Version control

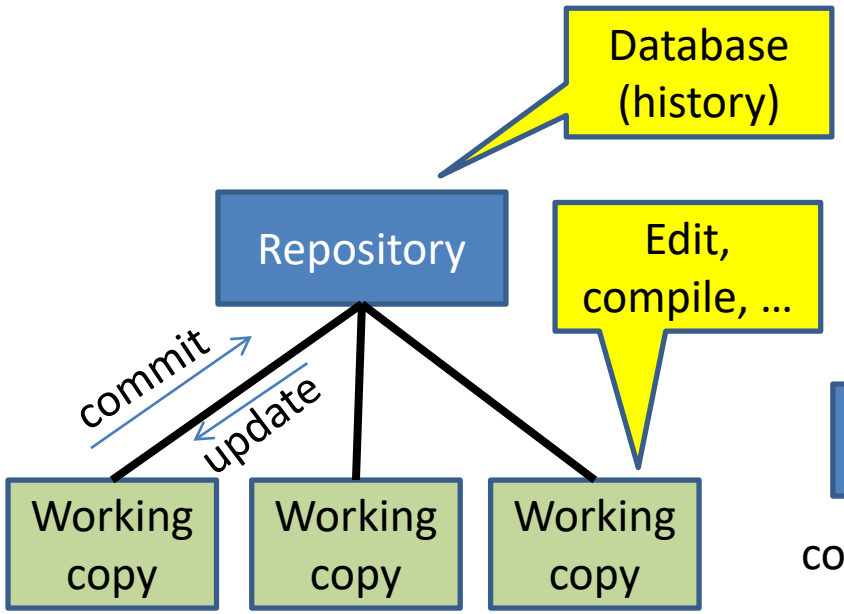
CSE 403

Goals of a version control system

- Keep a history of your work
  - Explain the purpose of each change
  - Checkpoint specific versions (known good state)
  - Recover specific state (fix bugs, test old versions)
- Coordinate/merge work between team members
  - Or yourself, on multiple computers or multiple features

