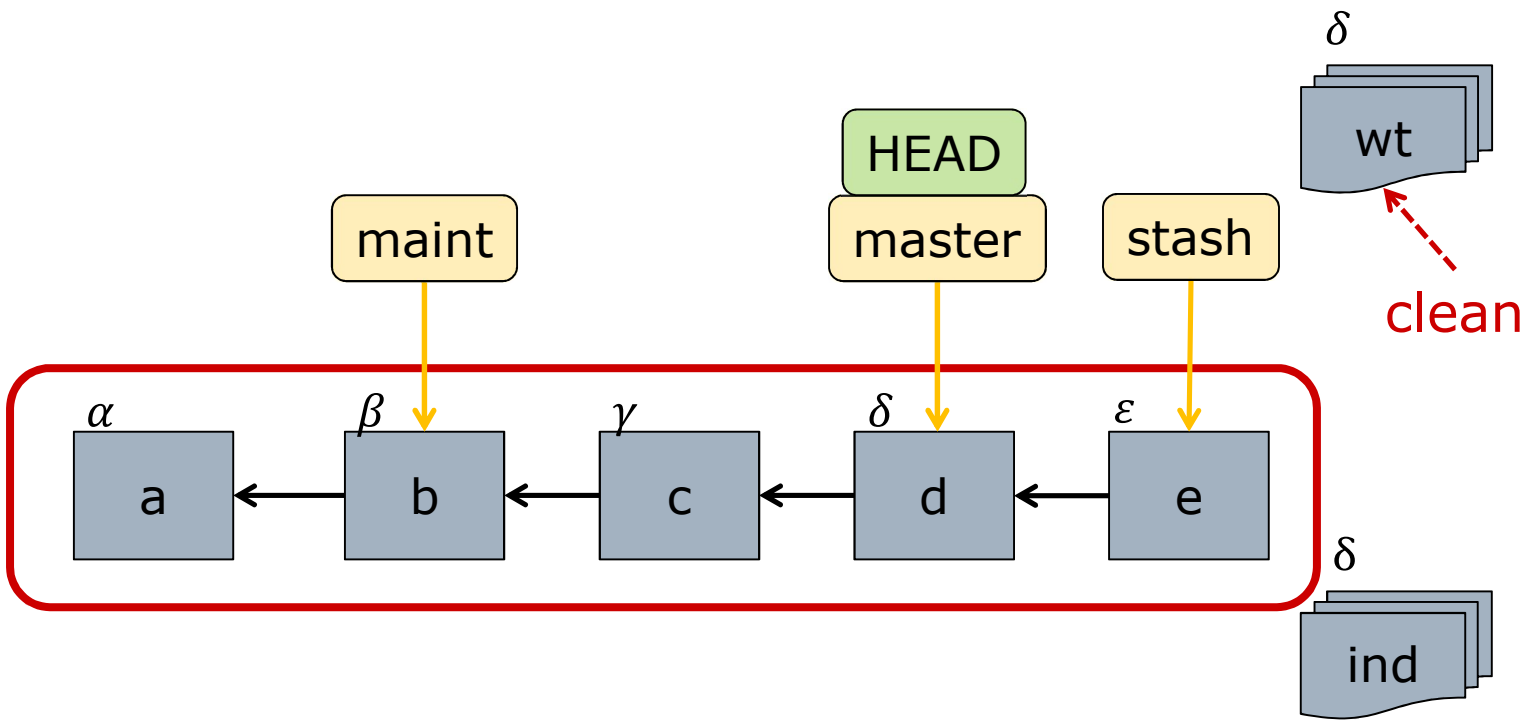
Advanced: Temporary Storage

- Say you have uncommitted work and want to look at a different branch
- Checkout won't work!



Stash: Push Work Onto A Stack

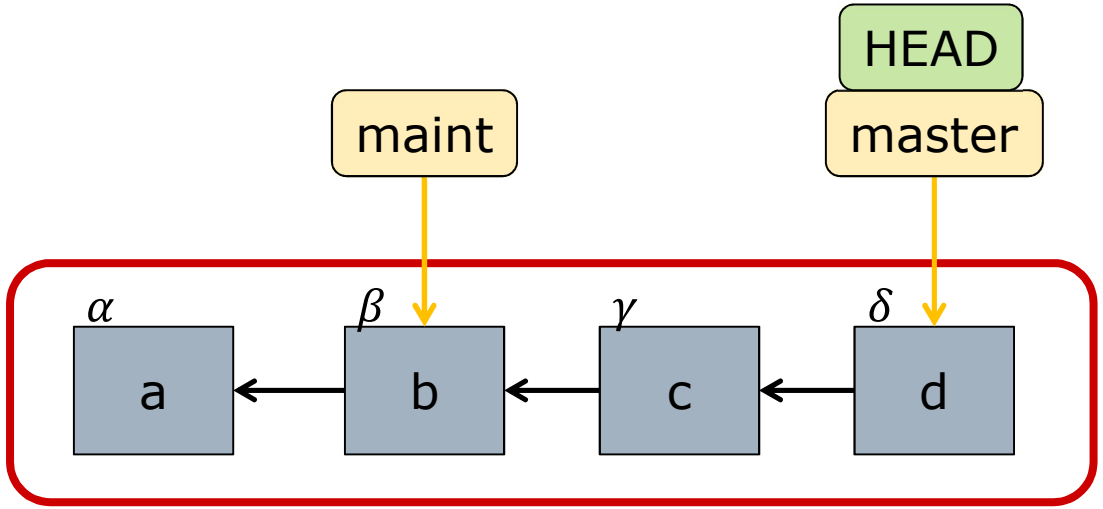

```
$ git stash # repo now clean
$ git checkout …etc… # feel free to poke around
```



Stash: Pop Work Off the Stack

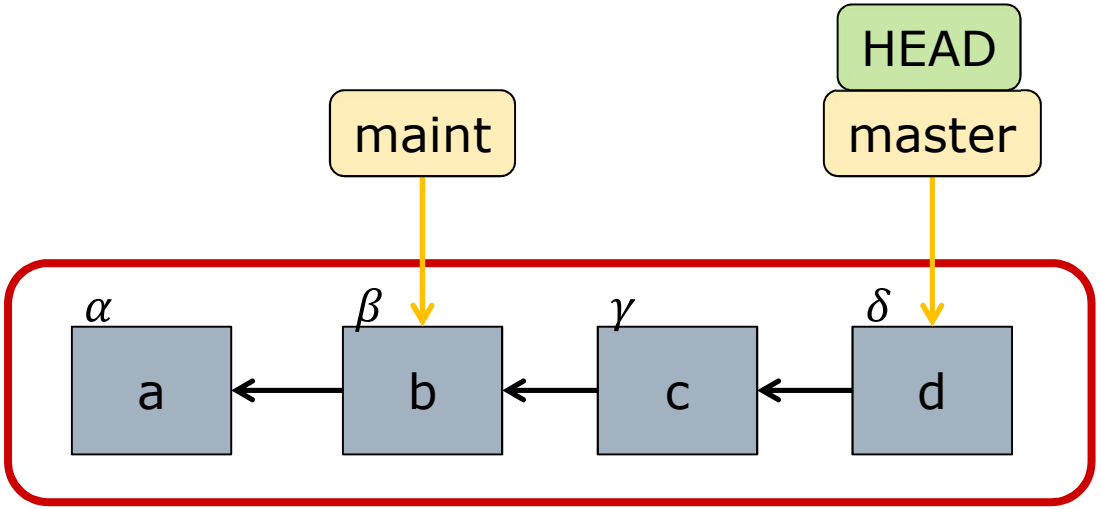

```
