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?

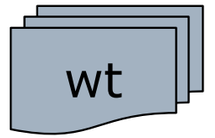
Key Idea: A Repository

- *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

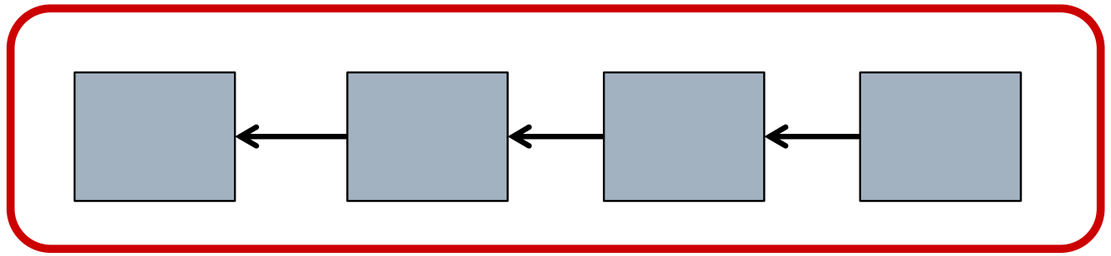
File Structure of a Repository


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

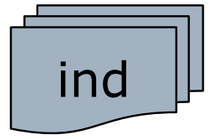
Conceptual Structure



working tree
~/my-app/

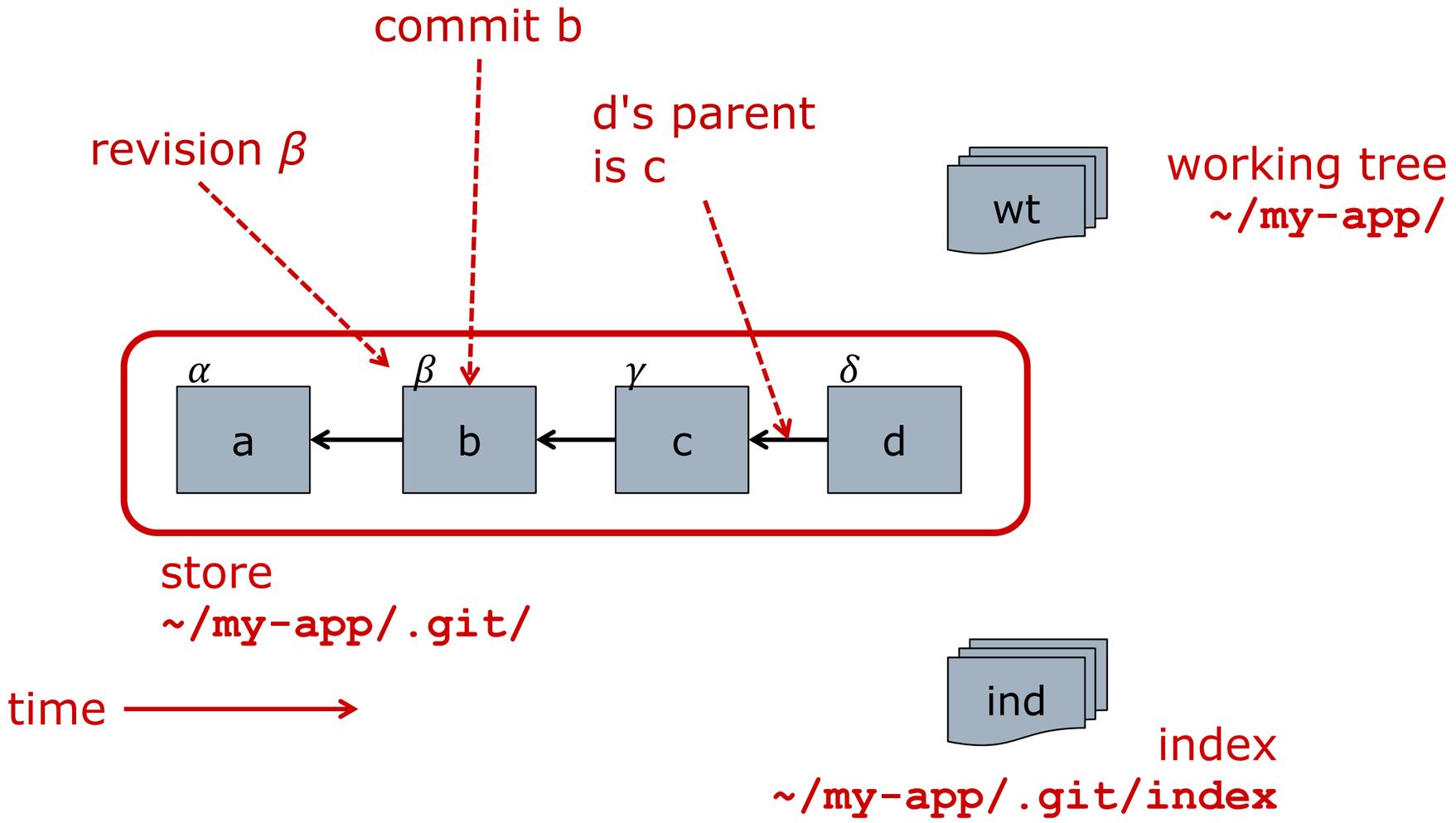


store
~/my-app/.git/



index
~/my-app/.git/index

A History of Commits

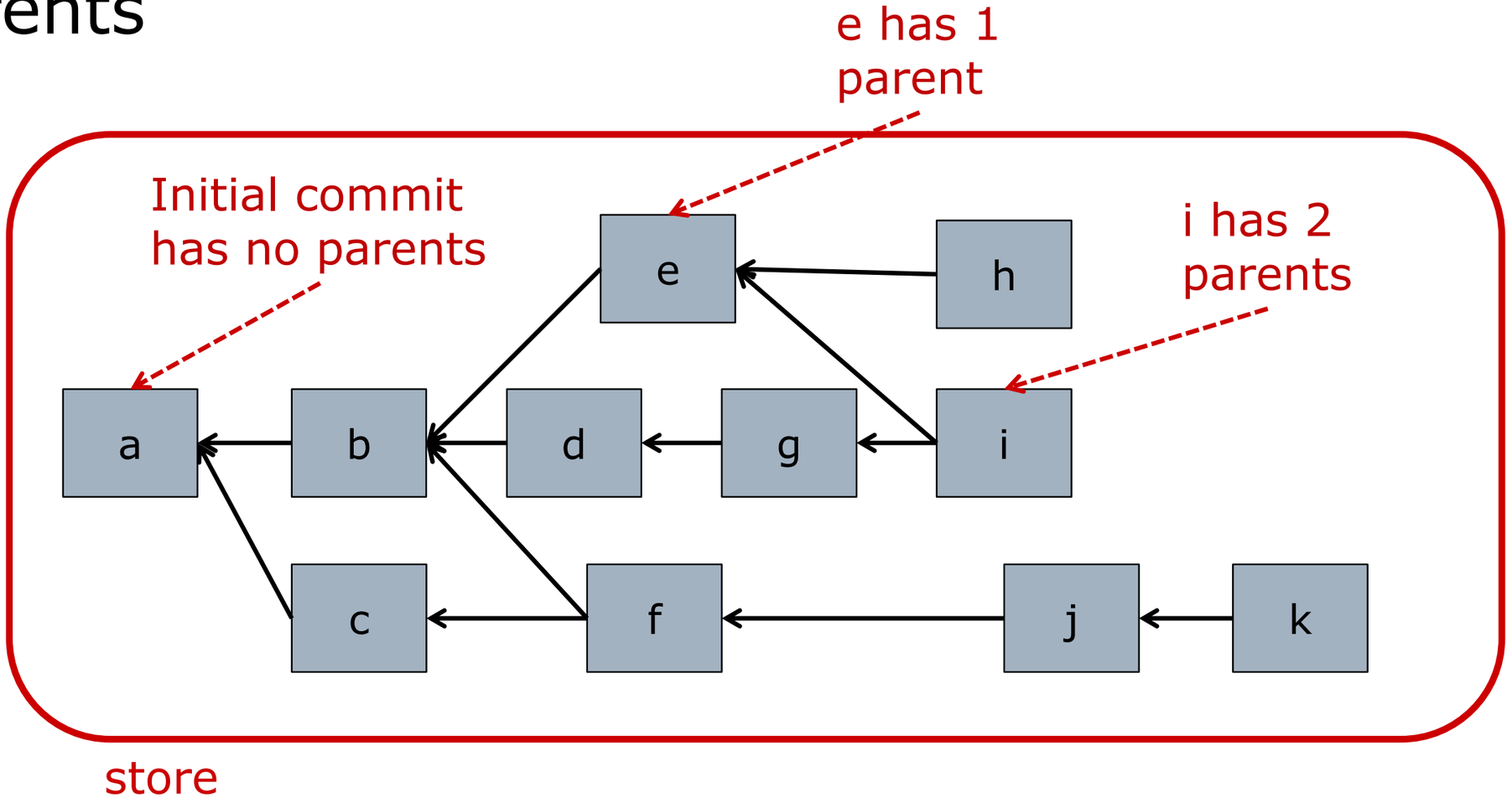


Commit: Snapshot or Delta?

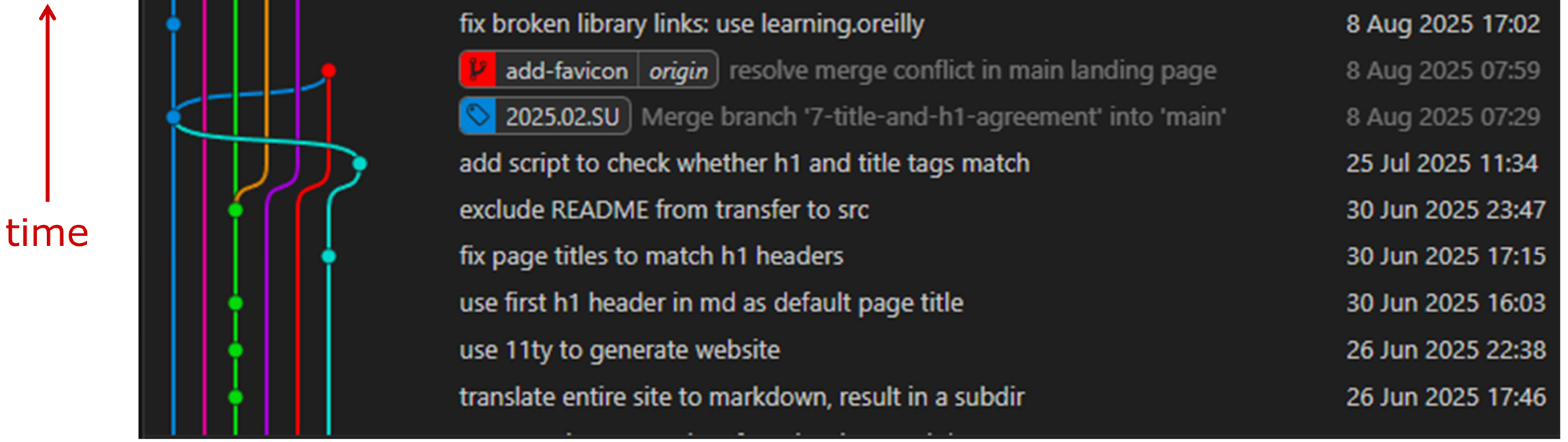
- A commit is simultaneously *both*
  1. A complete snapshot of the project at that point in time
     - aka a revision
  2. A delta of the changes made
     - aka a patch; the diff between two revisions
- Different git actions use different natures of a commit
  - **`checkout`**: Commit is a thing (e.g., "version 1.2")
  - **`show`**: Commit is a change (e.g., "fixes that bug")
- Clever data structures make both views available efficiently
  - Git objects (trees, blobs…) and references
  - Merkle tree

History is a DAG

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



Example View of DAG



Example View of DAG: ASCII Art


```
$ 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

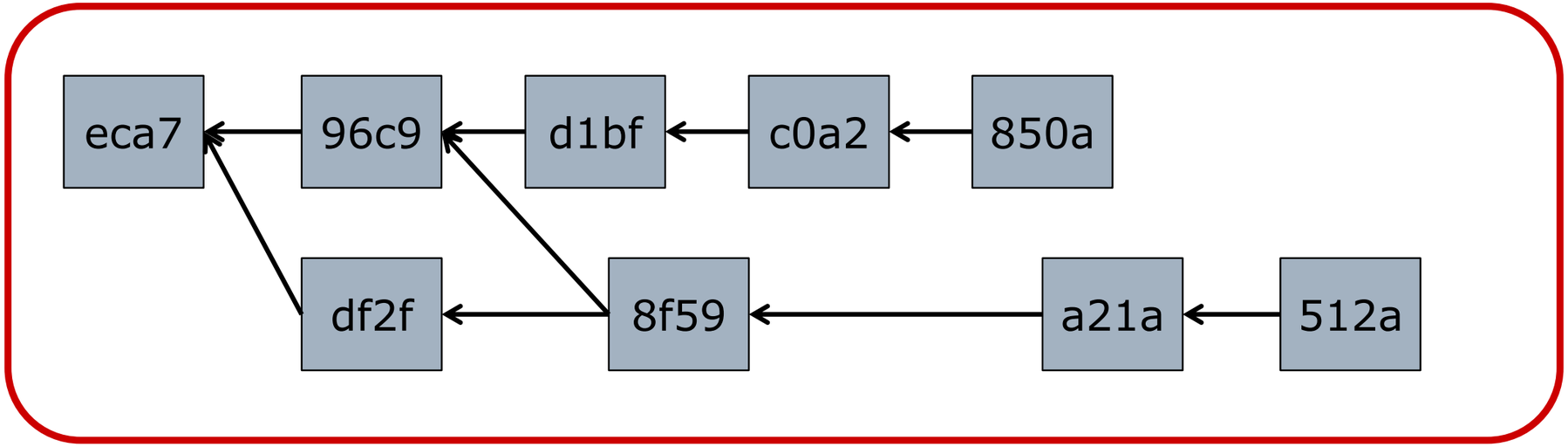
```
$ git show --name-only --no-decorate
commit 16188493c252f6924baa17c9b84a4c1baaed438b
Author: Brutus Buckeye <brutus@users.noreply.github.com>
Date:   Mon Mar 29 15:30:50 2023 +0200