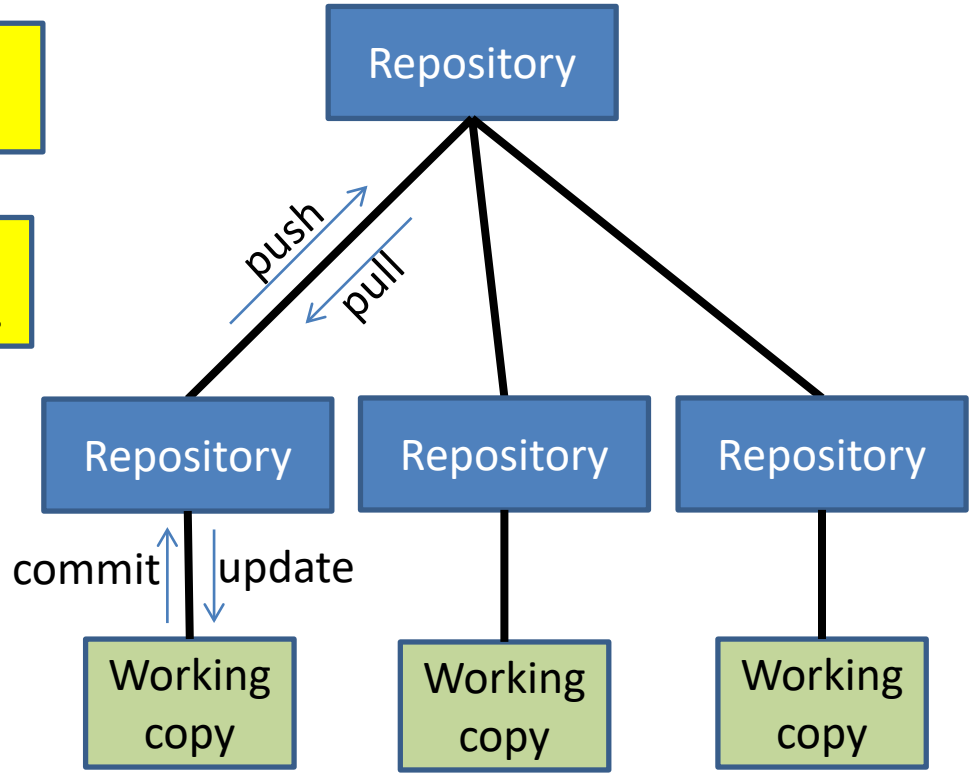
Varieties of version control system

Centralized VCS



- One repository
- Many working copies

Distributed VCS

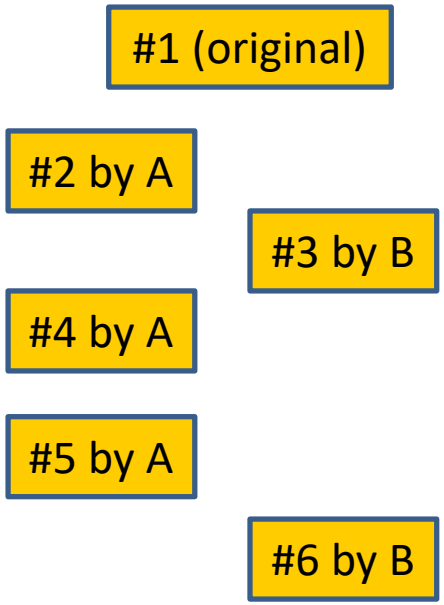


- Many repositories
- One working copy per repository

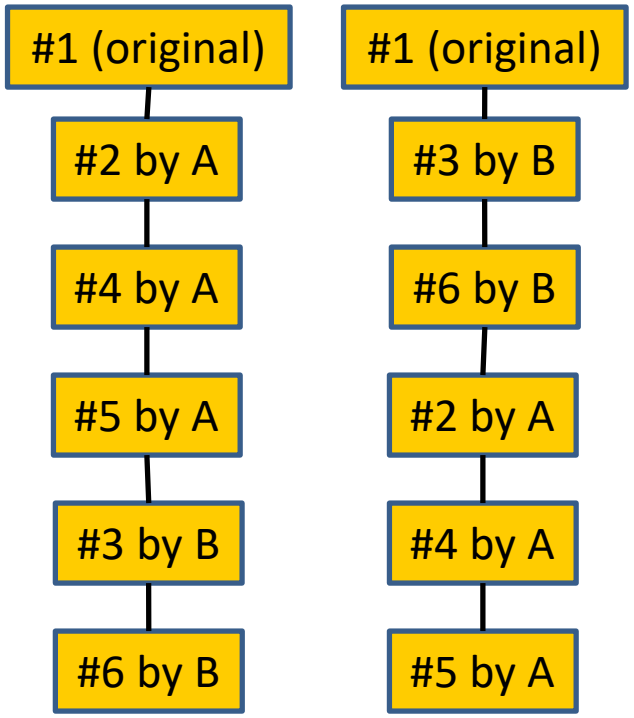
(More complicated topologies are possible)

Version control history

Reality

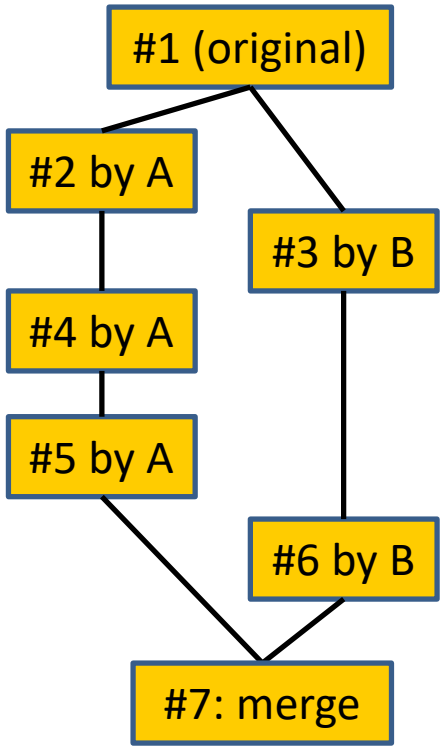


Centralized VCS (one of these)



- Rewrites history
- Multiple visible commits per dev.

