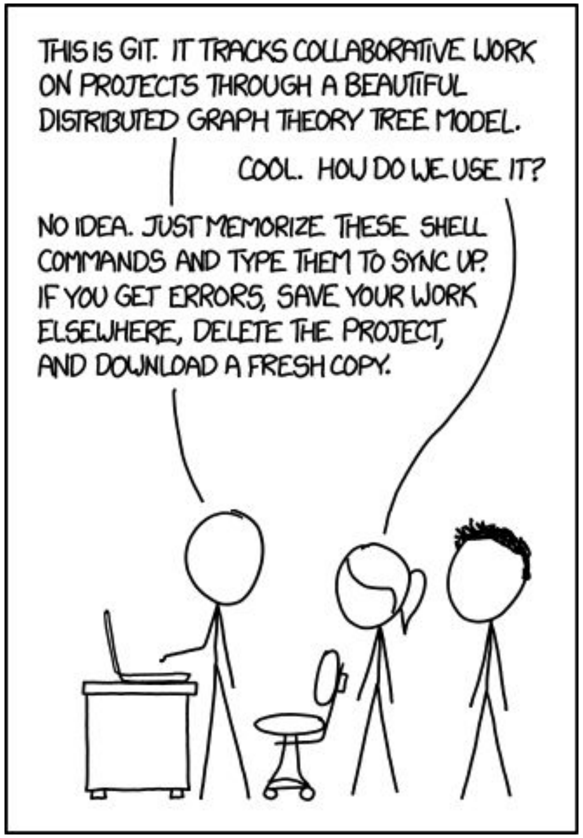
Version control and Git

CSCI-U1000052

Credit to UW CSE 403


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.

Why use version control?

Common App Essay

11:51pm

Why use version control?

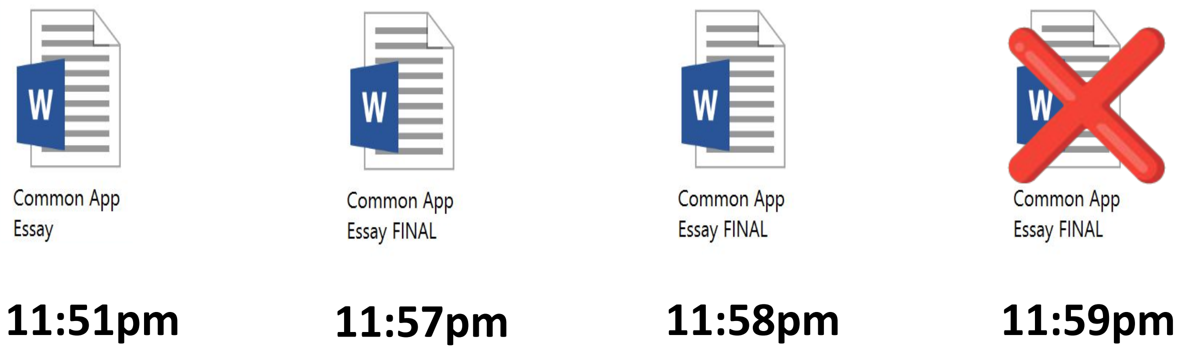
Common App Essay

11:51pm

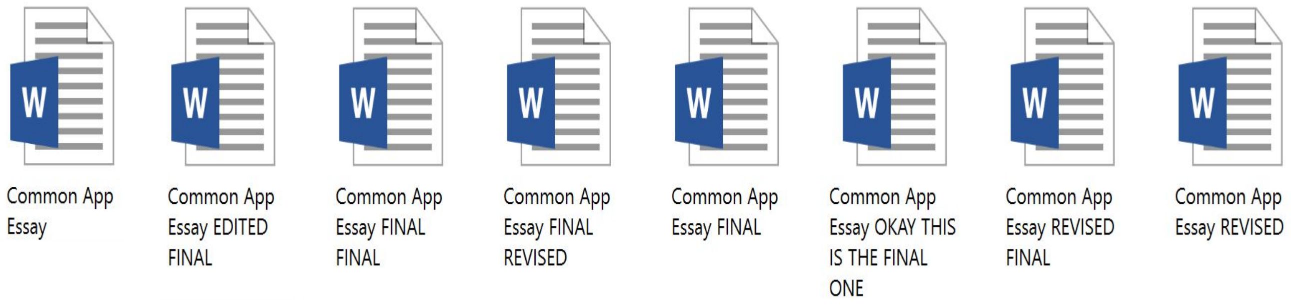
Common App Essay FINAL

11:57pm

Why use version control? – backup/restore



Why use version control? – teamwork



Common App Essay

Common App Essay EDITED FINAL

Common App Essay FINAL FINAL

Common App Essay FINAL REVISED

Common App Essay FINAL

Common App Essay OKAY THIS IS THE FINAL ONE

Common App Essay REVISED FINAL

Common App Essay REVISED

How are you going to make sense of this?

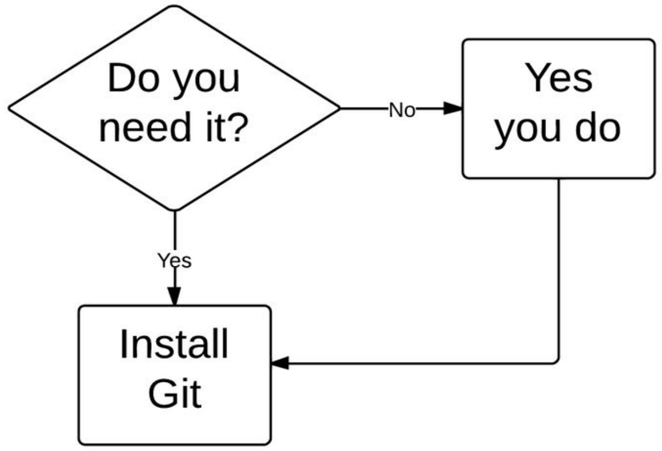
Goals of a version control system

Version control records changes to a set of files over time.

This enables you to:

- Keep a history of your work
  - Summary commit title
  - See which lines were co-changed
- Checkpoint specific versions (known good state)
  - Recover specific state
- Binary search over revisions
  - Find the one that introduced a defect
- Undo arbitrary changes
  - Without affecting prior or subsequent changes
- Maintain multiple releases of your product

Who uses version control?



Everyone should use version control

- Large teams (100+ developers)
- Small teams (2-10+ developers)
- Yourself (and your future self)
  - Multiple features or multiple computers

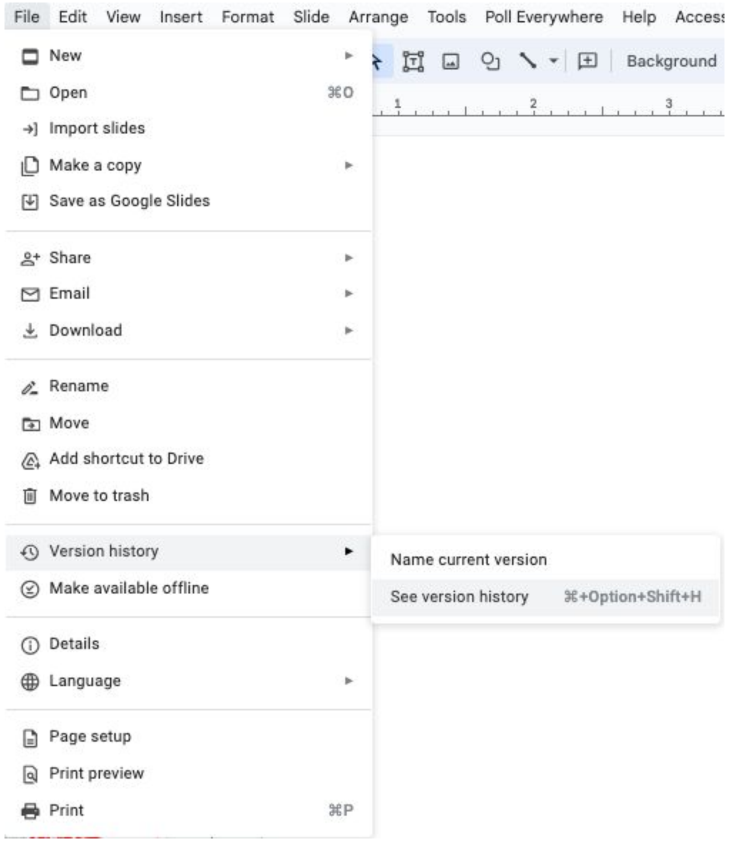
Example application domains

- Software development
- Hardware development
- Research & experiments (infrastructure and data)
- Applications (e.g., (cloud-based) services)
- Services that manage artifacts (e.g., legal, accounting, business, …)

Version control for documents

Common App Essay

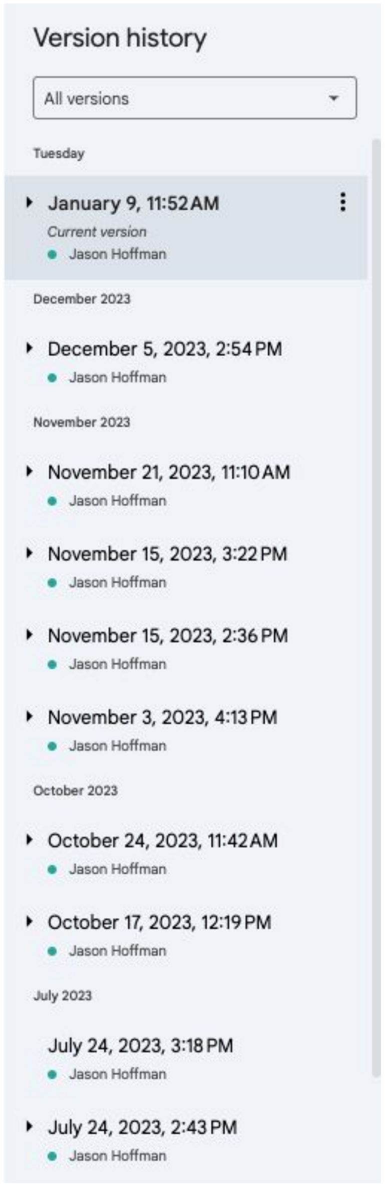
11:51pm


File Edit View Insert Format Slide Arrange Tools Poll Everywhere Help Access

New
Open ⌘O
Import slides
Make a copy
Save as Google Slides
Share
Email
Download
Rename
Move
Add shortcut to Drive
Move to trash
Version history
Make available offline
Details
Language
Page setup
Print preview
Print ⌘P