$ git stash pop # restores state of wt (and store)

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



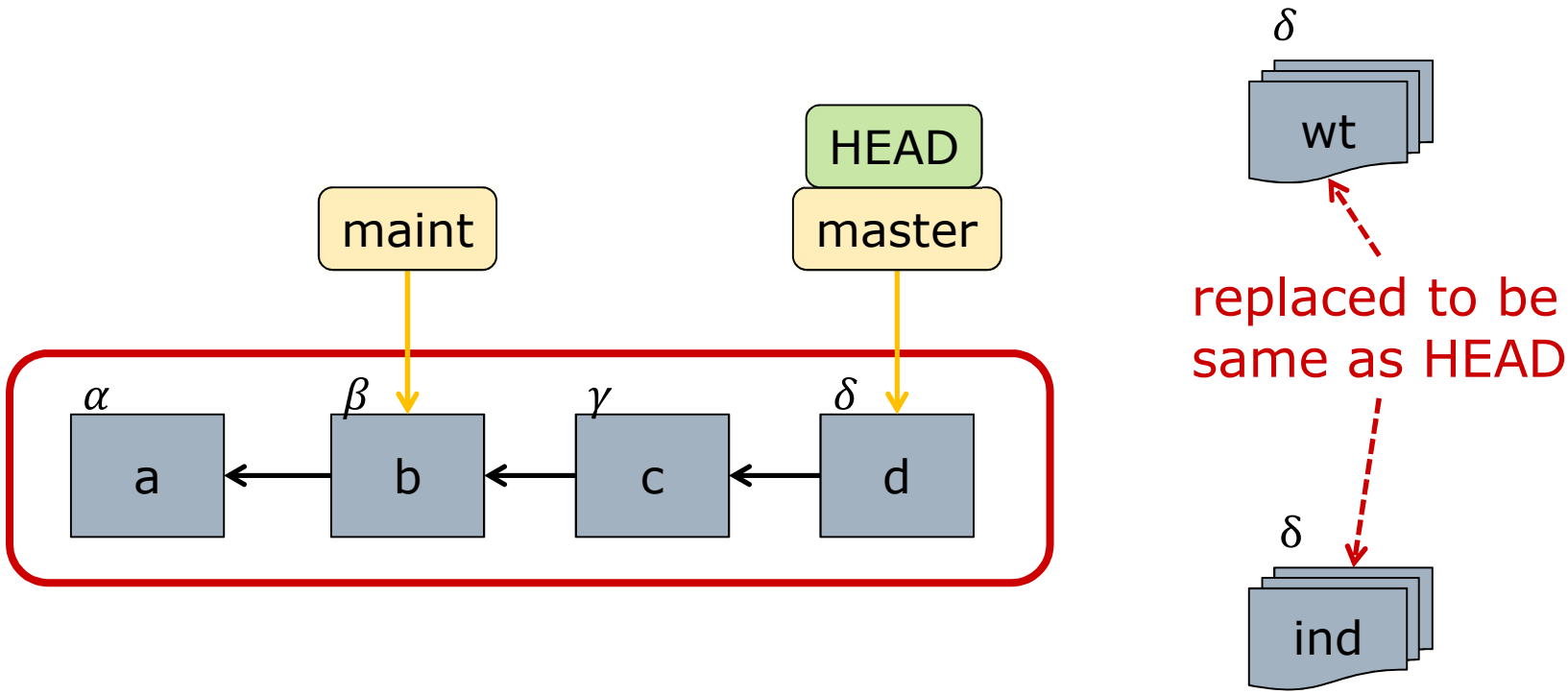
Advanced: Undoing Big Mistakes

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



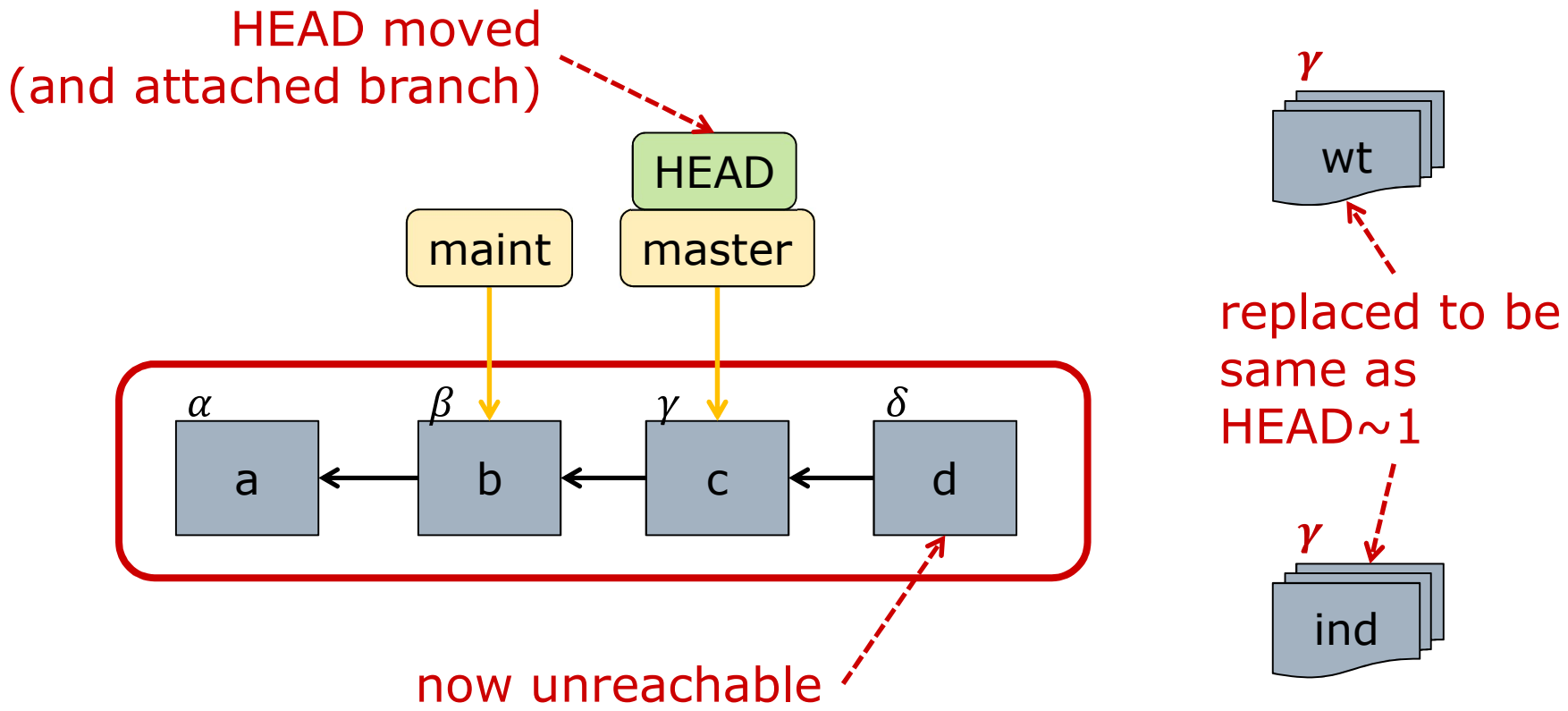
Reset: Discarding Changes


```
$ git reset --hard
$ git clean --dry-run # list untracked files
$ git clean --force # remove untracked files