    clean up css

source/stylesheets/_site.css
```

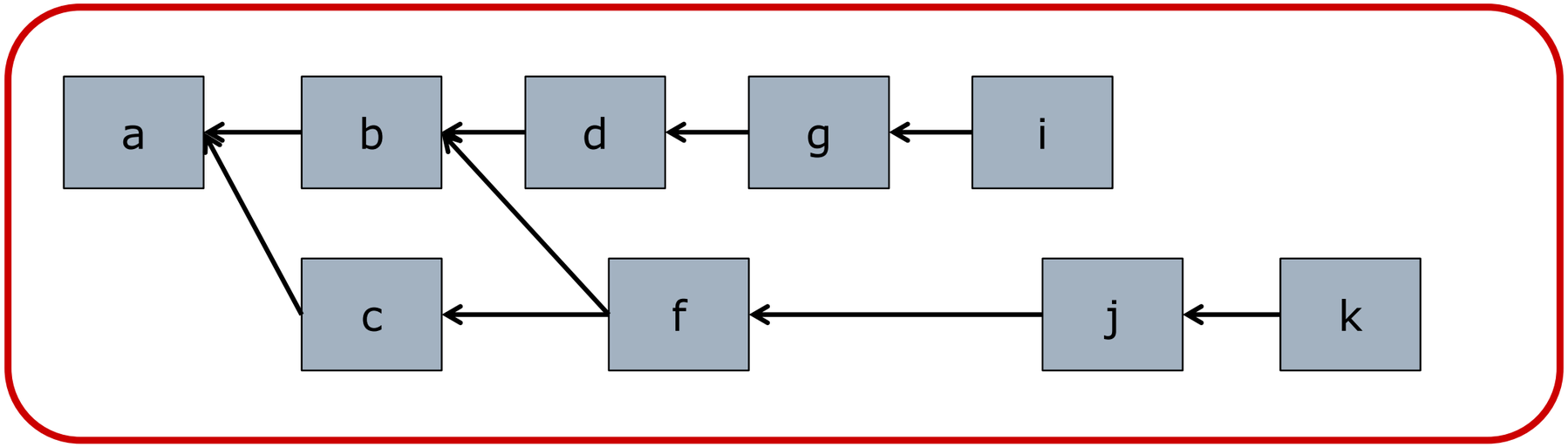
History is a DAG: With Prefixes

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



History is a DAG: Simplified

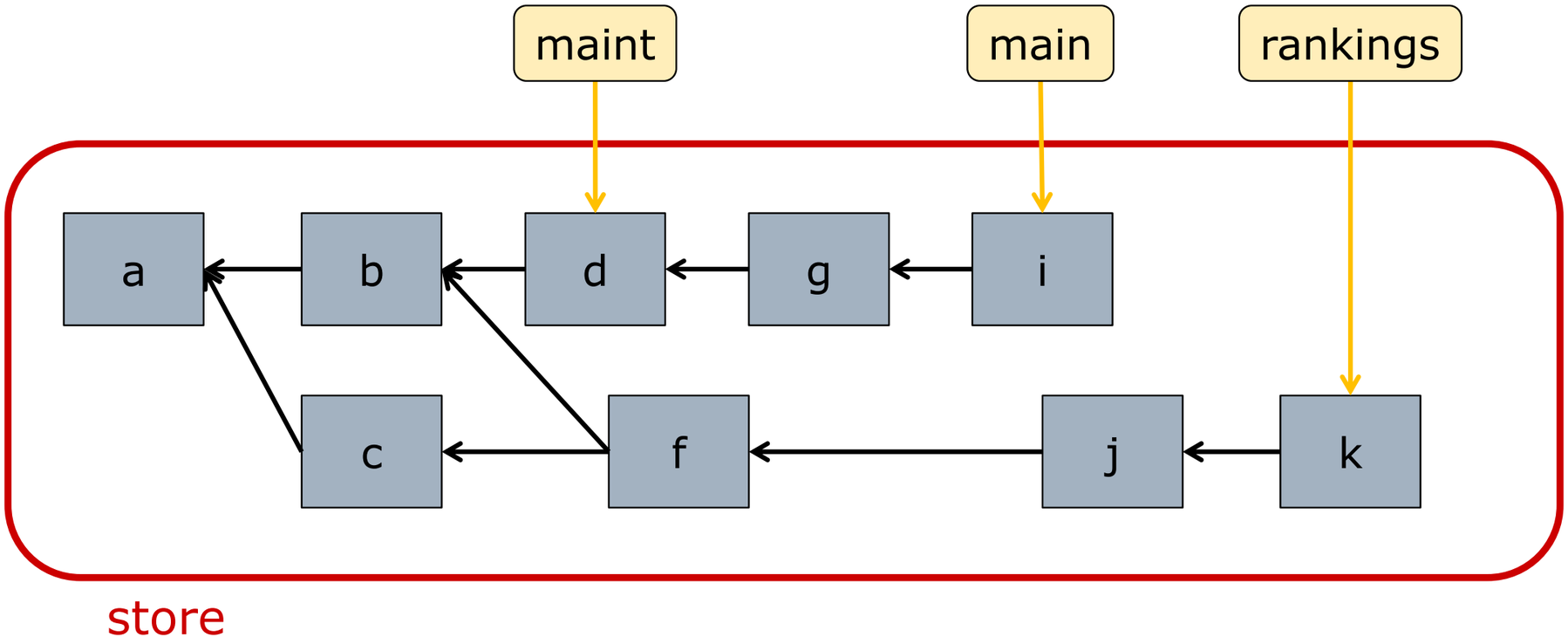
- A better picture would label each commit with its hash (prefix)



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

Nomenclature: Branch

- *Branch*: a pointer to a commit
- Not a “branch” in the DAG's shape



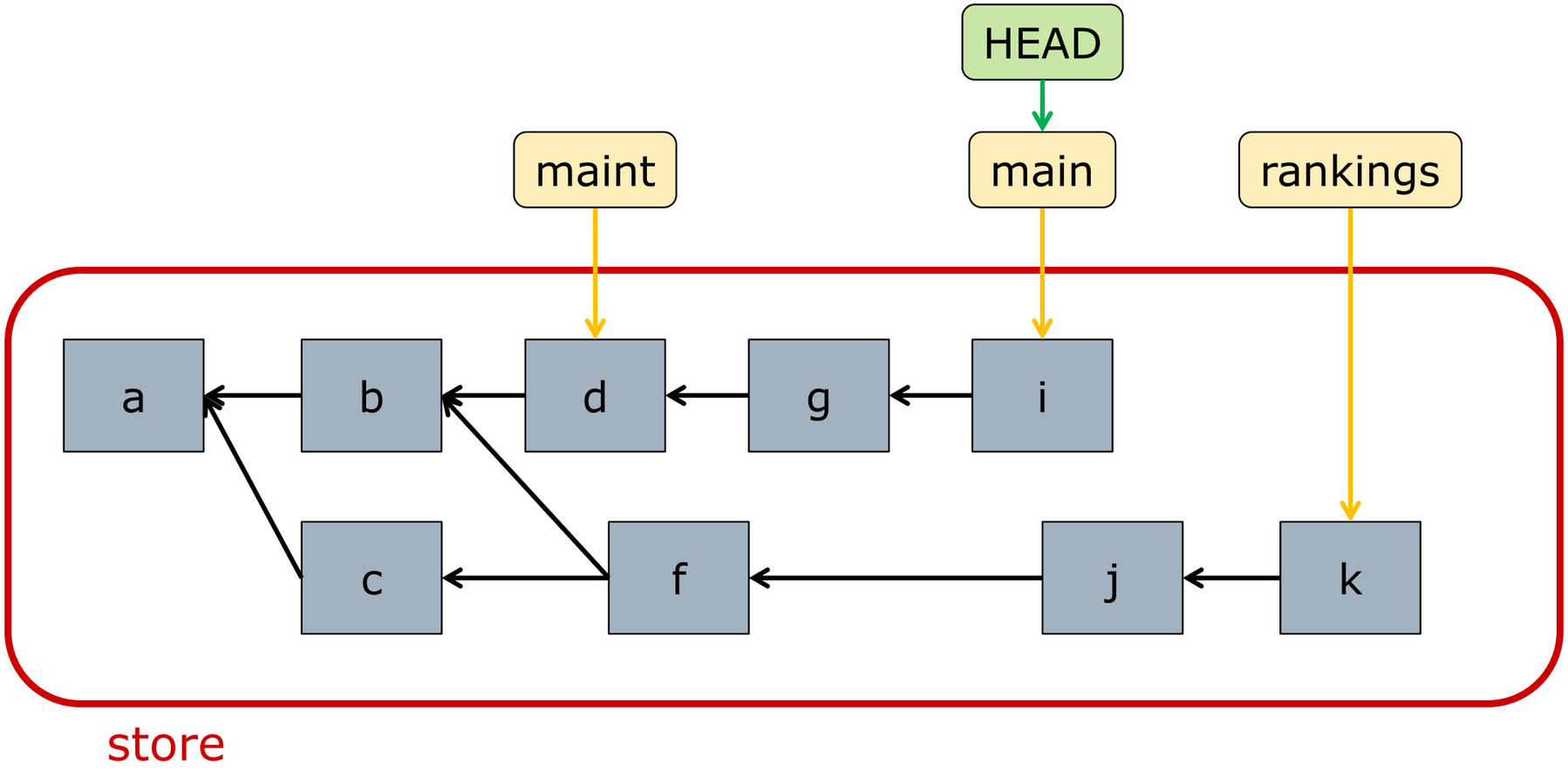
A Note on Naming Conventions

- 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
  - Default branch for many git commands
  - Convention: use “main” as the name of this default branch
  - Warning: Repos created locally use an old naming convention (“master”) by default
    - This default is user-configurable

```
init.defaultBranch main
```

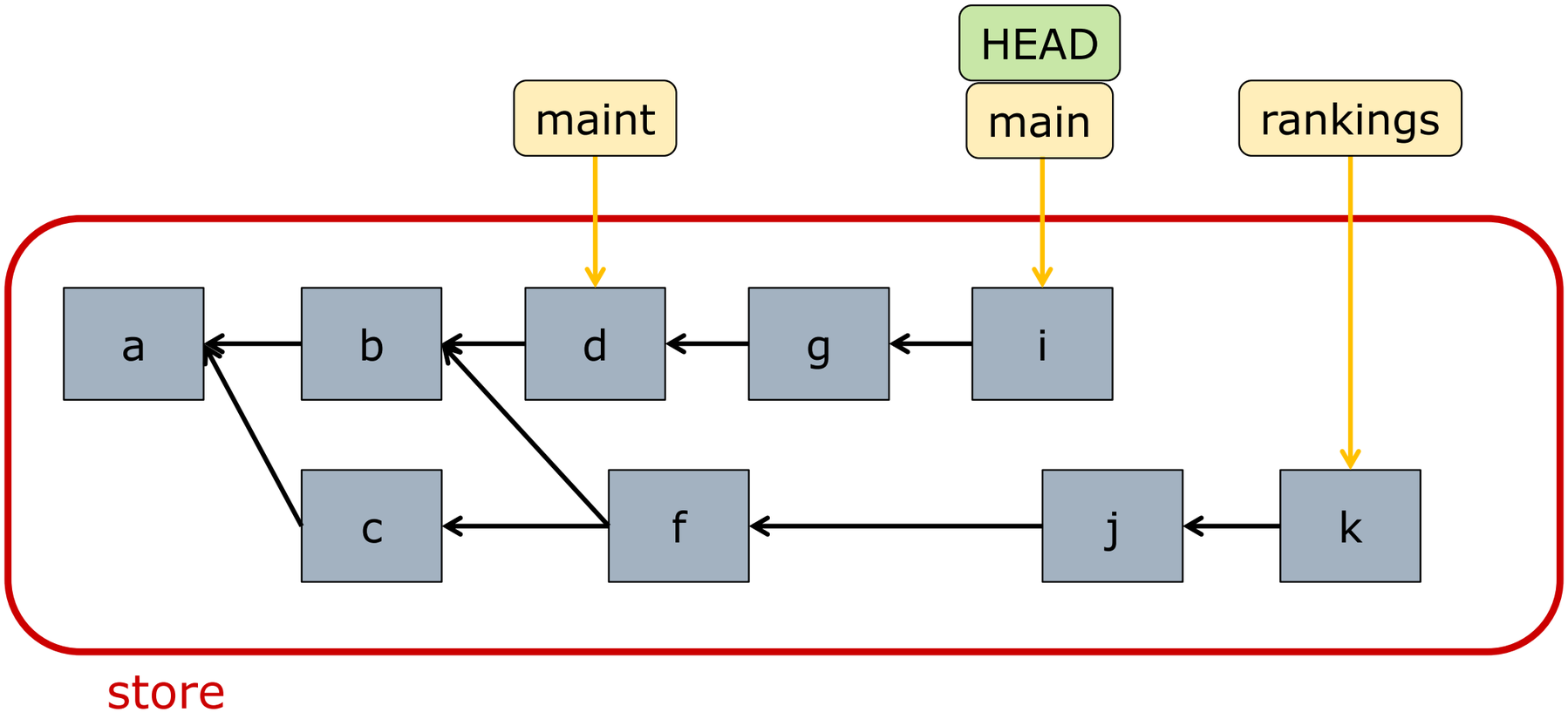
Nomenclature: HEAD

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



Nomenclature: HEAD (Attached)

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

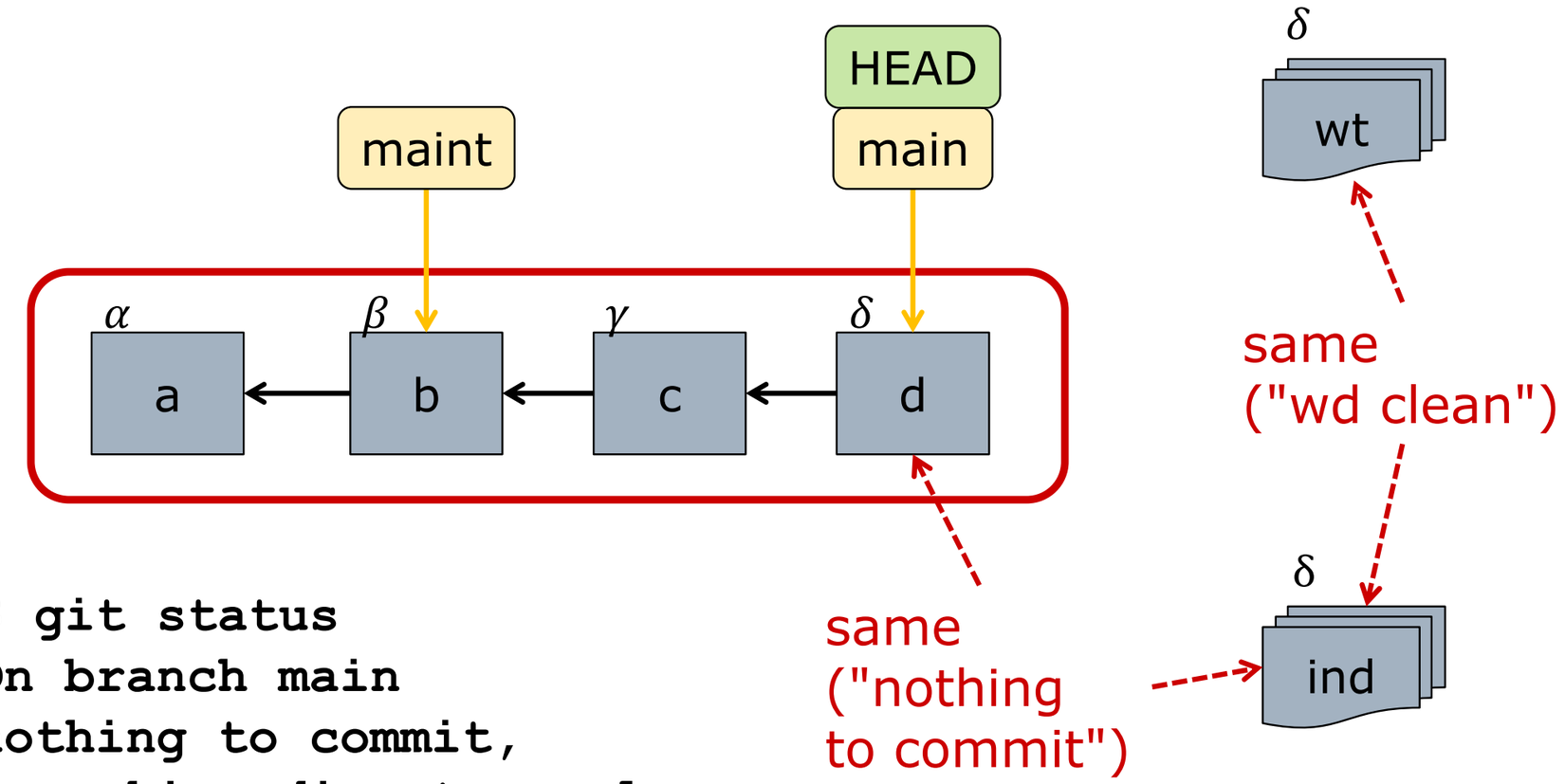


View of DAG with Branches


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

* 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
```

