# Introduction to Git

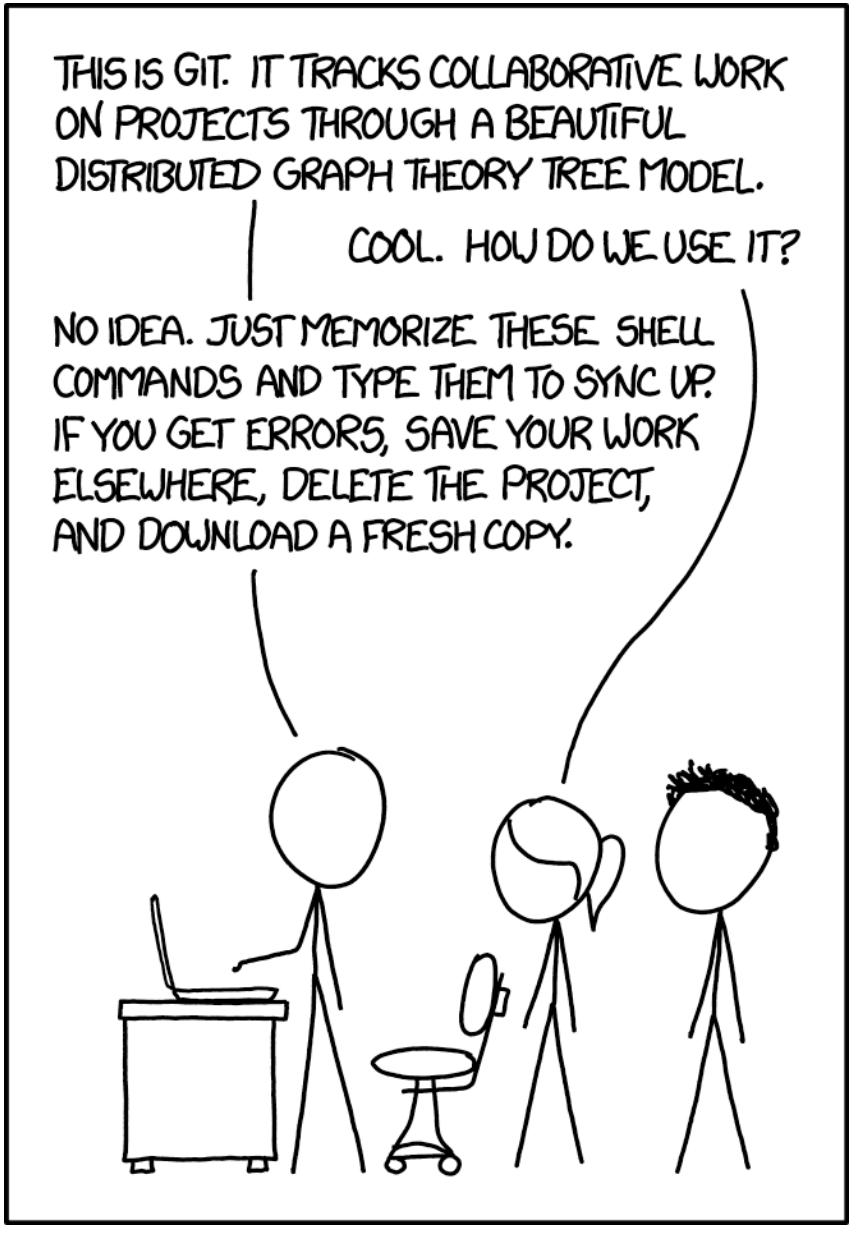
**"The stupid[1] content tracker"**

Naoki Pross — np@0hm.ch

XX. March 2025

[1] *git* (British) – a foolish or worthless person

# Obligatory XKCD




## Plan for Today

1. A tiny bit of graph theory and even less cryptography
2. Understand (instead of memorizing) Git
3. Flex on your friends by finding what caused a bug using a logarithmic search over the directed acyclic graph that represents the change history
4. Put it on your CV and profit

# Table of Contents

# What do we want?

## The Problem

Synchronize data across multiple computers, with multiple people working on (possibly the same) files.

## Linus' Wishes (The guy who invented Git)

- Synchronization *always* works
- Teamwork is possible and efficient
- Works offline
- Fast

neither *intuitive nor easy to use* were not on his list!

# Other Solutions?

## Popular at Linus' Time

**CVS** Slow to synchronize. CVS requires a centralized server which can get overloaded, was usually set up by the company IT.

**E-Mail** People sent patch files to each other via email.

## Popular Tools Today

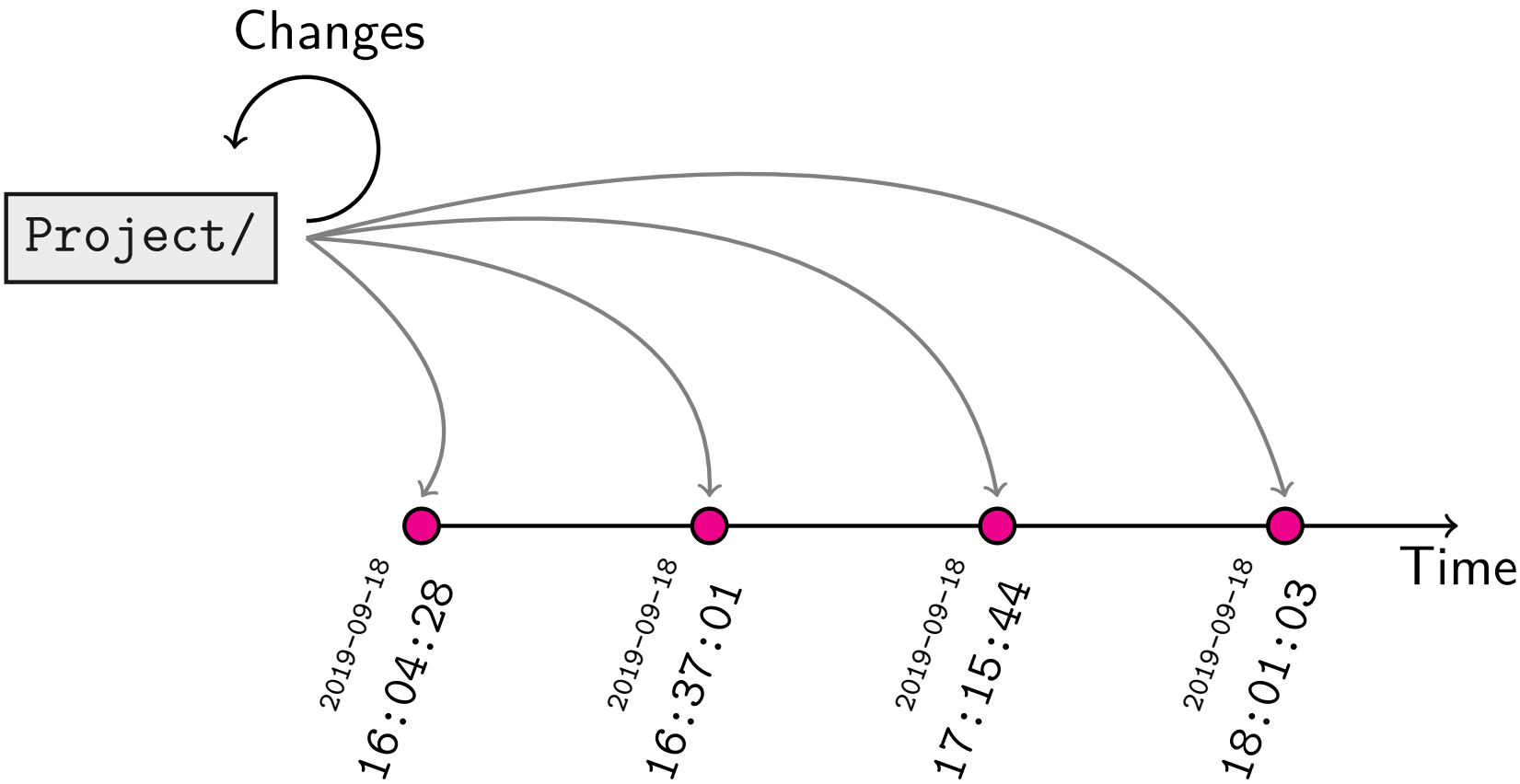
**Cloud Storage** Does not work offline. Their whole business model is against you. You have no (real) control over when to sync. Also, sharepoint is garbage. No way to compare changes.