Background

Name current version
See version history ⌘+Option+Shift+H


Version history

All versions

Tuesday

January 9, 11:52 AM
Current version
Jason Hoffman

December 2023

December 5, 2023, 2:54 PM
Jason Hoffman

November 2023

November 21, 2023, 11:10 AM
Jason Hoffman

November 15, 2023, 3:22 PM
Jason Hoffman

November 15, 2023, 2:36 PM
Jason Hoffman

November 3, 2023, 4:13 PM
Jason Hoffman

October 2023

October 24, 2023, 11:42 AM
Jason Hoffman

October 17, 2023, 12:19 PM
Jason Hoffman

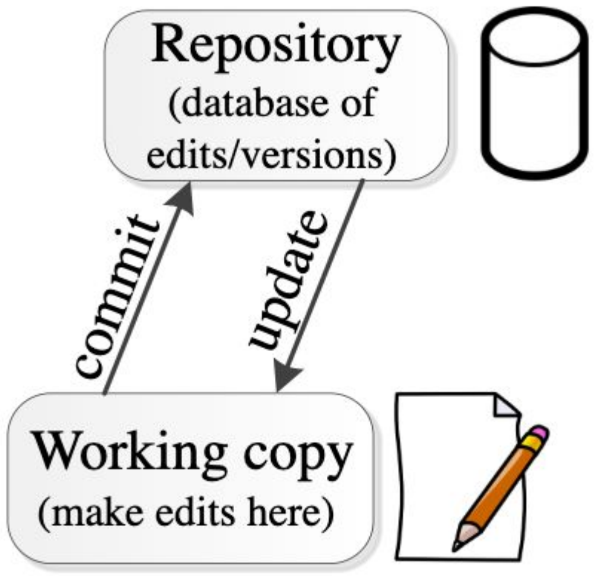
July 2023

July 24, 2023, 3:18 PM
Jason Hoffman

July 24, 2023, 2:43 PM
Jason Hoffman

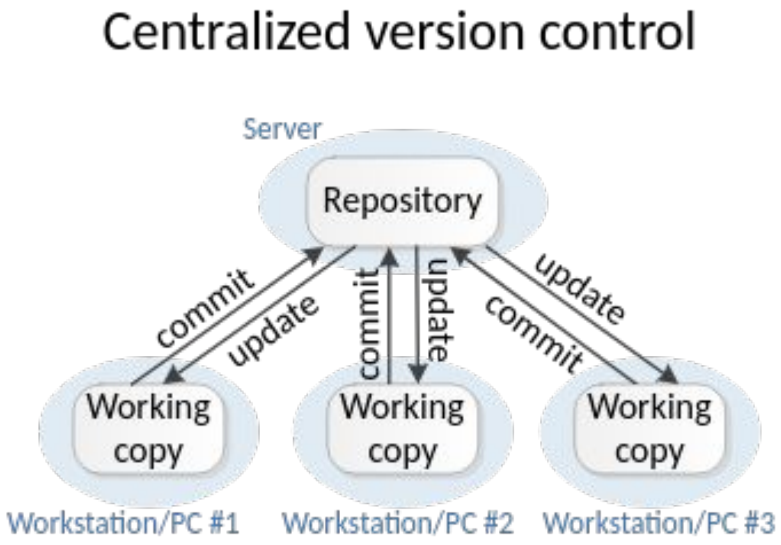
Version control

Working by yourself



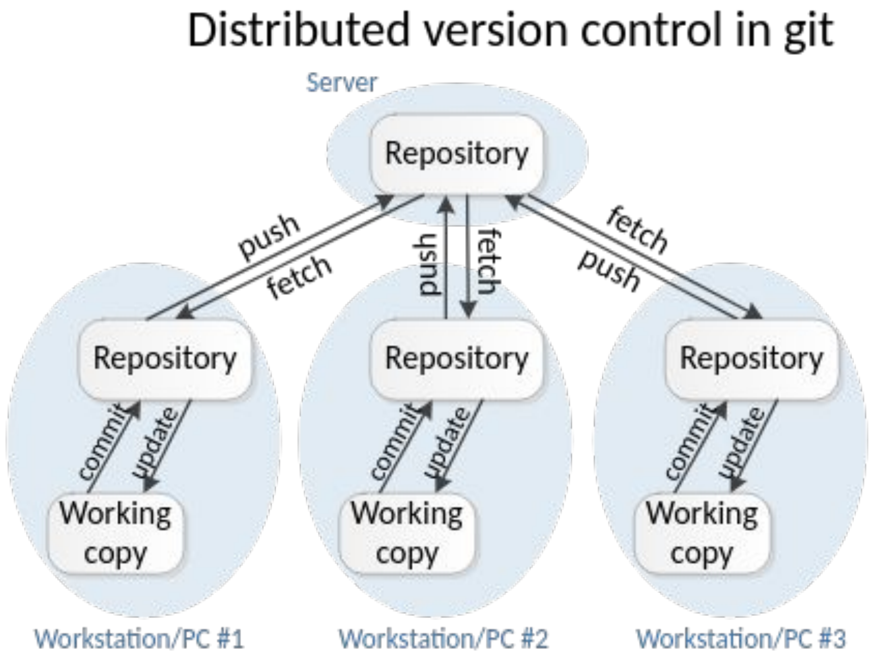
Centralized version control (the old way)

- **One central repository.**
  It stores a history of project versions.
- Each user has a **working copy**.
- A user **commits** file changes
  to the repository.
- Committed changes are immediately
  visible to teammates who **update**.
- Examples: SVN (Subversion), CVS.



Distributed version control (the new way)

- **Multiple copies of a repository**. Each stores its own history of project versions.
- Each user **commits** to a **local** (private) repository.
- All committed changes remain local unless **pushed** to another repository.
- No external changes are visible unless **fetched** from another repository.
- Examples: Git, Hg (Mercurial).



Typical workflow

`git pull`

`git branch` *name*

`git checkout -b` *name*

`git switch` *name*

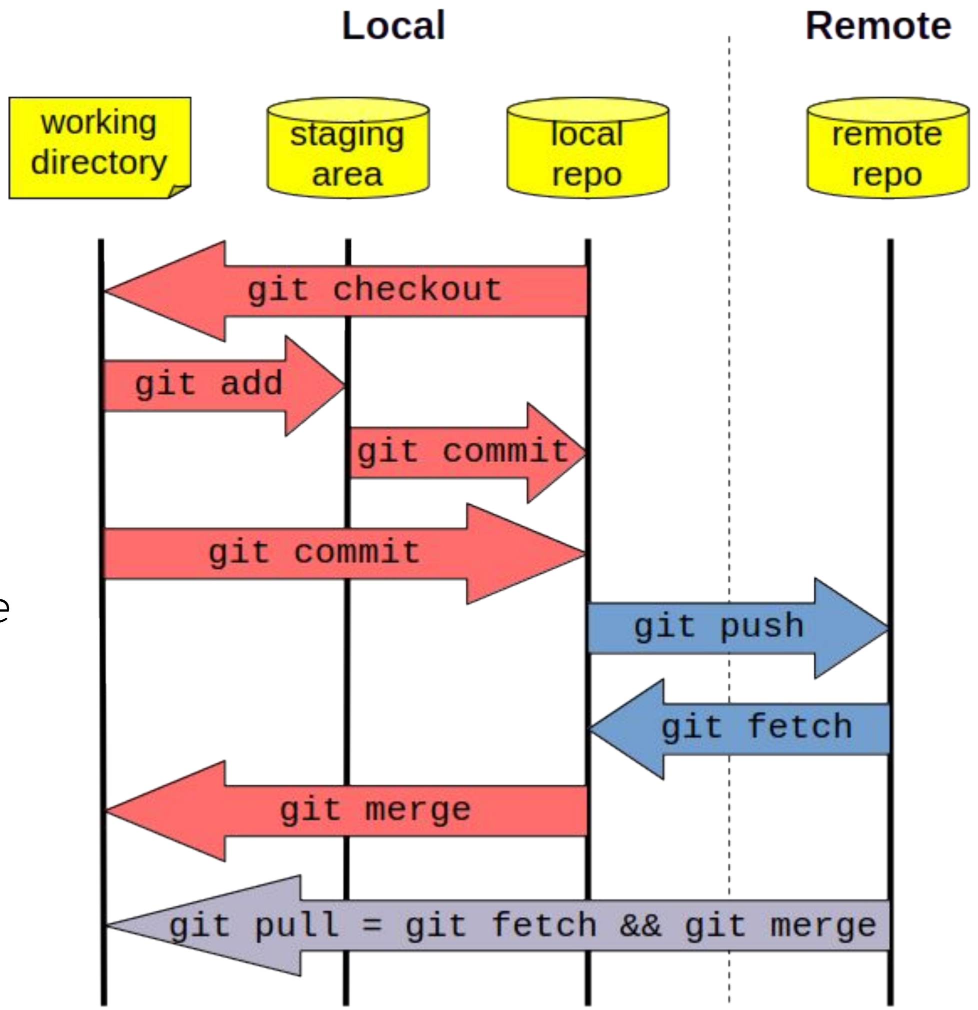
Repeat:

- <edit files, run tests>
- `[git add]`
- `git commit` *filename*
- `git commit`
- `git pull`

<run tests again>

`git push`

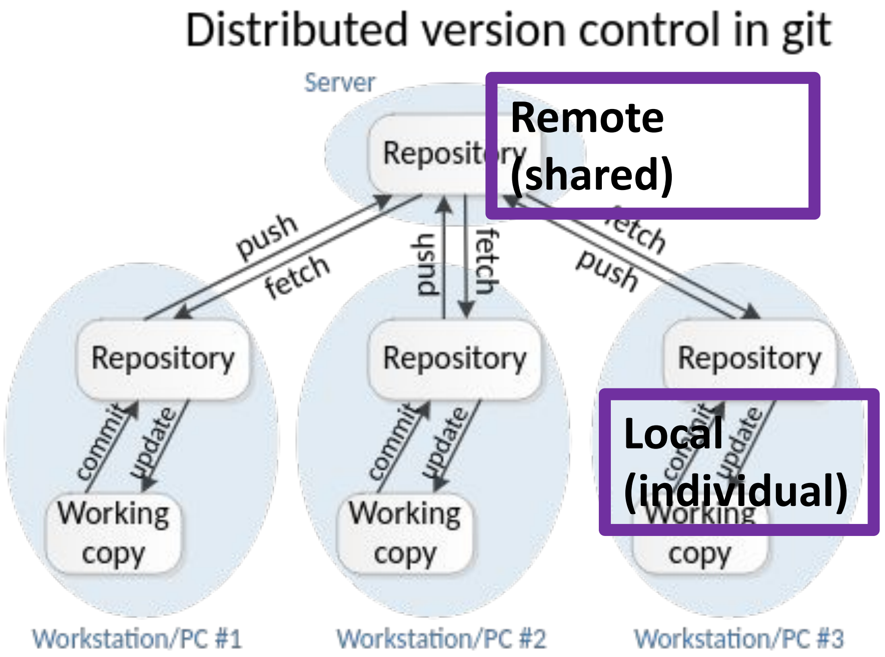
<make a GitHub pull request>



Detail introduction of how to use git: https://www.runoob.com/git/git-basic-operations.html

An example git workflow

- **git clone** (copy remote repo locally)
- Create branch: **git branch *name***
- Switch to branch: **git switch *name*** OR **git checkout *name***
- Create & switch: **git checkout -b *name***
- Loop:
  - develop
  - **git add** (stage changes)
  - **git commit** (local commit)
  - **git pull** (merge changes in remote with local)
  - resolve conflicts
- **git push** (copy local changes to remote repository)
- Make GitHub pull request



Other useful git commands

git diff: what changes are in the working copy?

git diff --staged: what changes are in the staging area?

git status: what files are in the working copy & staging area?

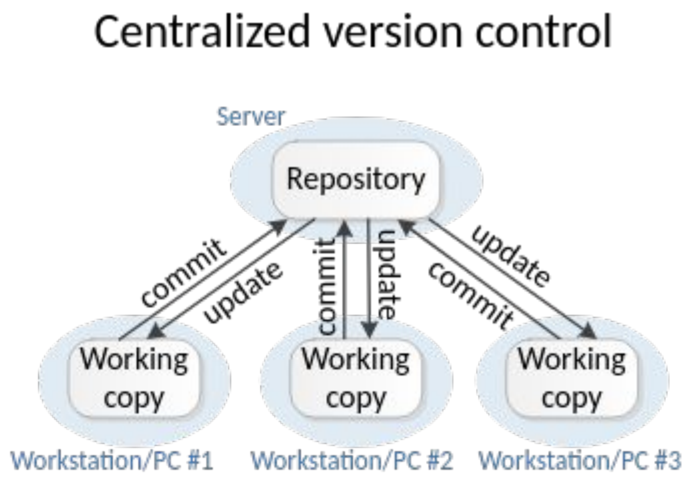
git log [--graph]: see the history of commits

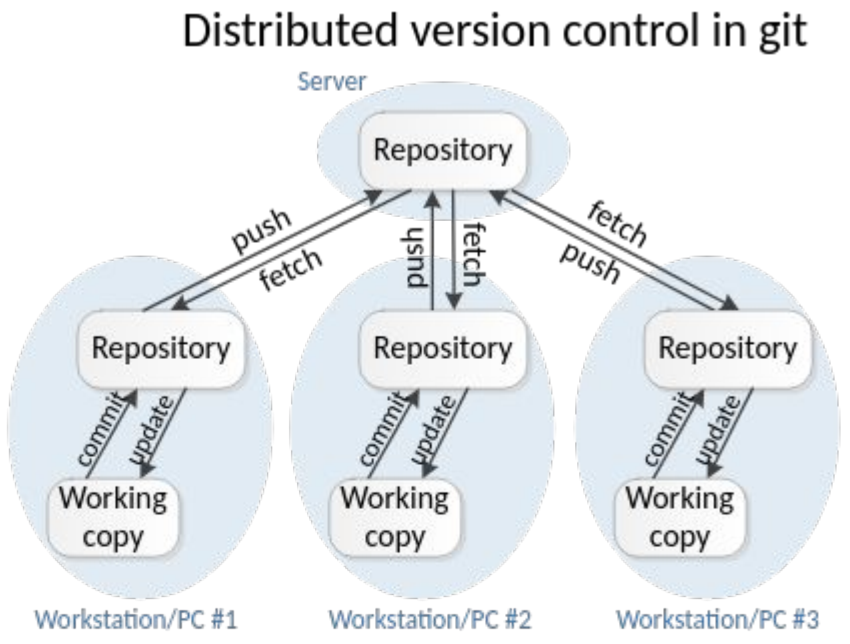
git {annotate,blame,praise}: who last changed each line?

git cherry-pick: apply identified commits to current branch

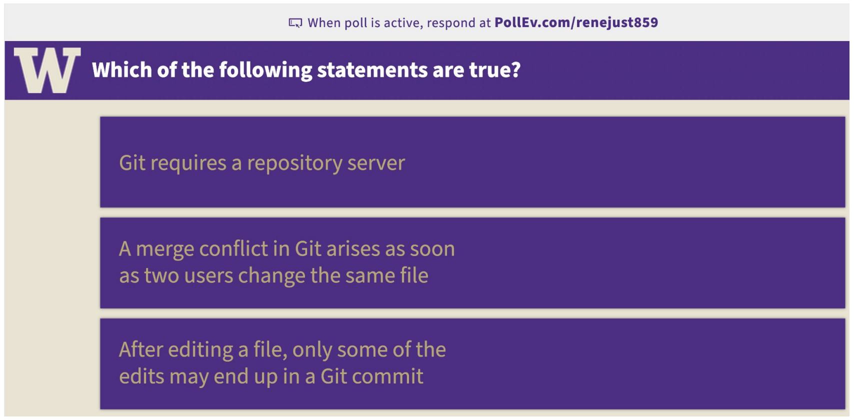
git bisect: run binary search to find a bad commit

2 different version control modes





A little quiz

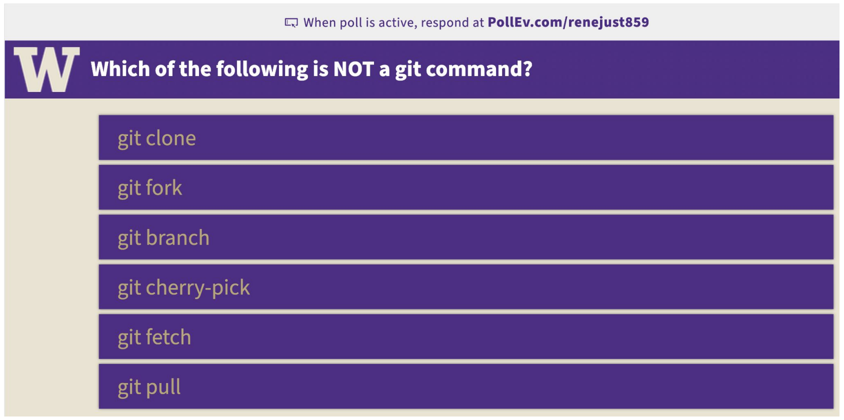

When poll is active, respond at **PollEv.com/renejust859**

Which of the following statements are true?

