Introducing Git version control into your team

Introduction

Short history of git

- Distributed version control system
- Mature, actively maintained open source project
- Originally developed in 2005 by Linus Torvalds

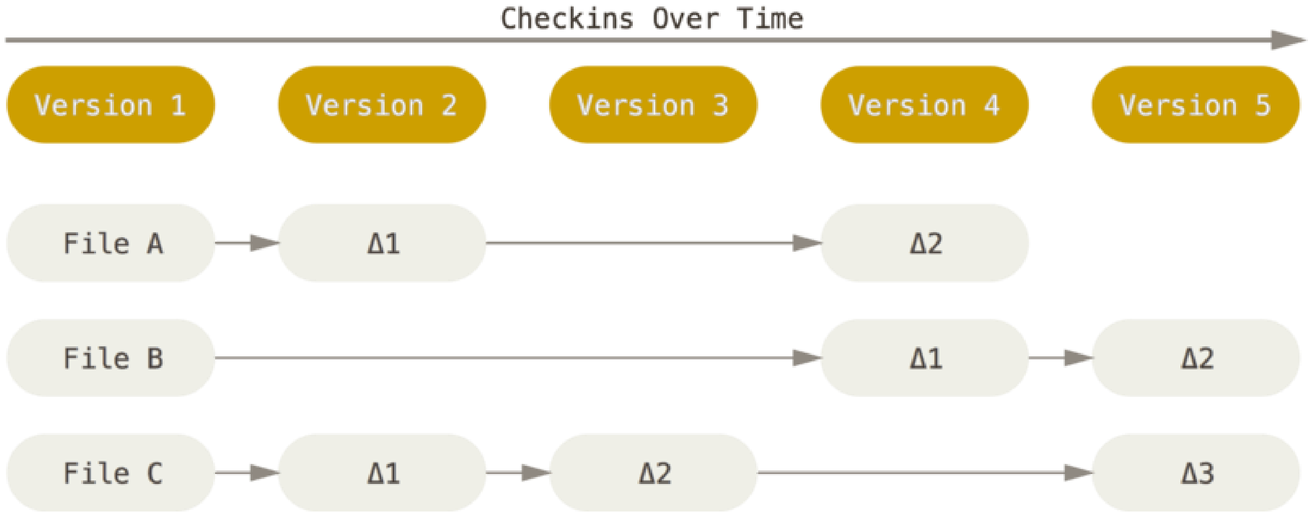
Core concepts

- Speed
- Simple design
- Strong support for non-linear development (thousands of parallel branches)
- Fully distributed
- Able to handle large projects like Linux kernel efficiently (speed and data size)

Git vs. SVN

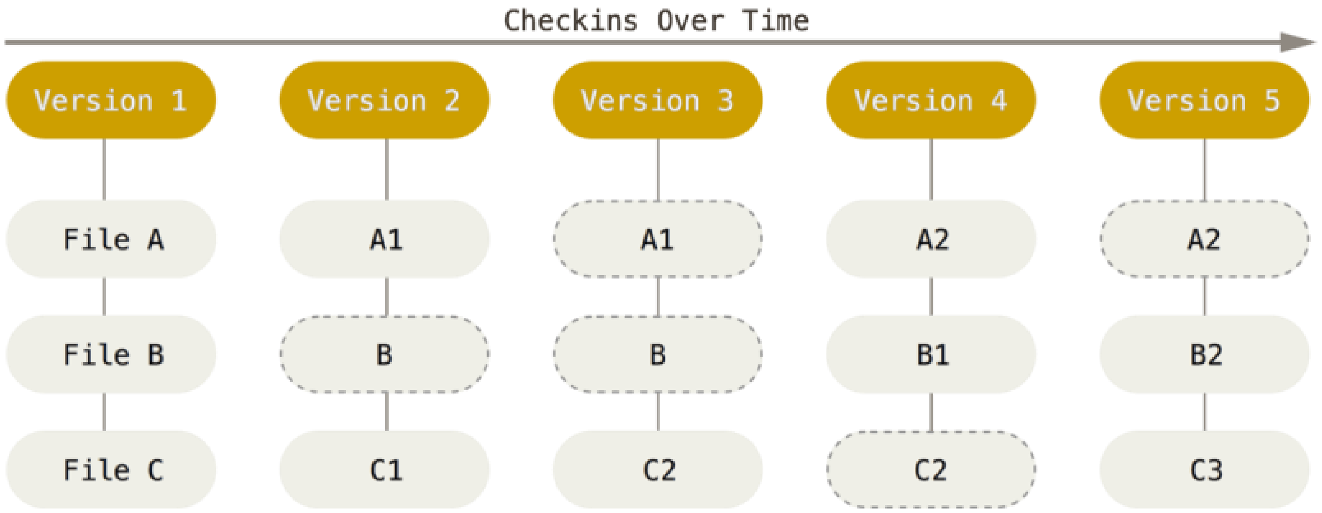
Snapshots vs. differences

- SVN stores information as a list of file-based changes



Snapshots vs. differences

- Git thinks of its data like a set of snapshots of a miniature filesystem



Work with single repo

Initialize repository

Initialize repository

```
$ mkdir /repo
$ cd /repo

$ git init
```

First commit

Checking status of your files

```
$ git status
```

→

- No tracked and modified files
- No untracked files
- You are on branch master

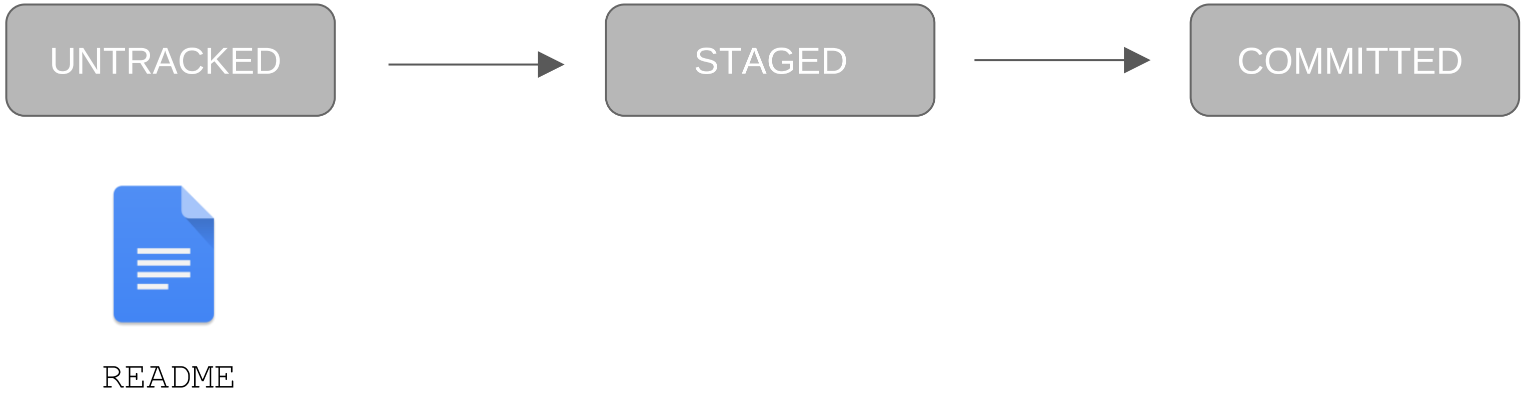
Checking status of your files

```
$ echo 'My Project' > README

$ git status
```