```



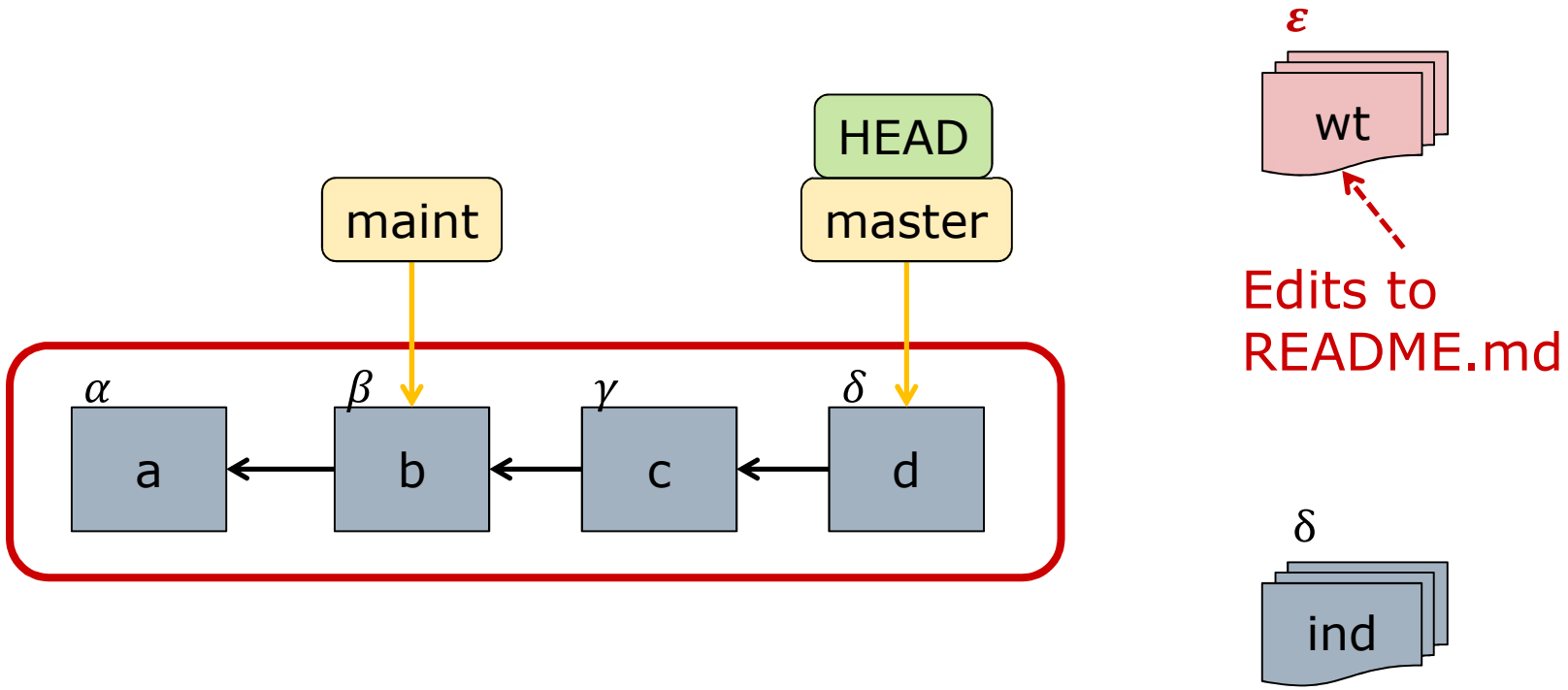
Reset: Discarding Commits


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



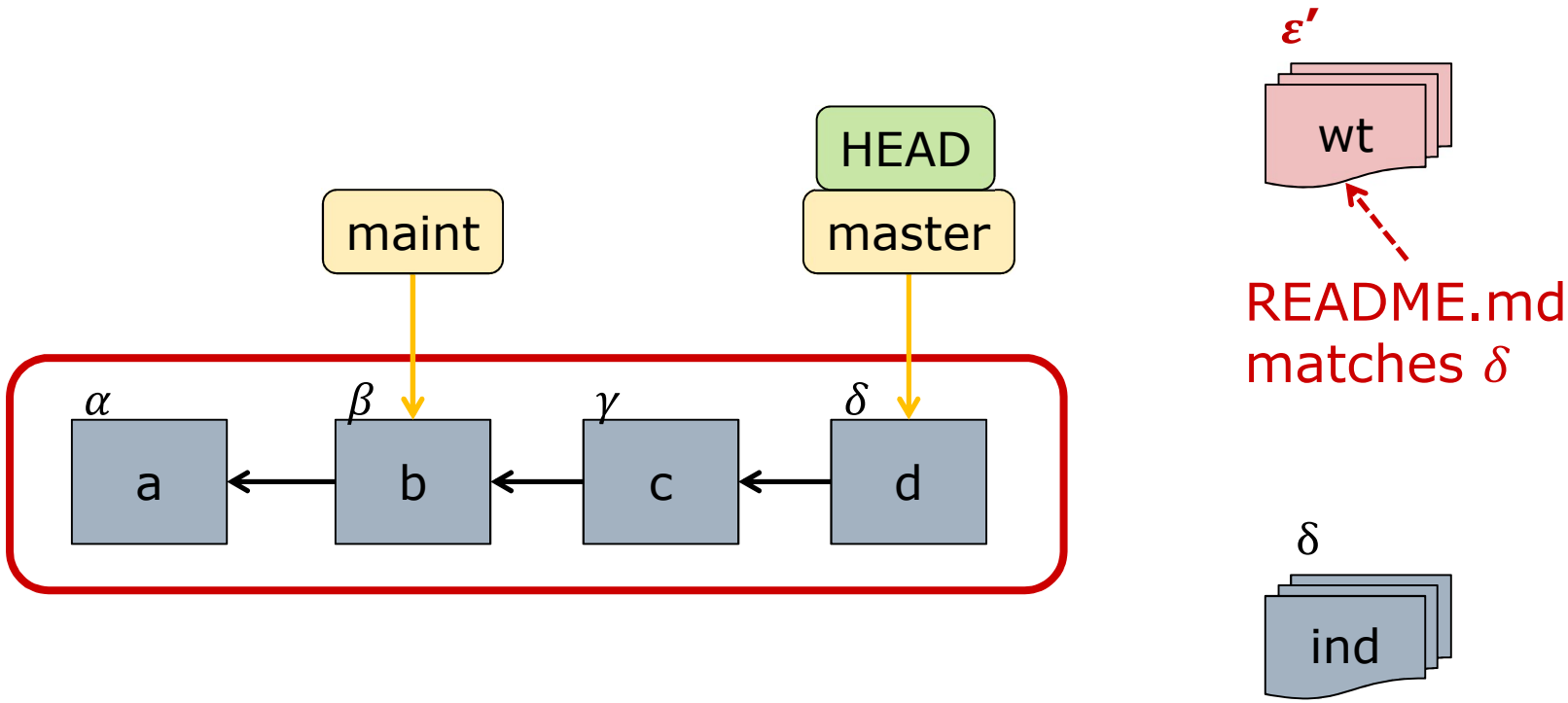
Advanced: Undo Small Mistakes

- 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

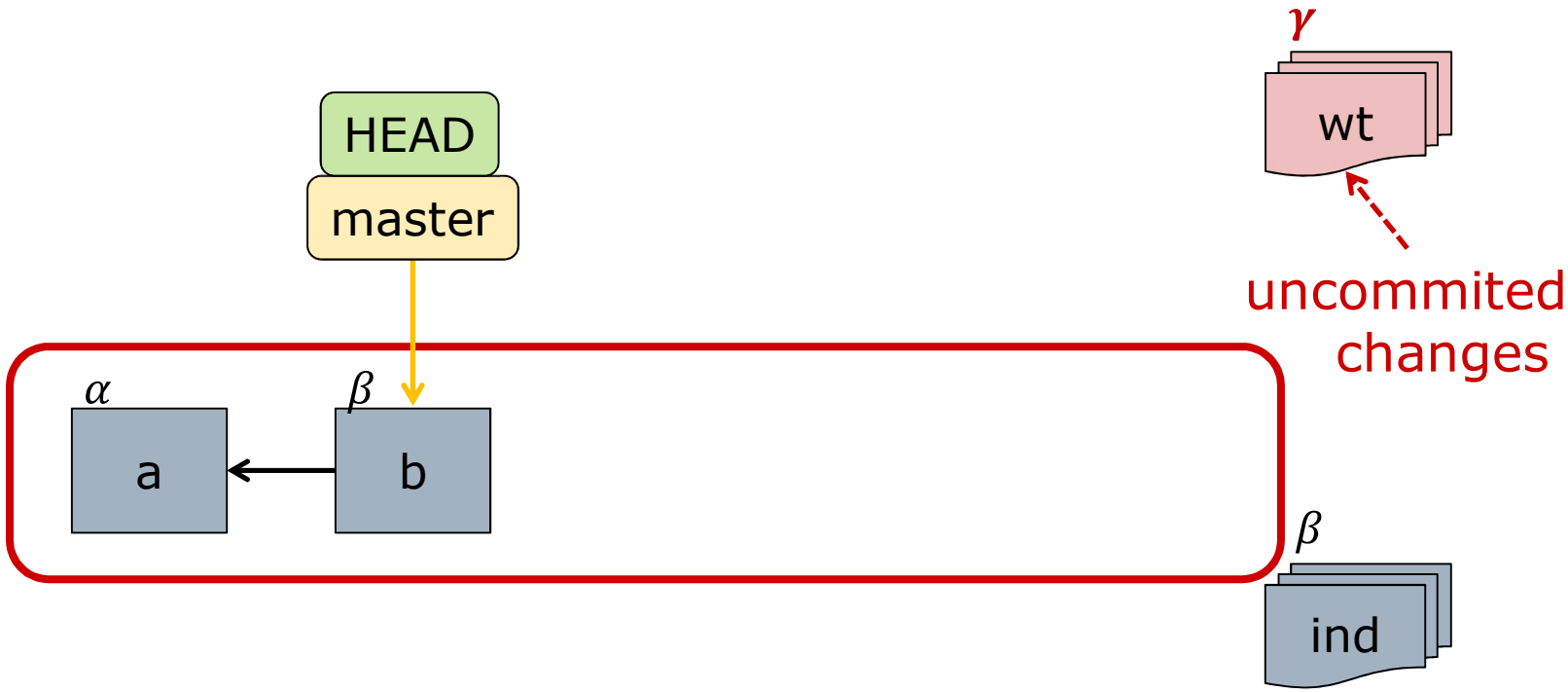
The Power to Change History

- 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!