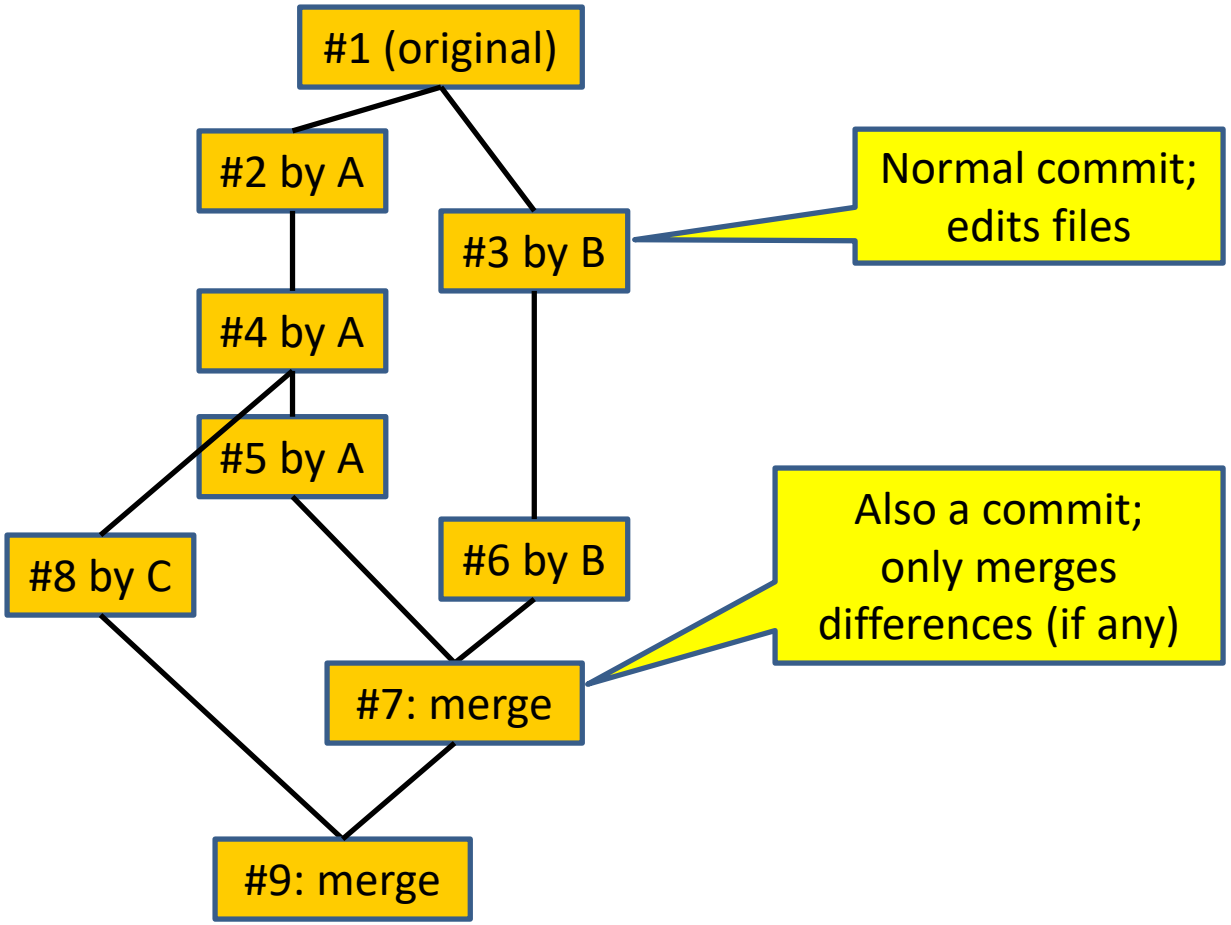
Distributed VCS



- Preserves history
- Multiple commits, one visible push per dev.

Distributed VCS history

Working copy can be `updated` to any revision in the history



Advantages of a distributed VCS

- checkpoint work without publishing to teammates
- commit, examine history when not connected to the network
- more accurate history
- more effective merging algorithms

Less important in CSE 403:

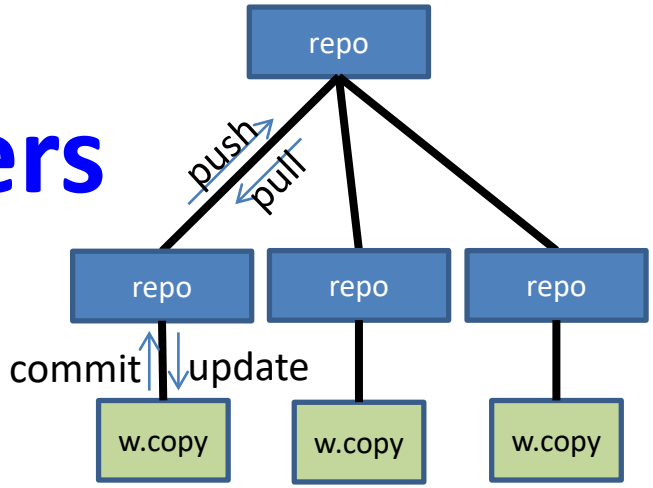
- share changes selectively with teammates
- flexibility in repository organization and workflow
- faster performance