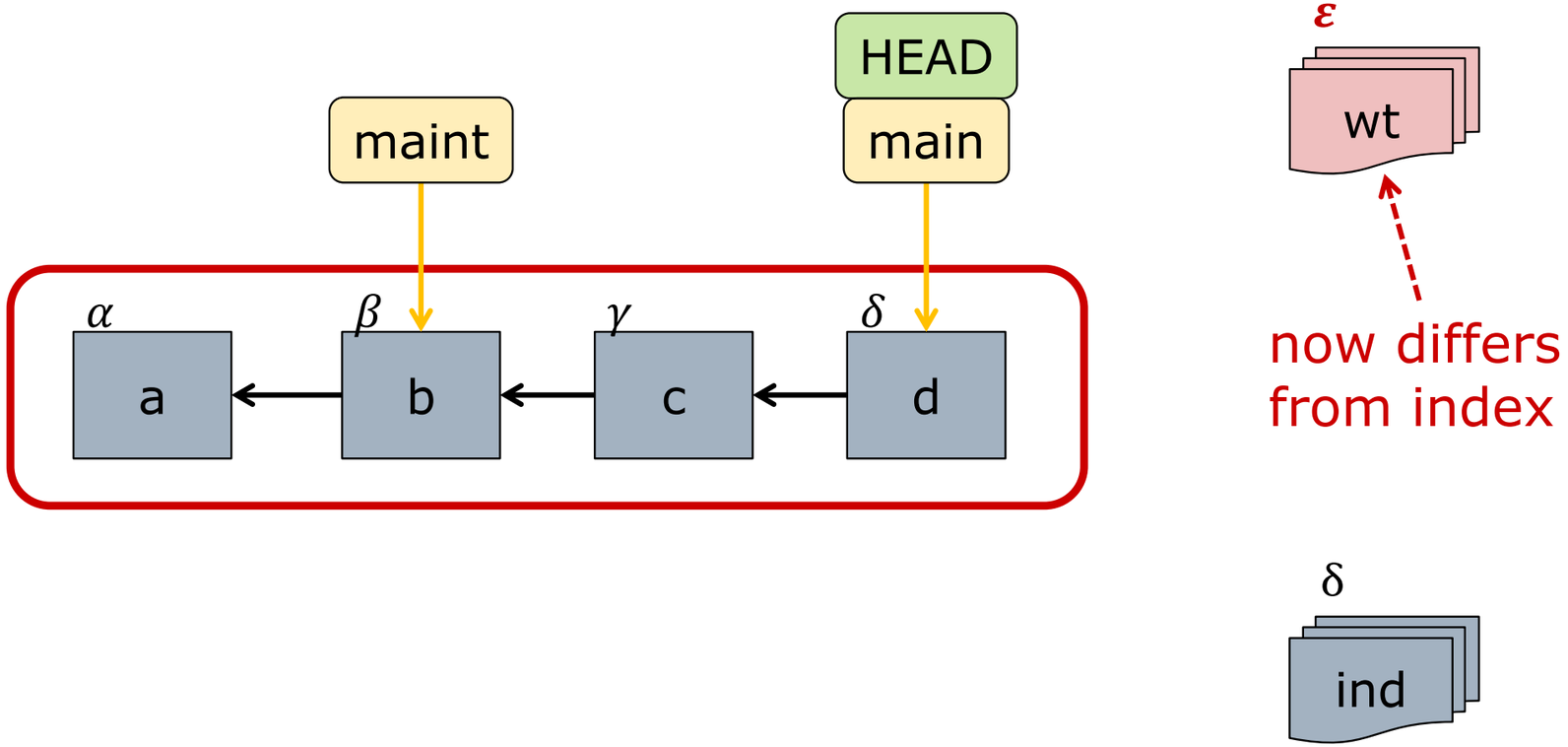
A “Clean” Repository




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

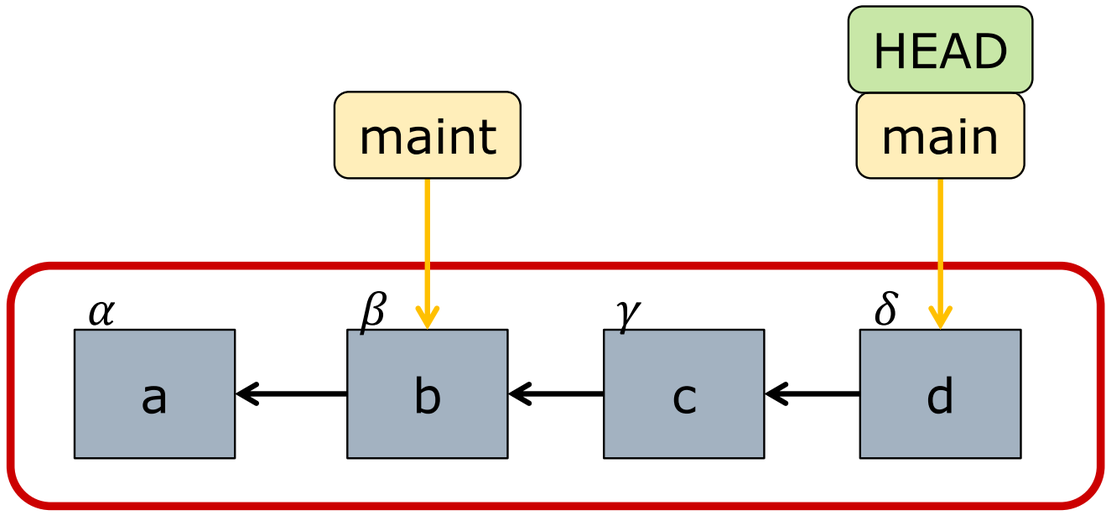
Edit Files in Working Tree

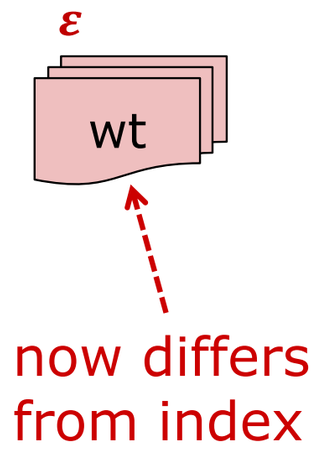
- Add files, remove files, edit files…

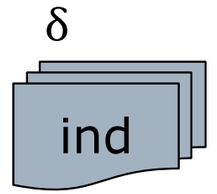


Edit Files in Working Tree: Check Status

- Add files, remove files, edit files…



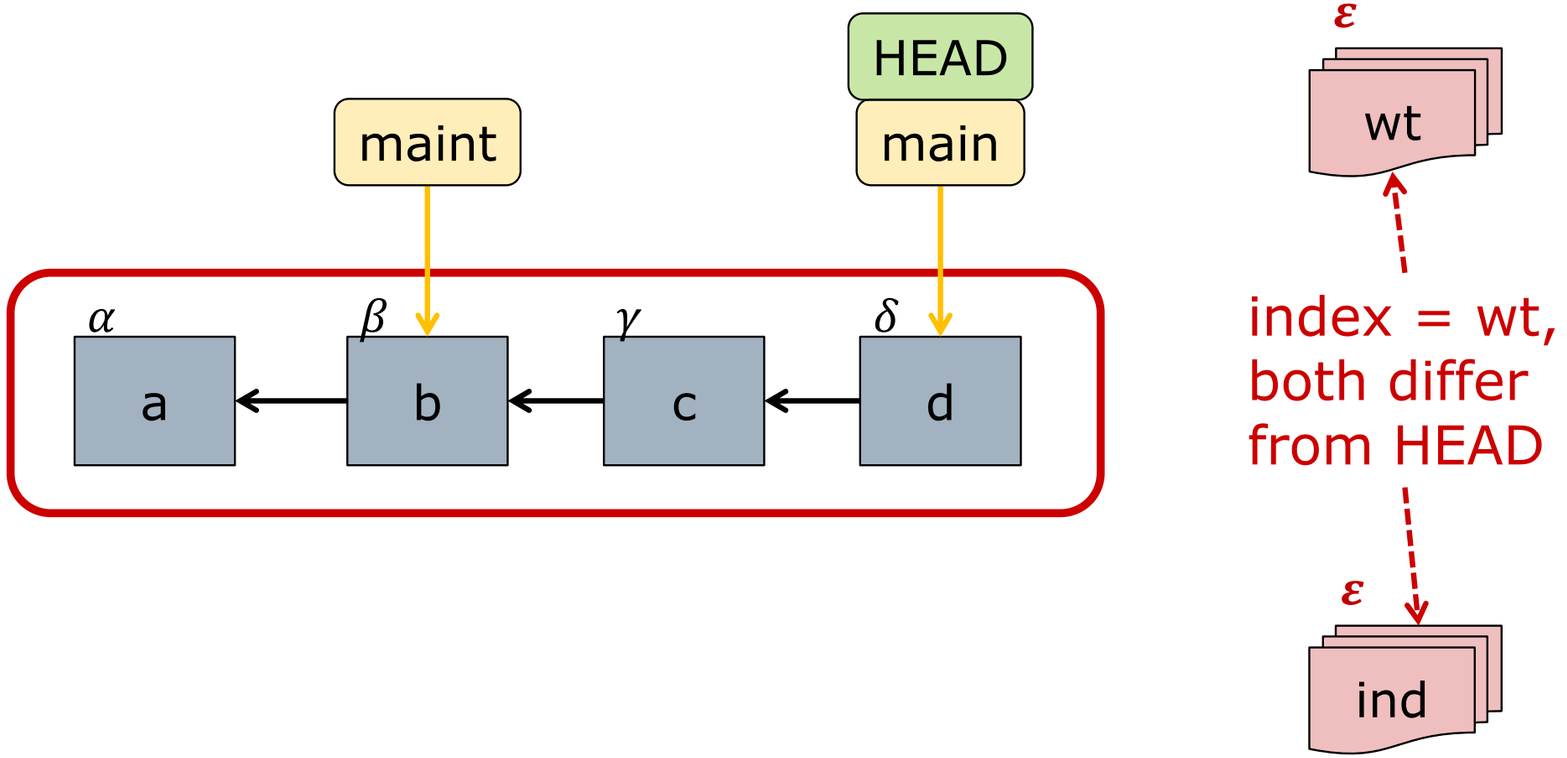





```
$ git status
On branch main
Changes not staged for commit:
    modified: css/demo.css
```