→

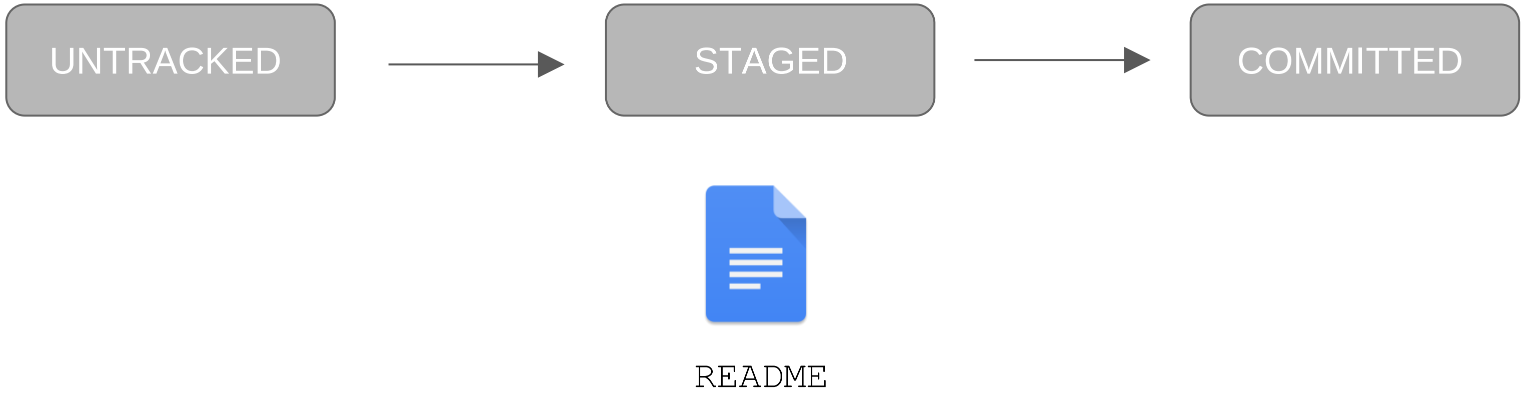
- No tracked and modified files
- Untracked files: README
- You are on branch master

Add files and commit



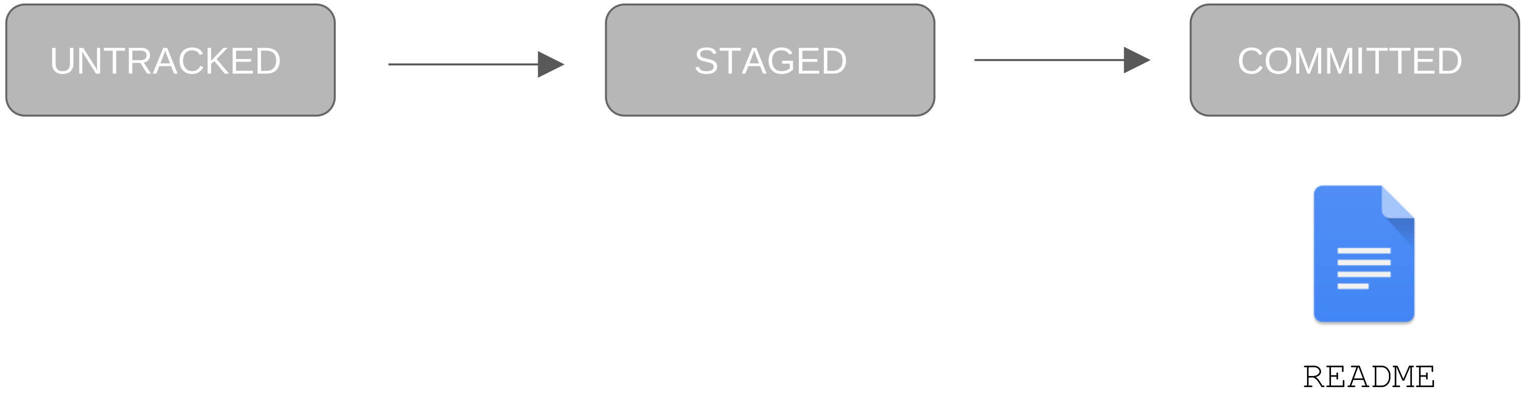
Add files and commit

```
$ git add README
```



Add files and commit

```
$ git commit -m “Initial commit”
```



Commands

Command	Description
`git status`	Print the current state of the project
`git add <file>`	Track file OR stage changes (add to next commit)
`git commit`	Create commit and save it in local repo (.git folder)
`git log`	Show the history of commits
`git diff`	Show diff between working version and staged version

Summary

- Git stores history as snapshots called **commits**
- To commit a new file you have to first track it with `git add` command
- Ones the file is changed, `git add` it again to include into next commit
- `git log` shows the history of your project

Ignoring files and folders

.gitignore

- Everything listed in .gitignore file will be ignored by git
- .gitignore is a simple text file that lists files and folders one per line:
    - useless_folder
    - *.log
- .gitignore supports patterns: *.log - means all files ending with .log
- It’s a good idea to add .gitignore itself to version control

Configuration

Git configuration

Your Identity

```
$ git config --global user.name “Yoda”

$ git config --global user.email “yoda@gmail.com”
```

Your Editor

```
$ git config --global core.editor emacs
```

Checking Your Settings

```
$ git config --list
```

Git configuration

- Get and set configuration variables that control all aspects of how Git looks and operates.
- These variables can be stored in three different places:
  - **/etc/gitconfig file** - contains values for every user on the system and all their repositories (--system)
  - **~/.gitconfig or ~/.config/git/config file** - specific to your user (--global)
  - **config file in the Git directory** - specific to that single repository

Debugging

File annotation

- Find a buggy method in your code
- Annotate the file with **`git blame`**
- See when each line of the method was last edited and by whom

```
$ git blame README
```

Viewing commit history

Commit history

- **`git log`** is the most basic and powerful tool to view commit history
- By default, with no arguments, git log lists the commits made in that repository in reverse chronological order

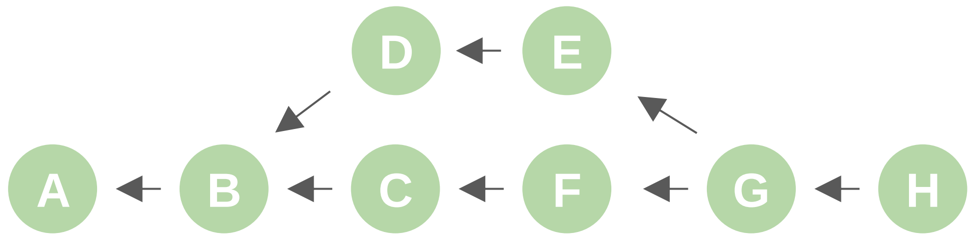
```
$ git log
```

Flags for `git log`

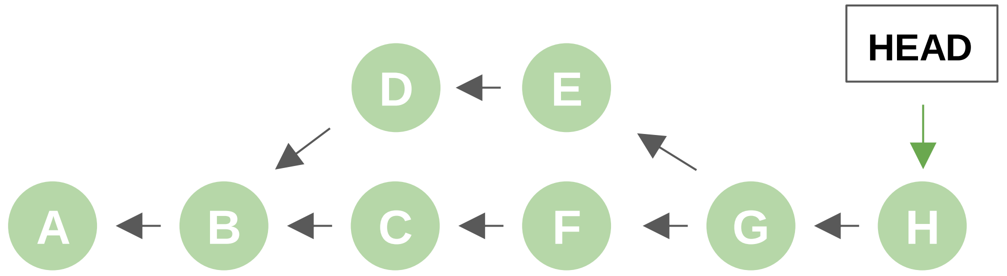
Flag	Description
`--graph`	Show log as graph
`--abbrev-commit`	Show only first few symbols of SHA-1 string
`--date=relative`	Switch date to readable relative format
`--pretty=oneline`	Compact one-line format
`--decorate`	Show branch names
`-5`	Show only 5 commits