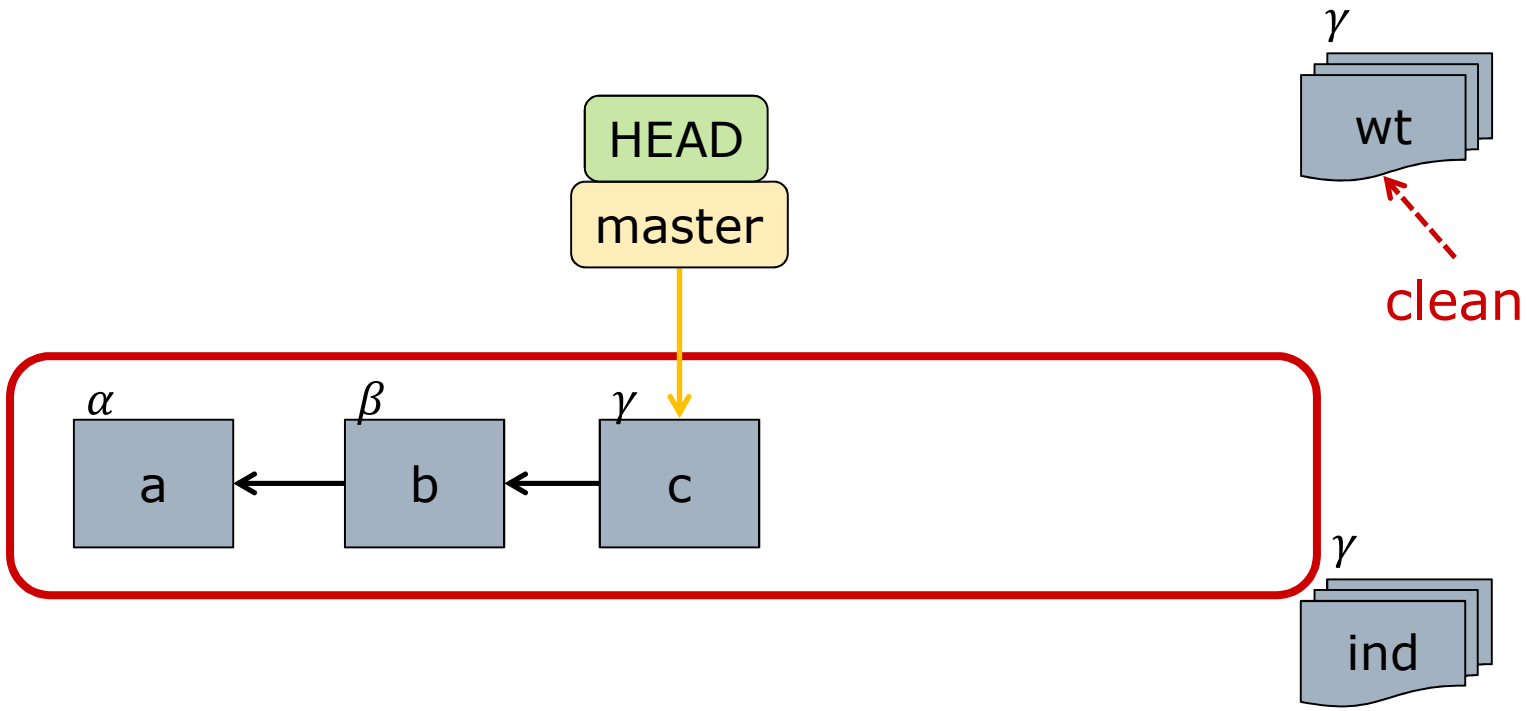
Advanced: Rewriting History

- Problem 1: Wrong or incomplete commit



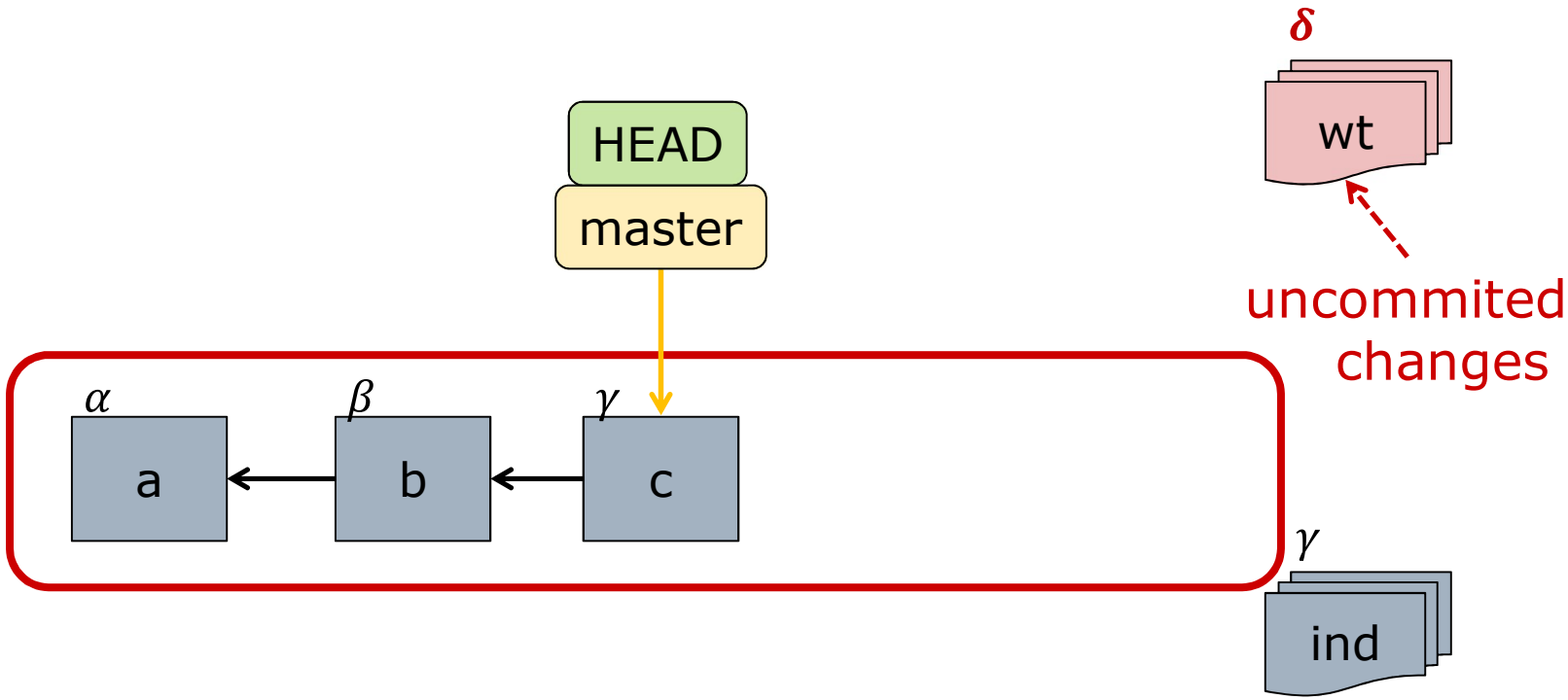
Advanced: Rewriting History

- Problem 1: Wrong or incomplete commit



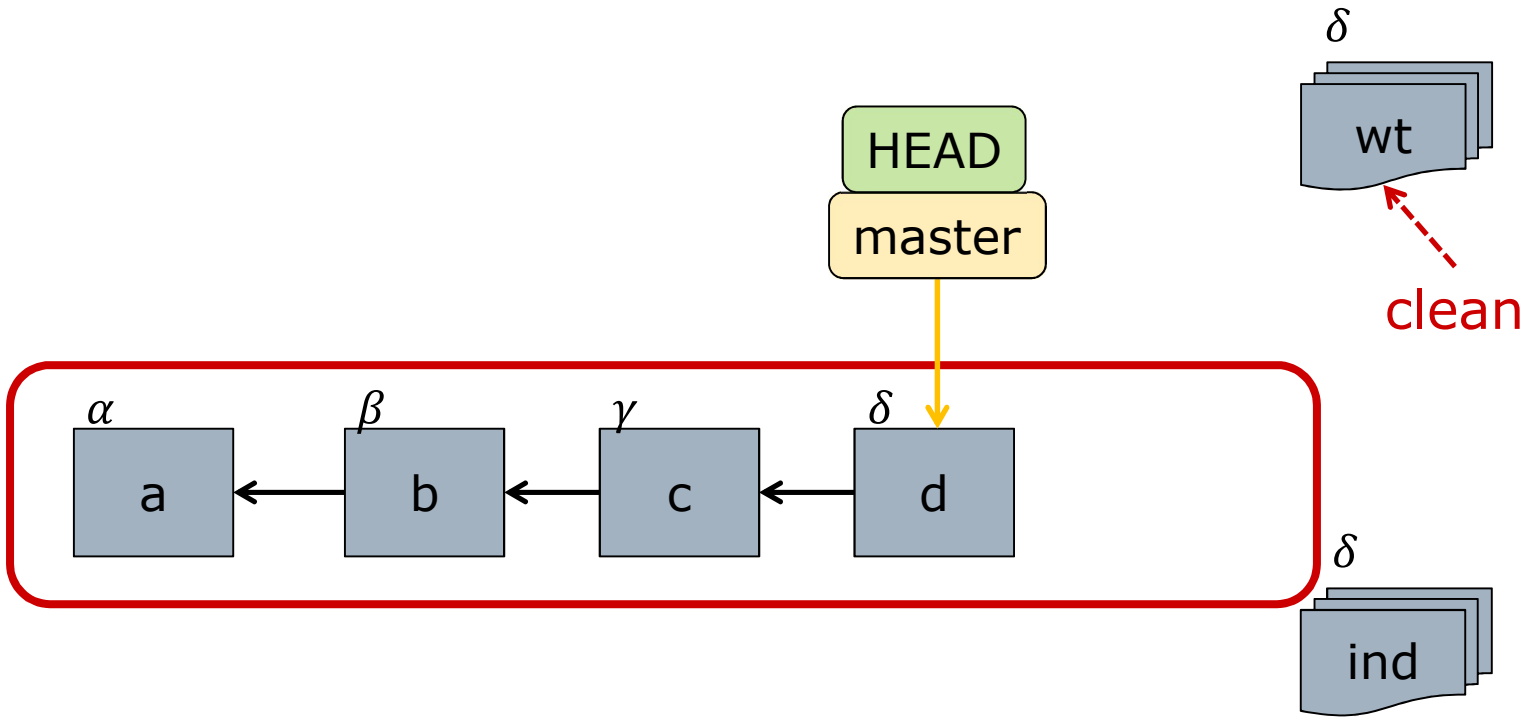
Advanced: Rewriting History

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



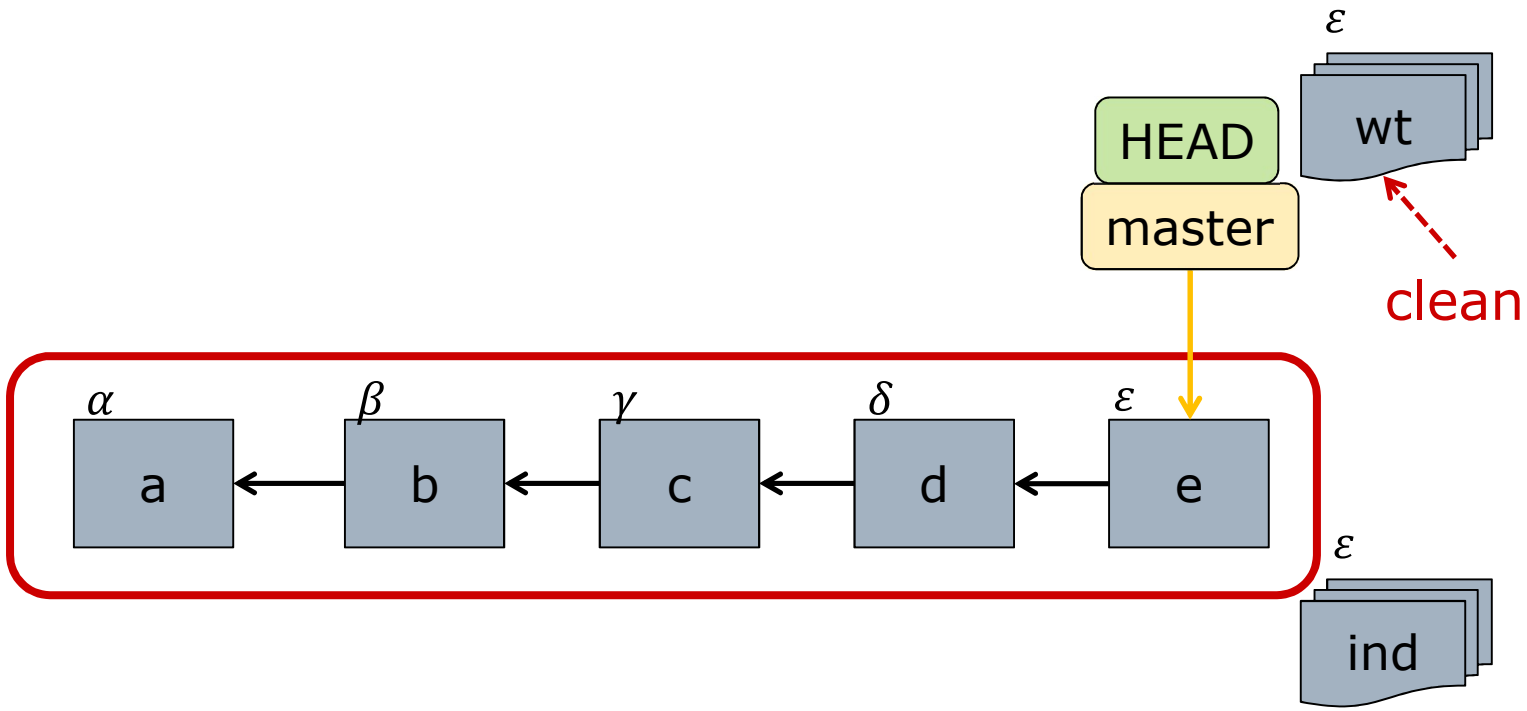
Advanced: Rewriting History

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



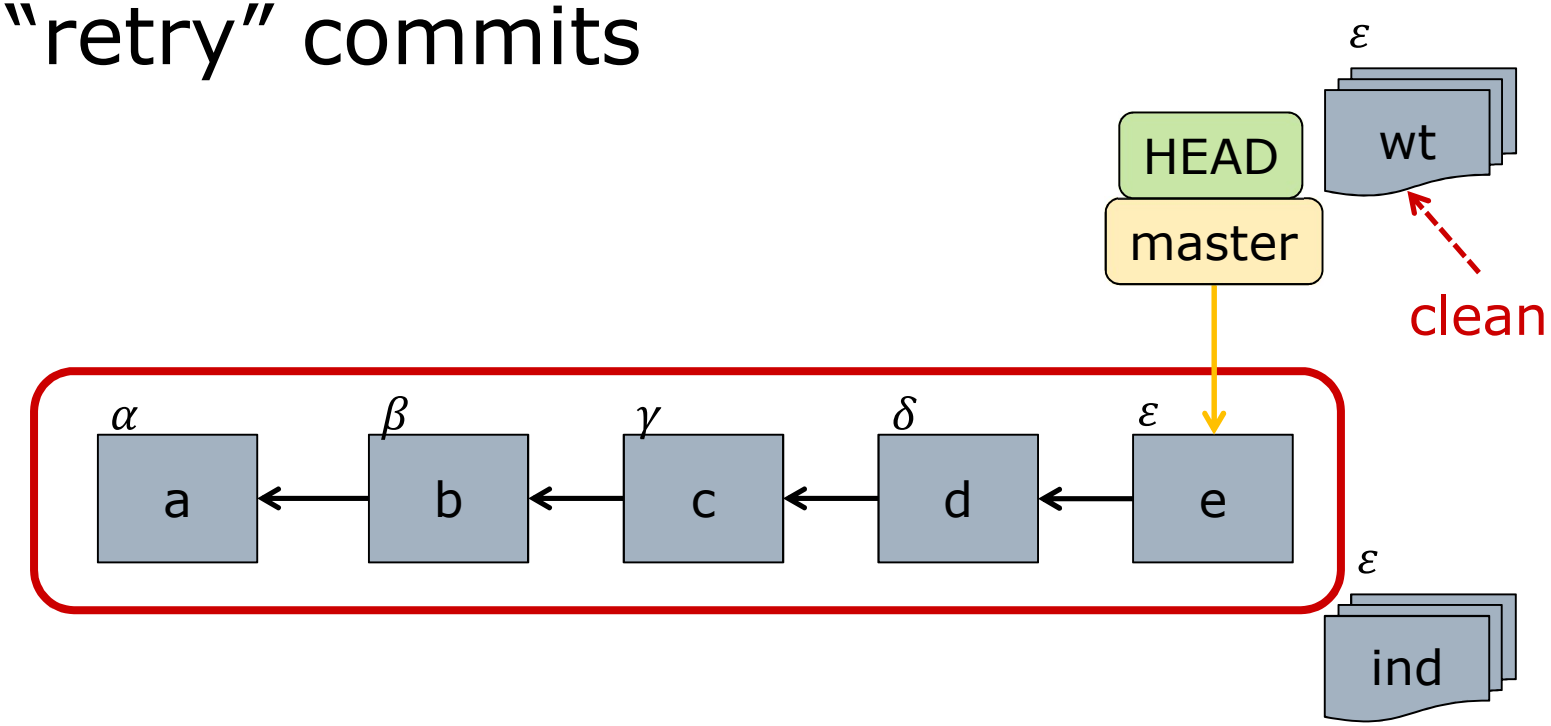
Advanced: Rewriting History