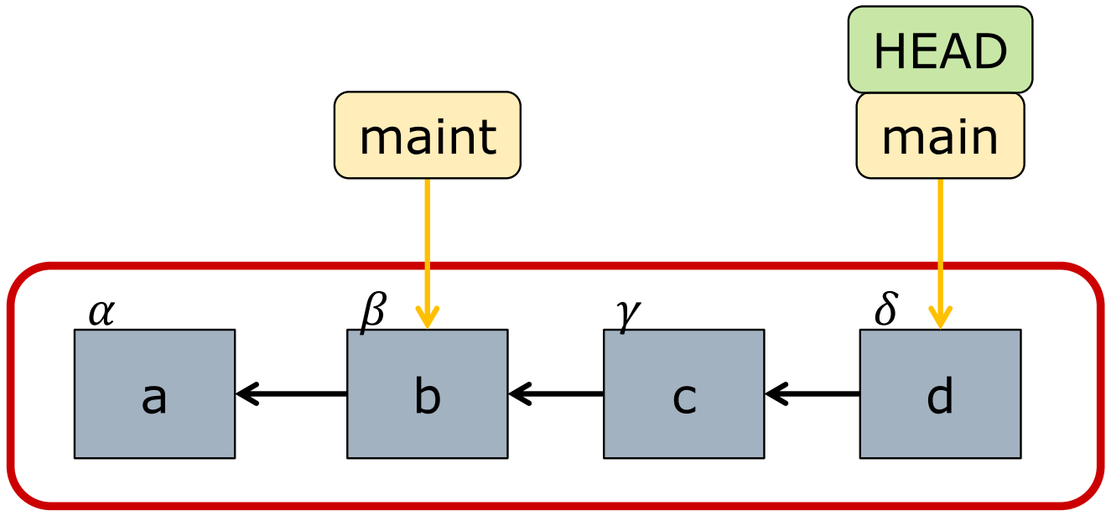
Add: Working Tree → Index

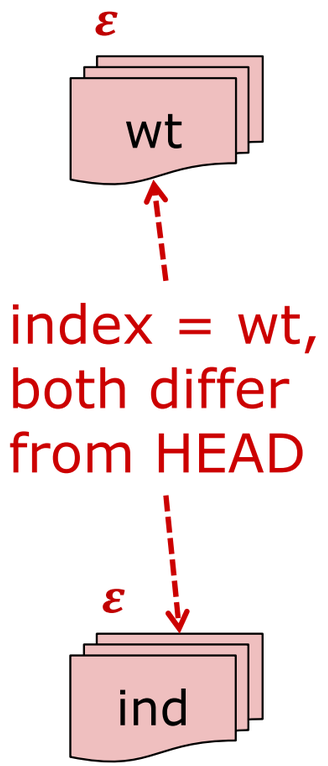

```
$ git add . # current directory, and below
```



Add: Working Tree → Index: Check Status


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

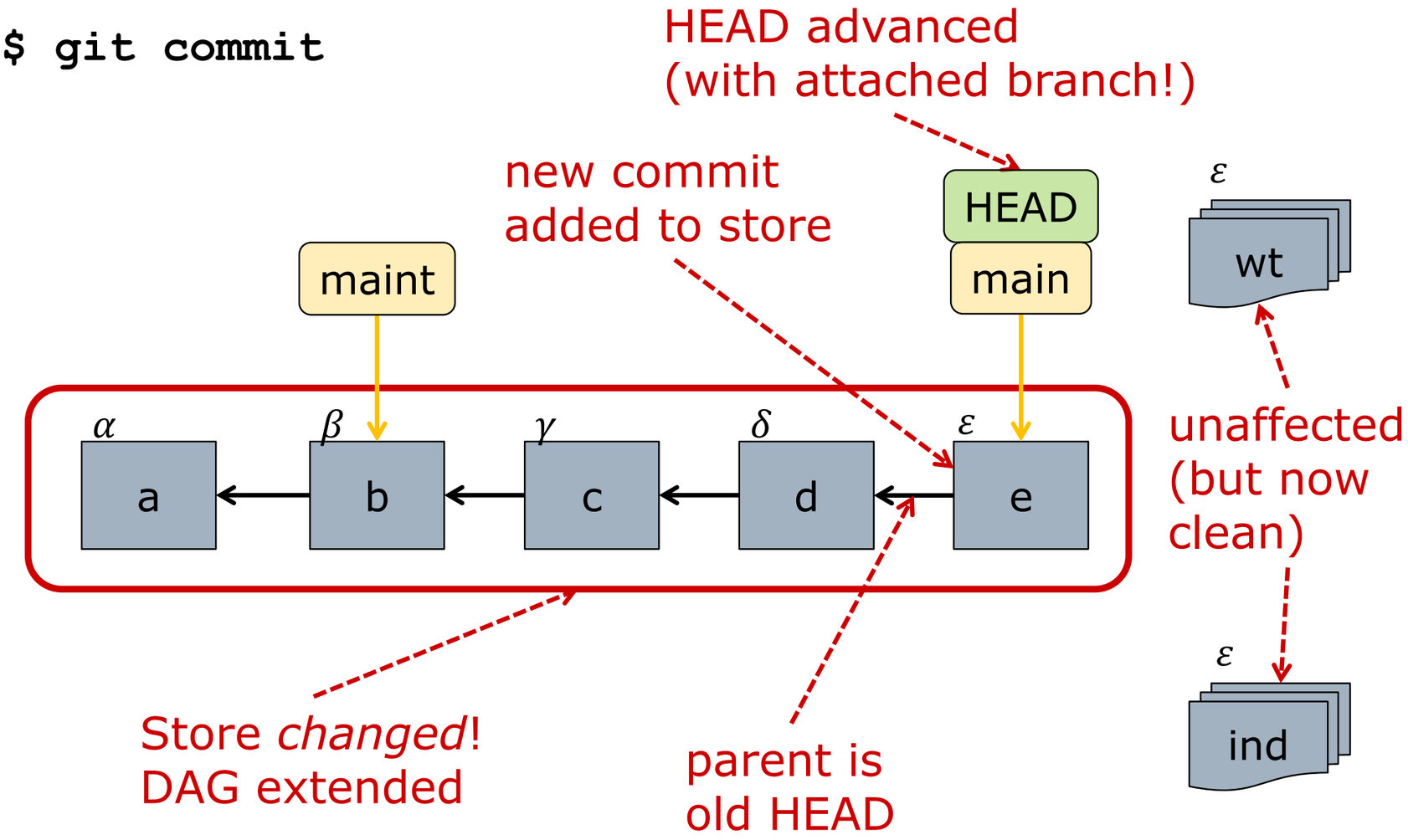




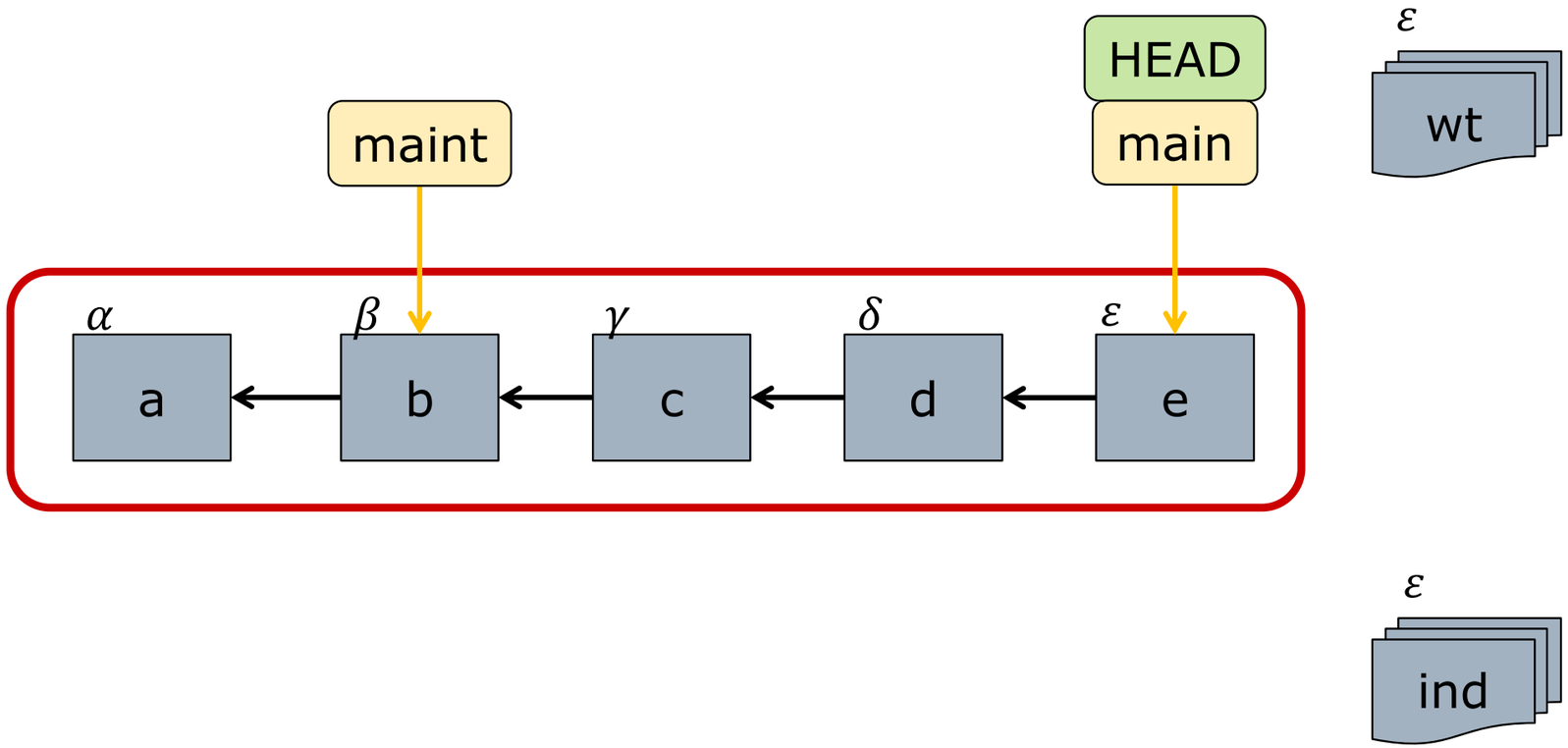

```
$ git status
On branch main
Changes to be committed:
    modified: css/demo.css
```