**Mercurial** Less popular than Git, used by Mozilla.

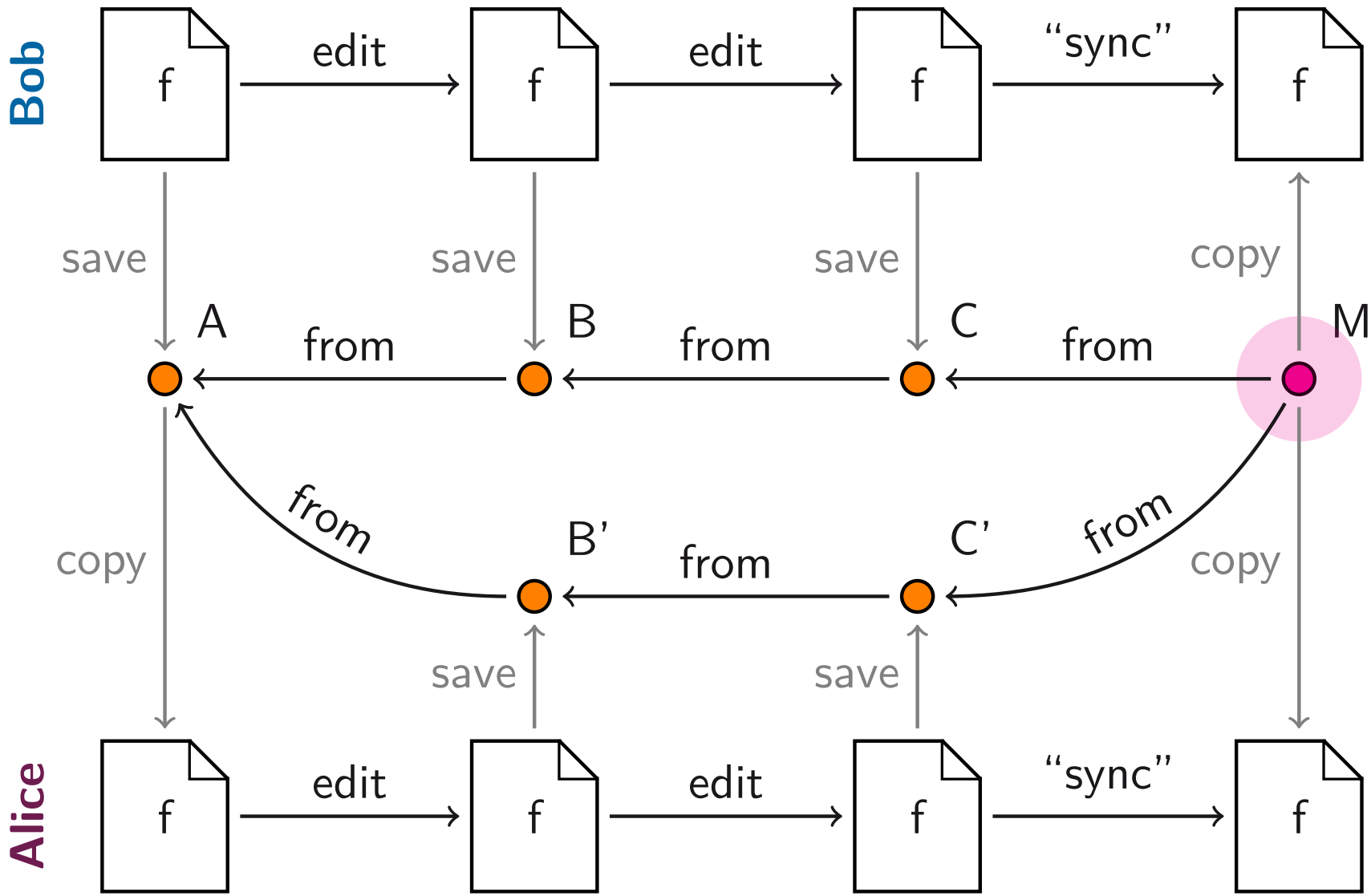
**Jujitsu** Git-compatible VCS, even less popular and very new.

# Table of Contents

# Solving the Problem: Snapshots



# Solving the Problem: Concurrent Changes I



# Solving the Problem: Concurrent Changes II

## High Level Overview

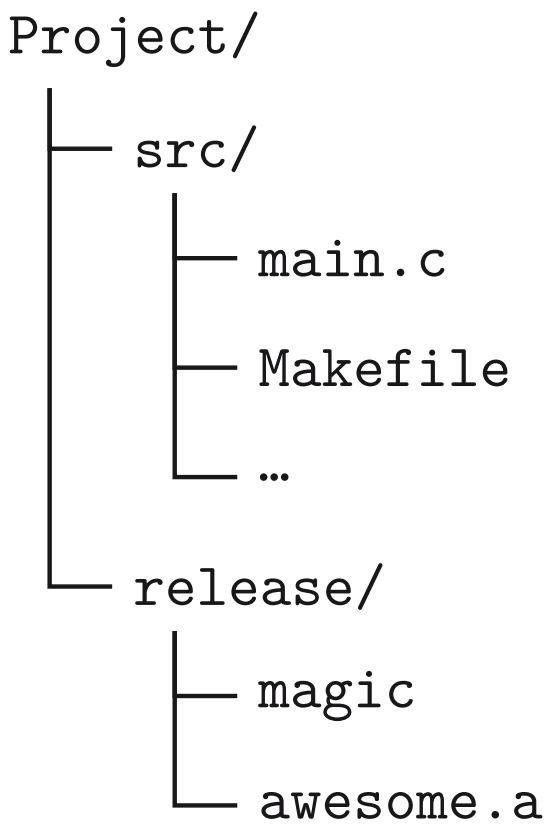
Store changes using a *directed acyclic graph* (DAG) called the *commit graph*.

- Nodes are saved points in time called *commits*
- Arcs point to state from which change was made
- Commits with multiple children (A) are *branching commits*
- Commits with multiple parents (M) are *merge commits*

## Problems

1. We care about file content not the files itself
2. How do we merge changes?
3. Alice and Bob are not working on the same computer

# Solving the Problem: Multiple Files




## Filesystem Jargon

**Tree** Folder / Directory

**Blob** Binary Large OBject, raw data (bits) of file content[a]

**File** Blob + Metadata (Name, Date, …)

## Solution

Treat all blobs as single entity with metadata. Examples:

- Rename file ⇒ Same blob, commit name change
- Move file ⇒ Same blob, commit change tree

[a] Demo: `hexdump` vs `stat`

# Mathematical Digression: DAG

## Directed Acyclic Graph

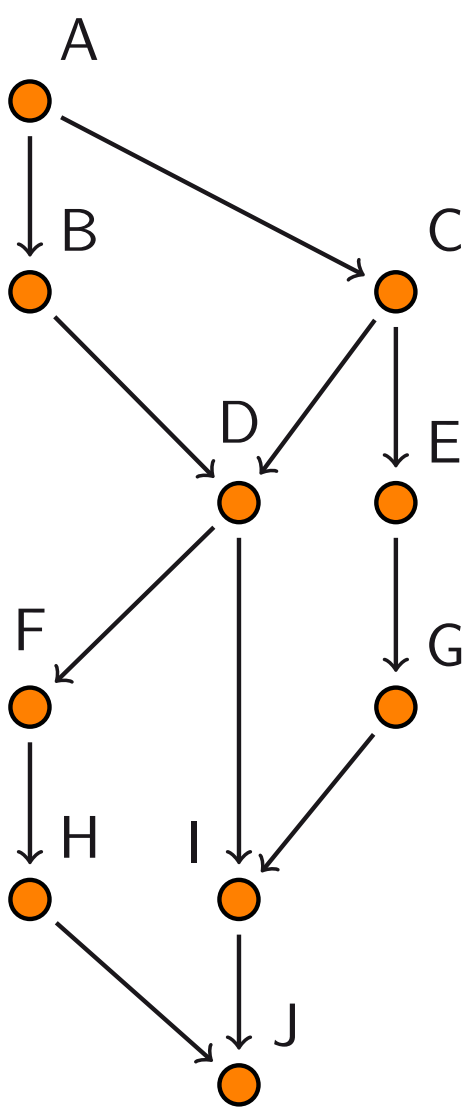
A DAG $G = (V, A)$ is defined by a finite set of vertices $V$ and a finite set of *arcs* $A$ and may not contain loops.

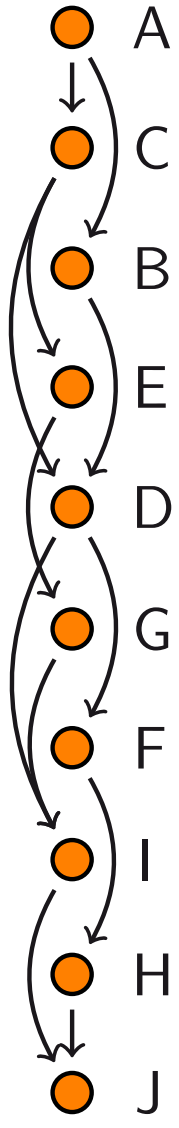
## Partial Order

DAG have a partial order relation $u \succ v$ for comparable $u, v \in V$.

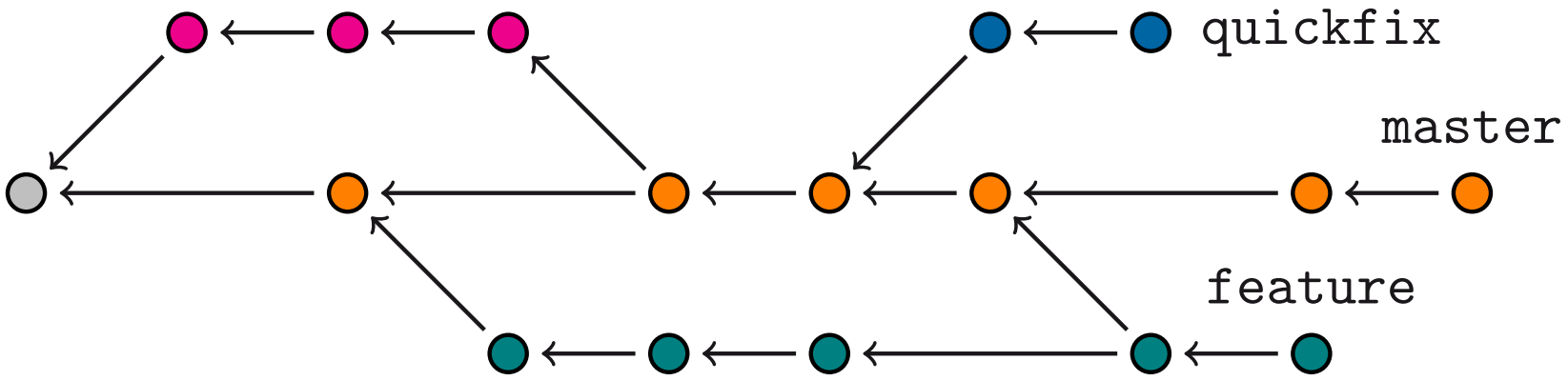
## Topological Order

A DAG $G = (V, A)$ has a total order $\succ^*$ by having that for all $(u, v) \in A$ $u \succ^* v$. If $G$ has a Hamiltonian path $\succ^*$ is unique.



# Solving the Problem: Concurrent Changes III




## Branch (informal)

Branches are subgraphs (subtrees) from a common anchestor in the commit graph.