Git requires a repository server

A merge conflict in Git arises as soon as two users change the same file

After editing a file, only some of the edits may end up in a Git commit


When poll is active, respond at **PollEv.com/renejust859**

Which of the following is NOT a git command?

git clone

git fork

git branch

git cherry-pick

git fetch

git pull

Branch
vs
Clone
Vs
Fork

Multiple versions of your program

What if you have to support:

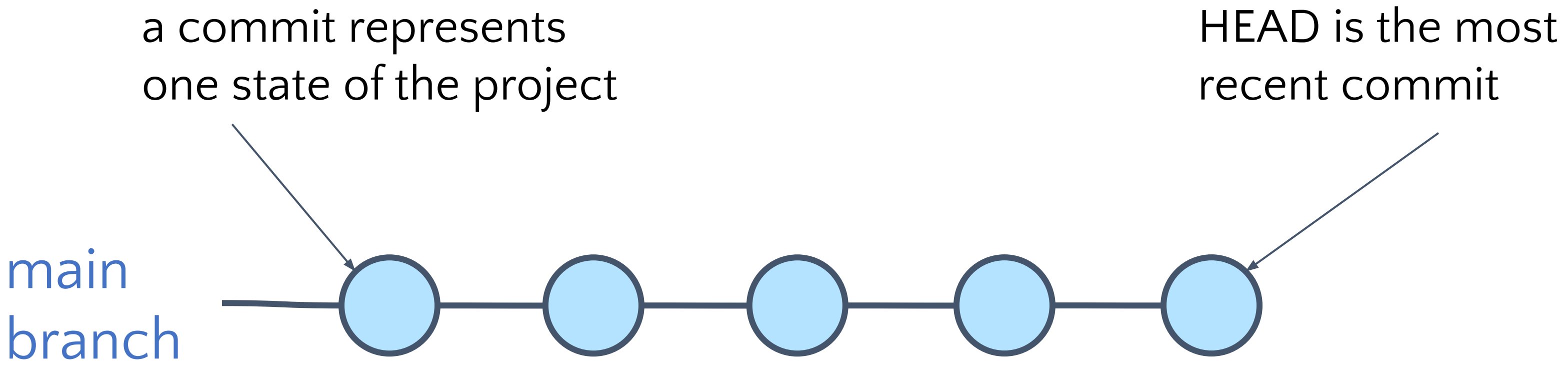
- Version 1.0.4 and version 2.0.0
- Windows and macOS
- Adding a feature
- Fixing a bug

Git has 3 ways to represent multiple histories:

- **Branch**: Start a parallel history of changes to the code in the repository
- **Clone**: Make a copy of the repository to work on code changes
- **Fork**: Make a copy the repository that will not necessarily be merged back with original (but can be through a pull request)

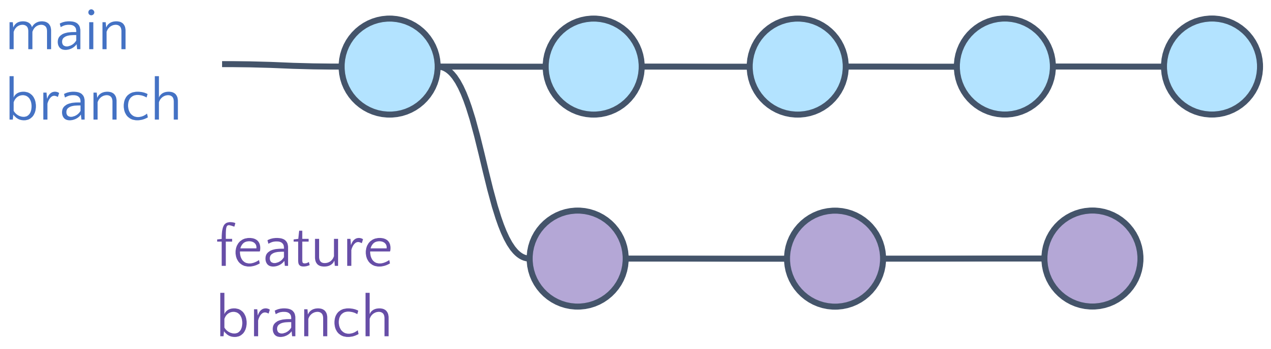
Branches

- A branch is a history of program versions
- There is one main development branch (main, master, trunk)
    - It should always pass tests and be ready to ship or deploy


a commit represents one state of the project
HEAD is the most recent commit
main branch

Branches

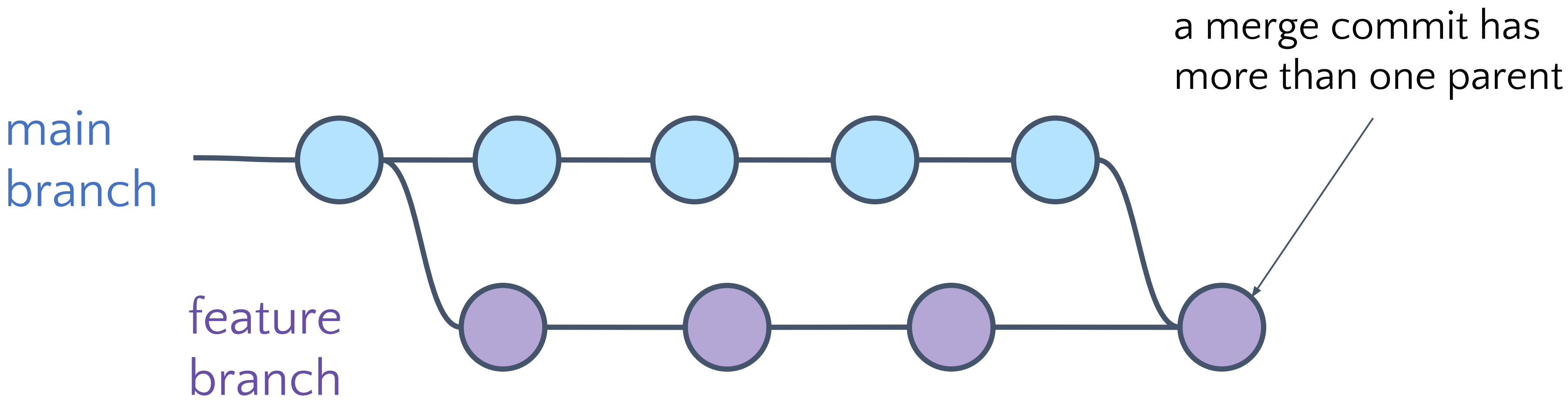
- Other branches are alternate histories
- You can create many branches
    - Lightweight - every work item (feature, bug fix) has its own branch
        - Use of many branches prevents cross-pollution
- Branches (histories) can get out of sync



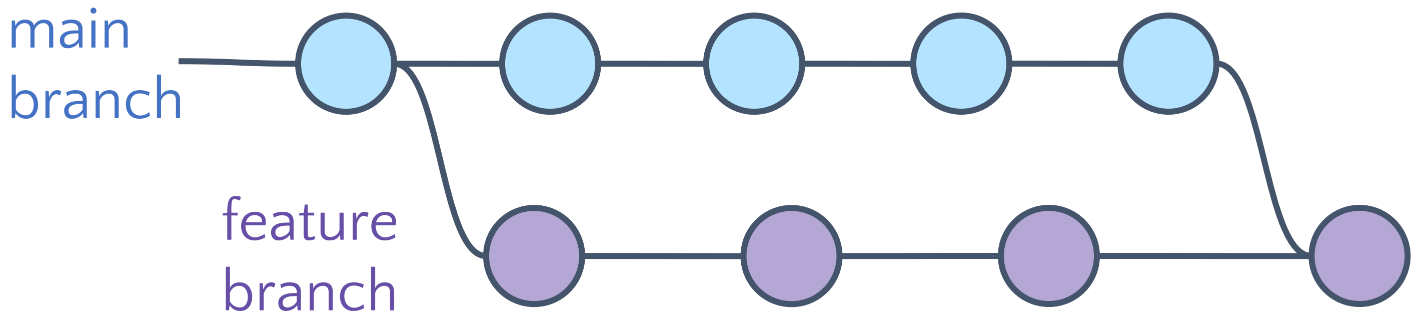
Merging branches

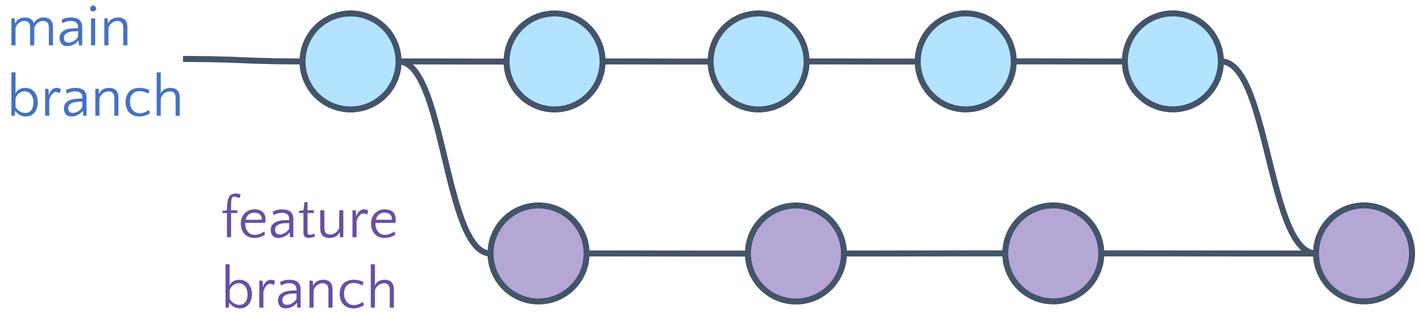
- Branches can get out of sync
- **Merge** incorporates changes from one branch into another
- From a feature branch, run: `git merge main`

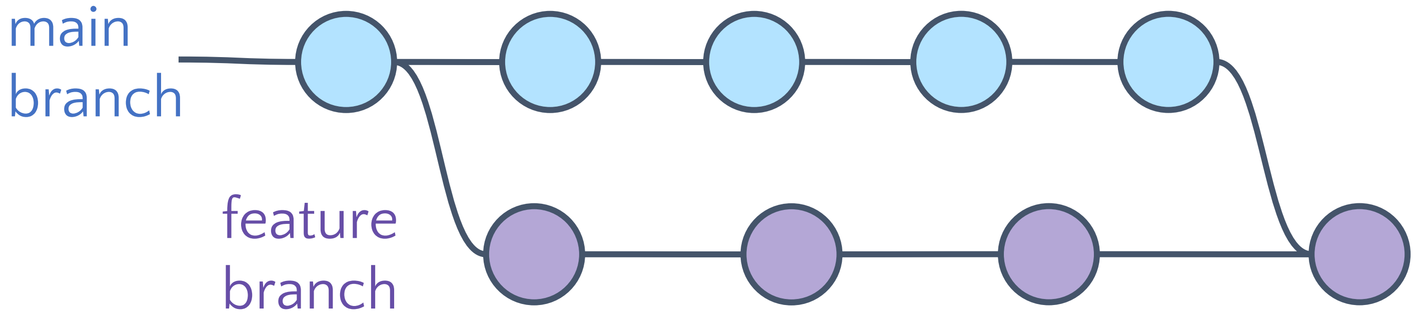
- Life goal of a branch is to be merged into main and deleted as quickly as possible
  - Done via a **pull request**, not via `git merge`