Commit: Index → Store


```
$ git commit
```

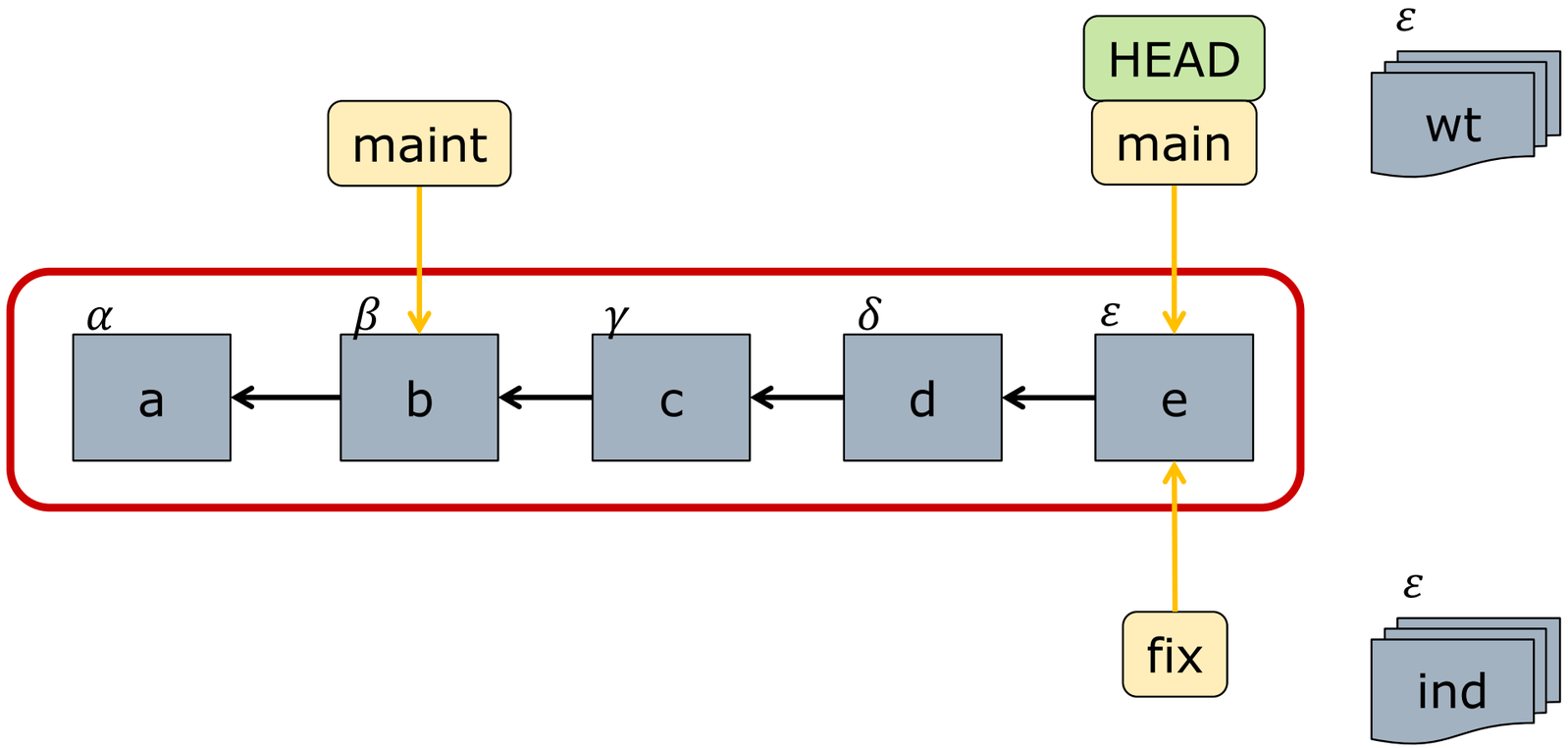


The (New) State of Repository



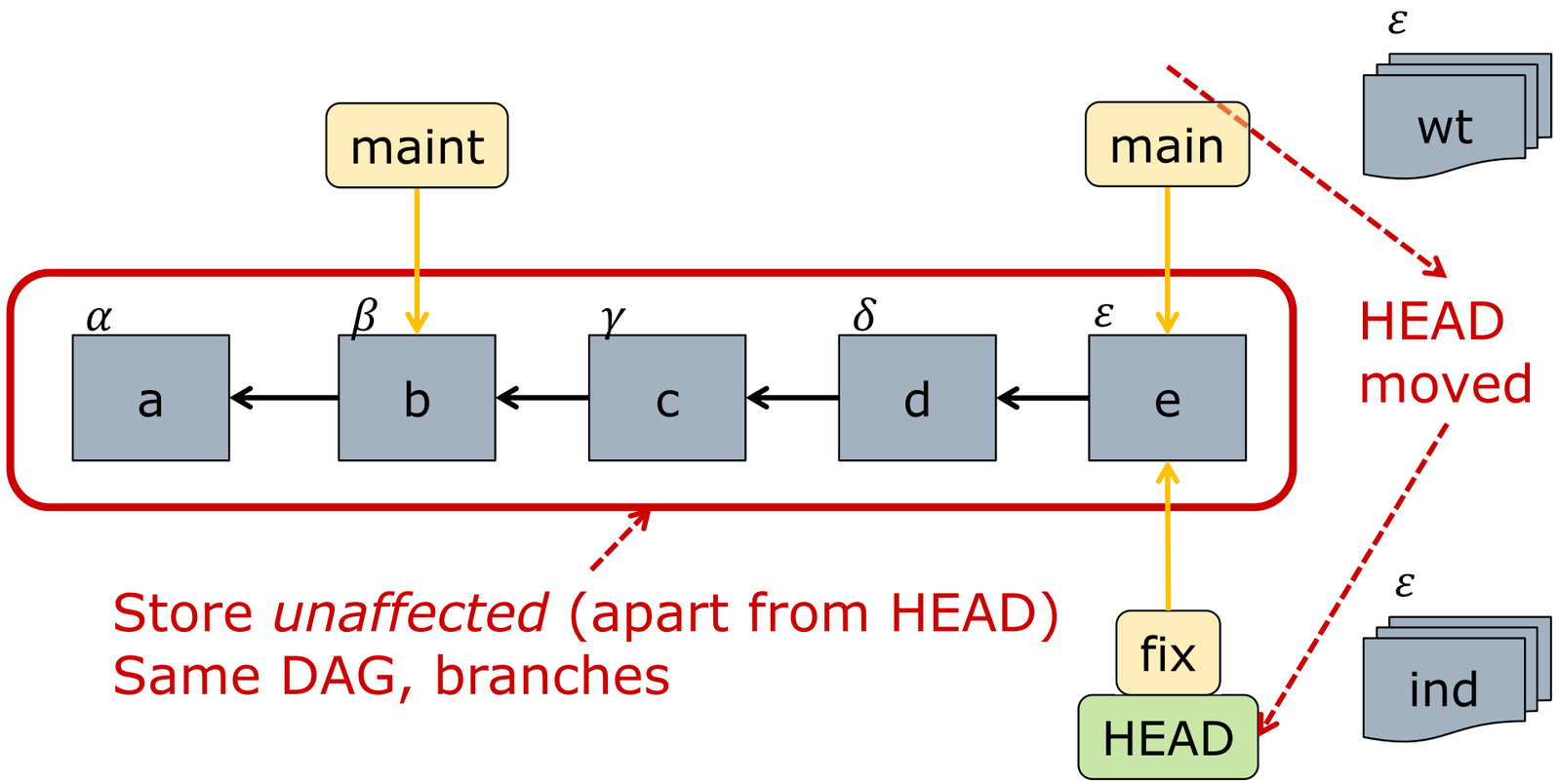
Creating a New Branch


```
$ git branch fix
```



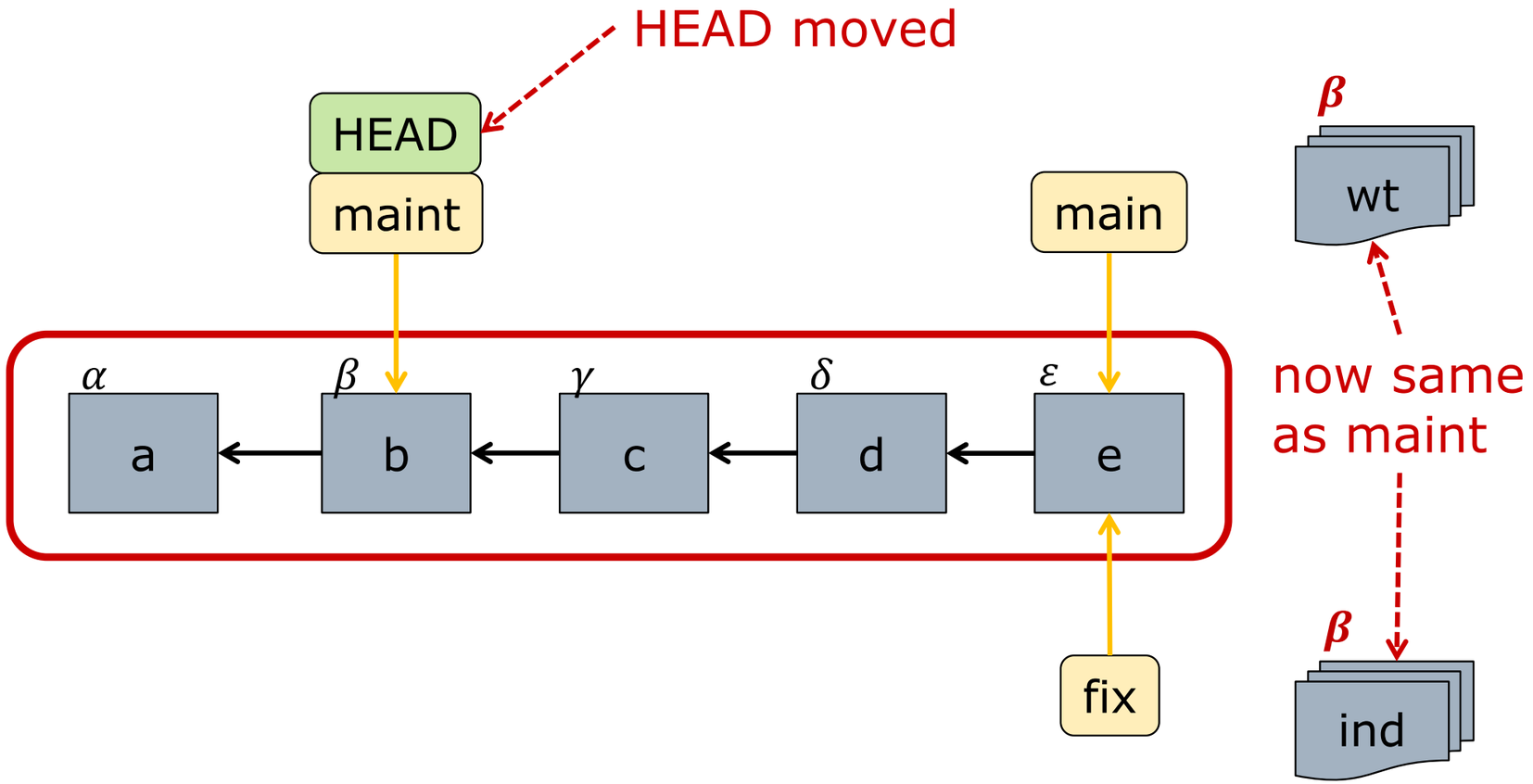
Checkout: Changing Branch (1)


```
$ git checkout fix
```



Checkout: Changing Branch (2)

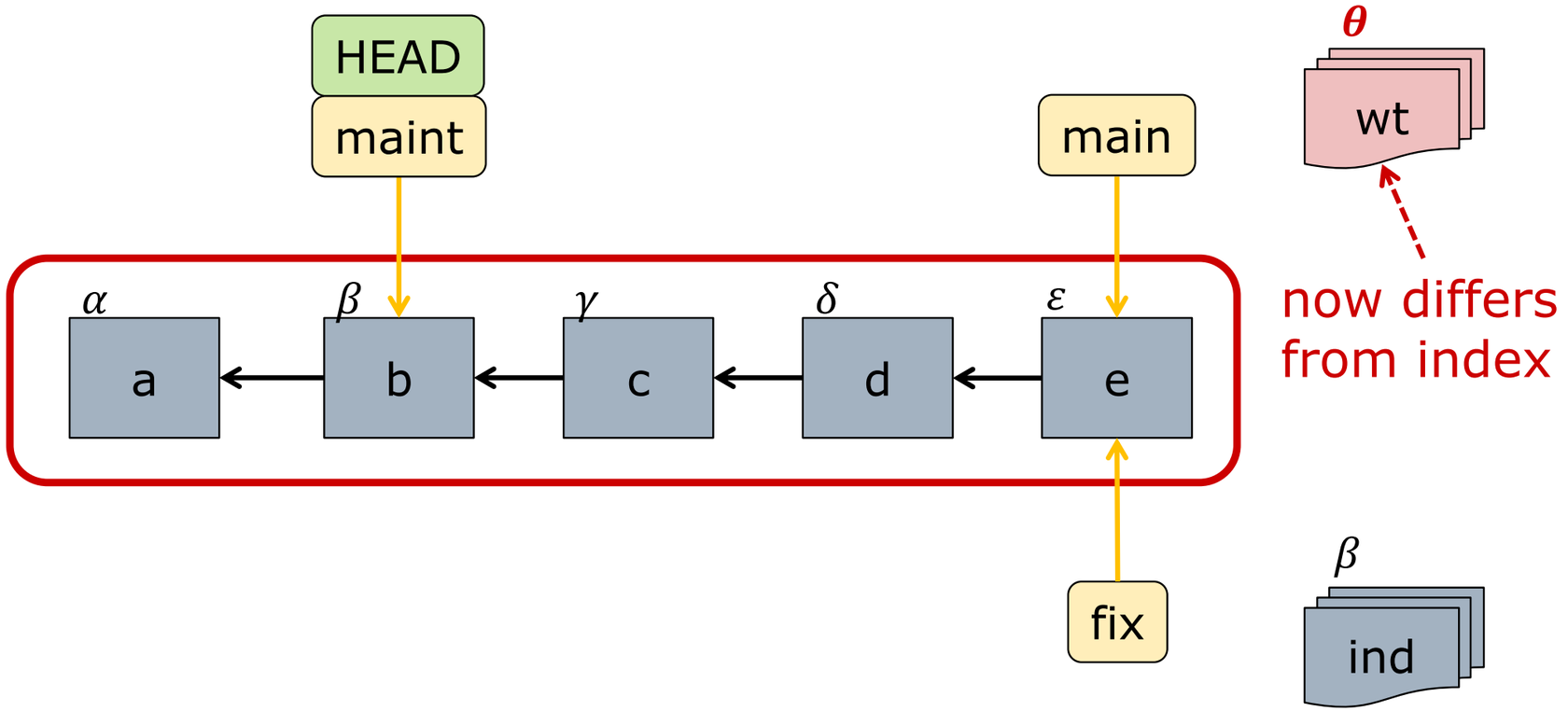

```
$ git checkout maint
```



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

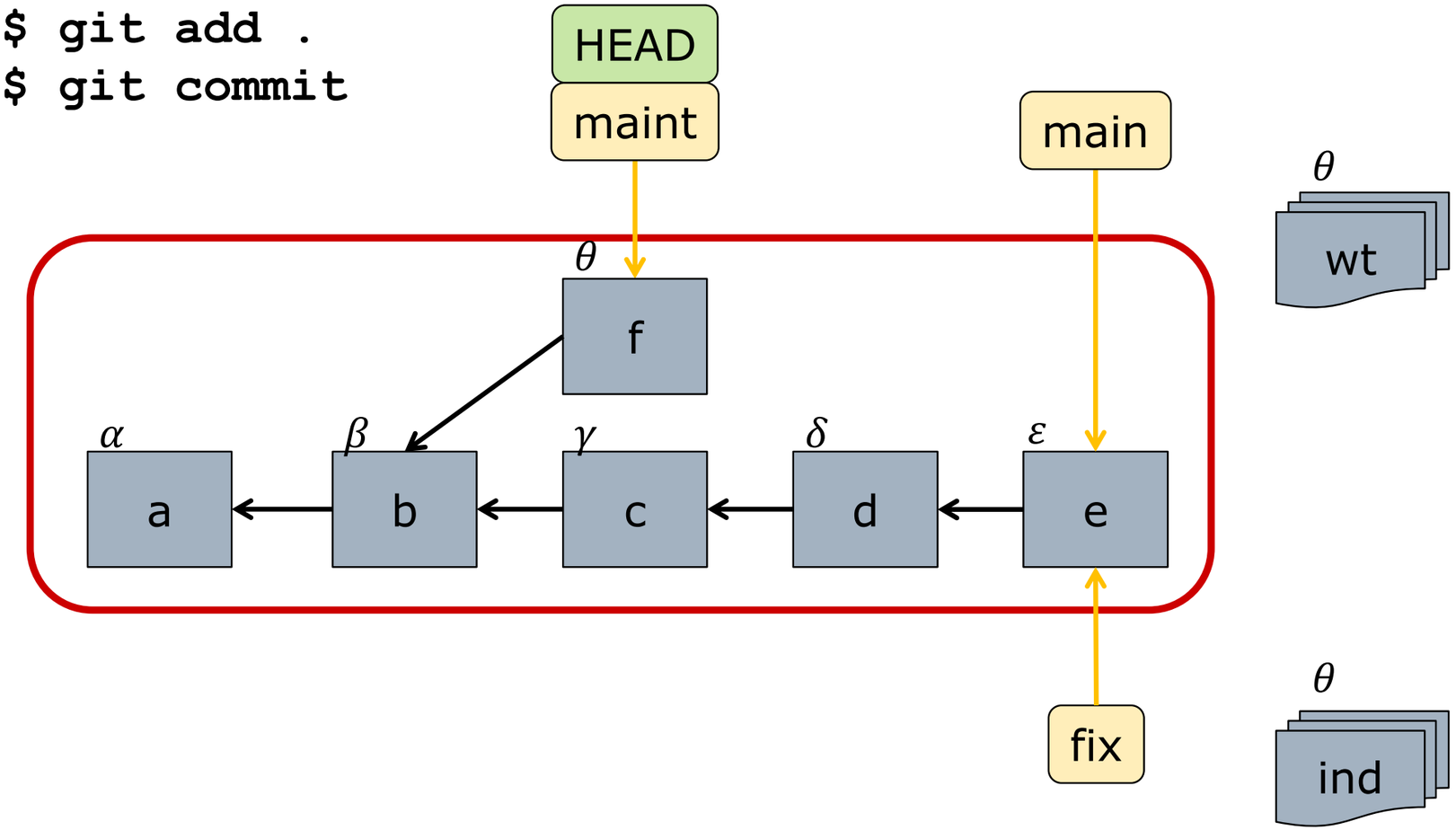
Edit Files in Working Tree: Different Branch