How git stores history

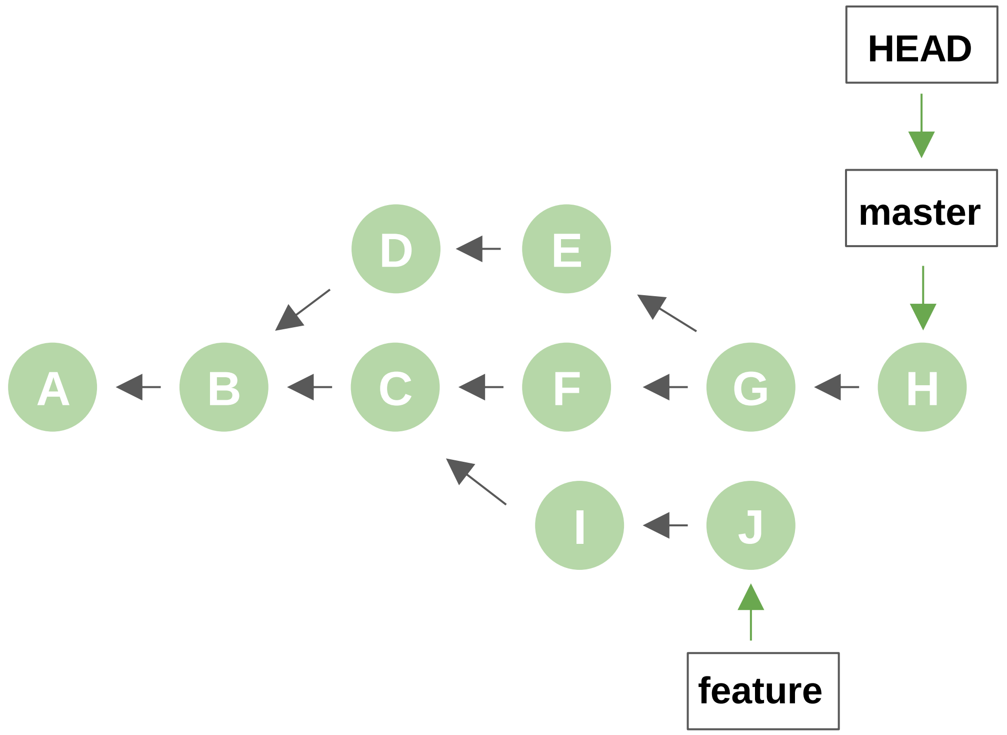
Git stores its commits as **DAG** (Directed Acyclic Graph)



DAG has **references** or **labels**



References can point to other references - that's how **branches** work



Summary

- Git stores history as DAG - Directed Acyclic Graph
- Nodes of the graph are commits
- Commits are immutable
- All you do in git is move around graph and add new nodes

What is commit

Commits never change

commit a31fd4…

Author: Yoda <yoda@gmail.com>

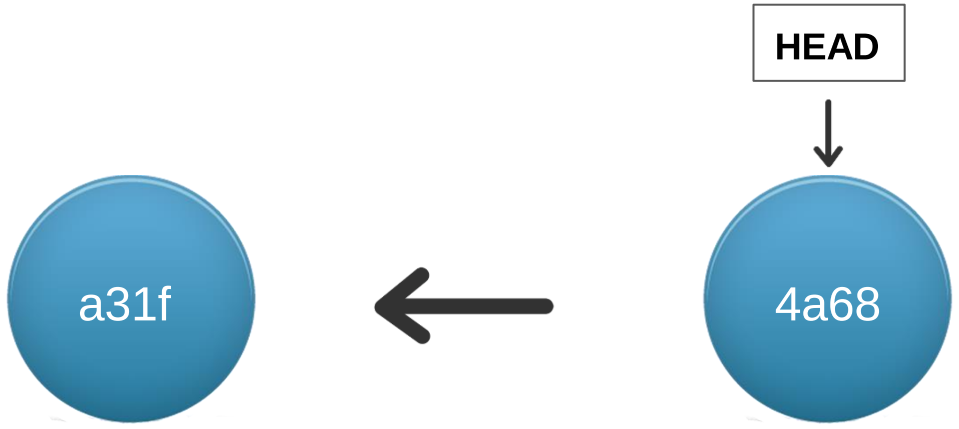
Date: Mon Oct 31 ...

commit a31fd4…

Author: Yoda <yoda@gmail.com>

Date: Mon Oct 31 ...

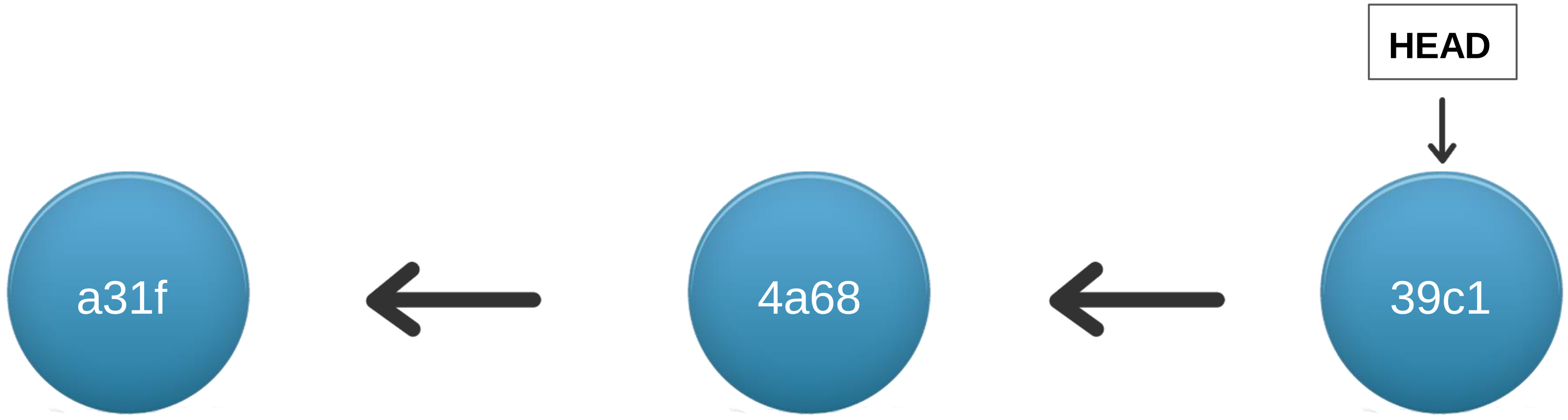
ID = content + author + date + log + previous commit



commit a31fd4…

Author: Yoda <yoda@gmail.com>

Date: Mon Oct 31 ...



commit a31fd4…

Author: Yoda <yoda@gmail.com>

Date: Mon Oct 31 ...

Commands

Command	Description
`git log`	Show the history of commits
`git show <commit>`	Show details of a commit

Summary

- Git stores history as snapshots called **commits**
- Each commit stores the **whole** state of a project at a point of time
- Commits are identified by their SHA-1 hash
- Commits know about their **parent** commits
- HEAD is a pointer that refers to your current commit

What is index

Working dir

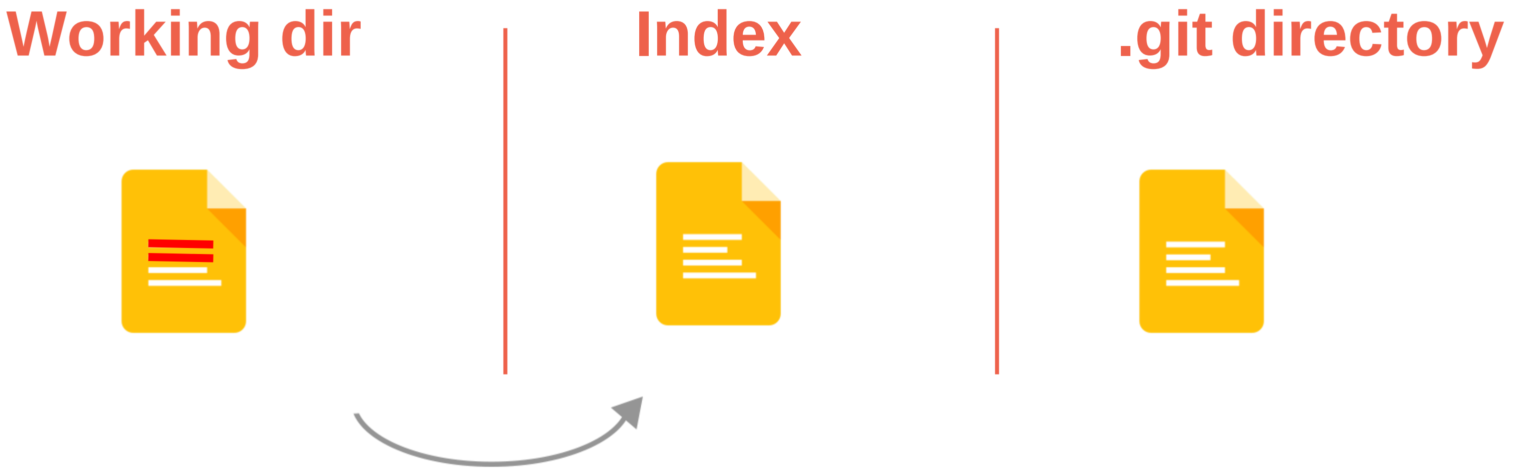
These files are placed on disk for you to use or modify

