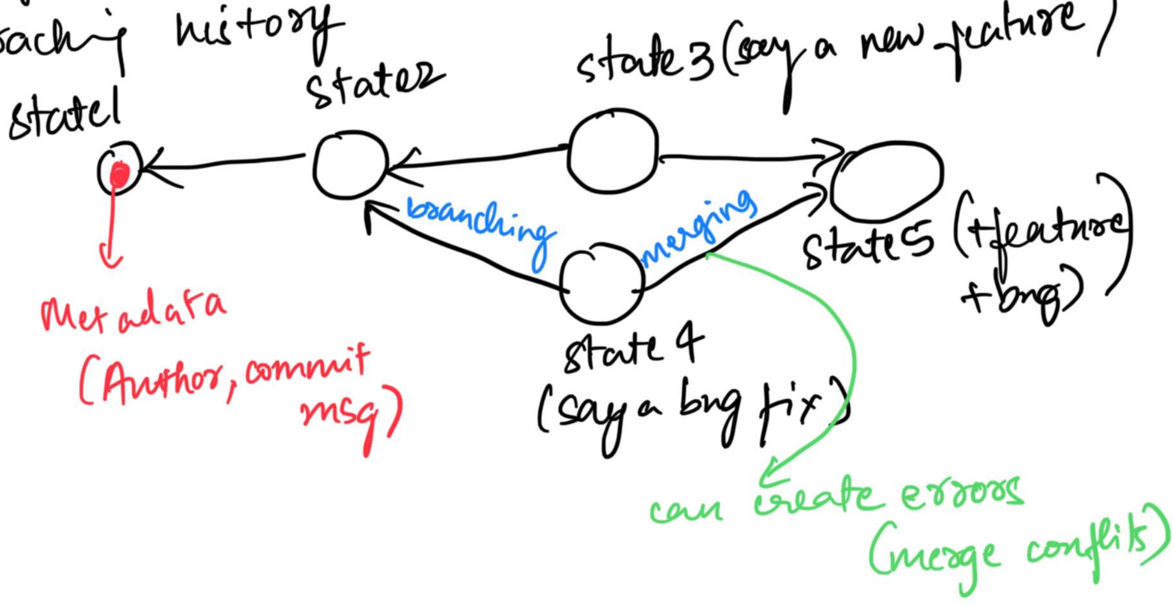
All about version control system

- Missing semester (MIT)

folders → "trees" }
files → "blobs" } terminology.

Tracking history



type blob = array<byte>
type tree = map<string, tree | blob>
type commit = struct { (some contents)
type parents = array<commit) (what preceded a commit)

message
author
Snapshot : tree

Objects = map<string, object>

blob tree commit

objects are "content" addressed.

key is hash

---

let us say we have an 'object store' to store all the objects.

To store :

```
def store(o):
    id = sha1(o)
    objects[id] = o
```

hash

To retrive :

```
def load(id):
    return objects[id]
```

This is git's "On disk data store".

- references : map<string, string>

both are human readable.

<hash, human readable name>

These references are 'mutable'

COMMANDS

① git init initialise empty git rep

② git help command

③ git status Info on the branch, commits

④ git add ./filename/dir name etc

Adds files to staging area

⑤ git commit

Creates a snapshot (author, timestamp, etc)

hash(returned by git commit)

⑥ git log --all --graph --decorate --oneline

⑦ git cat-file -p <hash> this is the hash you get after git commit.

Gives the details of the snapshot

<hash> is the hash of the commit.

Basically you can get a hash of anything, a number, a file (beauty of hash ♥), snapshot (in case of commit)

tree + other information

This tree furthur contains hashes of blobs

A useful version of git log command:

"git log --all --graph --decorate"

Gives a nice graphical version of git log.

⑧ git checkout <branch/commit> remember it's the hash

Move around version history/previous state

** This changes the references only**. So when we use the 'git log' after checkout we can still see the same history.

Be carefull with 'checkout' command though since you could be overwritting the changes (if not committed yet!)

⑨ git diff <file/commit/branch> #diff btn

eg: HEAD readme.md
<hash> readme.md

BRANCHING & MERGING

⑩ git branch <name> #list the branches in the repo

git branch -vv # verbose
git branch new-br # new-br points TO HEAD

git checkout -b <new-branch>

⑪ git merge <branch> checkout to master/previous state before.

Merge conflict **

-when git can't automatically figure out conflicts btn parallel branches while merging them use git mergetool

Git Remote Add remote repo for collaboration!

⑫ git remote

lists all the remotes for the current repo.

(name)
git remote add <origin><url>

tell the local repo that

Commands to interact with this repo:

① git push <remote> <branch>

② git clone <url> <folder name>

(remote repo name
③ git branch --set-upstream-to=<name>/<branch>

(advanced)

→ set up to which branch (in the remote repo to refer/track)

eg: So you can push without additional args like git push <origin> <master>|

④ git fetch #retrieve changes from remote to local repo

⑤ git pull
→ git fetch
→ git merge
→ retrieve and merge to local repo.

⑬ git config #you can add some personal info.
#can configure any way (eg. add documentation)

⑭ git clone <--shallow> #clone from a remote repo

Commands that are handy but not sure if I'll ever use!

⑮ git blame <file> # who edited what !!

(15) git

git show <commit> #info on the commit.

git stash #save the work but revert to the previous stage.

git stash pop # undo the stash.

git bisect (advanced)

#basically to debug.

(16) git ignore (1:22:00 onwards)

- ignore some set of files
- useful to specify patterns of files you want to ignore.

×