- Add files, remove files, edit files…



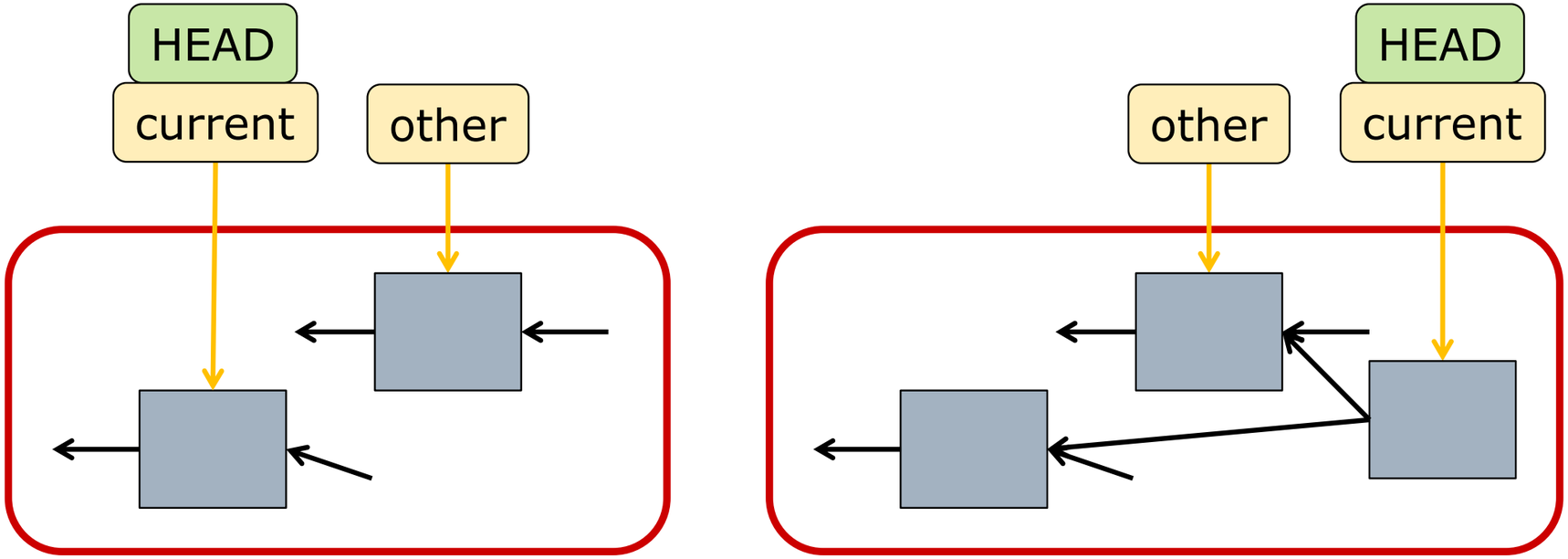
Add & Commit: Update Store


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



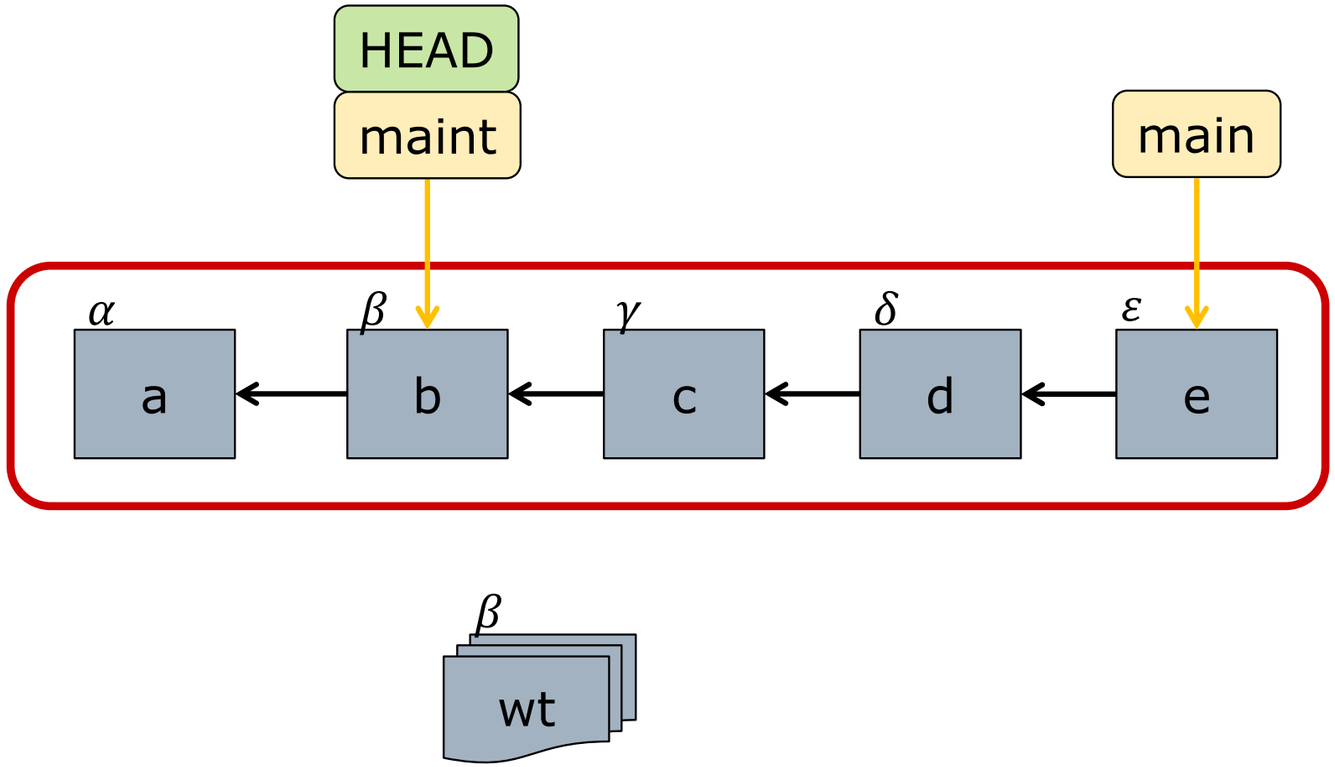
Merge: Bringing History together

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



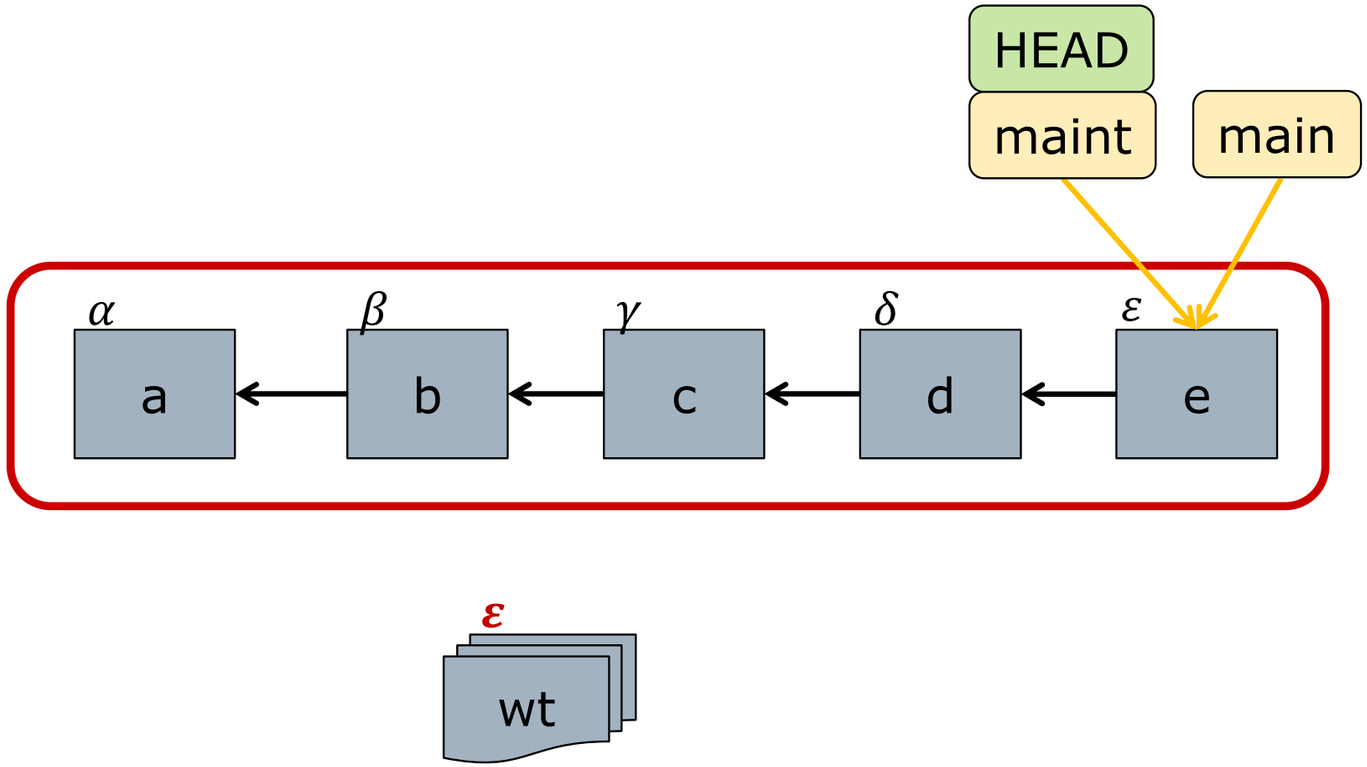
Merge – Case 1: Ancestor

- HEAD is an ancestor of other branch

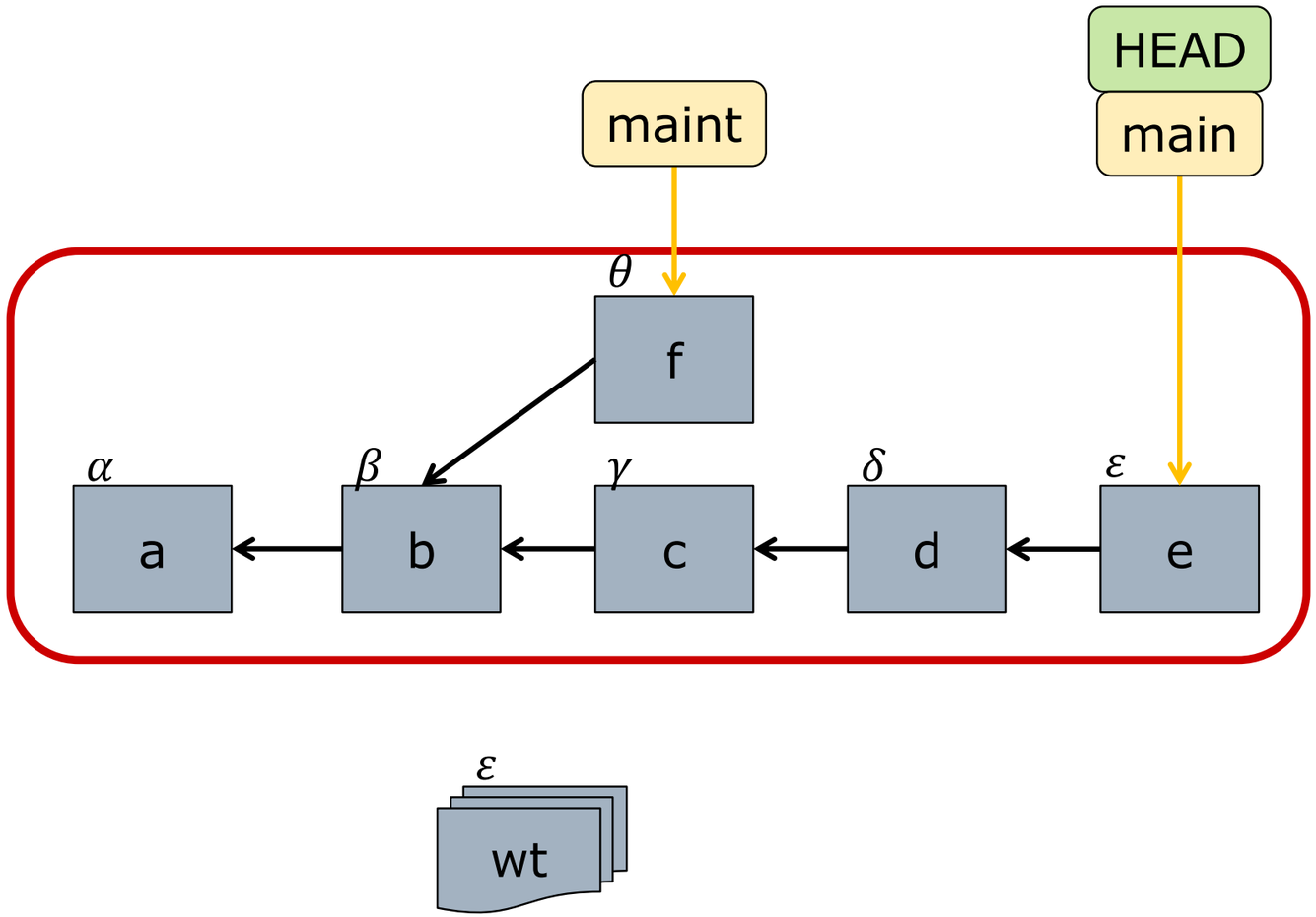


Fast-Forward Merge


```
$ git merge main