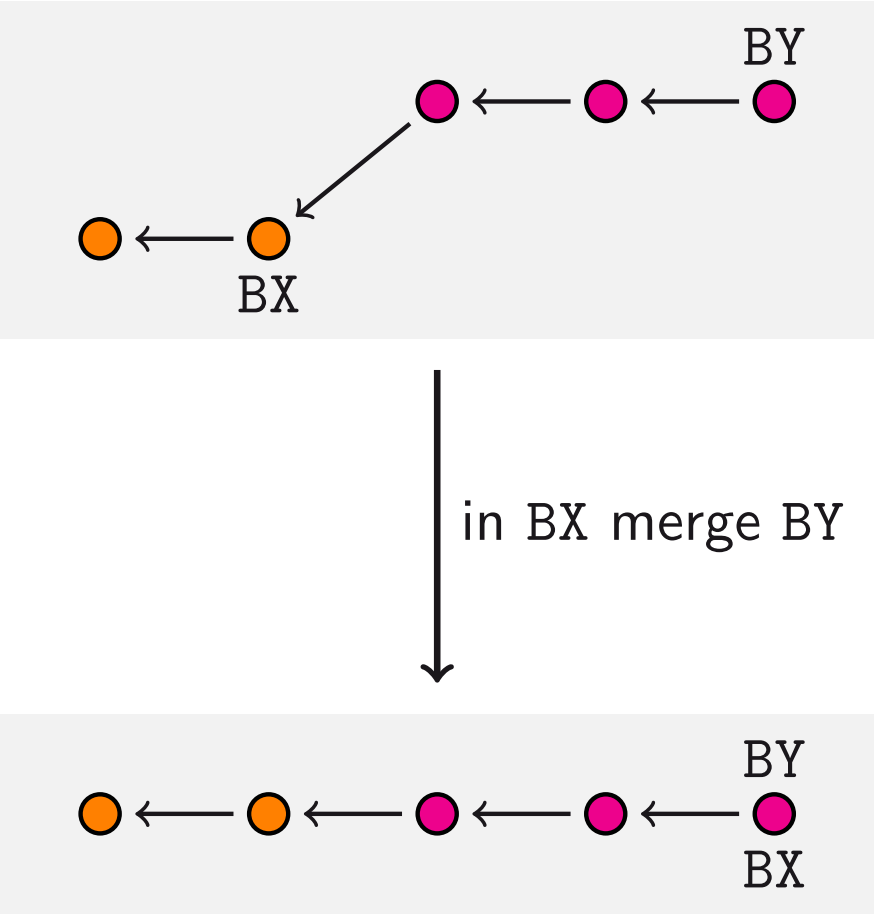
## Naming Branches

Branch names are labels on their most recent commit.

## Examples

- `quickfix` branch is from `master`
- Magenta (no name) branch was merged into `master`
- `master` branch was merged into `feature`

# Solving the Problem: Fast-Forward-Merge



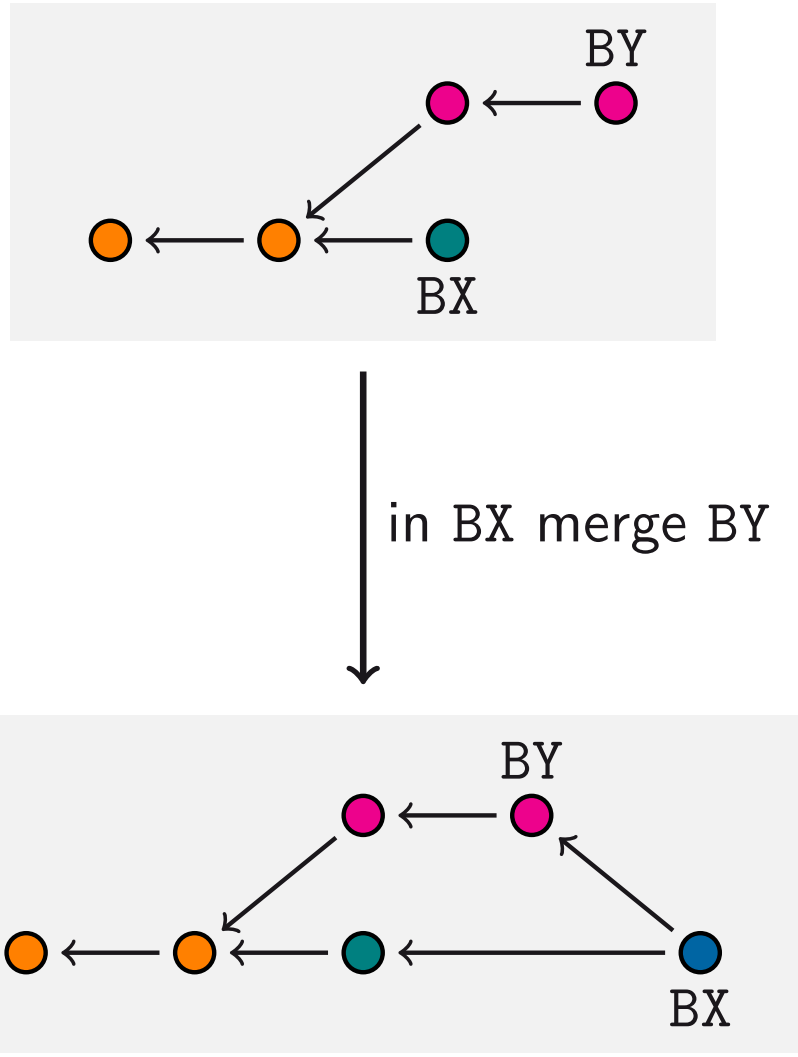

## History

1. From an existing branch BX (with orange commits) a branch BY added new commits (magenta)
2. We merge BY into BX

## FF-Merge

Apply changes of commits in BY starting at BX until you get to BY. Or BX just needs to "catch up" to BY. No new commits are created.

# Solving the Problem: 3-Way-Merge I




## History

1. Branches BX and BY have new commits (magenta and green resp.) and share a common history (orange)
2. We merge BY into BX

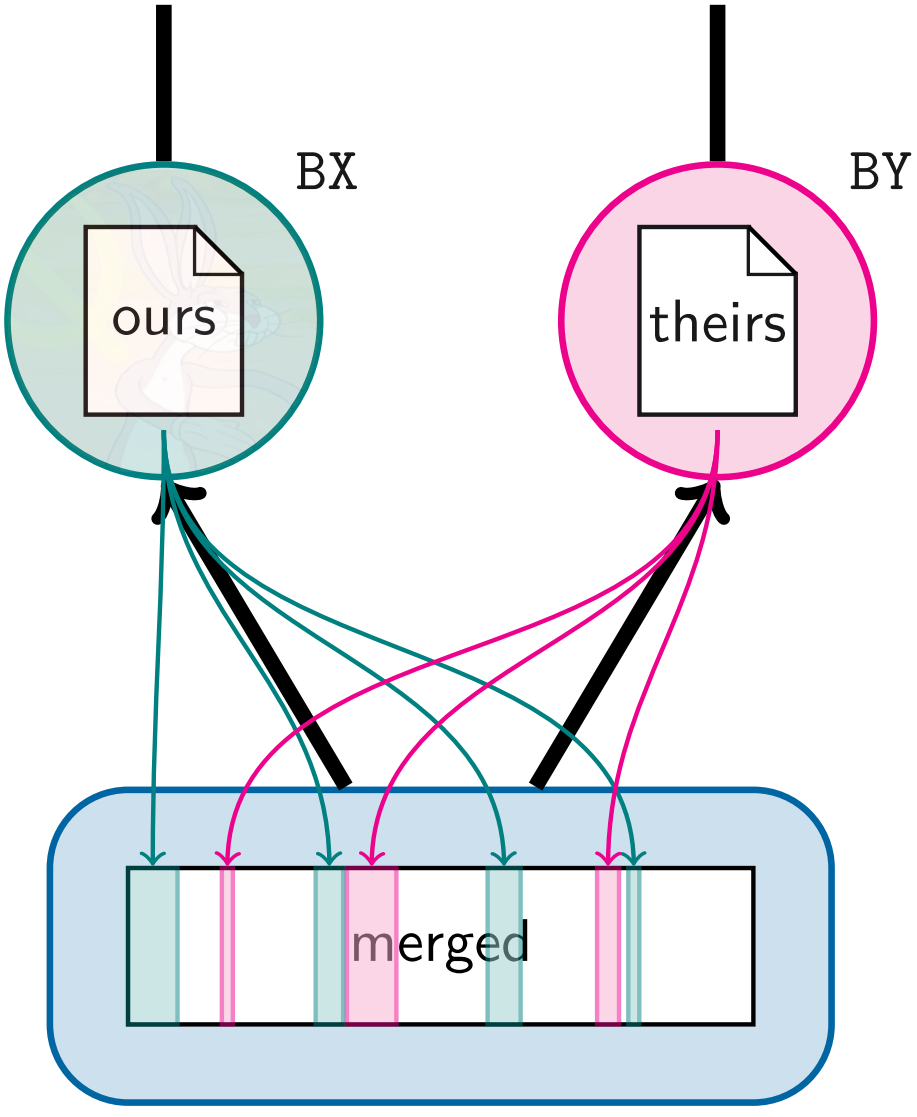
## Observations

When you merge you are in BX importing changes from BY

- "our" changes are from BX
- "their" changes are from BY

Need to make choices, which get saved in a new merge commit.