Index

The staging area (index) is a file that stores information about what will go into your next commit

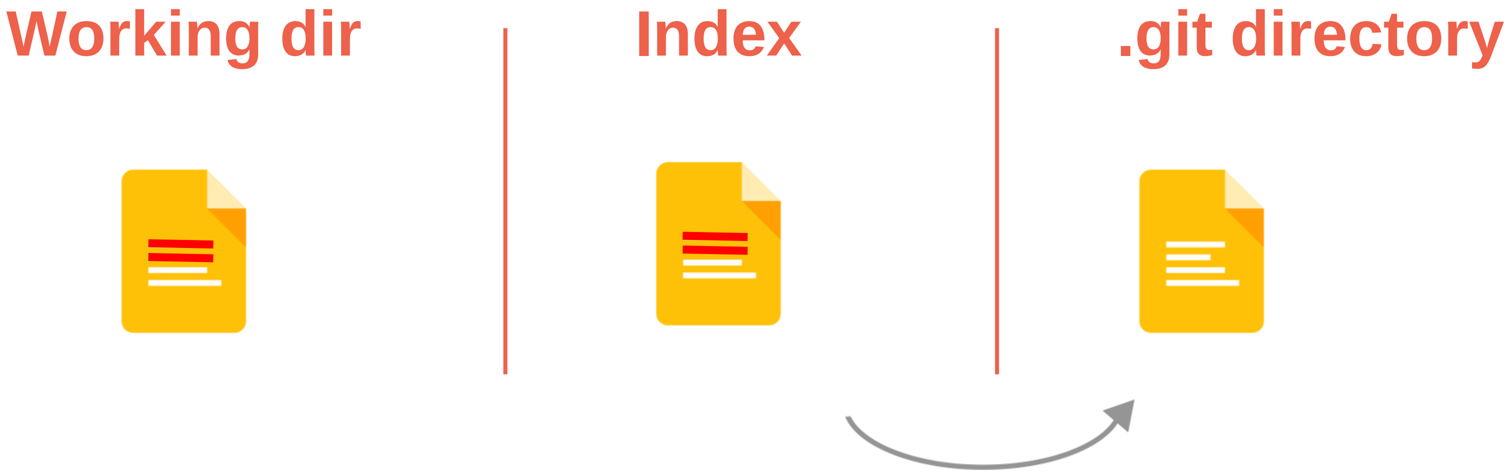
.git directory

The Git directory is where Git stores the metadata and object database for your project



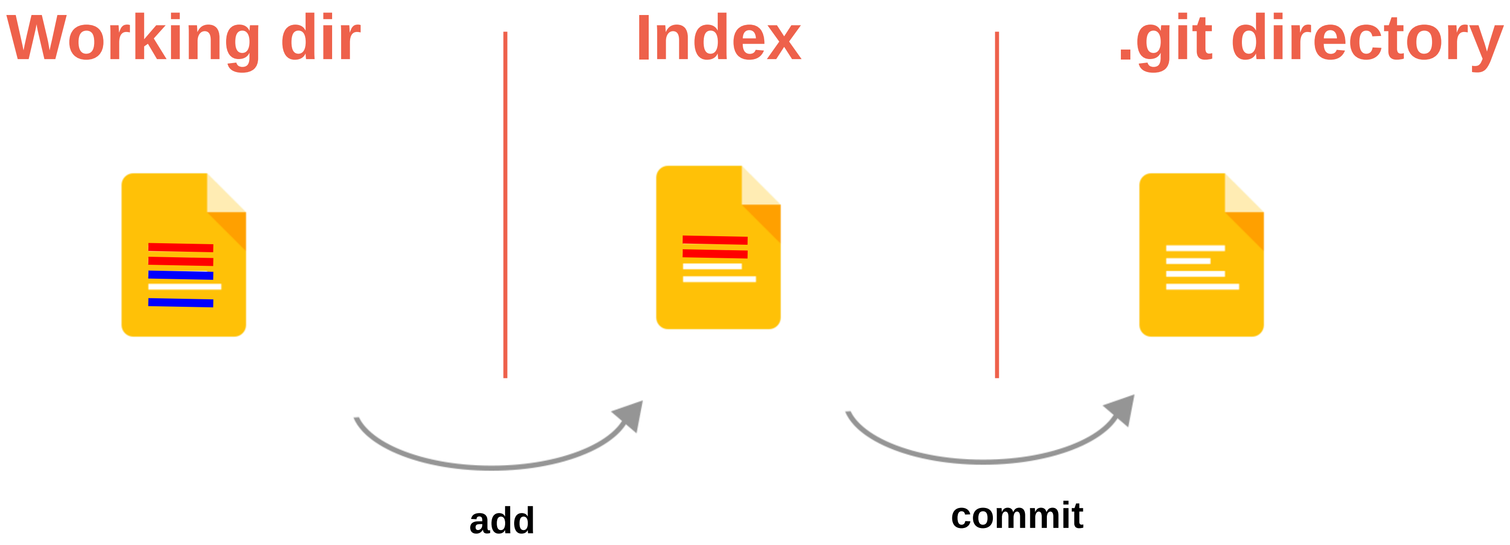
Edited file - status is **red**
(will not go to next commit)

```
To add to index:
$ git add <file>

To restore indexed
version:
$ git checkout -- <file>
```