```

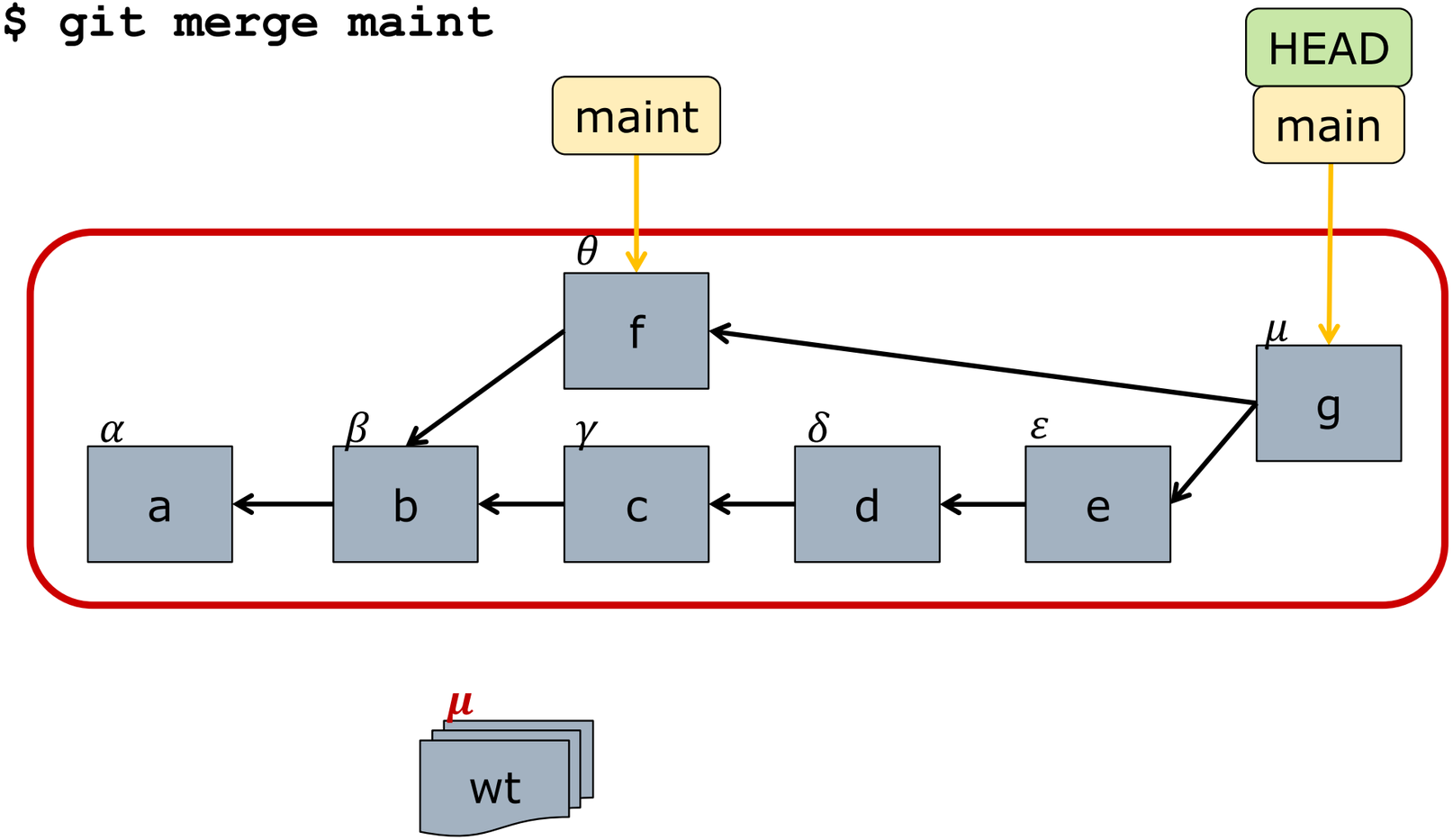


Merge – Case 2: No Conflicts



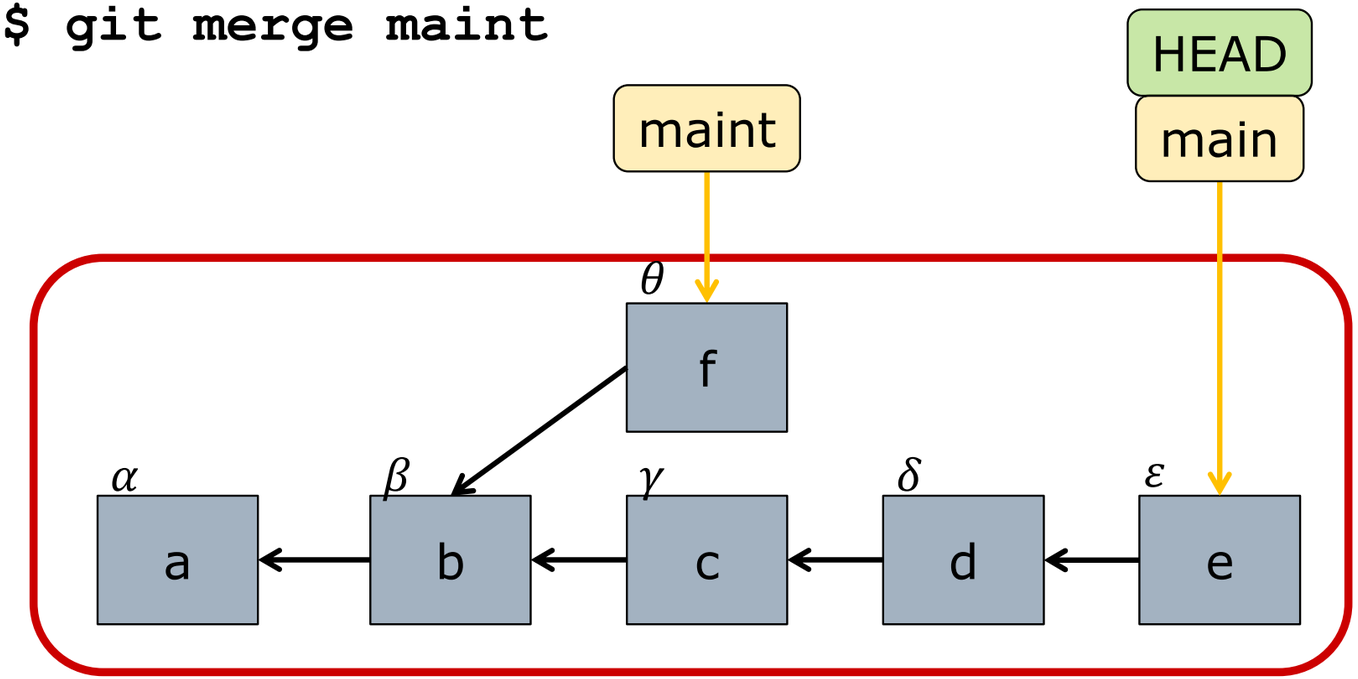
Merge Automatically Commits


```
$ git merge maint
```

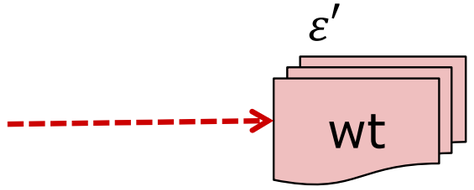


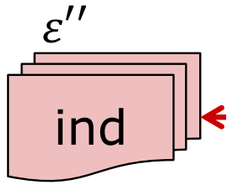
Merge – Case 3: Conflicts Exist


```
$ git merge maint
```



files with conflicts marked

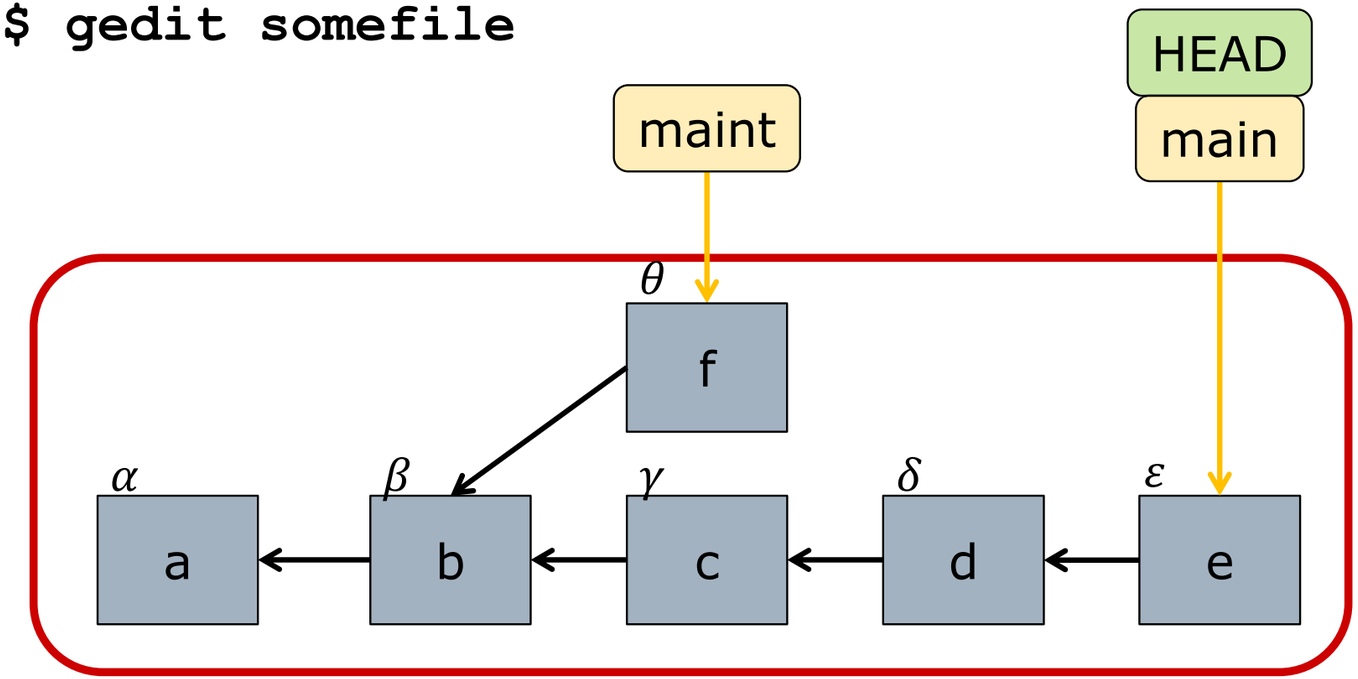




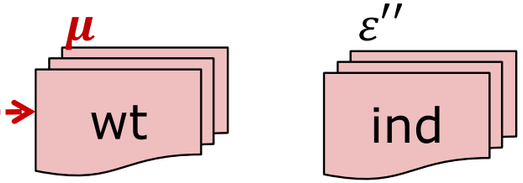
files that could be merged automatically