A DVCS prohibits* some operations

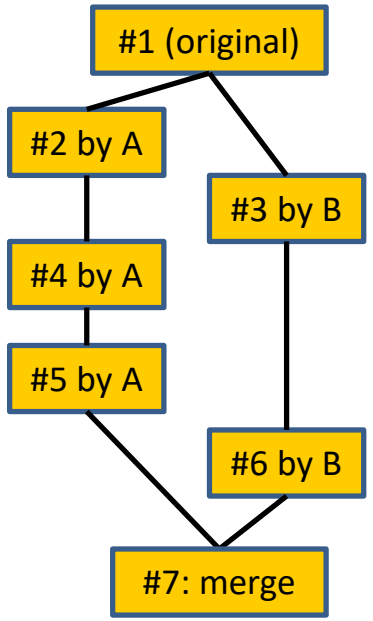
- No update if uncommitted changes exist
  - must commit first
- No push if not ahead of remote
  - must pull & merge first
- No partial update (e.g., updating just one directory)
  - update gets all changes in a changeset (= a commit)

- Rationale:
  - Maintain more accurate, complete history
  - Keep all users in sync
  - Avoid painful conflicts
  - Avoid loss of work

Coordinating with others



- **`pull`** incorporates others' changes into your repository
  - (**`update`** brings changes into your working copy)
  - (N.b.: **`git pull`** does pull, merge, *and* update)
- If you are behind, nothing more to do
  - Behind = your history is a prefix of master history
- If you have made changes in parallel, you must merge
  - Merge = create a new version incorporating all changes

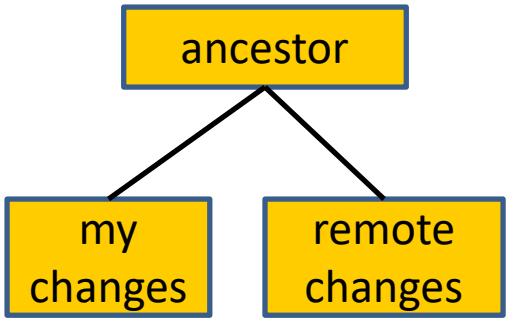


Do two changes conflict?

- Conflict-free
  - Changes are to different lines of a file
- Conflicting
  - Simultaneous changes to the same lines of a file
  - Requires manual conflict resolution

- "Conflict-free" is a textual, not semantic, notion
  - A *heuristic* about when to get the user involved
  - Could yield compile errors or test failures
- Git records changes at line granularity
  - Darcs can record word substitution (for code refactoring)
  - Git diff algorithm is customizable

Resolving conflicts



- There are three versions of the file:
- You decide which version to keep or how to merge them
- Many merge tools exist
- Configure your DVCS to use the merge tool that you prefer
  - Practice this ahead of time!

- Don't panic! Instead, think.
- You can always bail out of the merge and start over
  - You have the full local and remote history)

Popular DVCSes

- Git (`git`)
- Others: Mercurial (`hg`), Bazaar, Darcs, …

- Git is integrated with the GitHub hosting site and other tools
- Otherwise, similar functionality
- Git has an idiosyncratic command set

Hints

- Don’t forget to update after you pull
  - `git pull` does pull, merge, and update
    - Not symmetric with `git push`, but usually does what you want
- To use DVCS just like Subversion:

```
svn update = git pull
svn commit = git commit; git push
```

Binary files are not diffable

- The history database records changes, not the entire file every time you commit
  - The diff algorithm works line-by-line
- Avoid binary files (especially simultaneous editing)
  - Word .doc files
- Do not commit generated files
  - Binaries (e.g., .class files), etc.
  - Wastes space in repository
  - Causes merge conflicts

Synchronize with teammates often

- Pull often
  - Avoid getting behind the master or your teammates
- Push as often as practical
  - Don’t destabilize the master build
  - Use continuous integration (automatic testing on each push)

Commit often

- Make many small commits, not one big one
- Easier to understand, review, merge, revert
- How to make many small commits:
  - Do only one task at a time
    - commit after each one
  - Do multiple tasks in one clone
    - Commit only a subset of files (use Git's staging area)
    - Error-prone
  - Create a new clone for each simultaneous task
    - Can have as many as you like
  - Create a branch for each simultaneous task
    - Somewhat more efficient
    - Somewhat more complicated and error-prone
    - Easier to share unfinished work with teammates

More ways to avoid merge conflicts

- Modularize your work
  - Divide work so that individuals or subteams “own” a module
  - Other team members only need to understand its specification
  - Requires good documentation and testing
- Communicate about changes that may conflict
  - Don’t overwhelm the team with such messages