# Solving the Problem: 3-Way-Merge II




Merge BY into BX

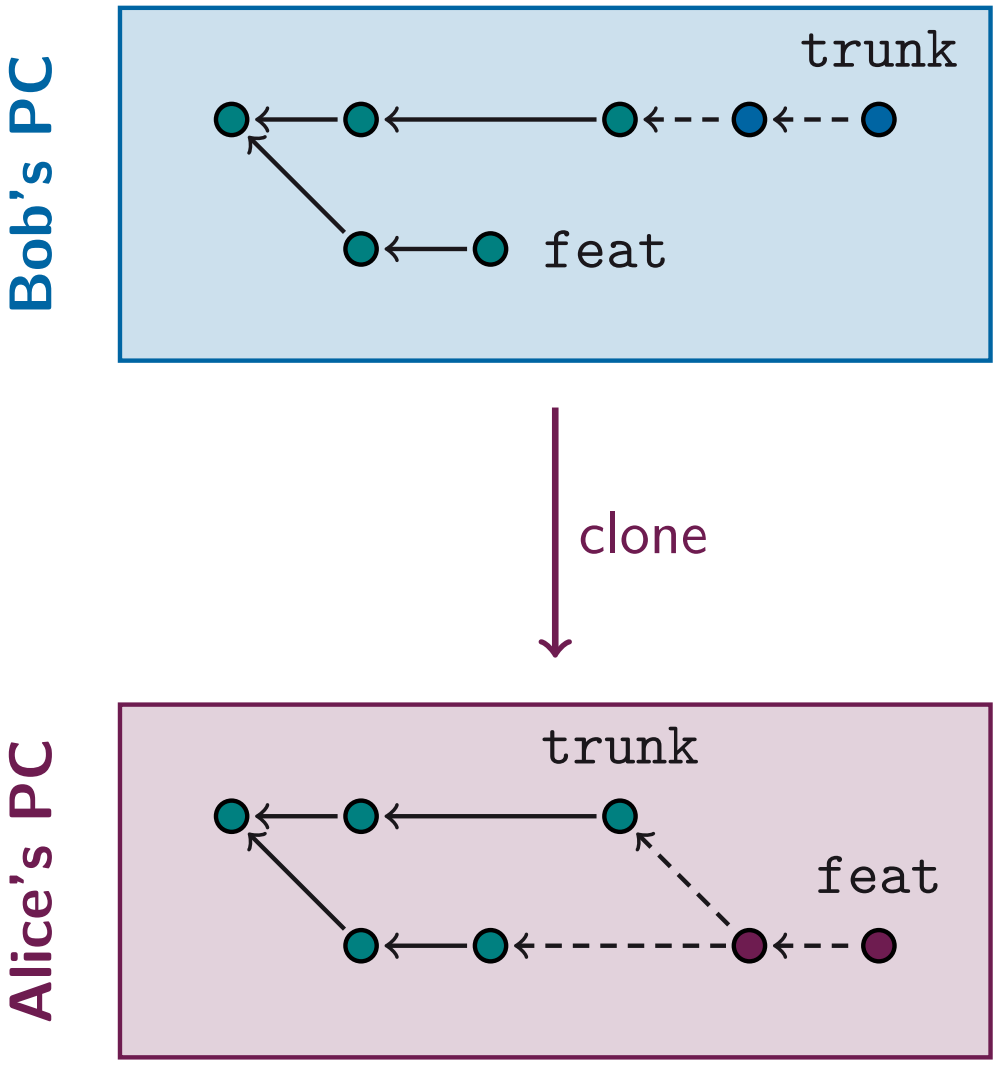
## 3-Way-Merge

- Use a (3-way-merge) algorithm to merge trees and blobs from each commit
- If not possible the user has to choose between 'our' changes and 'their' changes

## Merge Conflict

When the algorithm cannot merge the file automatically it is called *merge conflict*.

# Solving the Problem: Multiple Computers I



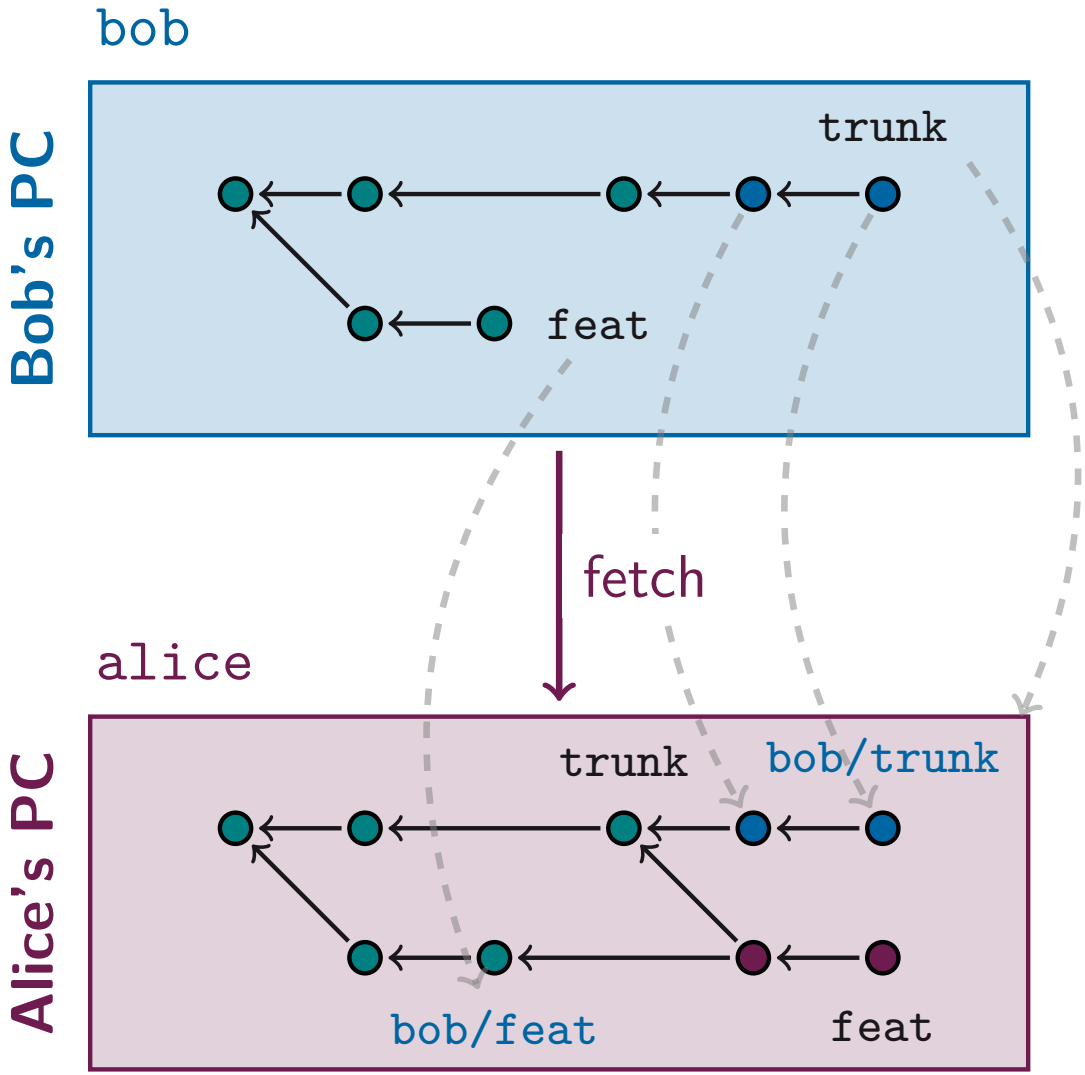

## Remotes and Clone

Other computers are called *remotes*. Clone means you copy the commit graph on the remote machine onto yours.

## Example

1. Alice has cloned Bob's (green) commit graph
2. Alice has merged `trunk` onto `feat` and made changes
3. Bob has also made changes on `trunk`

# Solving the Problem: Multiple Computers II




## Fetch

Copy the changes of the remote git graph into your local git graph.

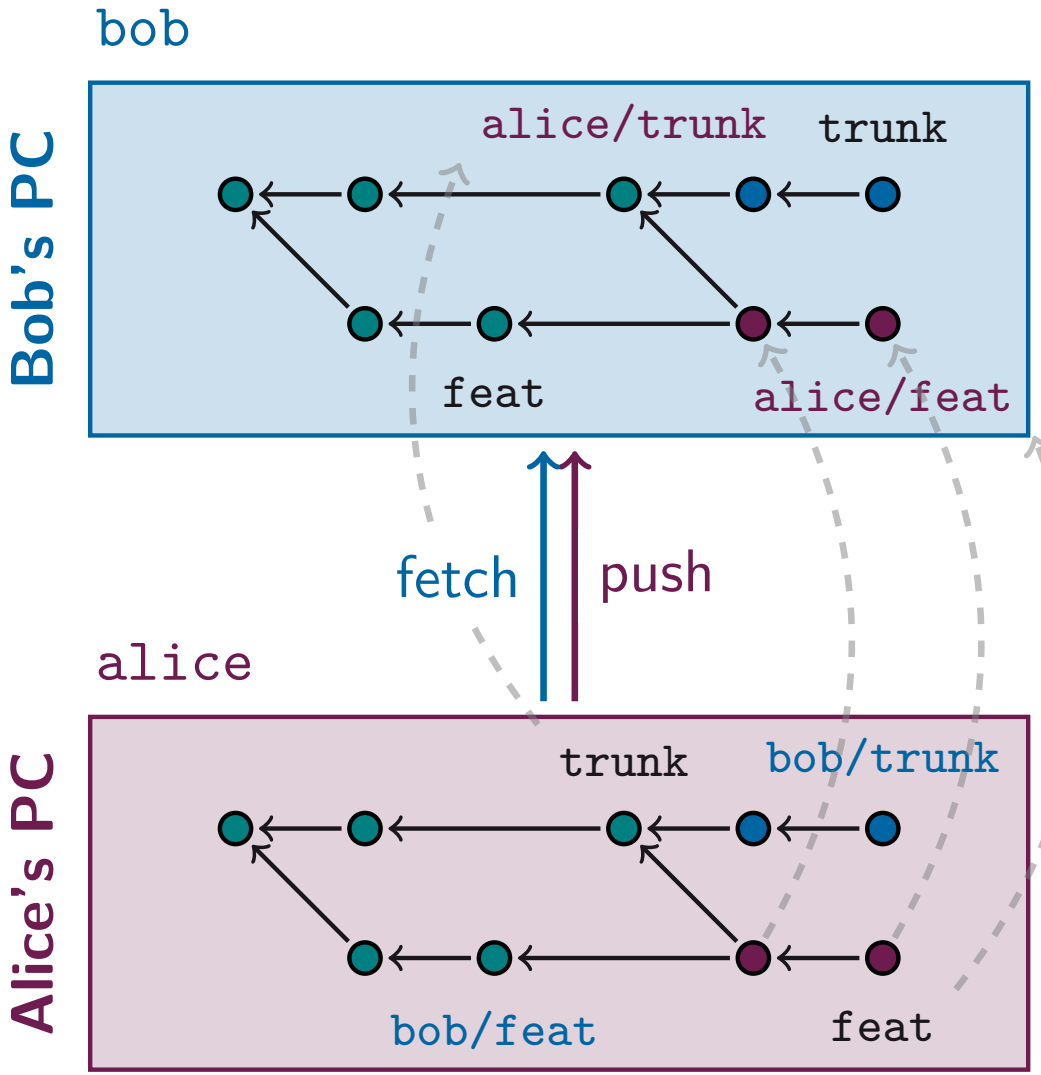
## Running Example

Alice fetches Bob's changes.

## Remote Branches

A branch that represents changes done in another machine. When a graph is cloned, the machine from which it was cloned has the default name `origin`.

# Solving the Problem: Multiple Computers III




## Push

Copy the changes of your local git graph to the remote machine.