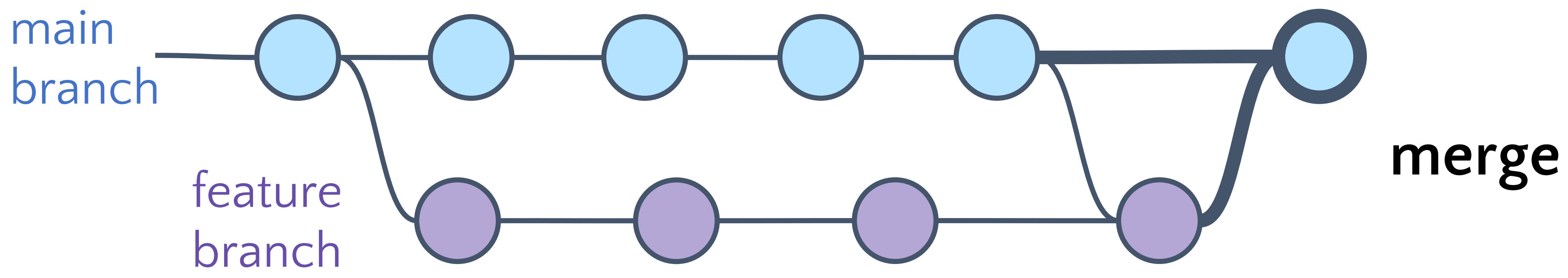
3 ways to resolve a pull request

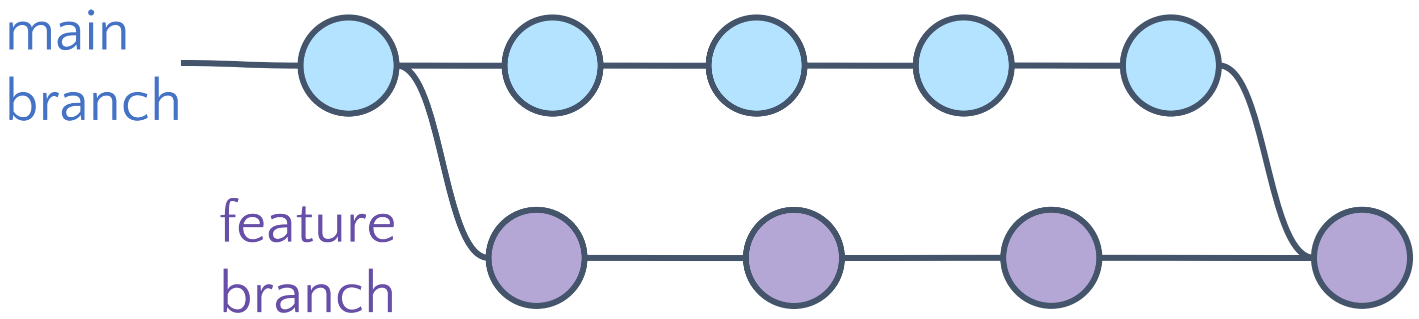


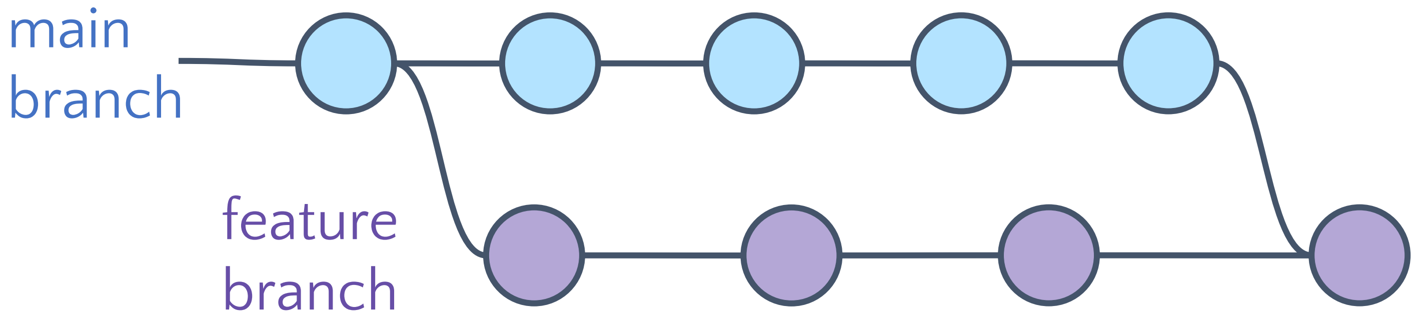




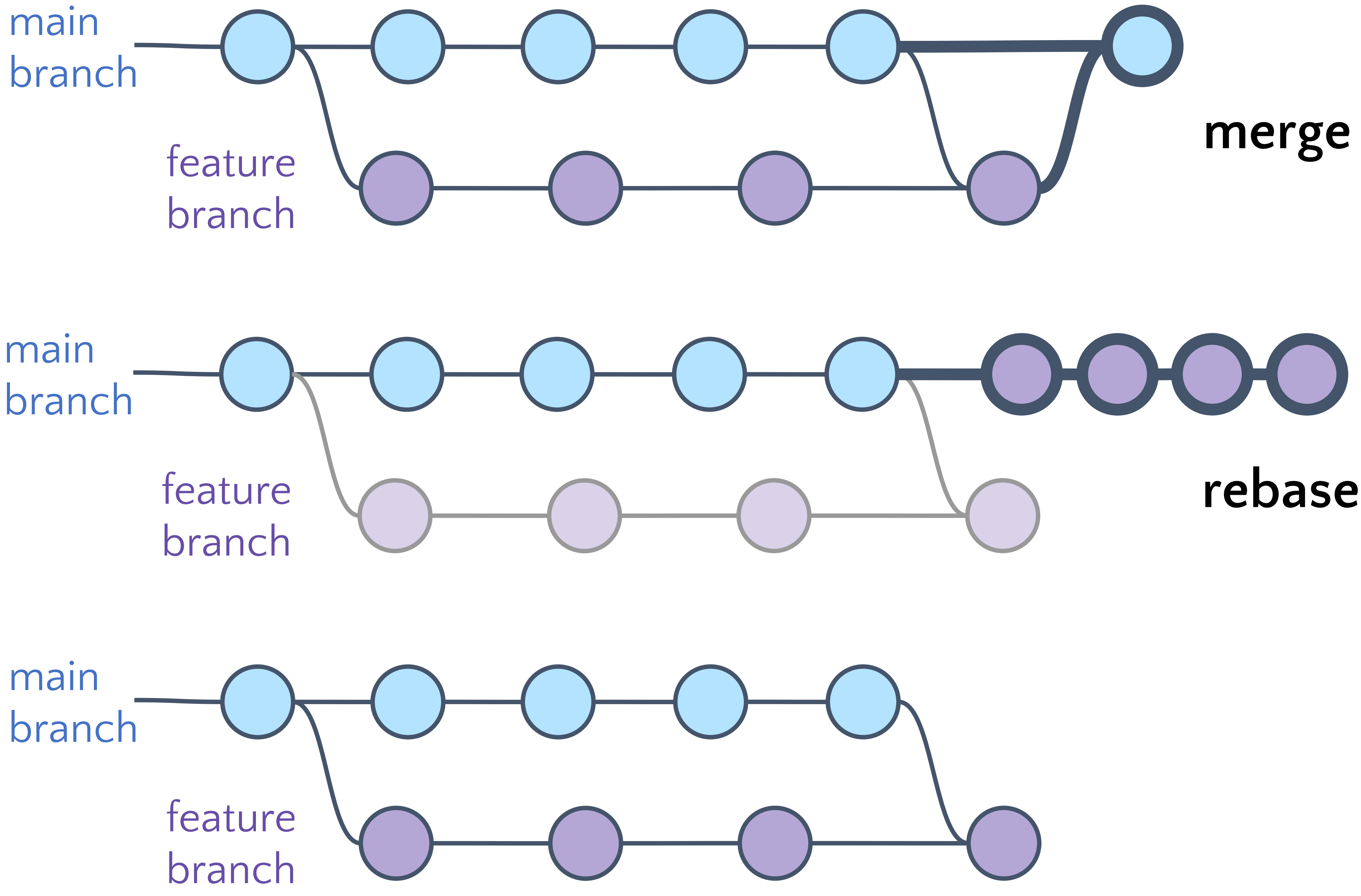
3 ways to resolve a pull request



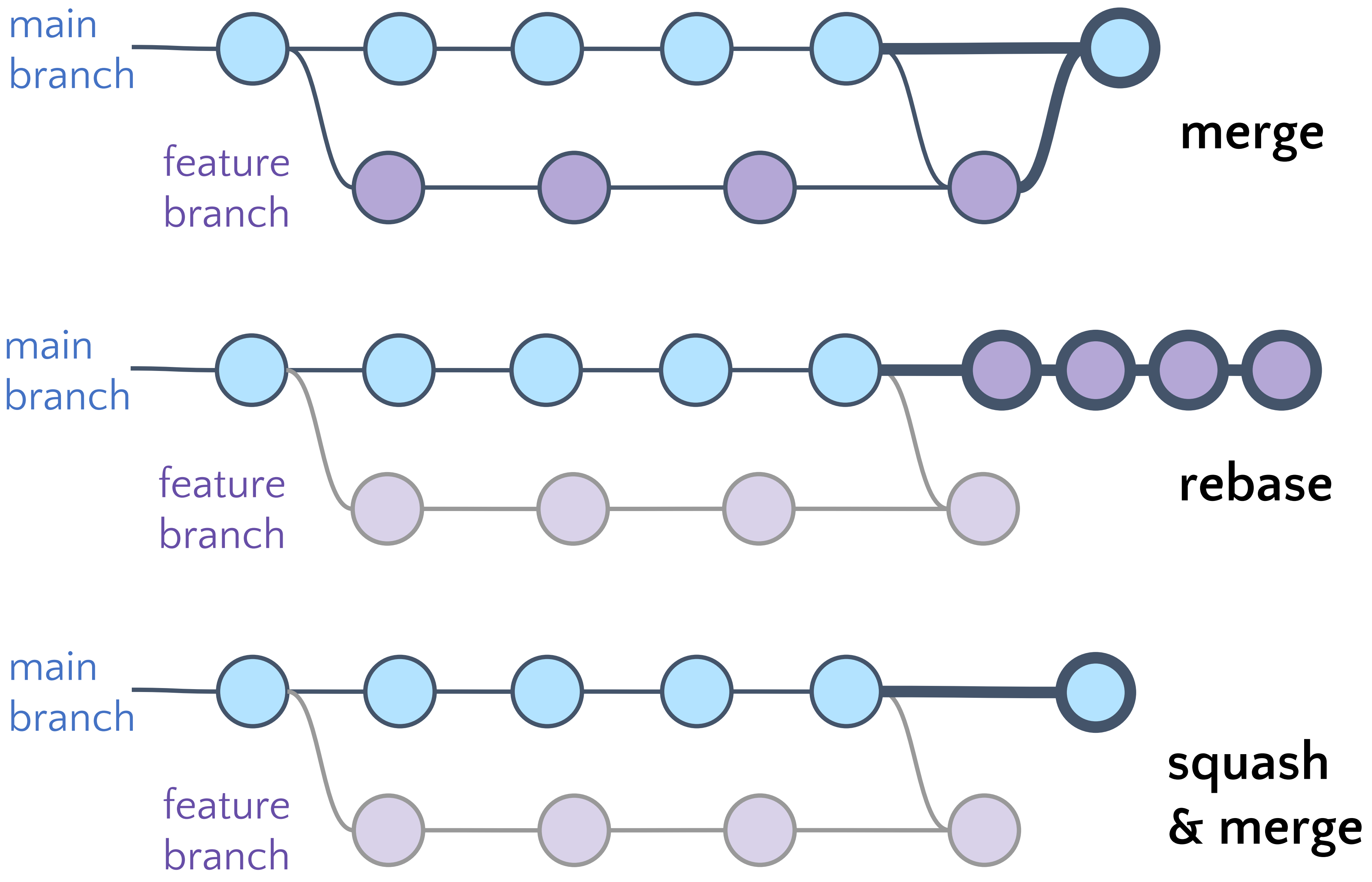




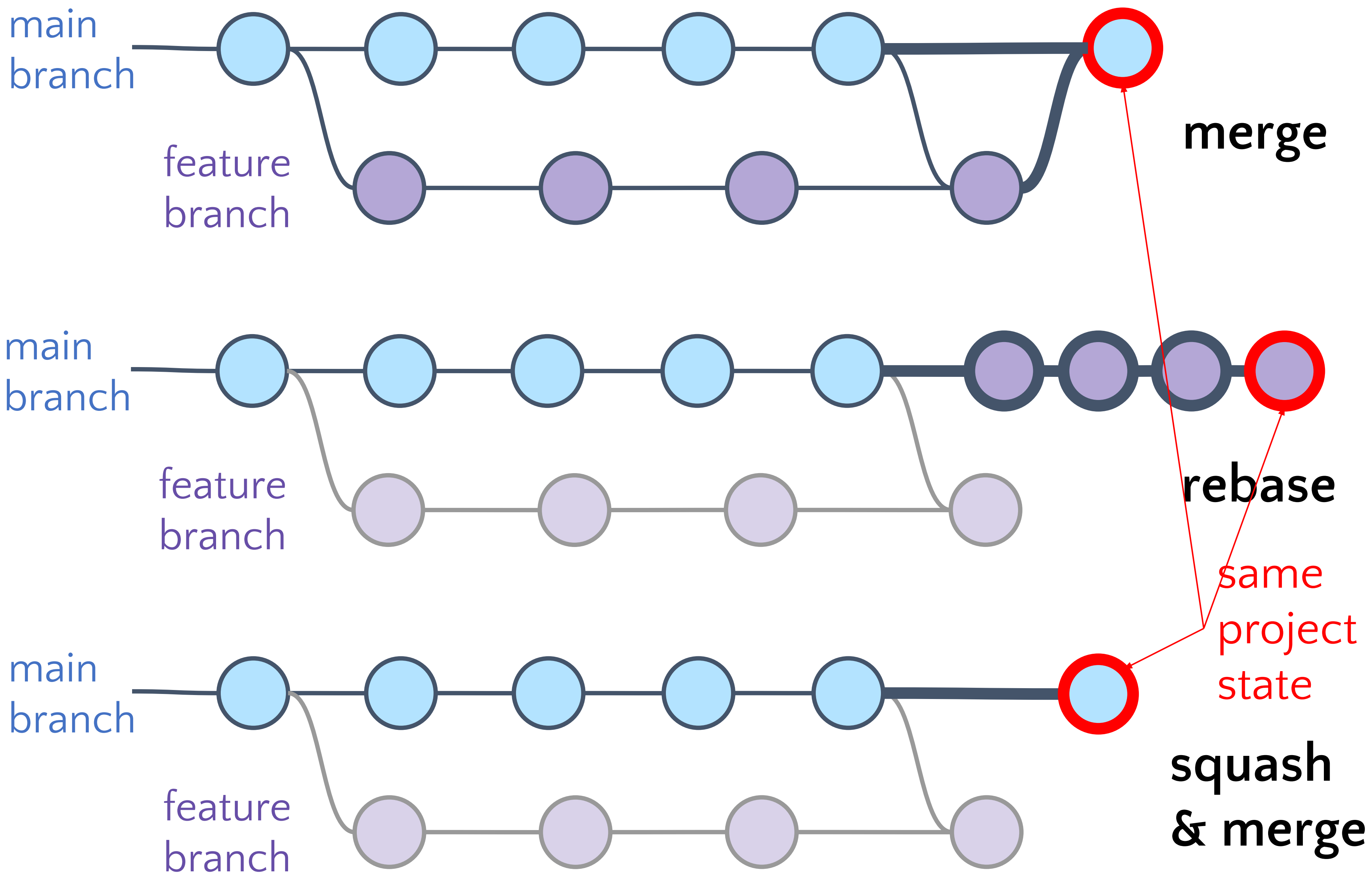
3 ways to resolve a pull request



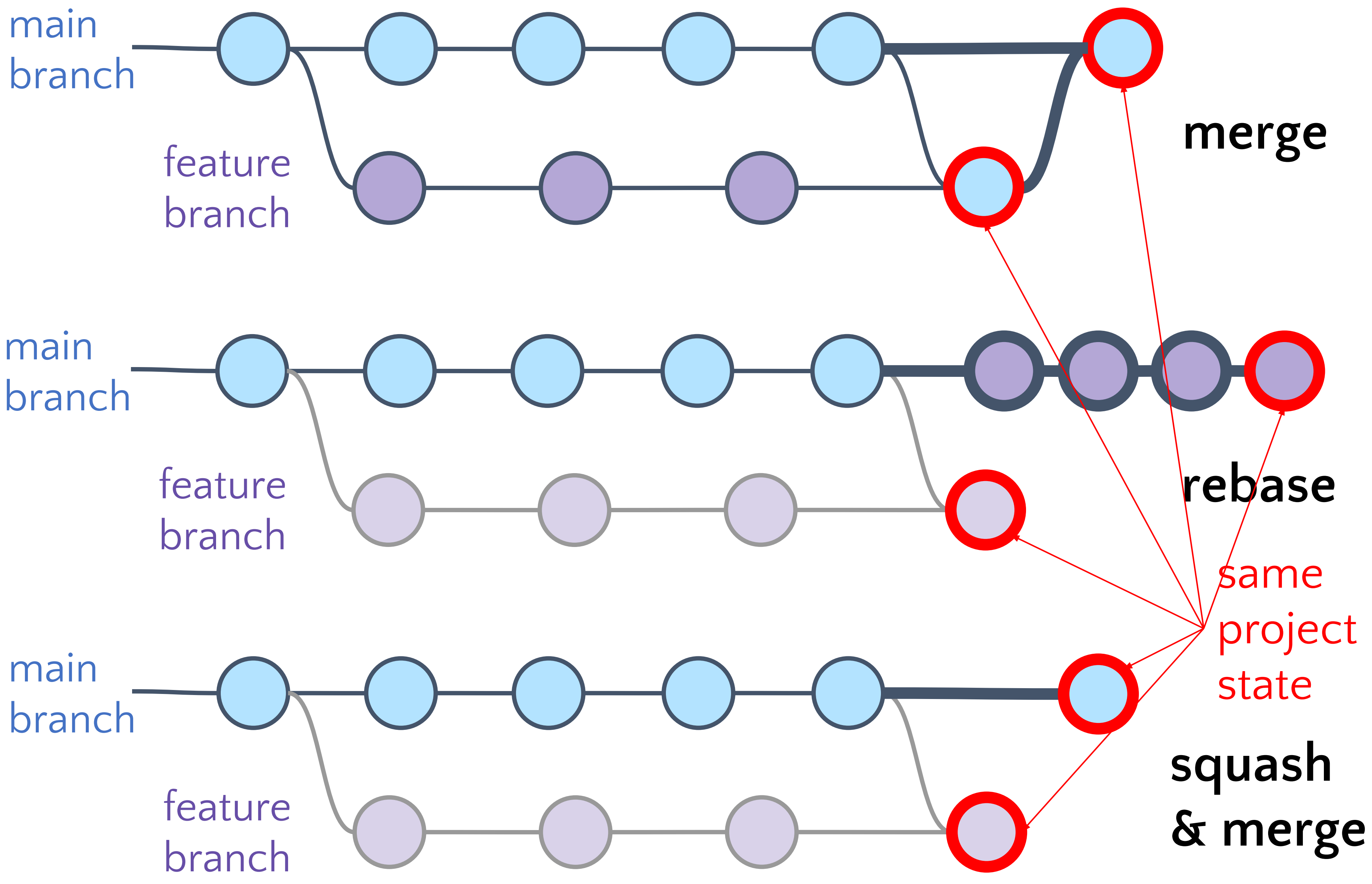
3 ways to resolve a pull request



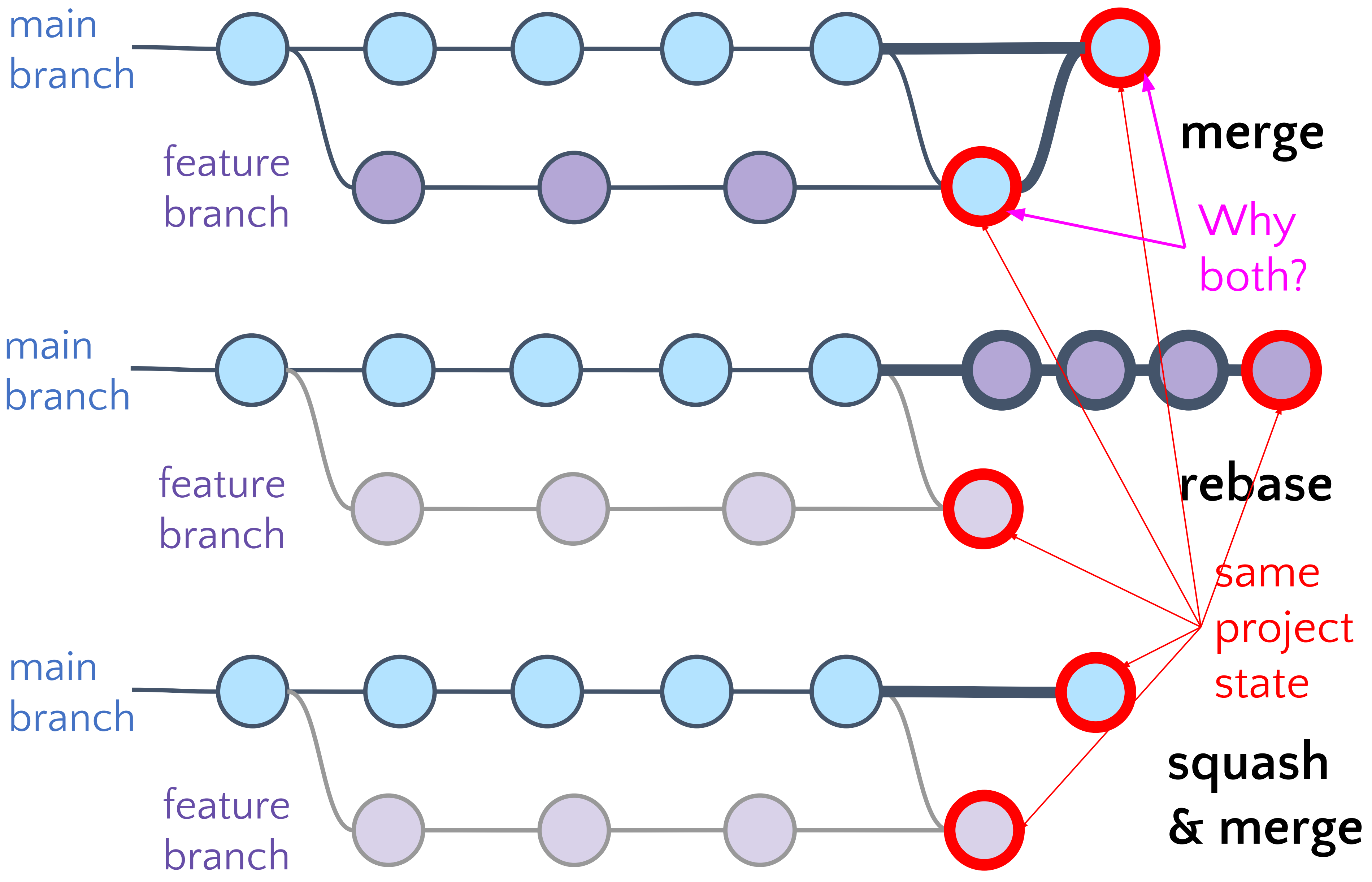
3 ways to resolve a pull request