Merge: Resolve Conflicts


```
$ gedit somefile
```

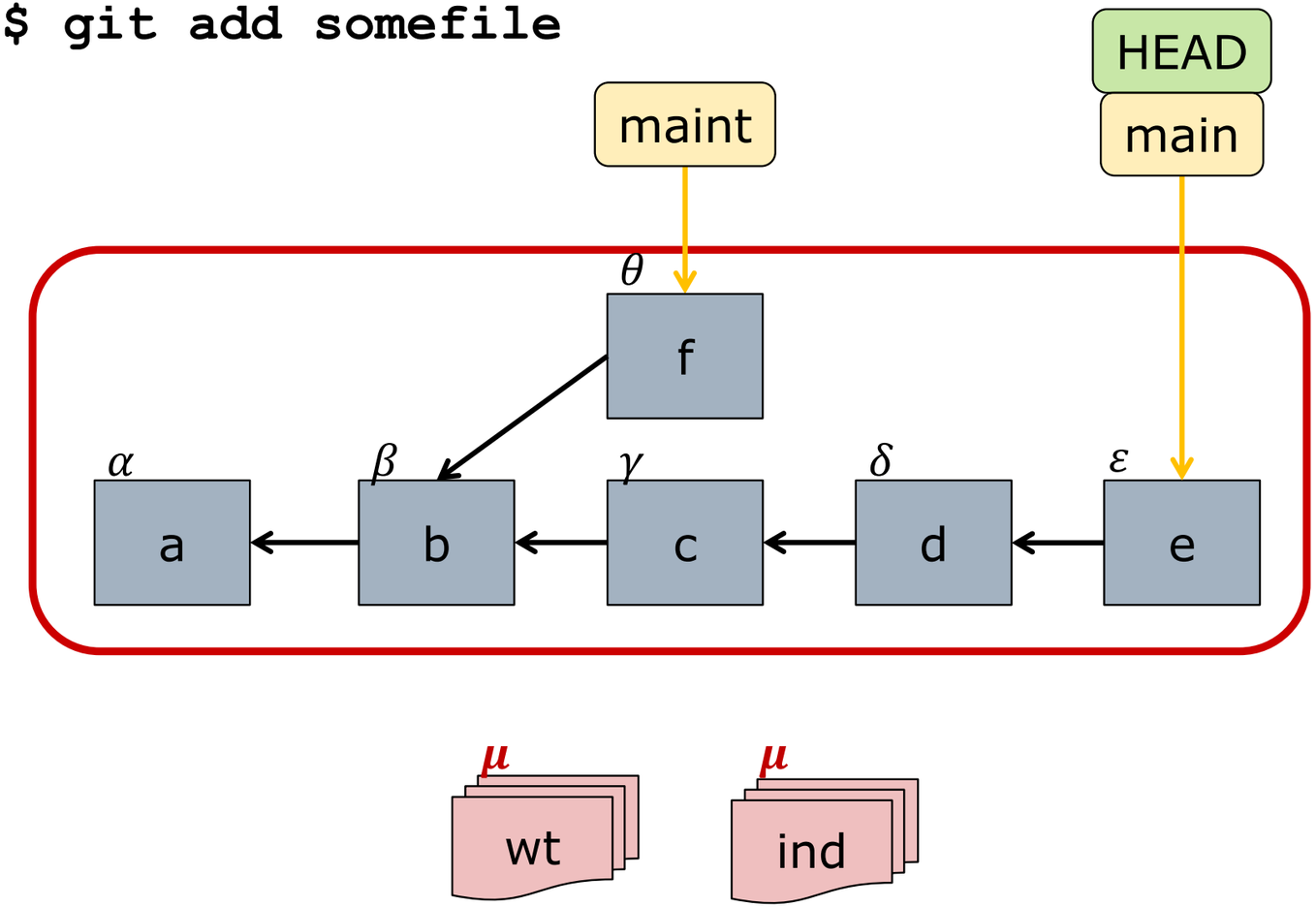


files with conflicts resolved



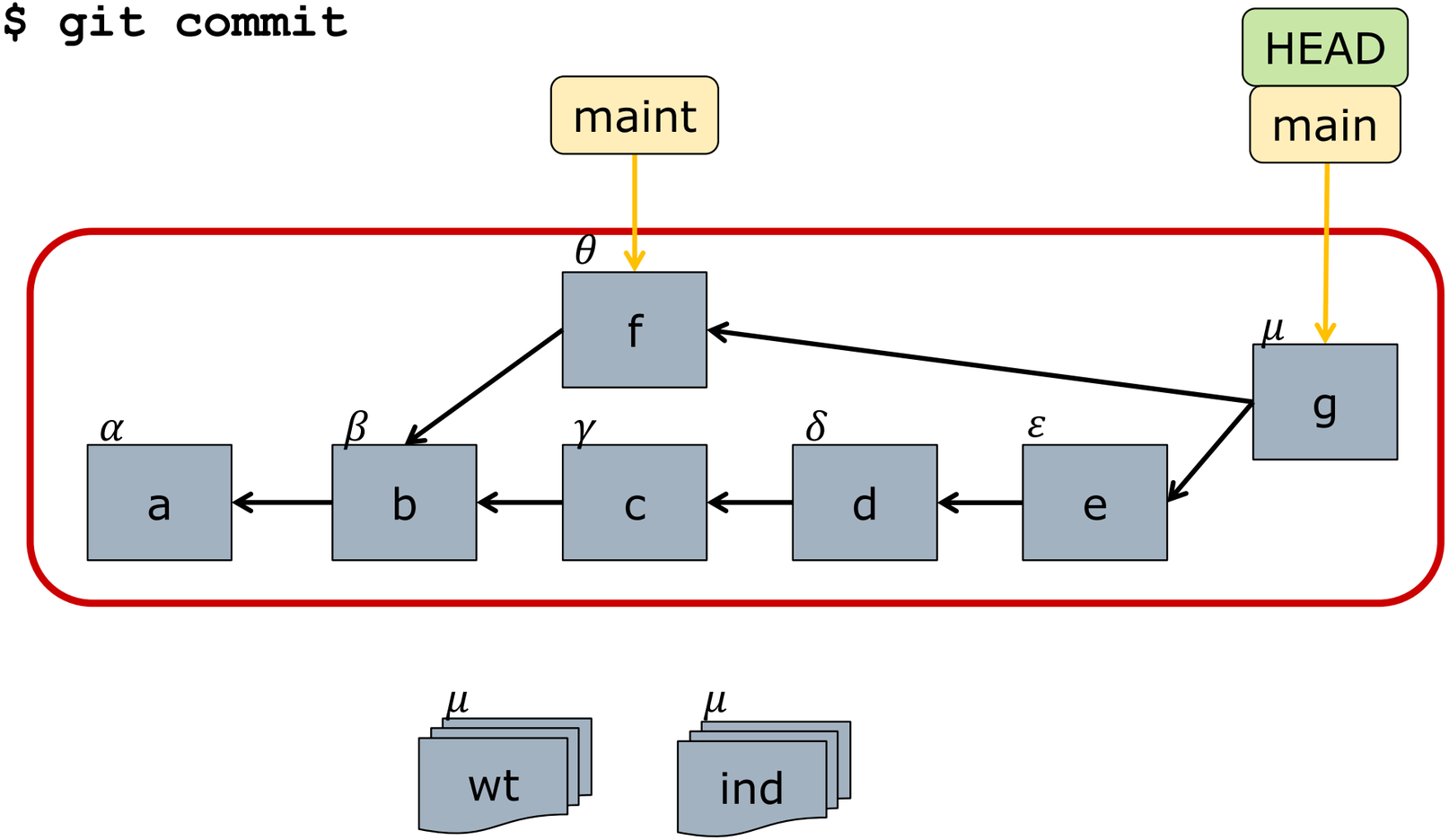
Merge with Conflicts: Add


```
$ git add somefile
```

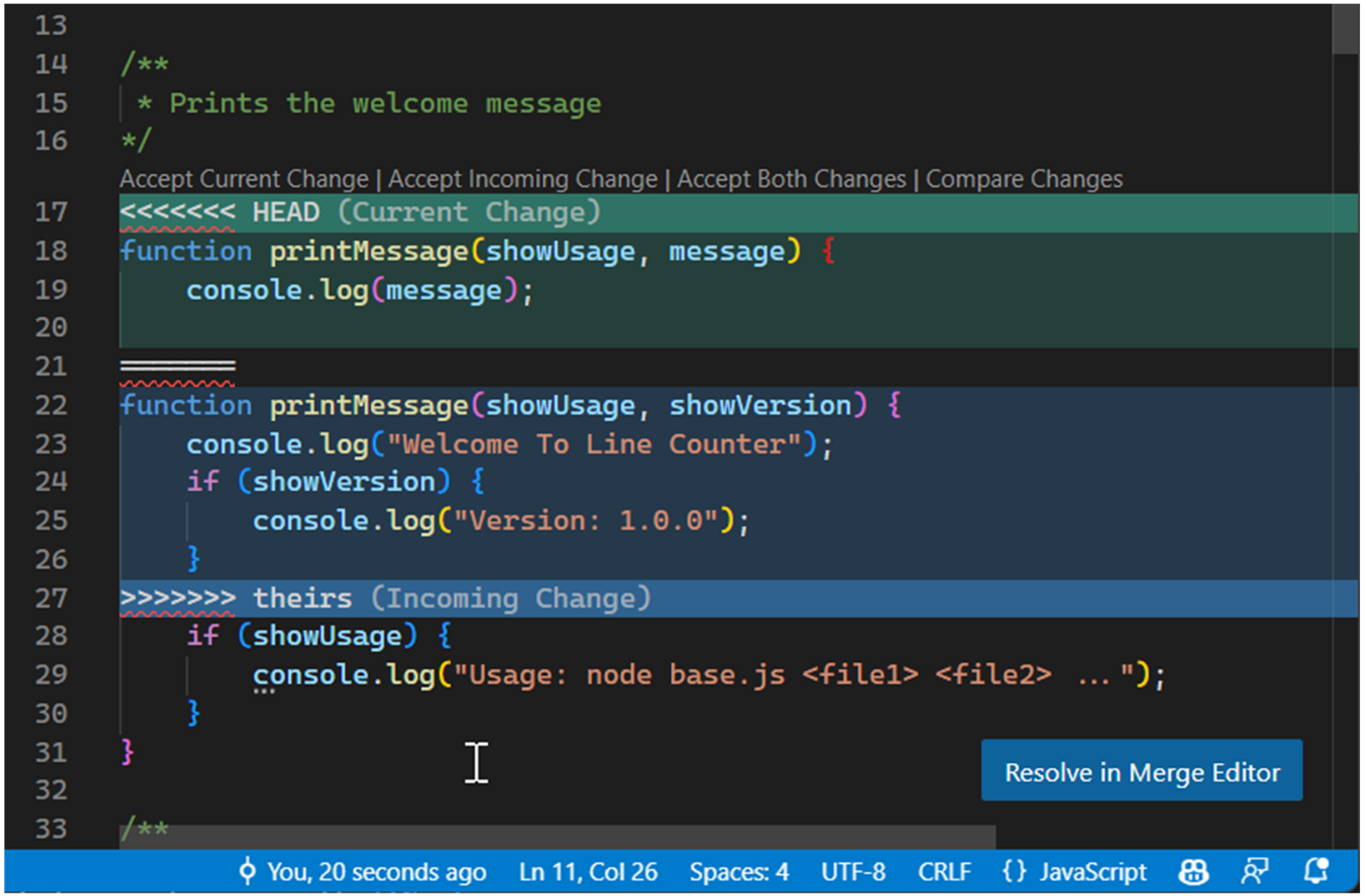


Merge with Conflicts: Commit


```
$ git commit
```

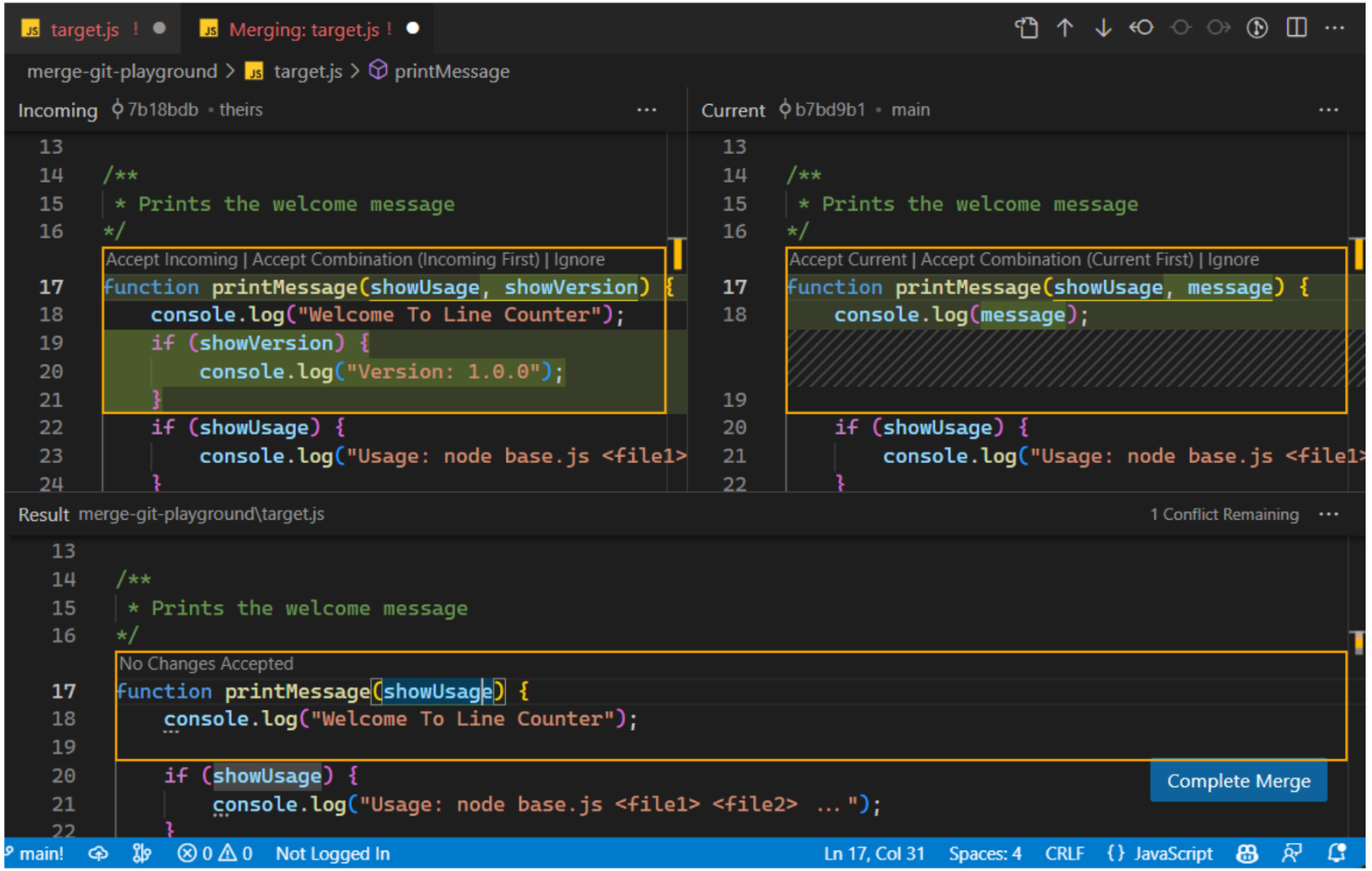


Merge: Edit to Resolve Conflicts



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

Merge: 3-way Merge Editor



MS demo resolving merge conflicts:
youtube.com/watch?v=HosPml1qkrg

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

Summary: Git Intro

- 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)