## Running Example

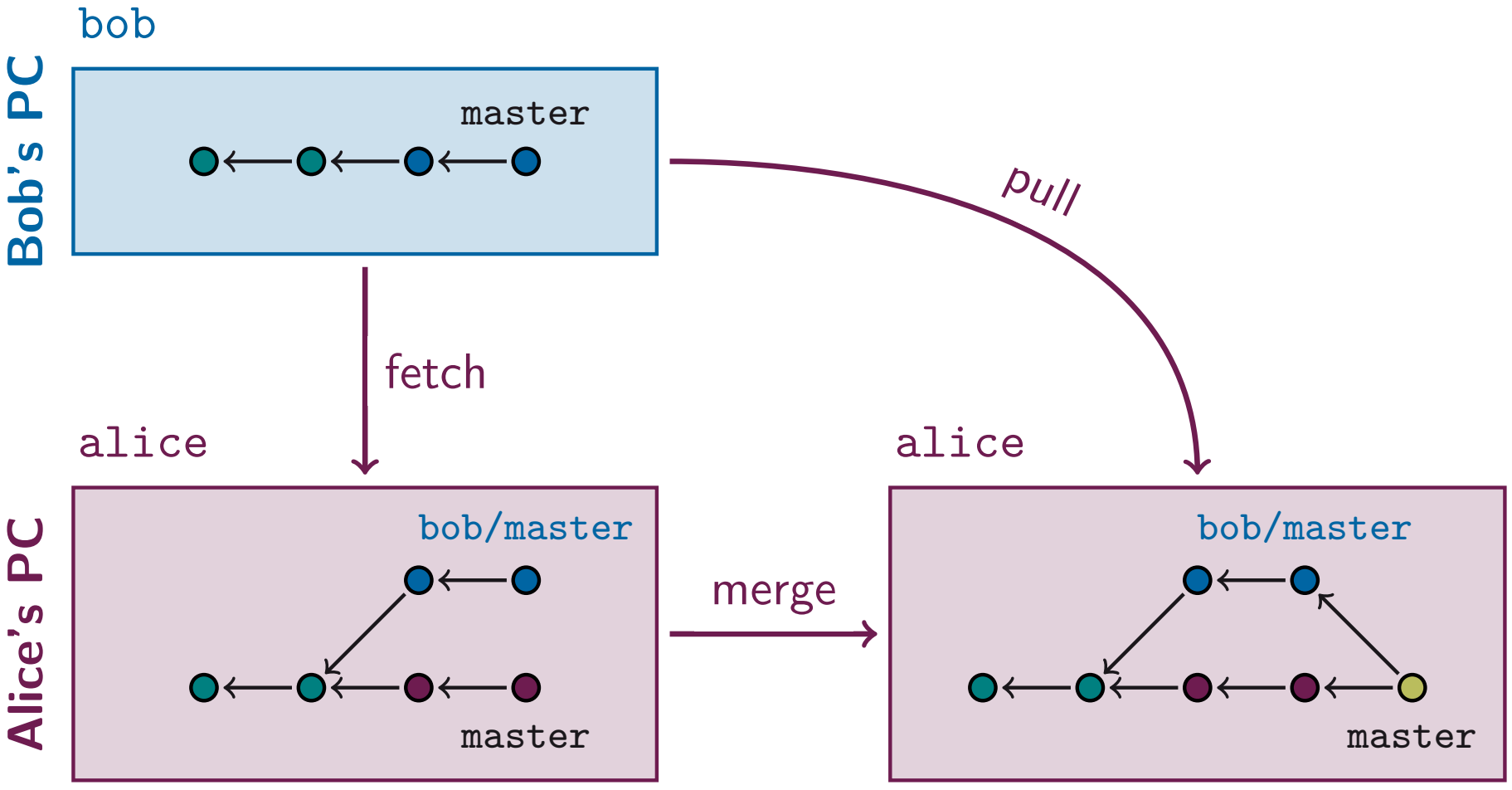
This is the same as if Bob had fetched Alice's changes.

## Network Access

In practice you cannot directly access other people's machines, so people use a third computer to which both parties have access (more later).

# Solving the Problem: Multiple Computers IV

Since it is very a common operation there is a shorthand to fetch and merge the remote branch with the same name as the current one.



# Table of Contents

# Mathematical Digression: Hashes and Merkle DAG

**"One-way fast" functions**

## Hash Function

A (cryptographic) *hash* function is an $h : \Omega \to \{0,1\}^d$ for a fixed hash length $d$ such that:

1. Given $y = h(x)$ it is hard to find $x$
2. It is hard to find $x, y \in \Omega$ s.t. $h(x) = h(y)$
3. Given $h(x)$ it is hard to find $y$ s.t. $h(x) = h(y)$
4. Given $h(x)$ and a function $f$ it is hard to find $h(f(x))$

Hashes are *not* unique!

## Merkle DAG

A Merkle DAG is a DAG $G = (V, A)$ with a hash

$$h : V \times \{0,1\}^d \to \{0,1\}^d$$

that defines a label function

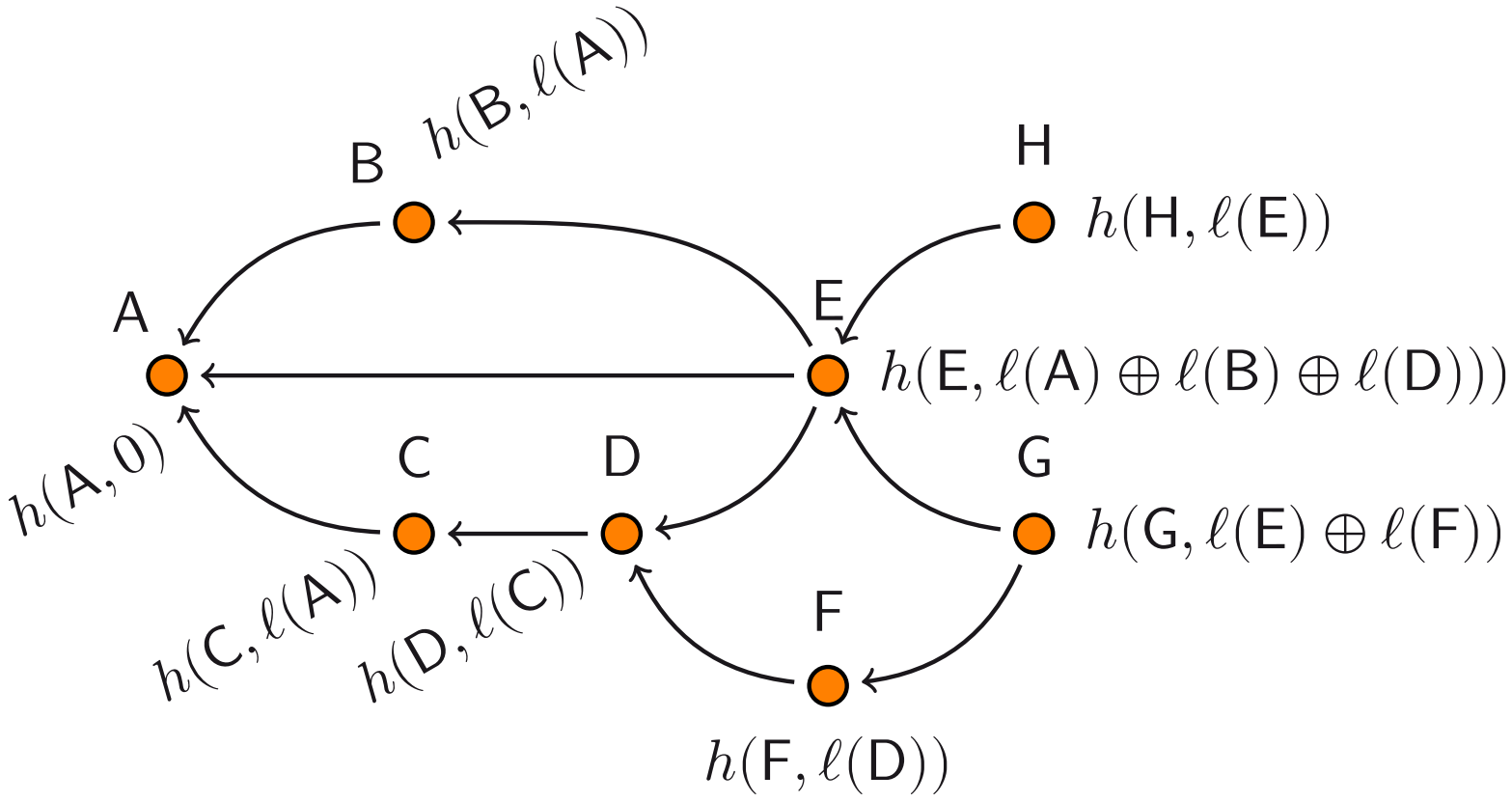
$$\ell(v) = h\left(v, \sum_{u \in \mathrm{n}^+(v)} \ell(u)\right)$$

## Properties

- Immutable data structure
- Cryptographic verification

# Mathematical Digression: Visualizing Merkle DAGs

To compute the label of a node, you need to first compute the label of all nodes on which it depends. Changing a label has a cascading effect on descendents.




Technicality: Sum symbol represents hash concatenation.

# Git Commits

## Commit Contents

- Content (Blobs and Trees) hash
- Parent(s) commit(s) hash(es)
- Metadata: Author, Date, Message