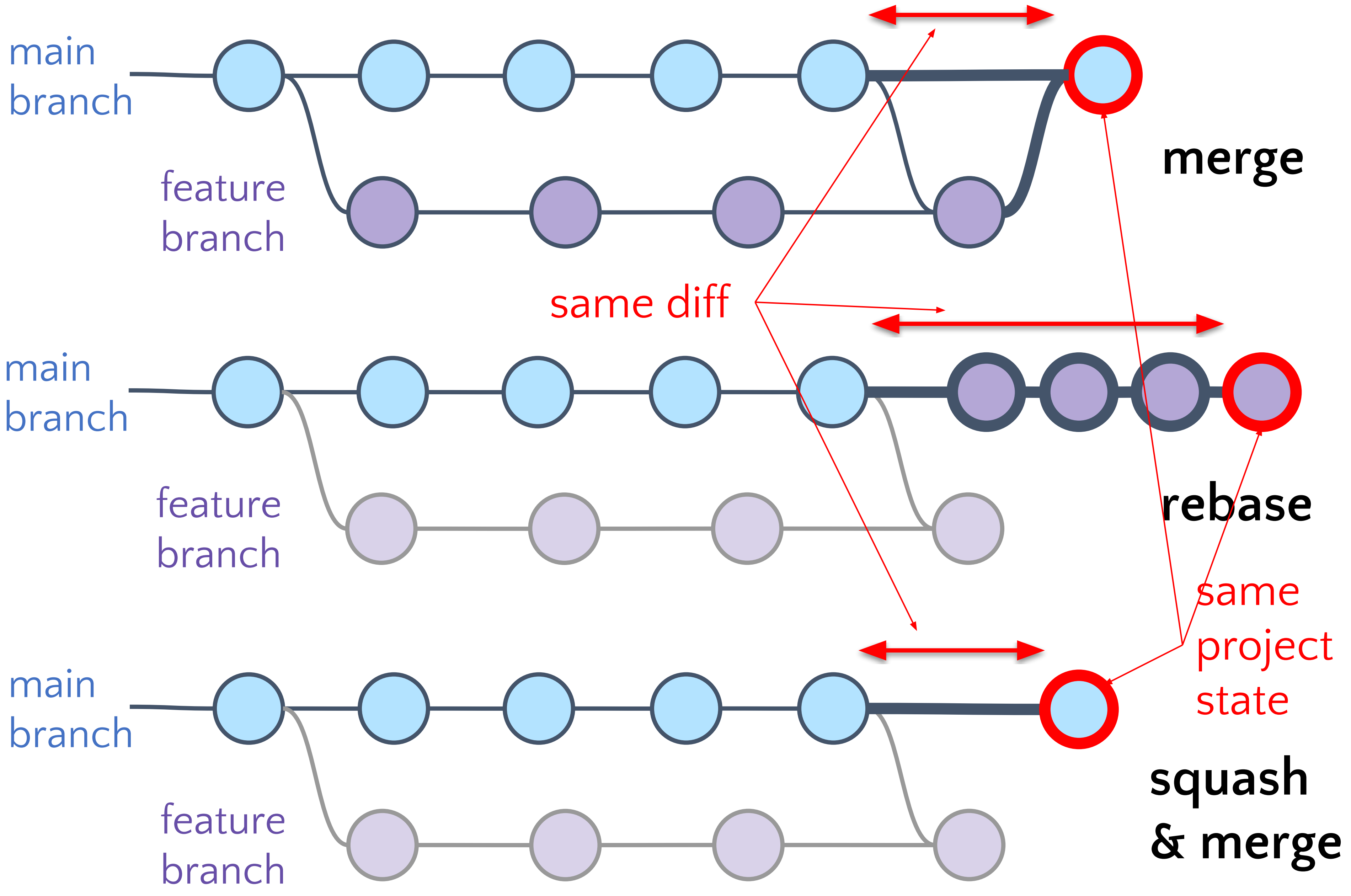
3 ways to resolve a pull request



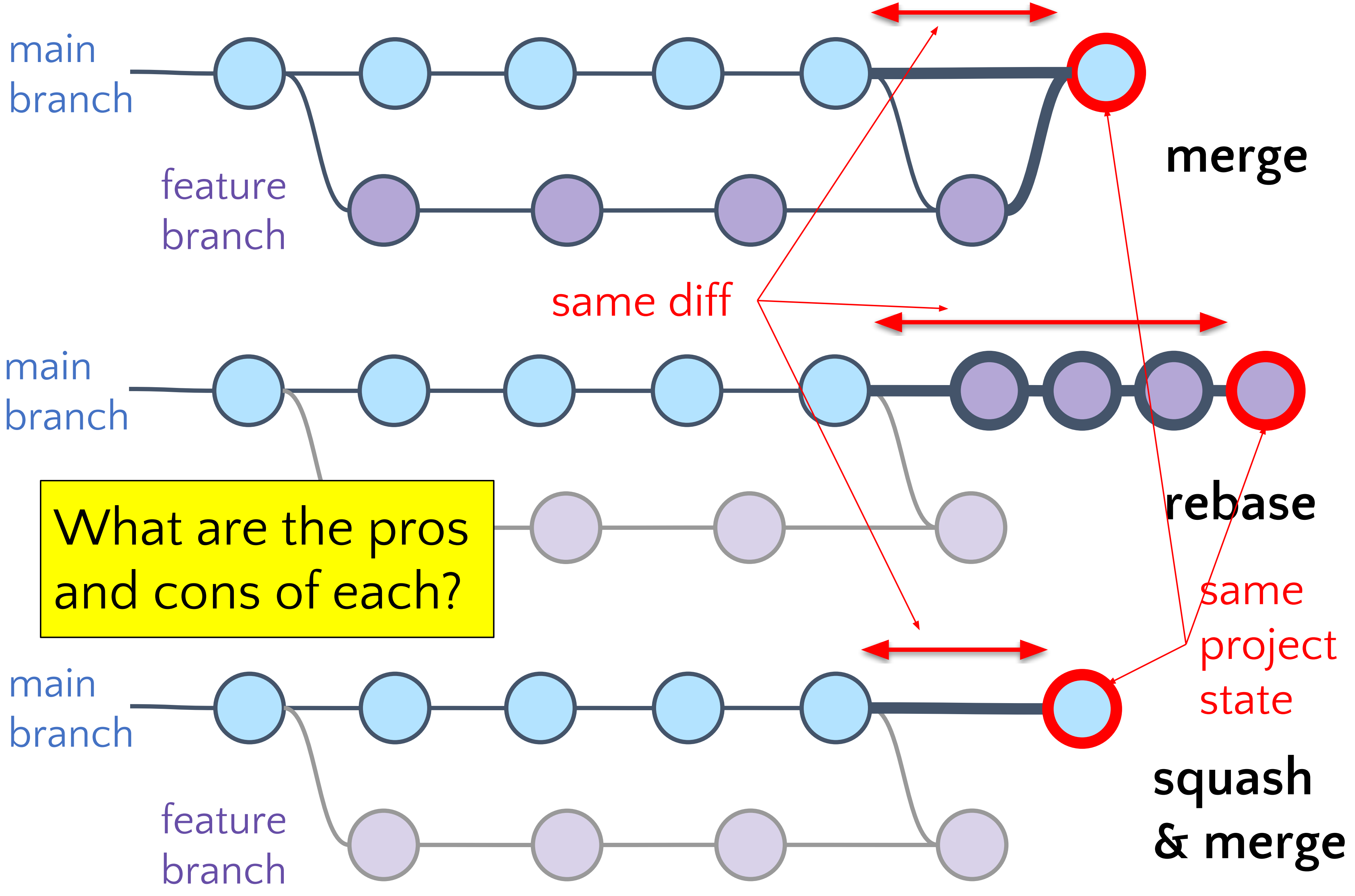
3 ways to resolve a pull request



3 ways to resolve a pull request

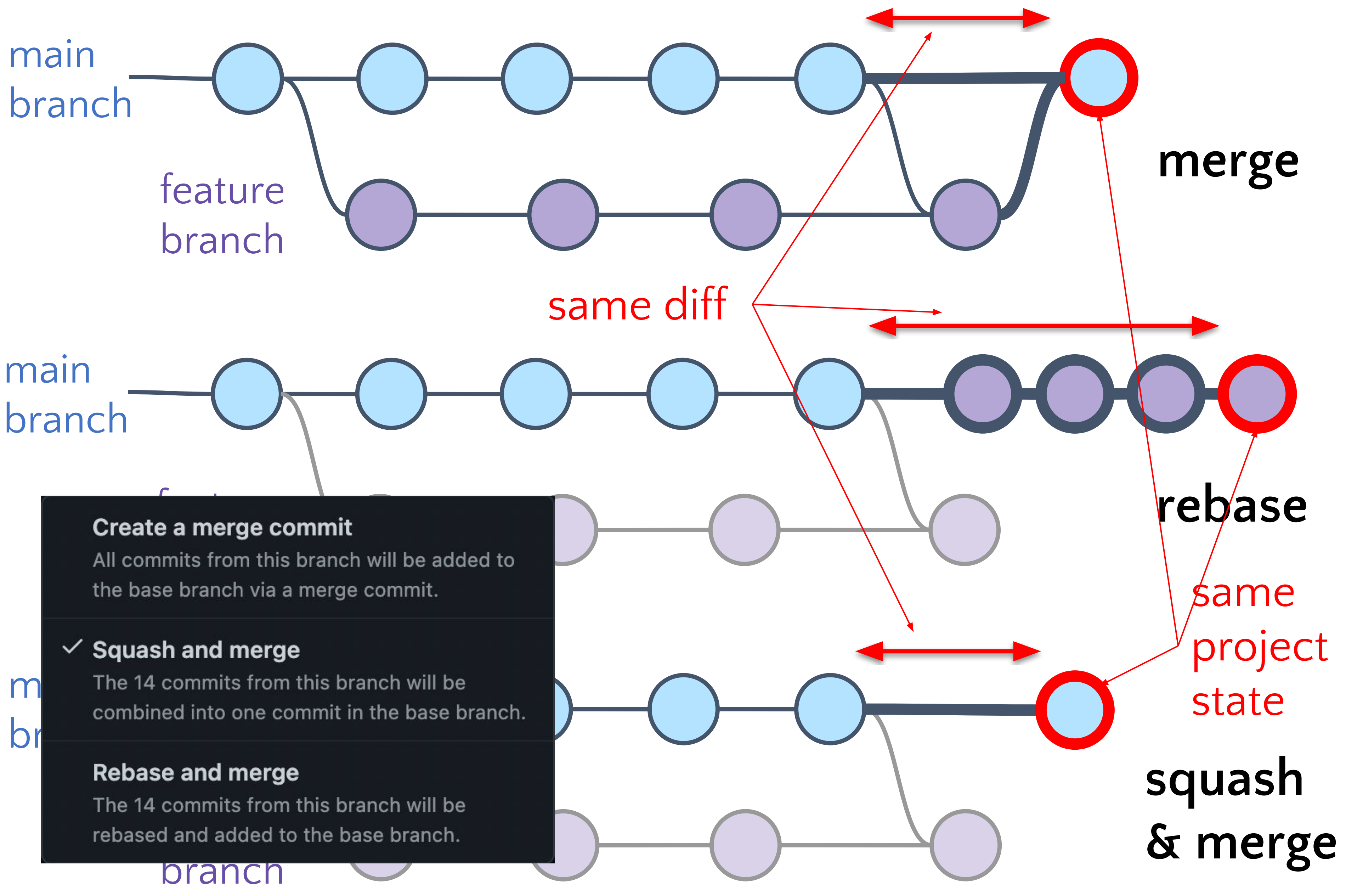


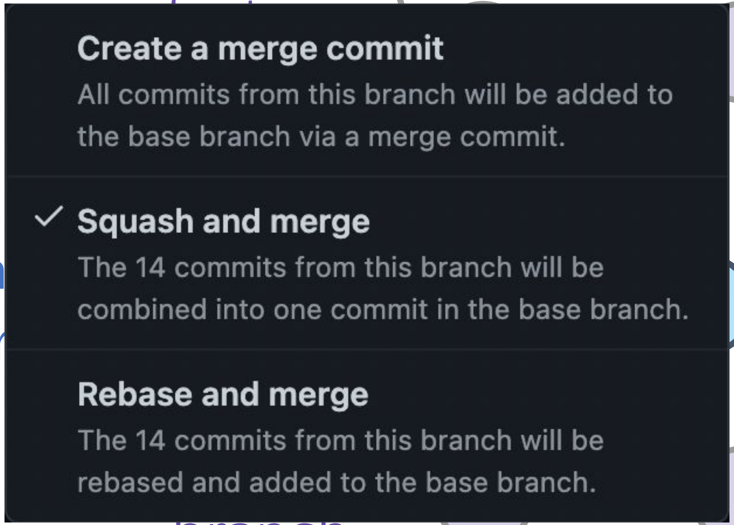
3 ways to resolve a pull request



What are the pros and cons of each?

3 ways to resolve a pull request




Create a merge commit
All commits from this branch will be added to the base branch via a merge commit.

✓ **Squash and merge**
The 14 commits from this branch will be combined into one commit in the base branch.

Rebase and merge
The 14 commits from this branch will be rebased and added to the base branch.

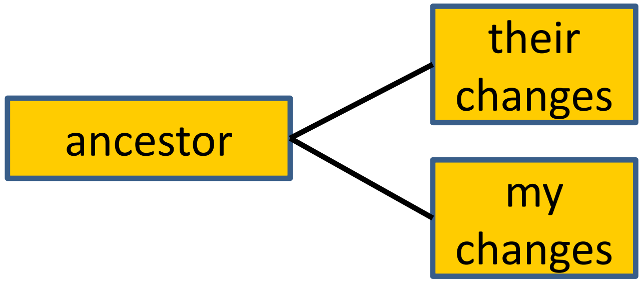
Merge conflicts

Parallel edits

You and a teammate edit at the same time.
Merging is integrating your changes, keeping all edits.
The VCS tries to merge the edits for you.

If the VCS fails, there is a **conflict**.
You must resolve the conflict manually.