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



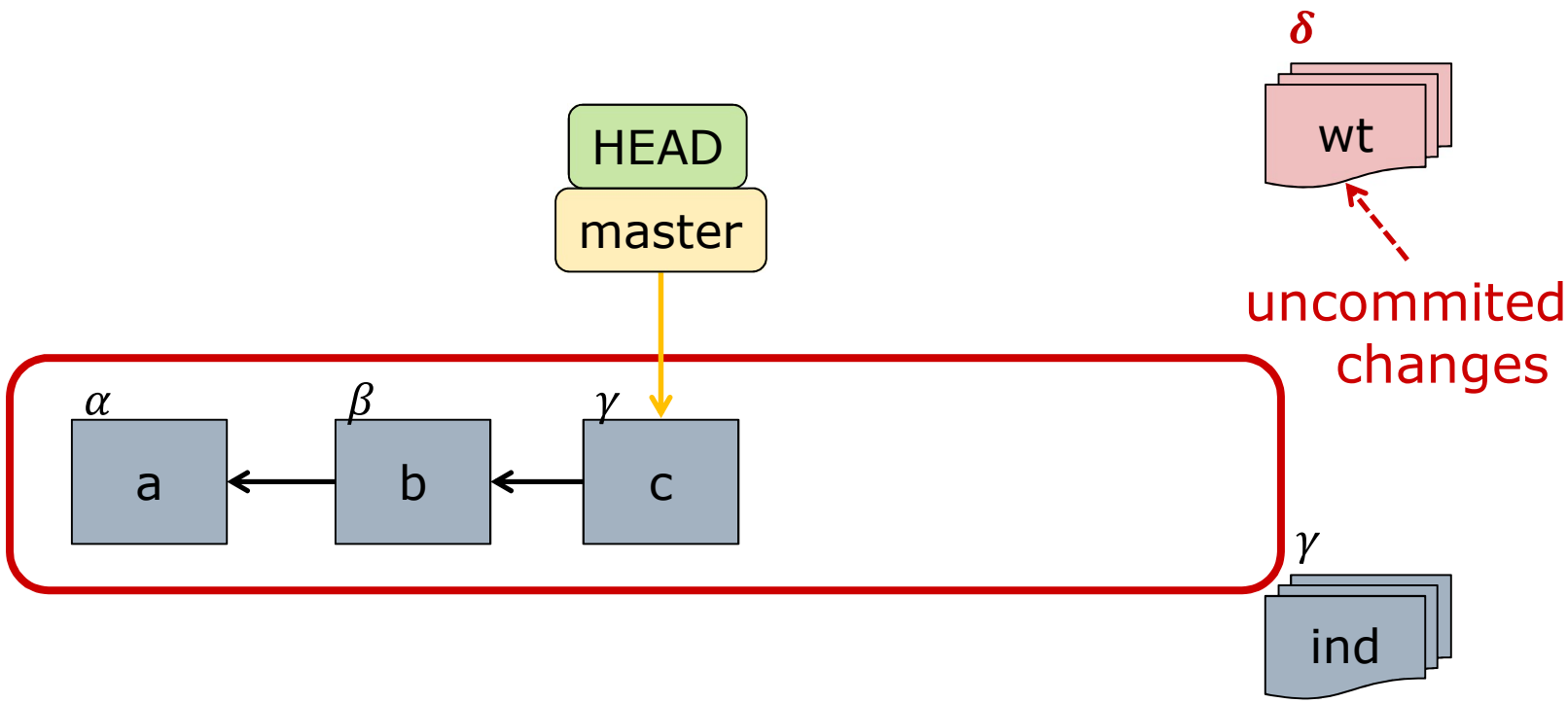
Advanced: Rewriting History

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



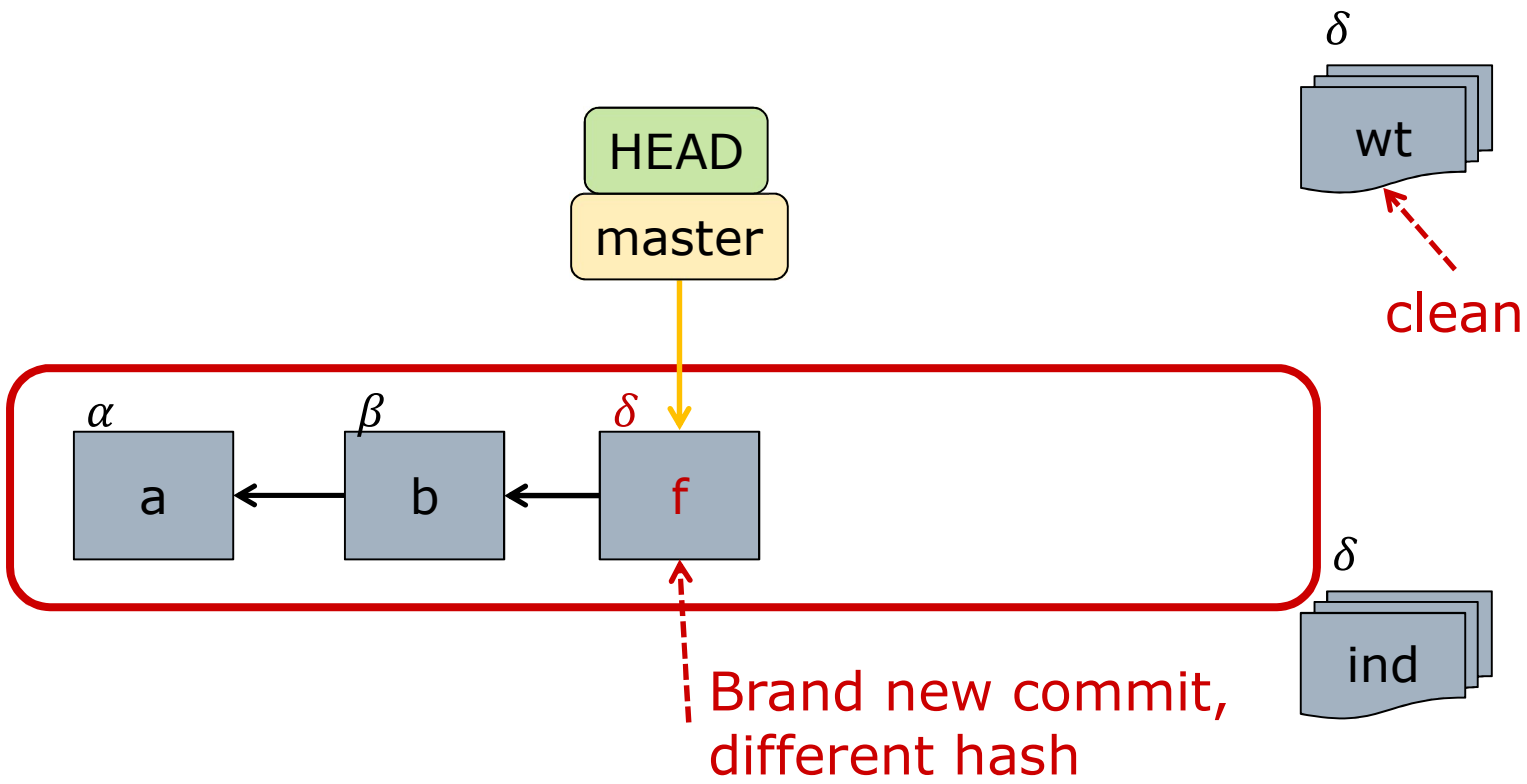
Commit --amend: Tip Repair

- Alternative: Change most recent commit(s)



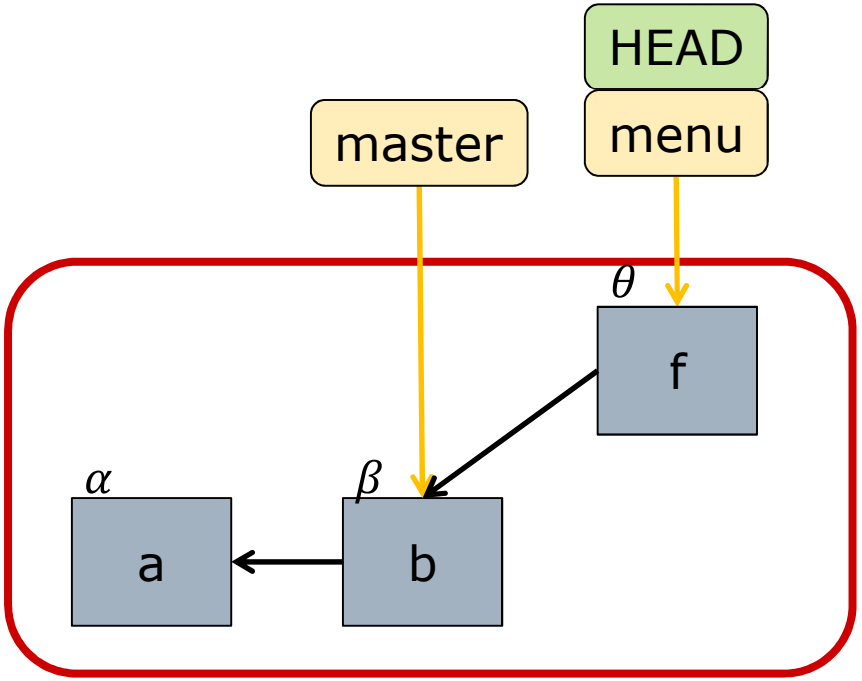
Commit --amend: Tip Repair


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



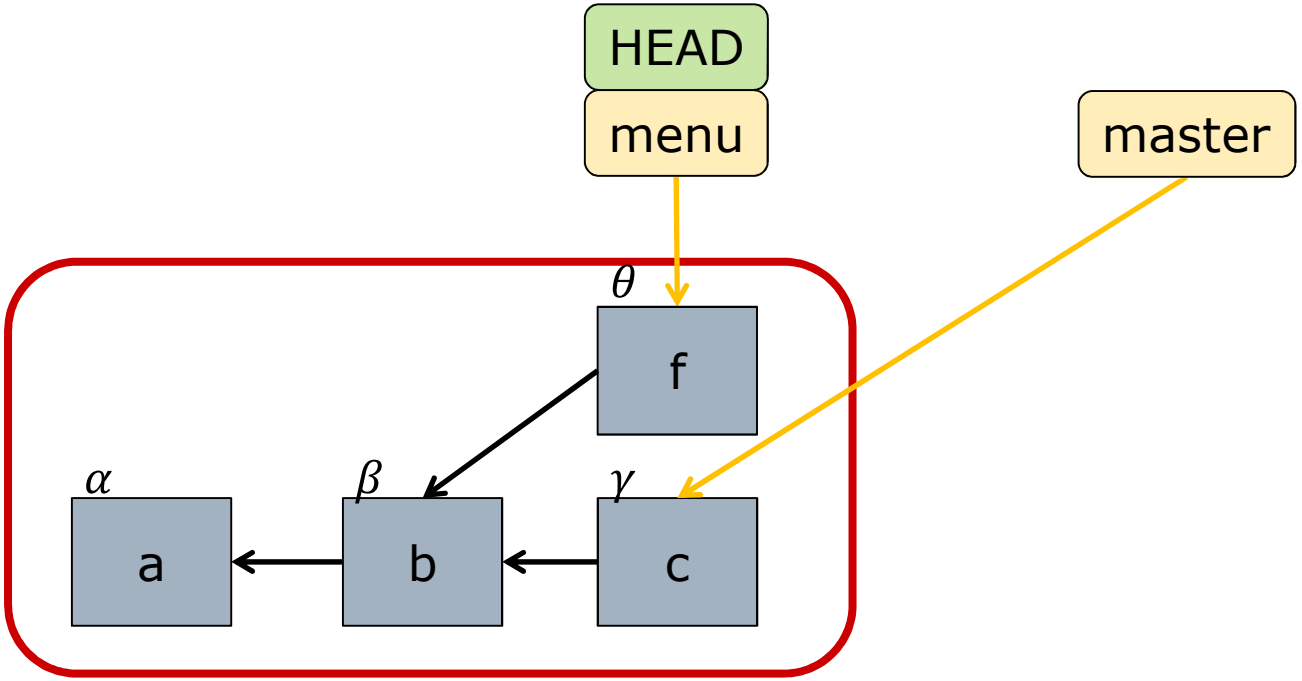
Advanced: Rewriting History

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