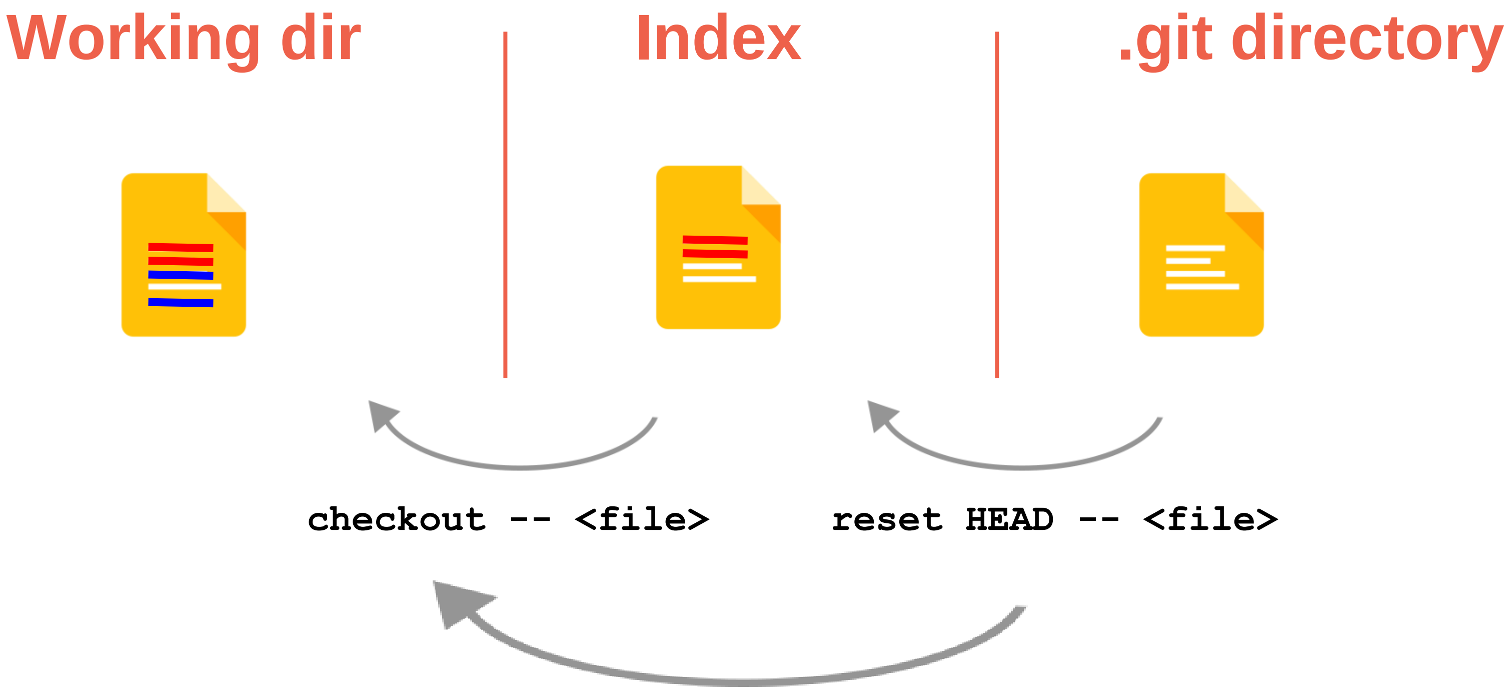


File is added to index - status is **green** (will be committed)

```
To commit:
$ git commit

To revert version in
index:
$ git reset HEAD -- <file>
```



File is edited again -
status is both green and **red**



```
checkout -- <file>
```

```
reset HEAD -- <file>
```

```
checkout HEAD -- <file>
(will also update index)
```

Working dir	Index	
A		`$ echo "A" >> myfile.txt`
A	A	`$ git add myfile.txt`
A B	A	`$ echo "B" >> myfile.txt`
A	A	`$ git checkout -- myfile.txt`
A	A	`$ git commit -m “Commit A”`
A C	A	`$ echo "C" >> myfile.txt`
A C	A C	`$ git add myfile.txt`
A C	A	`$ git reset HEAD -- myfile.txt`

Deleting files

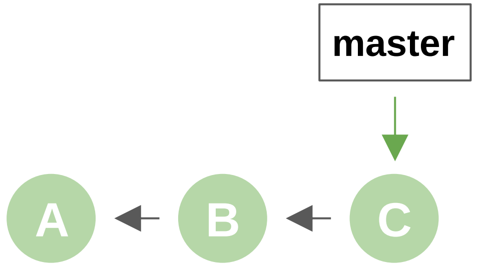
Commands

Command	Description
`git rm <file>`	Delete files from FS and from index
`git rm -r <folder>`	Delete from FS and index recursively
`git rm --cached`	Delete only from index, leave FS intact
`git commit -am “msg”`	Commit all tracked files with message