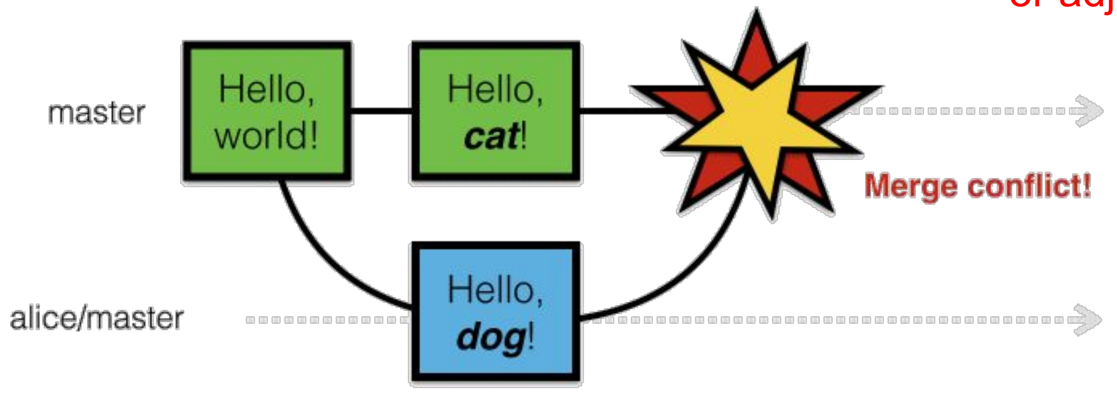
There are three versions of the code:



Conflicts

- When you run git merge, git attempts to retain all the changes from each branch
- A **conflict** arises when two users **change the same line** of a file

or adjacent lines

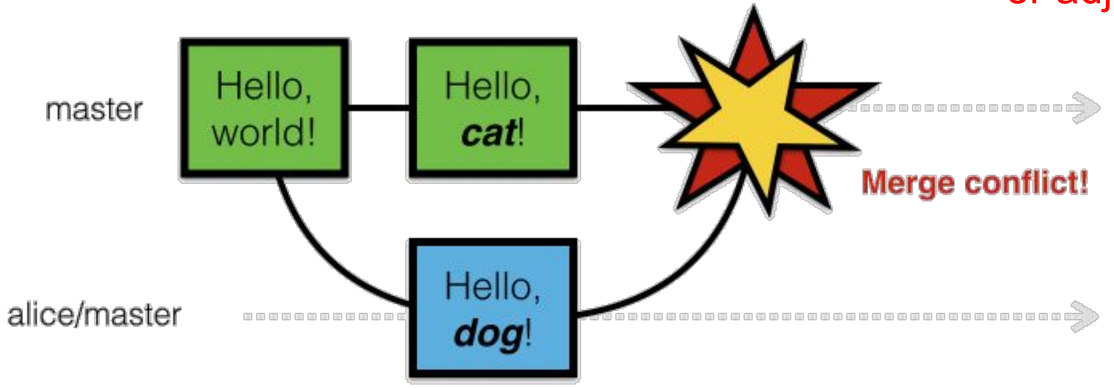


- The person doing the merge needs to resolve the conflict by manual editing

Conflicts

git’s merge tools
can make mistakes

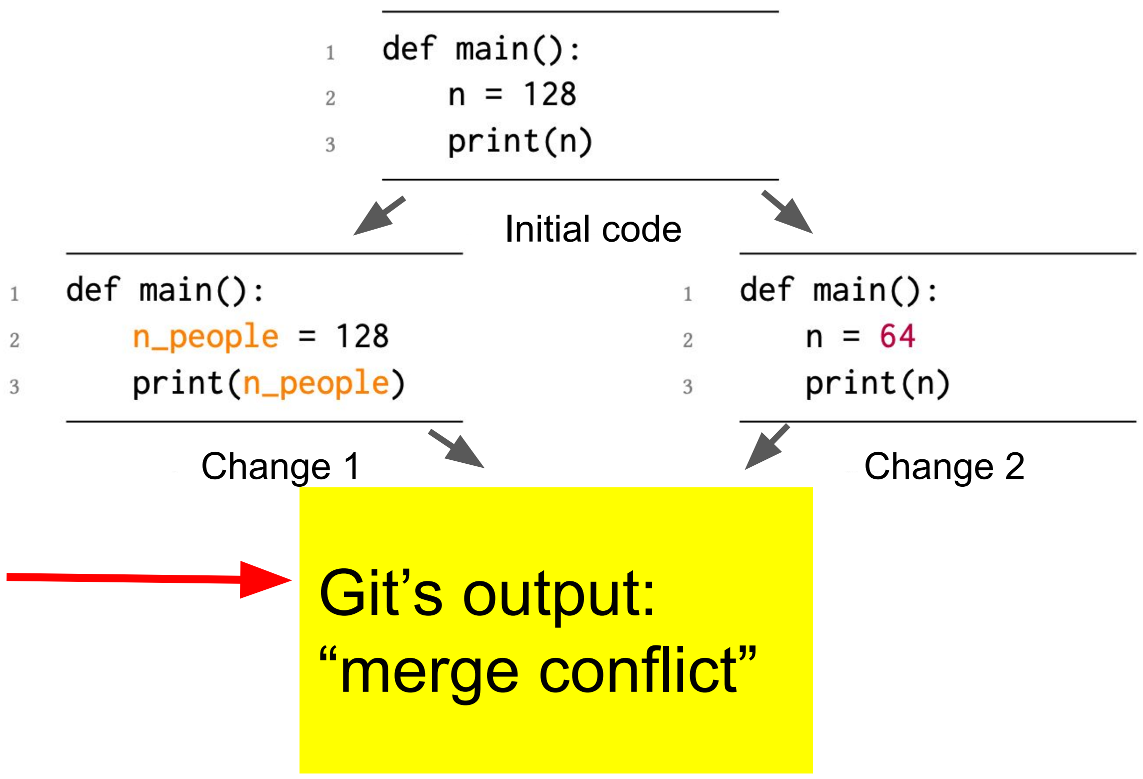
- When you run git merge, git attempts to retain all the changes from each branch
- A **conflict** arises when two users **change the same line** of a file
  or adjacent lines



- The person doing the merge needs to resolve the conflict by manual editing

Merge algorithm failure: unable to merge

- Line-by-line merge yields a conflict
- Inspection reveals they can be merged



```
def main():
    n = 128
    print(n)
```

Initial code

```
def main():
    n_people = 128
    print(n_people)
```

Change 1

```
def main():
    n = 64
    print(n)
```

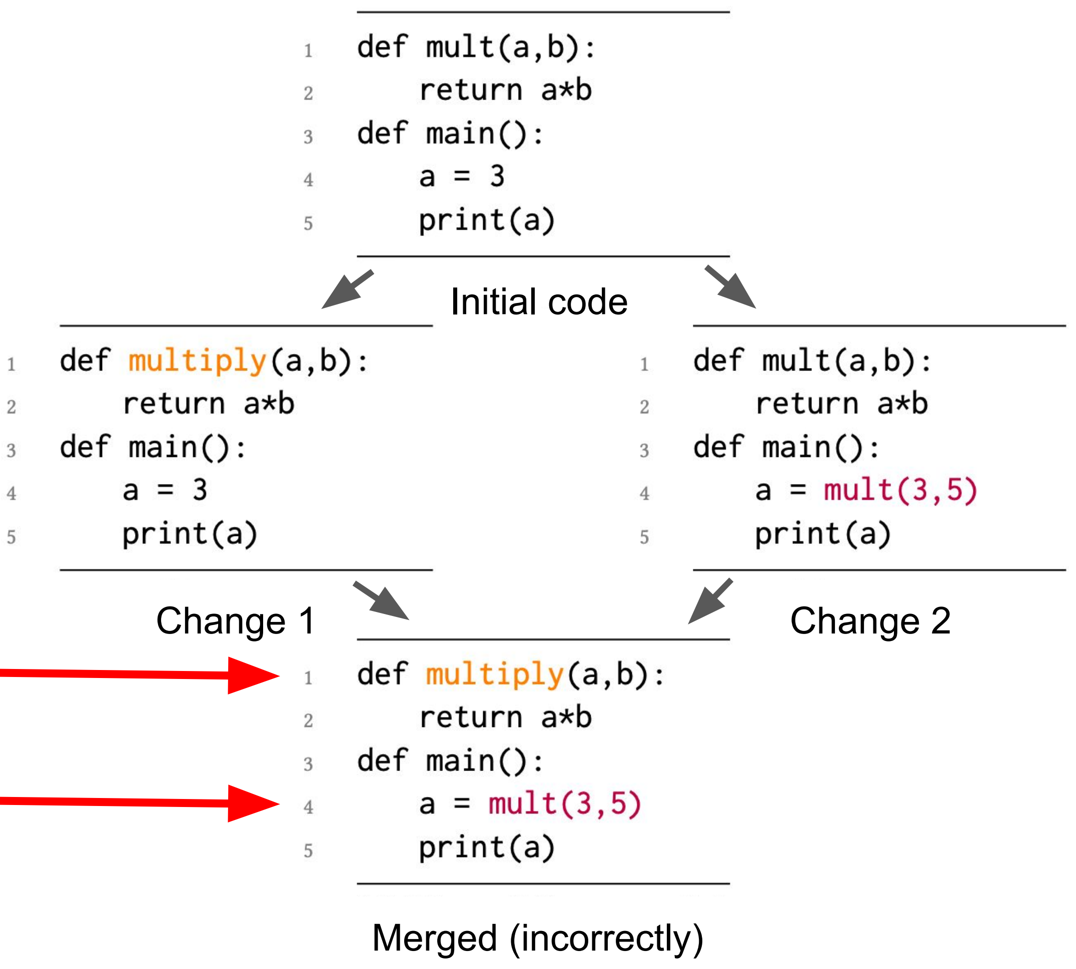
Change 2

Works despite changes on same line

Git's output: "merge conflict"

Merge algorithm failure: clean, incorrect merge

- Line-by-line merge yields no conflicts ("clean merge")
- Resulting code is incorrect



Initial code:

```
def mult(a,b):
    return a*b
def main():
    a = 3
    print(a)
```

Change 1:

```
def multiply(a,b):
    return a*b
def main():
    a = 3
    print(a)
```

Change 2:

```
def mult(a,b):
    return a*b
def main():
    a = mult(3,5)
    print(a)
```

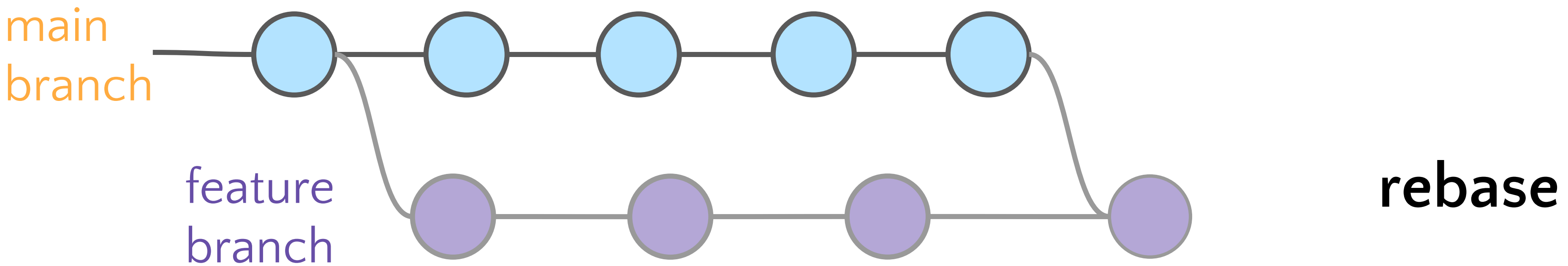
Merged (incorrectly):

```
def multiply(a,b):
    return a*b
def main():
    a = mult(3,5)
    print(a)
```

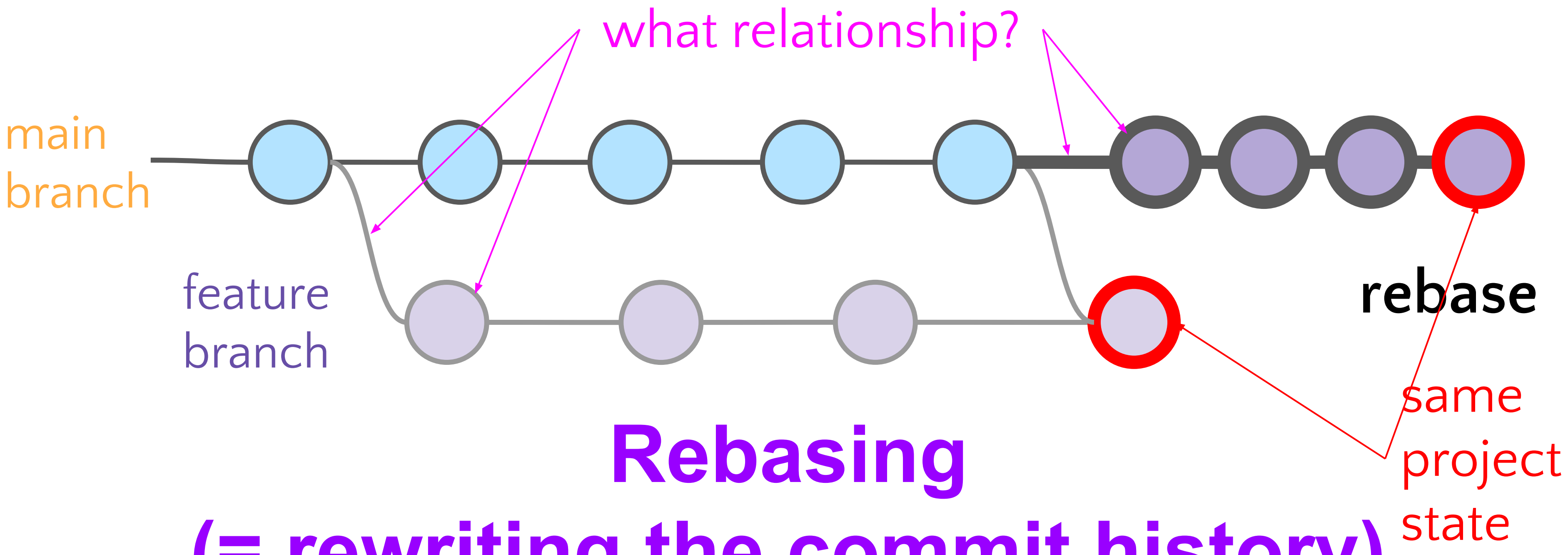
Function name changed

Function name not changed

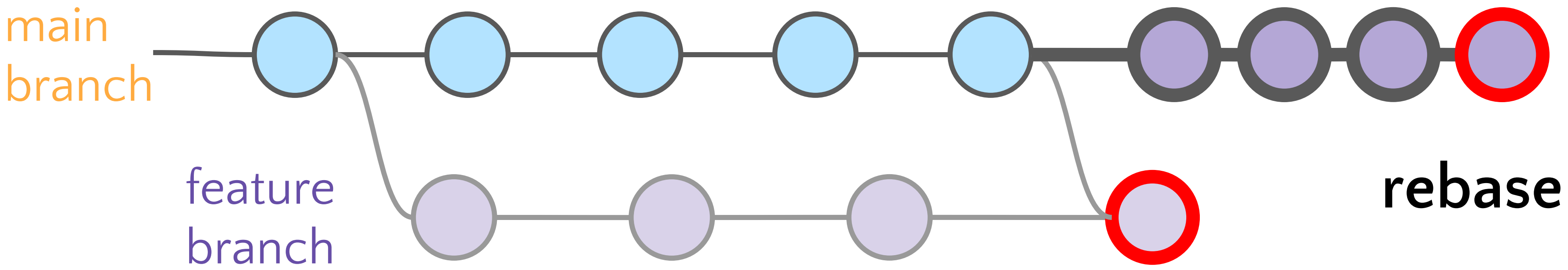
Darcs can record word substitution (for code refactoring)