## Example

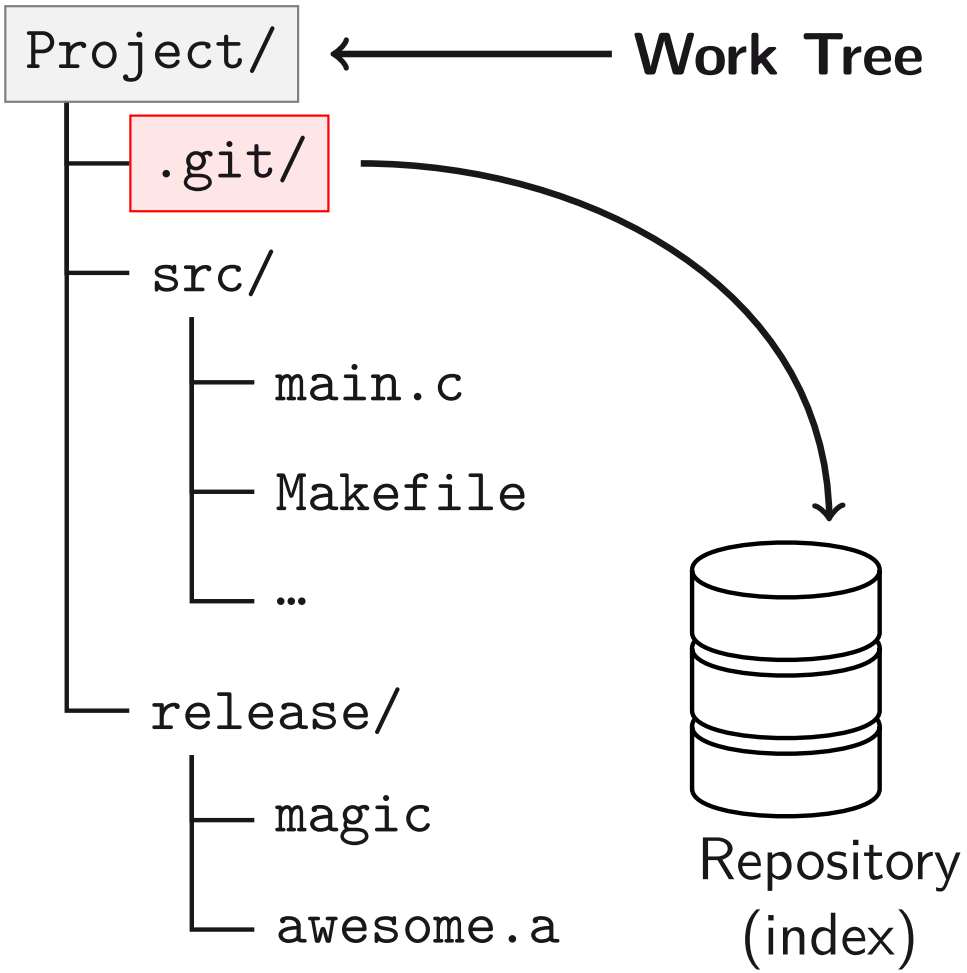
```
commit 1cfdf5c198f1c74c2f894067baf4670f5bca8e70
Author: Nao Pross <np@0hm.ch>
Date:   Wed Feb 9 19:53:06 2022 +0100

  Fix arrayobject.h path on Debian based distros

  On Debian Linux and its derivatives such as Ubuntu and LinuxMint, Python
  packages installed through the package manager are kept in a different
  non-standard directory called 'dist-packages' instead of the normal
  'site-packages' [1].

  To detect the Linux distribution the 'platform' library (part of the
  Python stdlib) provides a function 'platform.freedesktop_os_release()'
```

# Git Repositories




## Work Tree

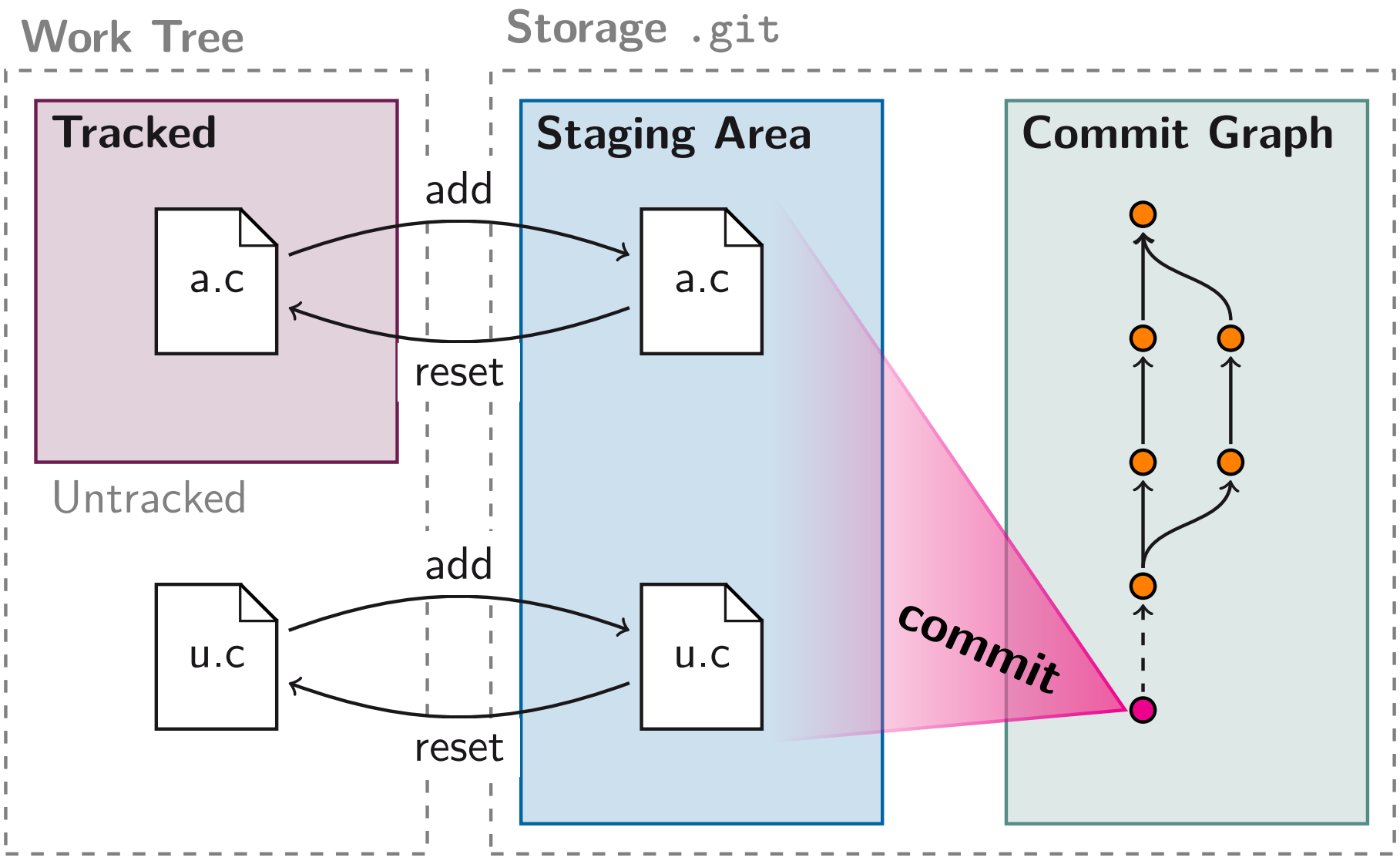
Root of your project, contains (hidden) `.git`.
**Never delete `.git`.**

## Repository

- Commit graph (Blobs, ...)
- Staging Area (will come next)

# Table of Contents

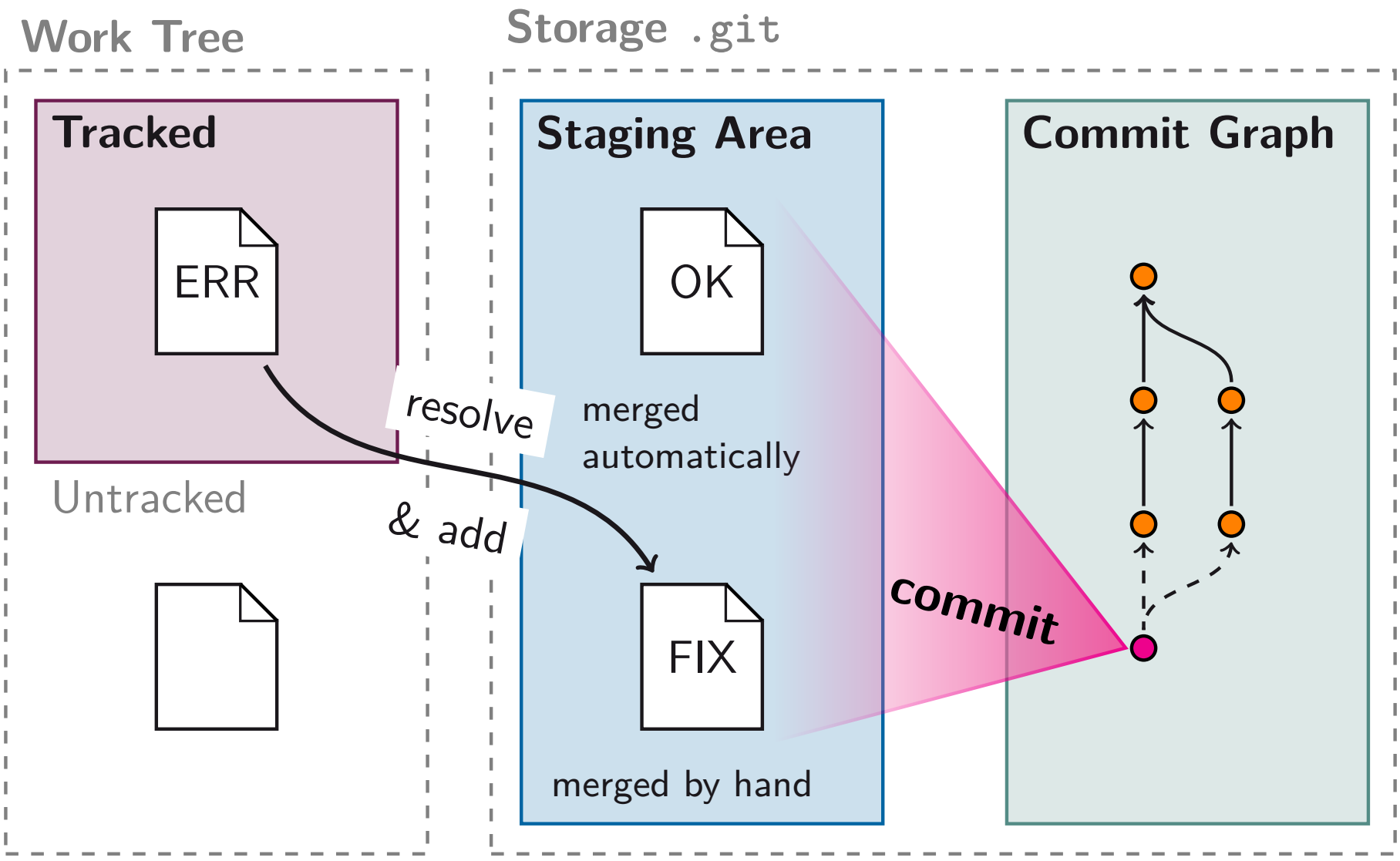
# The 3 (or 4) Conceptual Areas of Git



# Branches, Remotes and your HEAD

- Setup git the first time
  - `git config --global user.name "Your Name"`
  - `git config --global user.email your@email`
  - `git config --global core.editor your-editor`
  - `git config --global alias.graph "log --all --decorate --graph --oneline"`
- Your work tree state is at commit where you have your HEAD
  - `git status`
  - `git log`
- Adding changes to the staging area and committing them
  - `git add [-u]`
  - `git commit [-a] [-v]`
- Branching and the detached HEAD state
  - `git branch`
  - `git switch`
  - `git checkout`
  - `git merge`
- Managing remotes and Cloning
  - `git remote add / ...`
  - `git clone`
  - `git fetch / push / pull`

# Automatic Merge Failed (Conflicts)



# Restoring Changes from the Past

# Graphical User Interfaces

## Command Line Interface

If you learn to use Git on the terminal you are set forever, but

- you have to think (tip: abuse `git status` and read, always!)