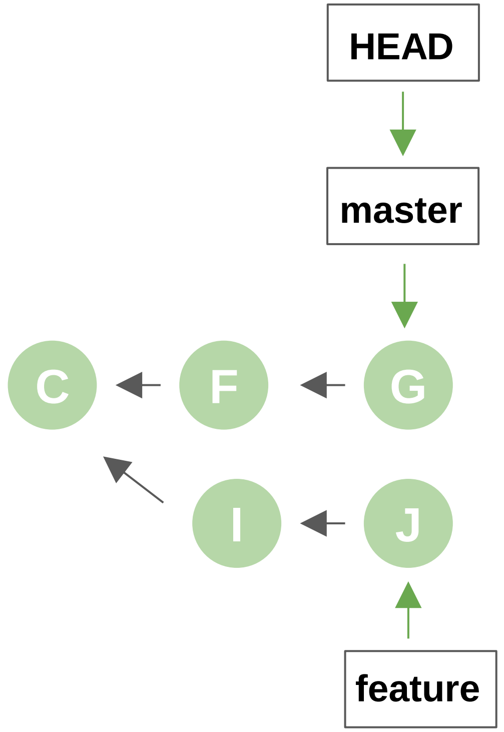
Work with local branches

What is a branch

Branch is a **pointer** to commit

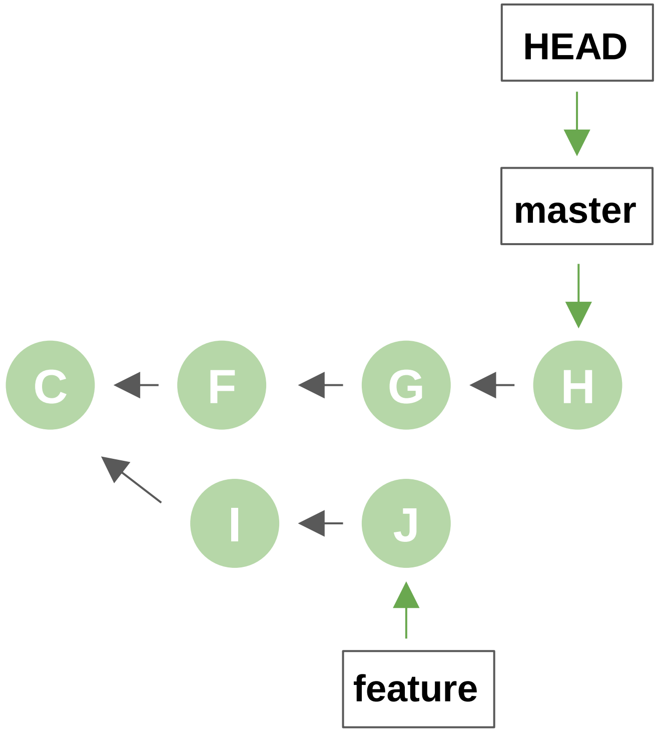


There can be many pointers in DAG

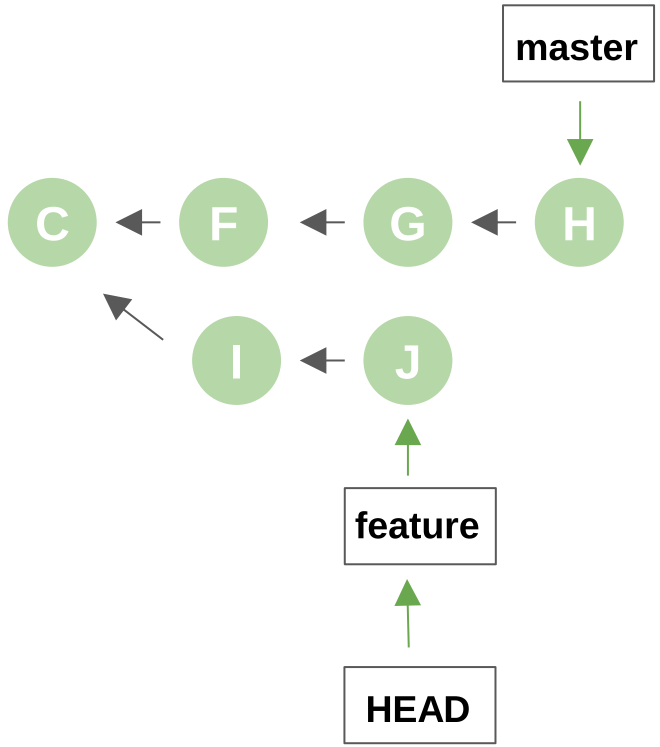


HEAD points to current branch

Adding commit **moves pointers**

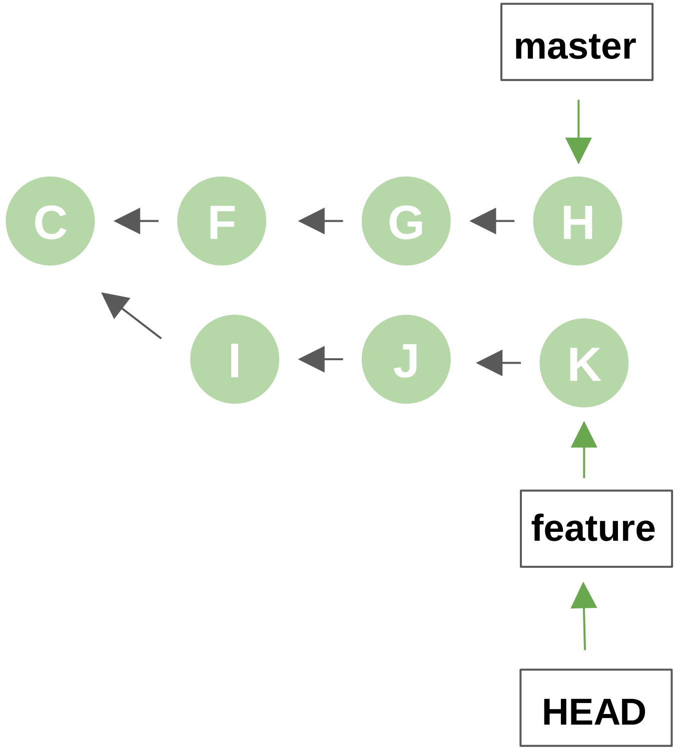


You can switch branches with **`checkout`**

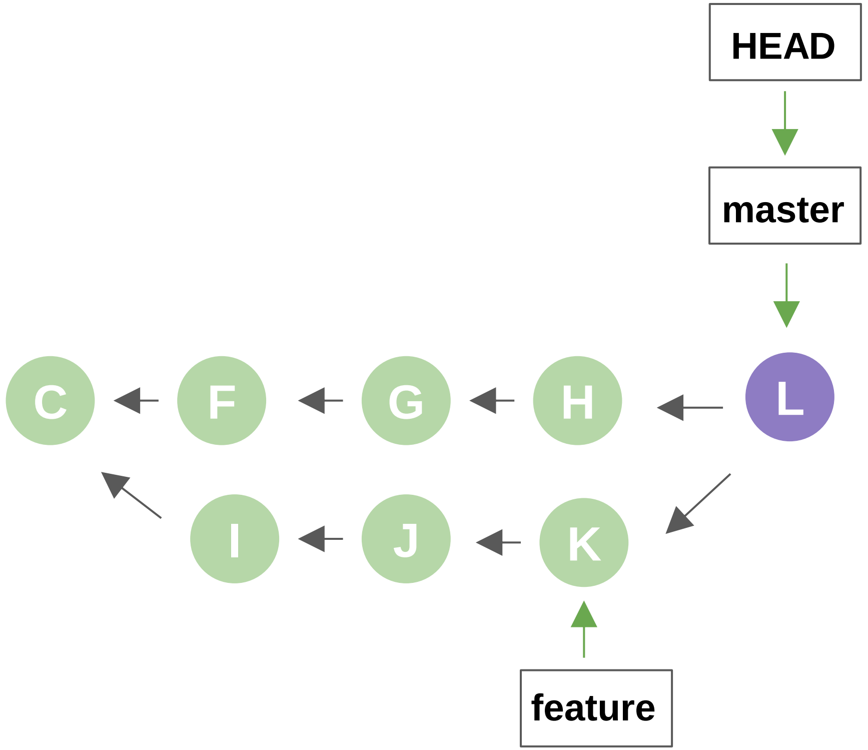


```
$ git checkout feature
```

New commits will be added to current branch

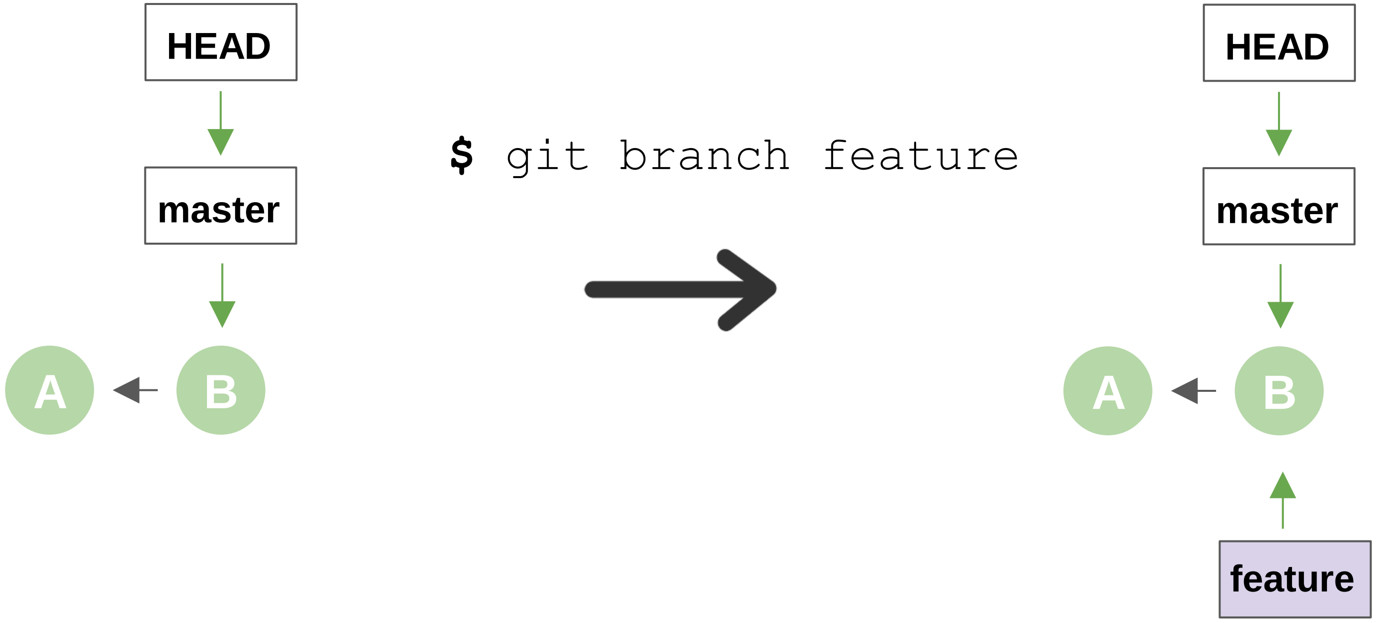


Once the work is done - merge it



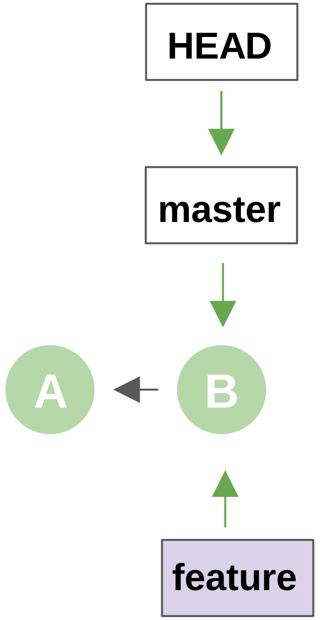
Creating branch

STEP 1: Create a new pointer to commit

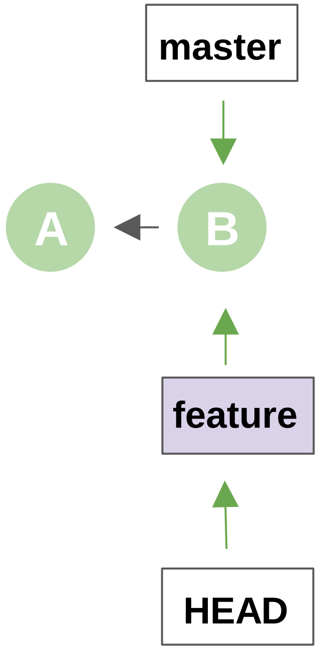


```
$ git branch feature
```

STEP 2: Checkout to created branch

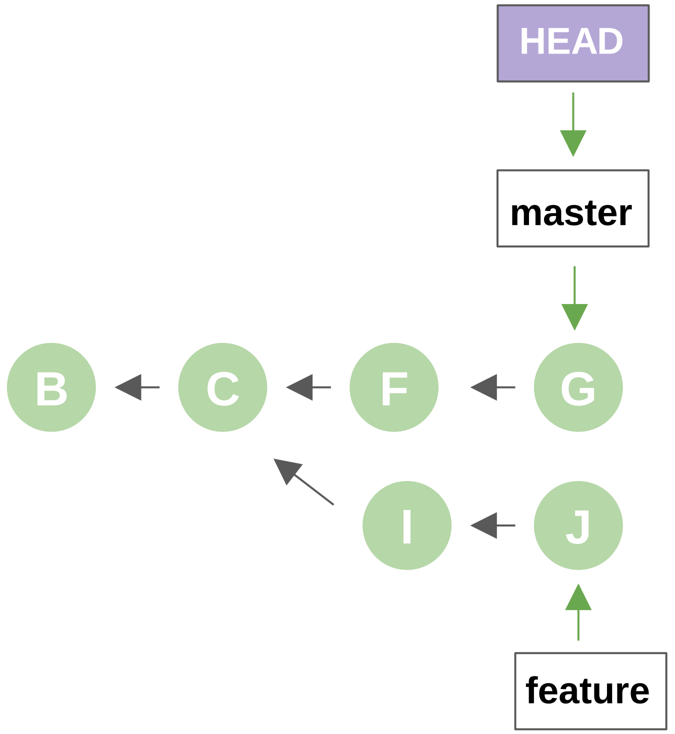


```
$ git checkout feature
```



Merging

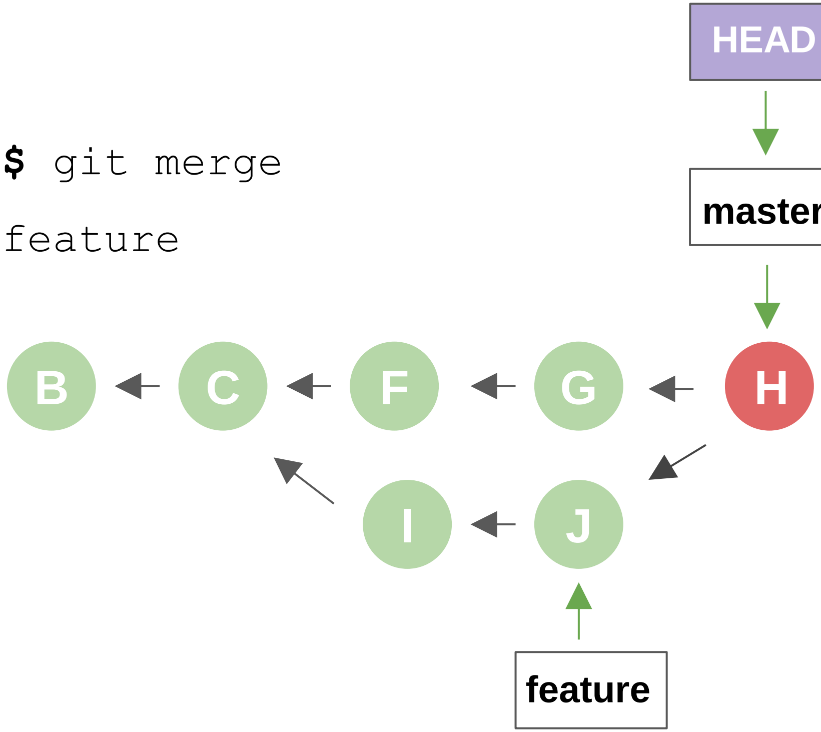
Merge