Rebasing
(= rewriting the commit history)



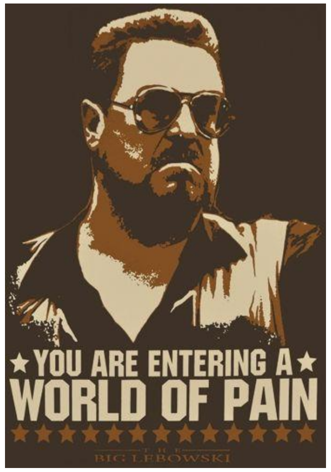
Rebasing
(= rewriting the commit history)



Rebasing
(= rewriting the commit history)

Don’t.

Any questions?



More seriously: why not?

How to avoid merge conflicts

Synchronize with teammates often

- Pull often
  - Avoid getting behind the main branch
- Push as often as practical
  - Don’t destabilize the main branch (don’t break the build)
  - Use continuous integration
    - automatic testing on each PR and push, even for branches
  - Avoid long-lived branches (make frequent, small pull requests)

Commit often

- On the main branch (or any long-lived branch):
  1. Every commit should address one concept (see next slide); commits created *only* via pull requests
  2. Every concept should be in one commit
  3. Tests should always pass
- On feature/bugfix branches:
  1. Each merged branch should address one concern
  2. Don’t worry about the commit history
  3. Get changes into main via a PR; squash and merge it

Make single-concern branches and commits

They are easier to understand, review, merge, revert.

Ways to achieve single-concern branches and commits:

- Do only one task at a time
  - Commit after each one
- Create a branch for each simultaneous task
  - Easier to share work with teammates
  - Single-concern branch ⇒ Single-concern commit on main
  - Requires a bit of bookkeeping to keep track of them all (but worth it)
  - Potential for merge conflicts
- Do multiple tasks in one branch ☹️
  - Commit only specific files, or only specific parts of files
    - use Git's "staging area" with `git add`; can interactively choose *parts* of files

I create a working copy per branch.

Do not commit all files

Use a `.gitignore` file

Don’t commit:

- Binary files
- Log files
- Generated files
- Temporary files

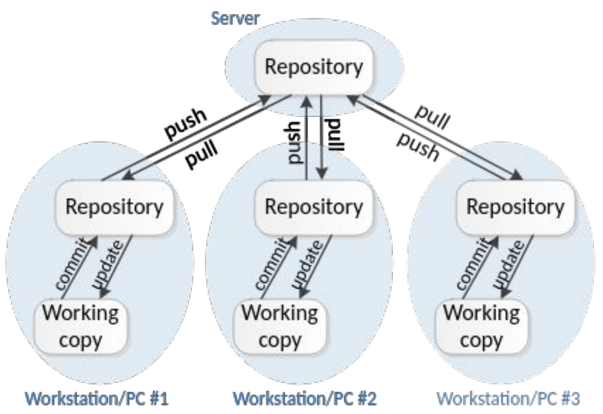
Committing would waste space and lead to merge conflicts

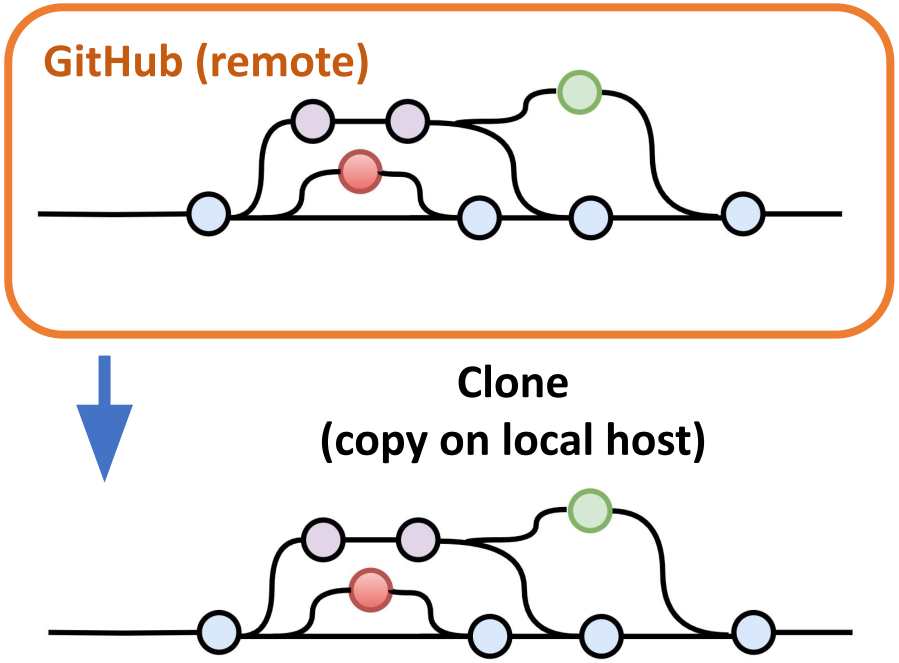
Plan ahead to avoid merge conflicts

- Modularize your work
  - Divide work so that individuals or subteams “own” parts of the code
  - Other team members only need to understand its specification
  - Requires good documentation
- Communicate about changes that may conflict
  - Examples (rare!): reformat whole codebase, move directories, rename fundamental data structures

Cloning

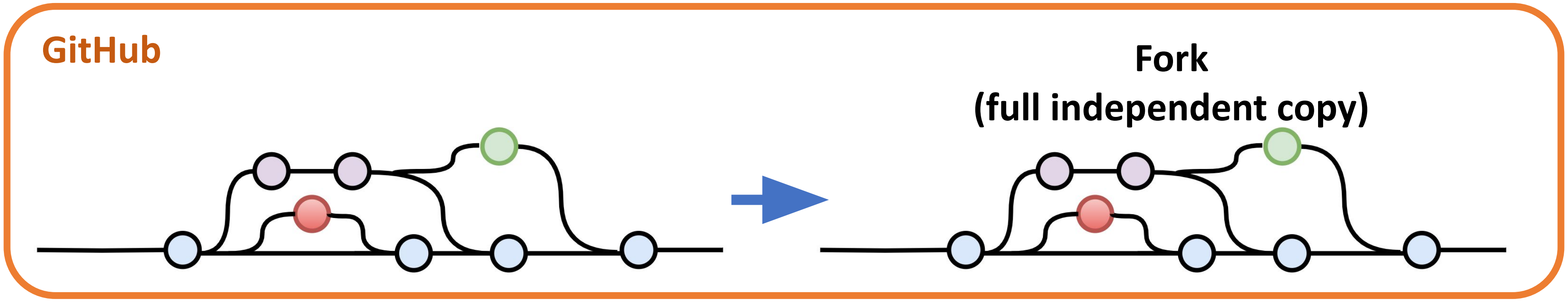
- `git clone` creates a **local copy** of the repo and a working copy of the files for editing
- Ideal for contributing to a repo alongside other developers
- `git push` sends local changes to remote repo





Forking (GitHub concept, not a git concept)

- Creates a **new, unrelated repository** (GitHub project) that is initially an exact copy (*including* SHAs)
- Changes to either repository *do not affect* the other
- You can evolve the fork without impacting the upstream
- If original repo is deleted, forked repo will still exist



- It's possible to update the original but only with **pull requests (original owner approves or not)**

Choose between branch, clone, or fork

Scenario: CSE403 Class Materials GitHub Repo

1. Fix bugs in assignment 1
2. Work on my laptop
3. CSE413 will build upon CSE403
4. A new quarter of CSE 403

What is this Git command?

```
NAME
       git-______ - ______ file contents to the index
SYNOPSIS
       git ______ [--dry-run | -n] [--force | -f] [--interactive | -i] [--patch | -p]
DESCRIPTION
This command updates the index using the current content found in the working
tree, to prepare the content staged for the next commit. It typically ______s the
current content of existing paths as a whole, but with some options it can also
be used to ______ content with only part of the changes made to the working tree
files applied, or remove paths that do not exist in the working tree anymore.
```

What is this Git command?

```
NAME
       git-add - Adds file contents to the index
SYNOPSIS
       git add [--dry-run | -n] [--force | -f] [--interactive | -i] [--patch | -p]
DESCRIPTION
This command updates the index using the current content found in the working
tree, to prepare the content staged for the next commit. It typically adds the
current content of existing paths as a whole, but with some options it can also
be used to add content with only part of the changes made to the working tree
files applied, or remove paths that do not exist in the working tree anymore.
```

Git: concepts and terminology

SYNOPSIS

git-diff-index [-m] [--cached] [<common diff options>] <tree-ish> [<path>…]

DESCRIPTION

git-diff-index compares the content and mode of the blobs found in a tree object with the corresponding tracked files in the working tree, or with the corresponding paths in the index.

Git: concepts and terminology

SYNOPSIS

git-diff-index [-m] [--cached] [<common diff options>] <tree-ish> [<path>…]

DESCRIPTION

git-diff-index compares the content and mode of the blobs found in a tree object with the corresponding tracked files in the working tree, or with the corresponding paths in the index.

SYNOPSIS

git-allocate-remote *[--derive-head | --massage-link-head | --abduct-commit]*

DESCRIPTION

git-allocate-remote allocates various non-branched local remotes outside added logs, and the upstream to be packed can be supplied in several ways.

SYNOPSIS

git-resign-index *[--snap-file] [--direct-change]*

DESCRIPTION

git-resign-index resigns all non-stashed unstaged indices, and the --manipulate-submodule flag can be used to add a branch for the upstream that is counted by a temporary submodule.

Git: concepts and terminology

SYNOPSIS

git-diff-index [-m] [--cached] [<common diff options>] <tree-ish> [<path>…]

DESCRIPTION

git-diff-index compares the content and mode of the blobs found in a tree object with the corresponding tracked files in the working tree, or with the corresponding paths in the index.

SYNOPSIS

git-allocate-remote *[--derive-head | --massage-link-hea[illegible] --abduct-commit]*

DESCRIPTION

git-allocate-remote allocates va[illegible] non-[illegible]anched l[illegible] l[illegible]em[illegible]es outside added logs, and the upstream to be packed can be s[illegible] in se[illegible]l wa[illegible]

FAKE

SYNOPSIS

git-resign-index *[[illegible]na[illegible]file] [[illegible]direct[illegible]nge]*

DESCRIPTION

git-resign-index resigns all non-stashed unstaged indices, and the --manipulate-submodule flag can be used to add a branch for the upstream that is counted by a temporary submodule.

Git’s confusing vocabulary

- **content**: git tracks **what is in a file, not the file itself**
- **tree**: git's representation of a file system
- **working tree**: tree representing the local working copy
- **commit**: a snapshot of the working tree (a database entry)
- **SHA**: a unique identifier for a commit
- **ref**: pointer to a commit object
- **branch**: just a (special) ref; represents a line of development
- **HEAD**: a ref pointing to the working tree
- **staged**: ready to be committed (you ran `git add`)
- **index**: staging area (located in .git/index)

Ask me anything

Learn more!

- Other resources: explanations, tips, best practices
  - GitHub git cheat sheet
  - Michael Ernst: VC Concepts and Pull Requests
  - Atlassian merge vs rebase (but **don’t** rebase)
  - Git branching and merging
  - Video tutorial “Git, GitHub, & GitHub Desktop”
  - Learn Git Branching