## Graphical Interfaces

A good GUI that does not hide complexity

- Sublime Merge

Alternatives

- SourceTree, GitKraken, lazygit (terminal UI)
- TortoiseGit (integrates with Windows Explorer)

Bad GUI (why? It tries to hide complexity until you inevitabily screw up something, and then you have no clue what is going on)

- GitHub Desktop

More at `https://git-scm.com/downloads/guis`

# What is a Commit Anyways?

When you work with git, a commit should be a

## Logical Unit of Work

i.e. you think of a specific thing you need to do, you do it, then immediately commit the changes, repeat. The more people in the team, the more commits you should do.

### Bad

- Huge commits that change multiple unrelated files / classes / functions / ...
- Meaningless messages like “fix”, “update”, “wip” or “misc”
- Messages in past tense

Demo: Browse Linux kernel git commits

### Good

- Small, modular changes, change only one file / class / function / ...
- Descriptive commit message, just title with bullet points are ok!
- Commit even if does not compile
- Message in imperative tense. Write as if you were the recipient of the commit: what will it do?

# Trunk, Feature Branches

This pertains software developement and project management.

## Simplest Workflow

- There is no team
- Commit everything on the same branch
- Extremely easy

## People Workflow

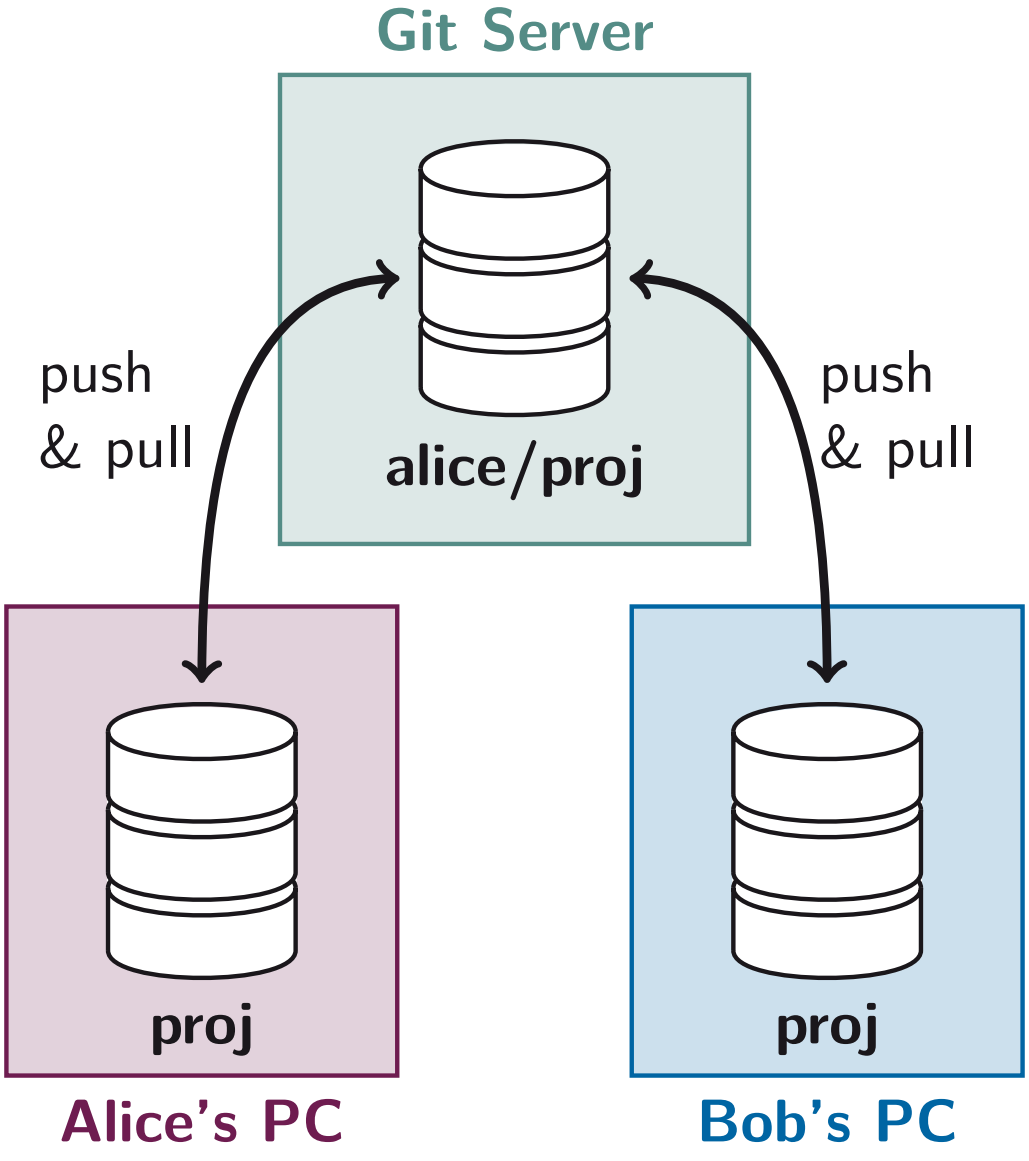
- For very small teams
- Very easy
- Each person has a branch
- Avoids mixing work

## Branching Workflow

Every time you want to add a feature, make changes in a short lived branch until feature is complete, then merge that branch into a “master” branch (also called “main” or “trunk”)

- For larger teams or organized people
- Changes are more organized
- Typically, master branch always compiles

# Git Services (GitHub, GitLab, ...)




## Git Server

Because of IPv4 NAT, Bob can't push directly to Alice's computer. So they use a third computer that hosts a *git server* that can be reached by both.

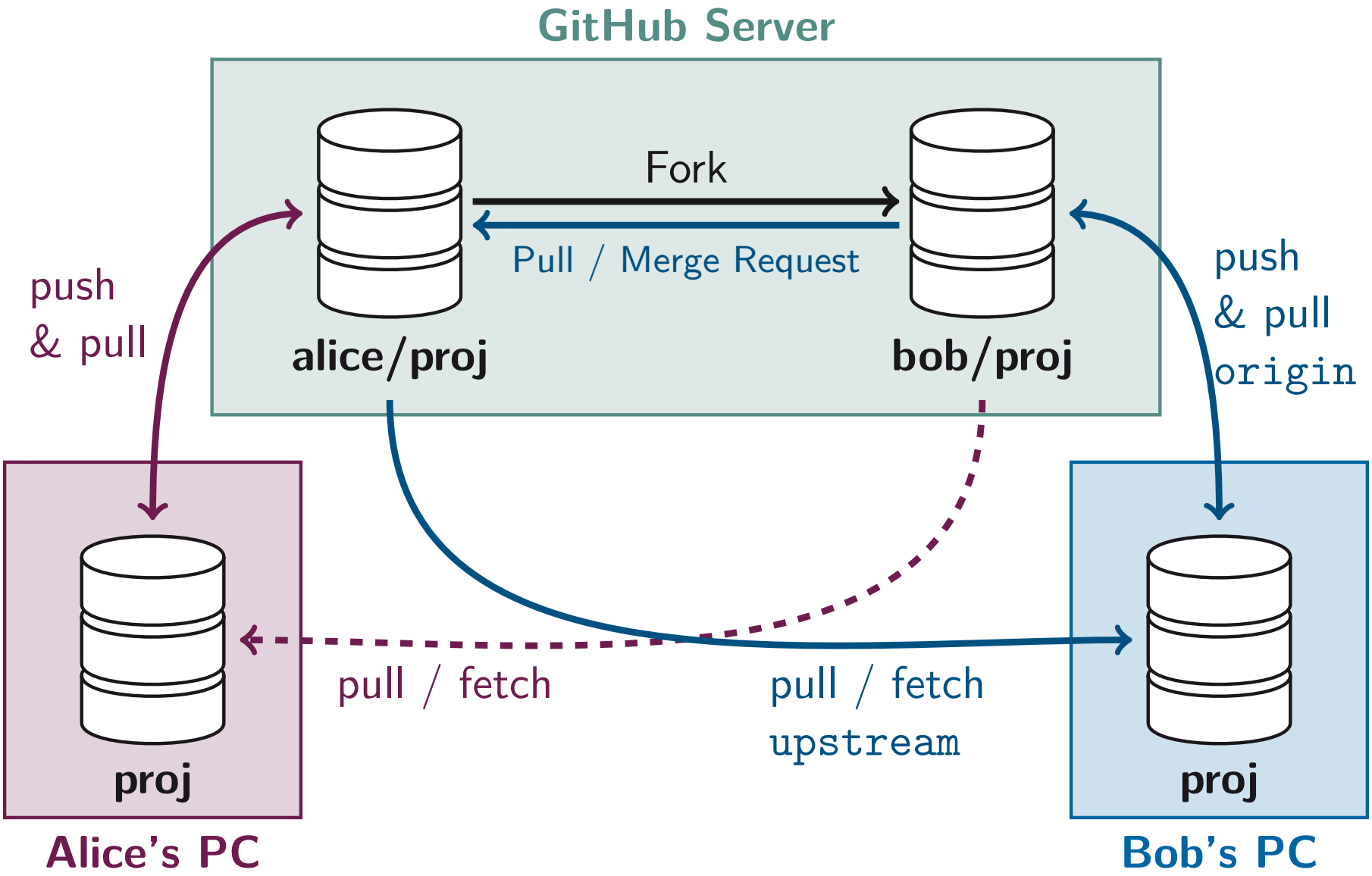
## Interface

The server (usually) has a web interface with a login to manage your hosted repositories. In this example Alice has a project **proj** under the username **alice**.

## Examples

GitHub, GitLab, Gitea, Codeberg, SourceHut, rgit.

# Forking and Pull / Merge Requests




Note: Actually the fork does not need to be on the same server, but let's keep this simple.