- Your work in feature branch is completed and ready to be merged into master branch.
- Run **`$ git merge feature`** to do that.

Merge (no fast-forward)

```
$ git merge feature
```

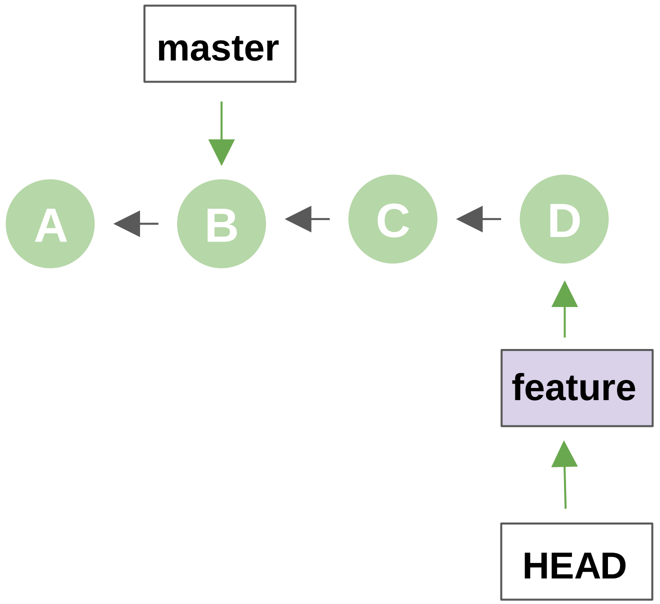


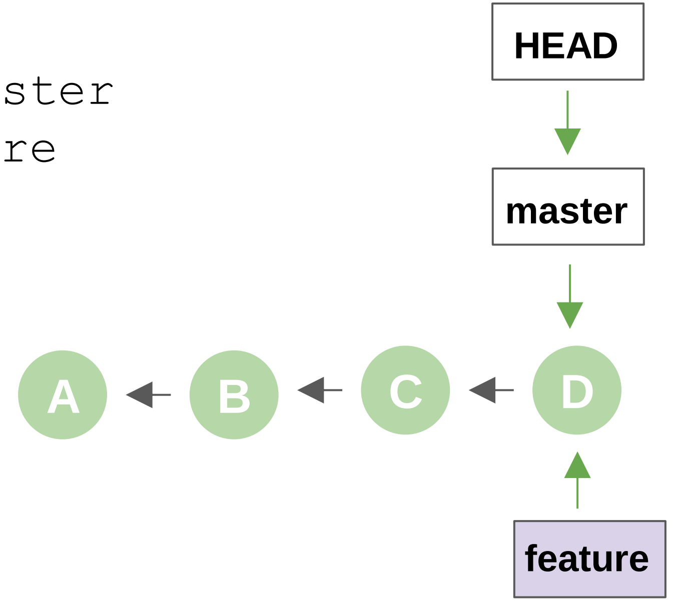
- Git creates a new snapshot that results from this merge and automatically creates a new commit that points to it (**H**).
- This is referred to as a merge commit (**H**), and is special in that it has more than one parent

Fast forwarding

Fast Forward is a merge

```
$ git checkout master
$ git merge feature
```





When all you have to do to merge is just move a pointer - it’s fast-forward

No fast forward merge

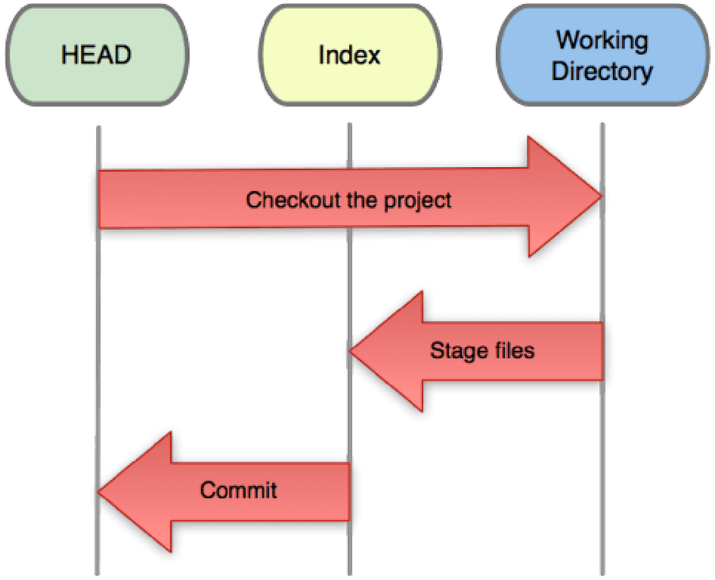
```
    // starting from master branch
$ git checkout -b feature
    // working in feature branch...
$ git commit -am "Feature added"
$ git checkout master
    // working in master branch...
$ git commit -am "Master changed"
$ git merge feature
```