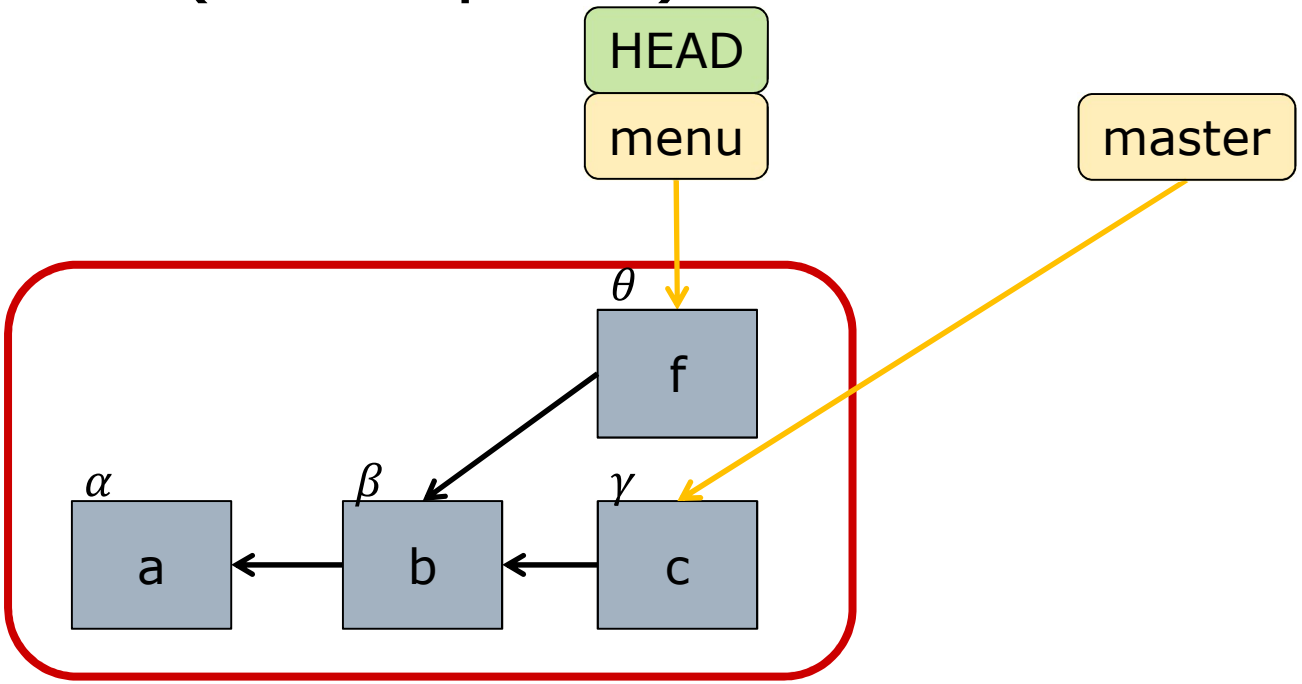
Advanced: Rewriting History

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



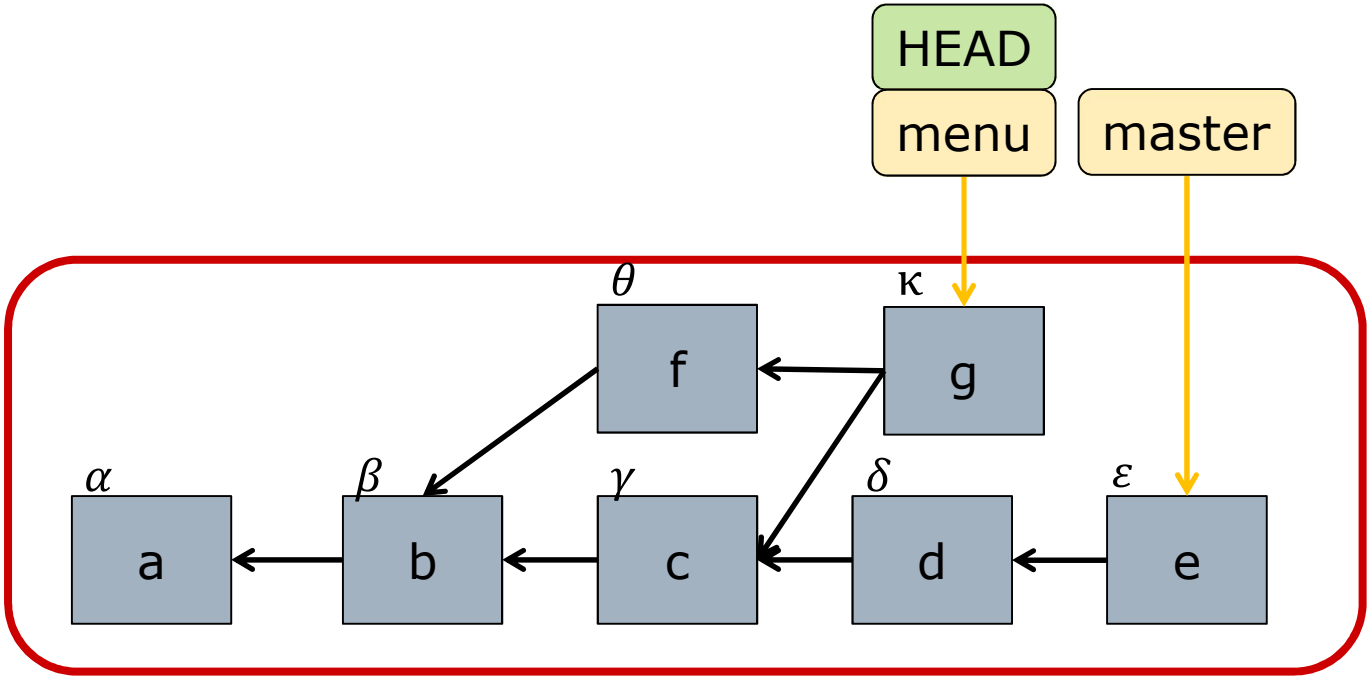
Advanced: Rewriting History

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



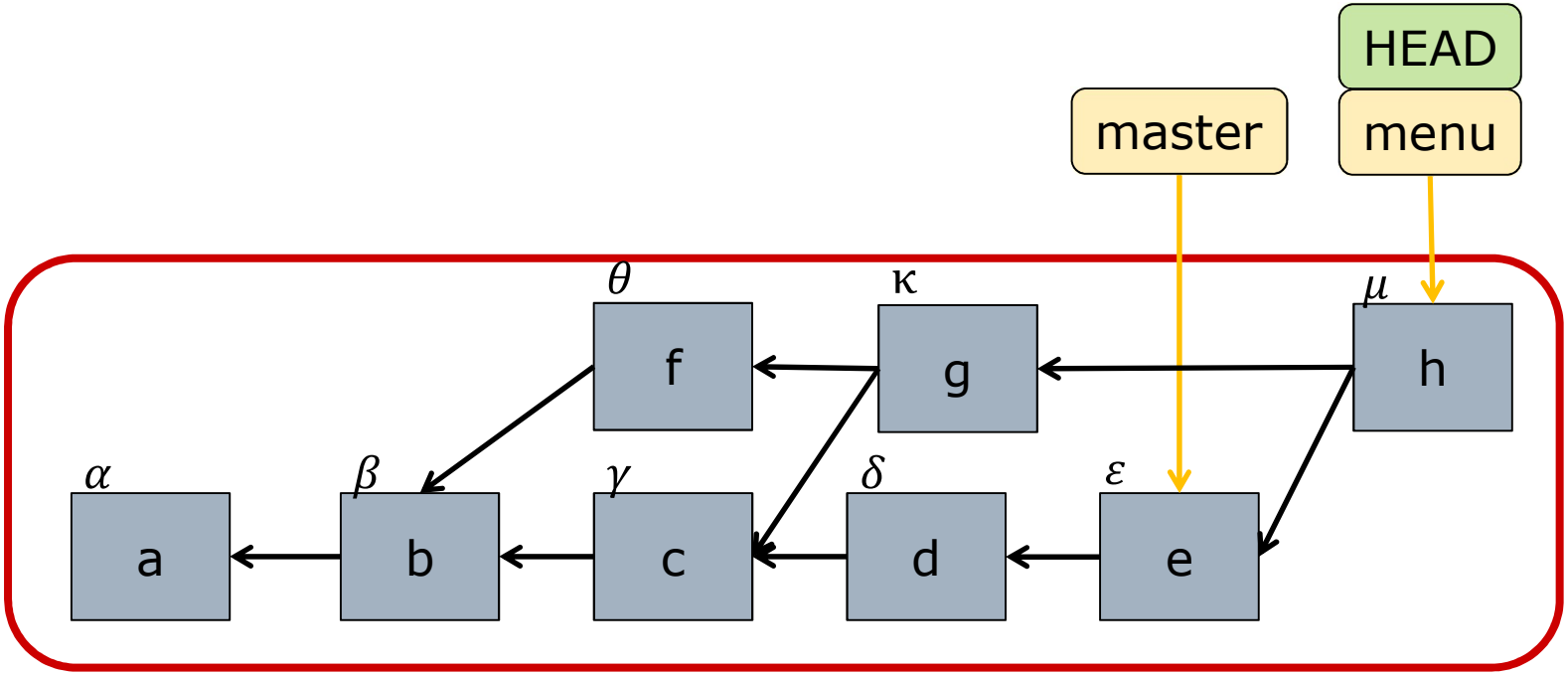
Advanced: Rewriting History

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



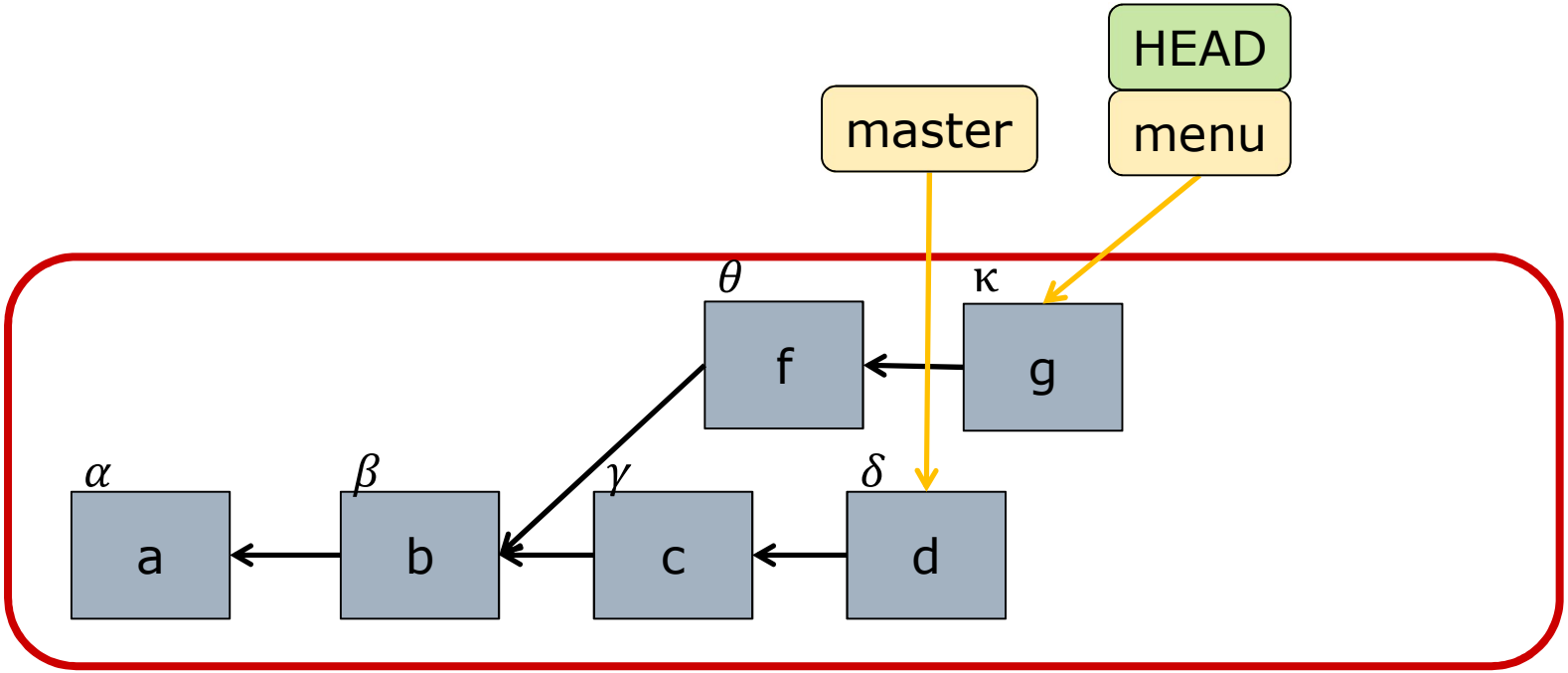
Advanced: Rewriting History

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



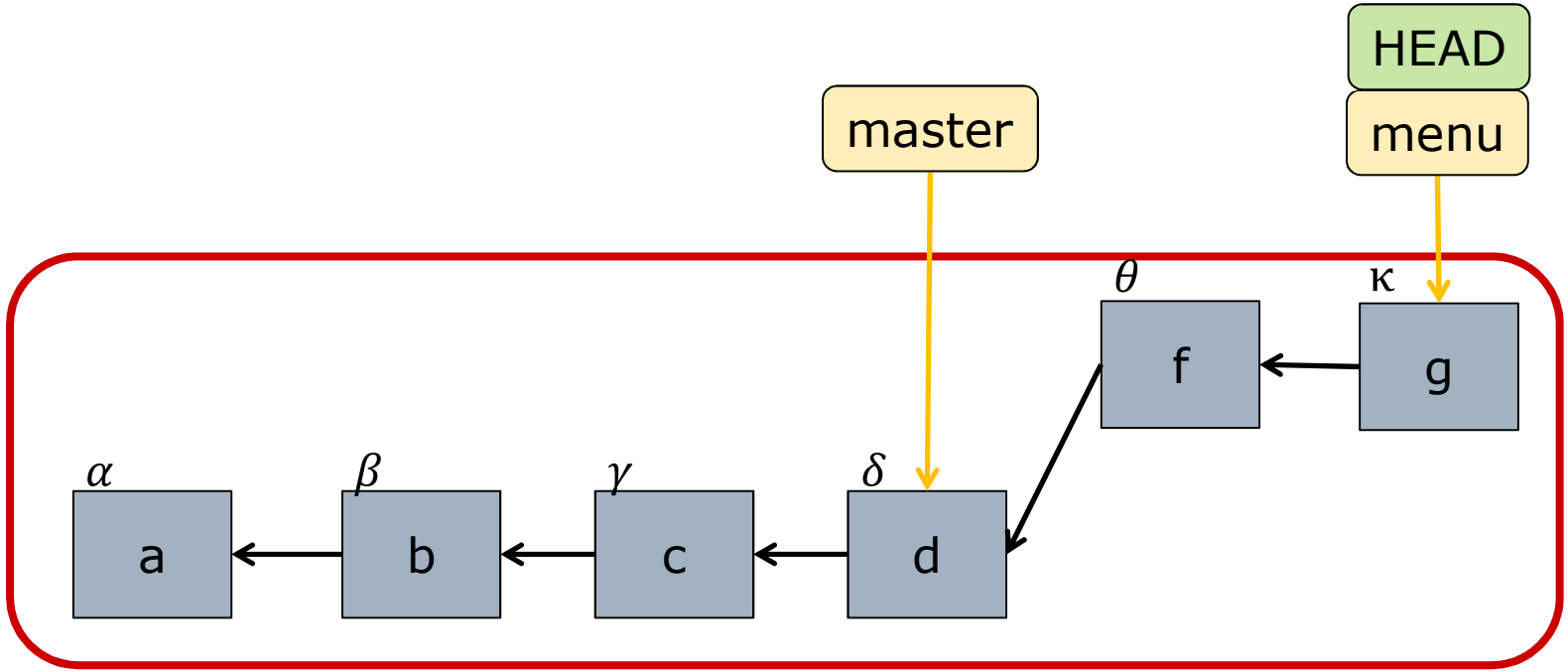
Rebase: DAG Surgery

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



Rebase: DAG Surgery


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



Git Clients and Hosting Services

- Recommended client: Command line!
- Alternative: IDEs
  - 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
  - But even free plan allows unlimited *private* repo's (and collaborators)
  - 3901 has an "organization" for your private repo's and team access
- Other services (eg GitLab, Bitbucket) have similar issues
- Public repo's containing coursework can create academic misconduct issues
  - Problems for poster
  - Problems for plagiarist

Mercurial (hg): Another DVCS

- Slightly simpler mental model
- Some differences in terminology
  - git fetch/pull ~= hg pull/fetch
  - git checkout ~= hg update
- Some (minor) differences in features
  - No rebasing (only merging)
  - No octopus merge (#parents <= 2)
- But key ideas are identical
  - Repository = working directory + store
  - Send/Receive changes between stores

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