# Forking and Pull / Merge Requests

Most services provide a way to fetch the pull requests as if they were remote branches on your repository. Assuming the remote is named[2] `origin` for GitHub this can be achieved by running

1. `git config --add remote.origin.fetch '+refs/pull/*/head:refs/remotes/origin/pr/*'`
2. `git fetch origin`
3. `git switch pr/5`

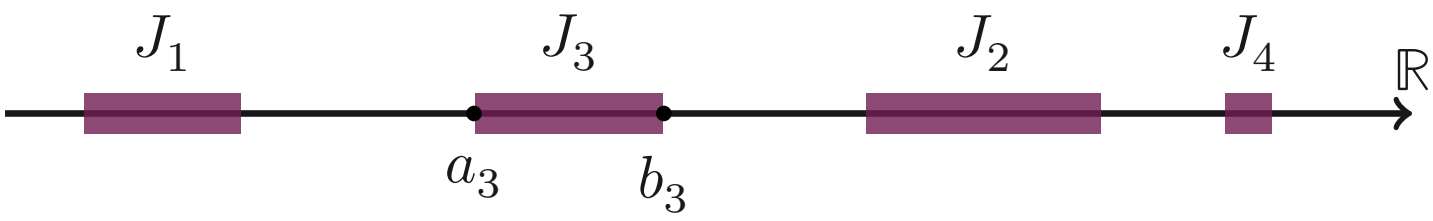
Then, you can check out any pull request using their ID, here for example #5. The same for GitLab:

1. `git config --add remote.origin.fetch '+refs/merge-requests/*/head:refs/remotes/origin/mr/*'`
2. `git fetch origin`
3. `git switch mr/5`

[2] If you remote has a different name, replace all occurences of `origin`.

# Table of Contents

# Mathematical Digression: Logarithmic Search I




## Toy Problem

Given a set of disjoint intervals $S = \{J_1, \ldots, J_n\}$, $J_i \subset \mathbb{R}$, $\log_2(n) \in \mathbb{N}$ find to which interval belongs $q \in \bigcup_i J_i$.

## Naive Solution

For every $J \in S$ interval check if $q \in J$. This is $O(n)$.

## Total Order in $S$

Intervals $[a, b) \in S$ can be ordered. Define $J_i \succ J_j$ if $a_i > a_j$.

## Logarithmic Search Intuition

If $q \notin J = [a, b)$ then either

- $q > a$ so $q \in J' \succ J$
- $q < a$ and $q \in J' \prec J$

# Mathematical Digression: Logarithmic Search II

## Logarithmic Search Intuition

If $q \notin J = [a, b)$ then either

- $q > a$ so $q \in J' \succ J$
- $q < a$ and $q \in J' \prec J$

## Idea

Recursively apply intuition.

## Complexity (Landau)

Base $b$ logarithmic search is $O(\log_b(n))$. In this case $b = 2$ (two options $q > a$ or $q < a$), so it usually called *binary* search.

## Logarithmic Search

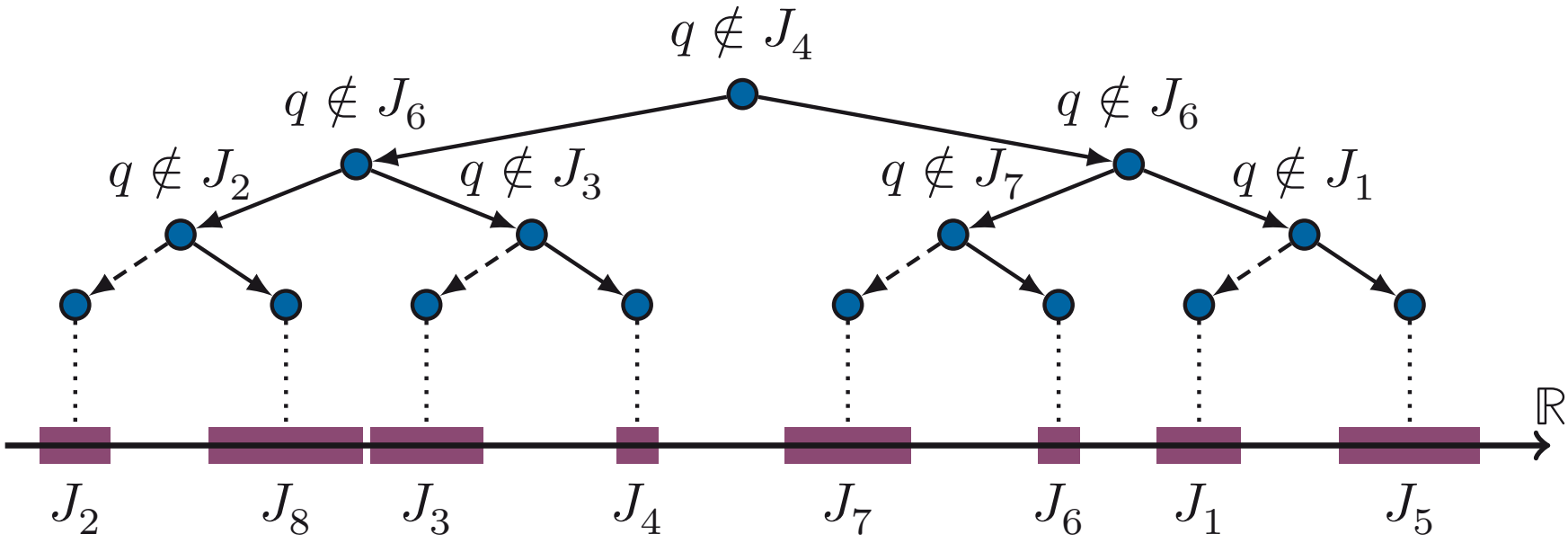
Start with $Q = S$ then

1. take $J \in Q$ in the "middle" of $Q$ and if $q \in J$ we are done
2. otherwise
   1. if $q > a$ repeat with $Q := \{J' \in Q : J' \succ J\}$
   2. if $q < a$ repeat with $Q := \{J' \in Q : J' \prec J\}$

Does not check every $J \in S$ (fast for large $n$!).

# Mathematical Digression: Logarithmic Search III

We can visualize the decisions of logarithmic searching as a tree. The decision goes to the left or right branch depending on whether $q < a$ or $q > a$ respectively. Obseve that the tree has depth $3 = \log_2(8)$.



# Git Bisect: Search for something in the past

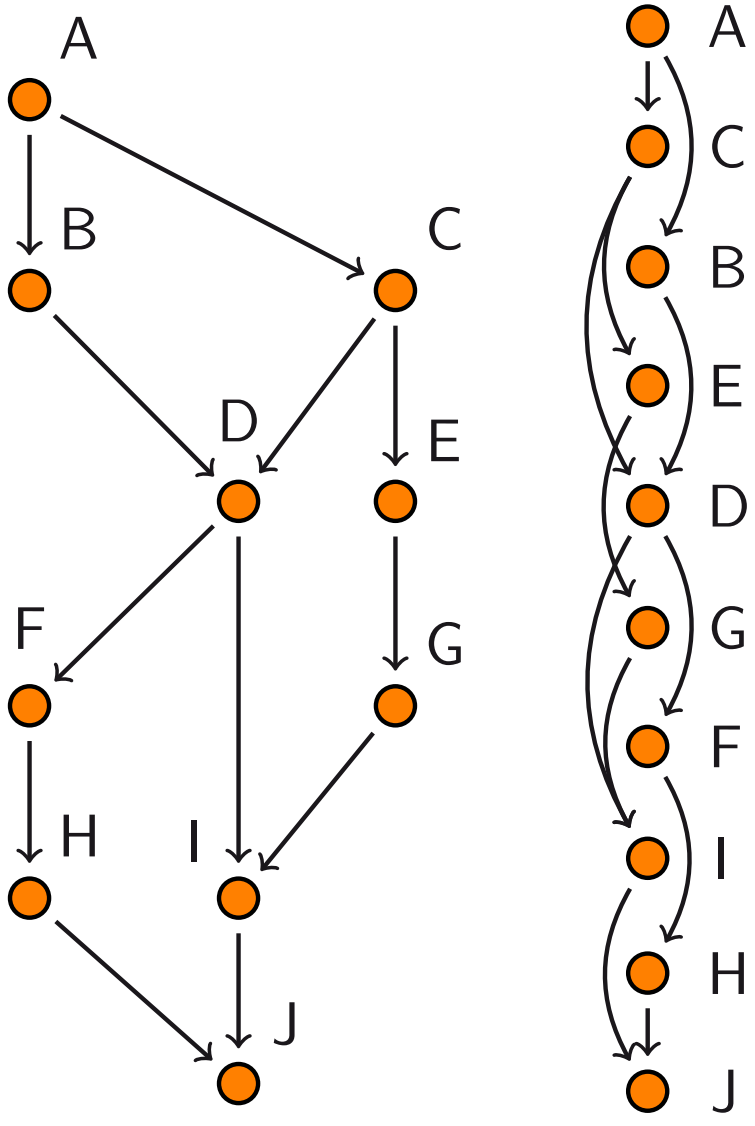
## Purpose

You are looking for a commit that caused something, e.g.

- Introduced a bug
- Deleted / added something
- Anything really

## Rough Idea

1. Take commit graph $G = (V, A)$ we want to find $\bar{v} \in V$ that did above
2. Topologically sort $G$, i.e. add order $\succ^*$ to $V$
3. Logarithmic search $\bar{v}$ in $G$



# Git Bisect Practice

You want to find the commit that did X. Initialization:

1. `git bisect start`
2. `git bisect bad` (current commit is bad, no X)
3. `git bisect good 258dbc1` (commit 258dbc1… was good, has X)

Git will checkout (go back in time to) a commit between the good one and the bad one and you have to say

- `git bisect good`
- `git bisect bad`
- `git bisect skip` (cannot test this commit for X)

Process repeats a few time ($\approx$ log of # of commits between good and bad). If you have a script e.g. `check.py` that returns 0 for good, 125 for skip, any other number for bad, it can be automated

- `git bisect run check.py`

# That's (most of) it

## Learn More

Git and its ecosystem have many more features

- Stash, Rebase, Blame, ...
- Submodules and subtrees
- LFS (Large File Storage) for big (gigabytes) files
- Email "old school" workflow (e.g. `sr.ht` and Linux Kernel)
- Integration with CI (e.g. GitHub Actions, GitLab Workers)