Fast forward merge

```
    // starting from master branch
$ git checkout -b feature
    // working in feature branch...
$ git commit -am "Feature added"
$ git checkout master
$ git merge feature
```

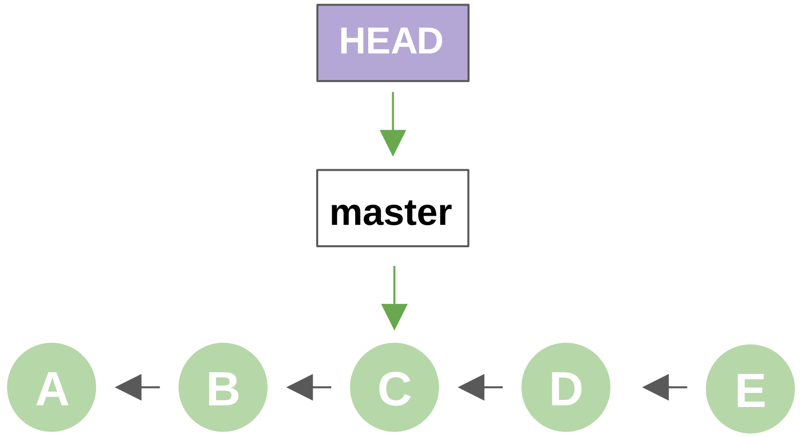
Git work process

Git work process



Reset

Reset



Starting from master on E

Reset to commit C:

```
$ git reset [sha_of_C]
```

or

```
$ git reset HEAD~2
```

Reset does 3 different operations:

1) Move whatever branch HEAD points to (stop if `--soft`)

2) THEN, make the Index look like that (stop here unless `--hard`)

3) THEN, make the Working Directory look like that

Practice. Lab 1

Parallel work

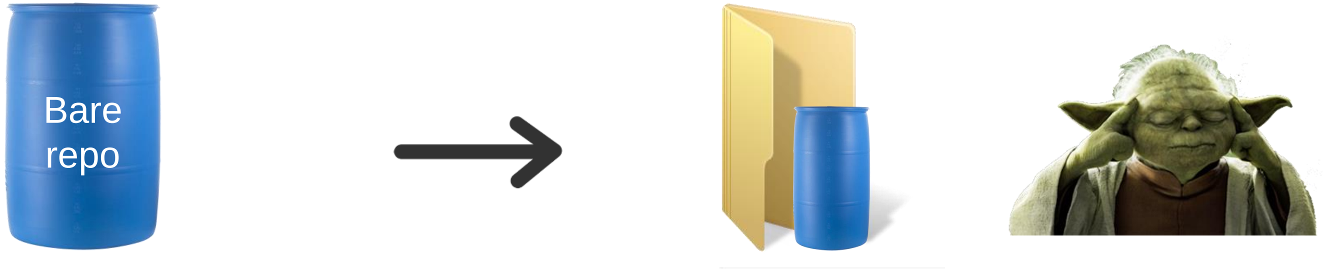
Creating remote repo

Create bare (shared) repository

```
$ mkdir /repos
$ cd /repos

$ git init --bare
origin.git
```

Create Yoda's working copy

```
$ mkdir ~/yoda
$ cd ~/yoda

$ git clone /repos/origin.git .
```



/repos/origin.git

Yoda's workspace

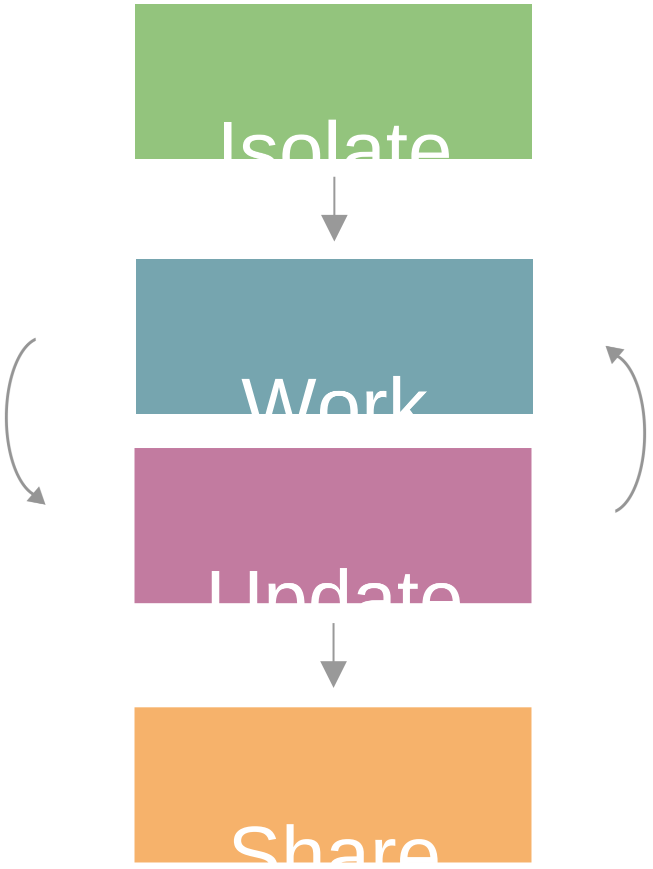
Commands

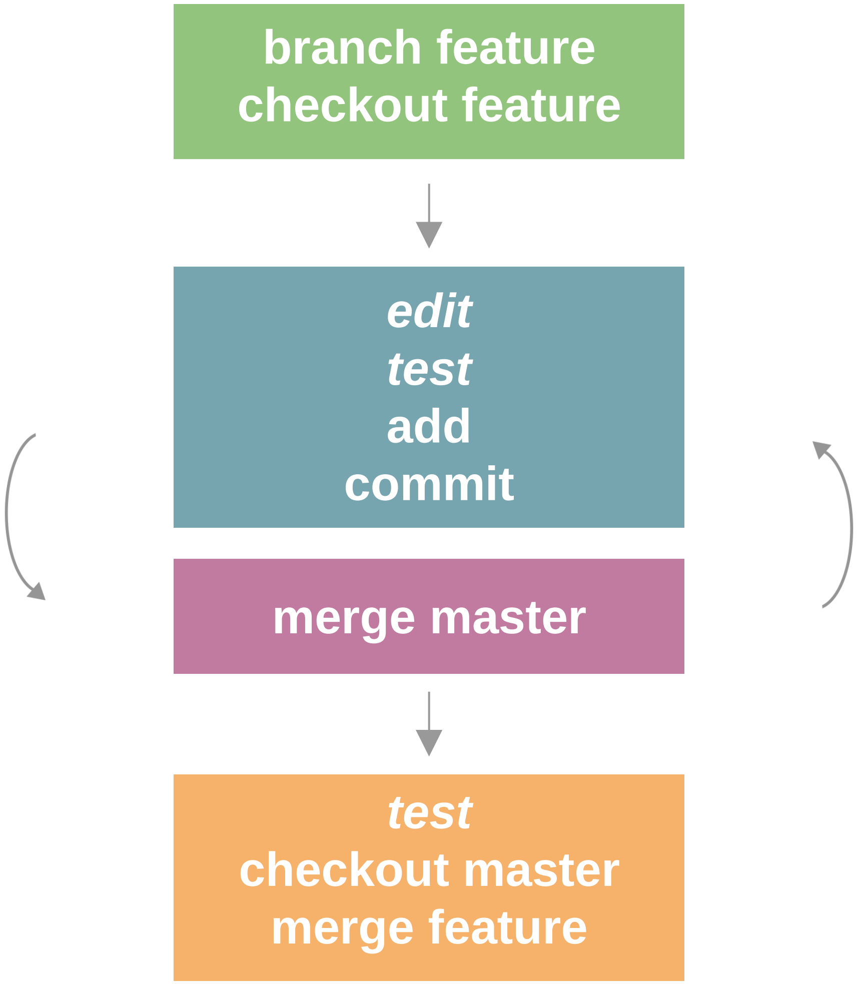
Command	Description
`git init --bare <name>`	Create a bare repository (used only for sharing)
`git clone <path>`	Create a working copy of repository

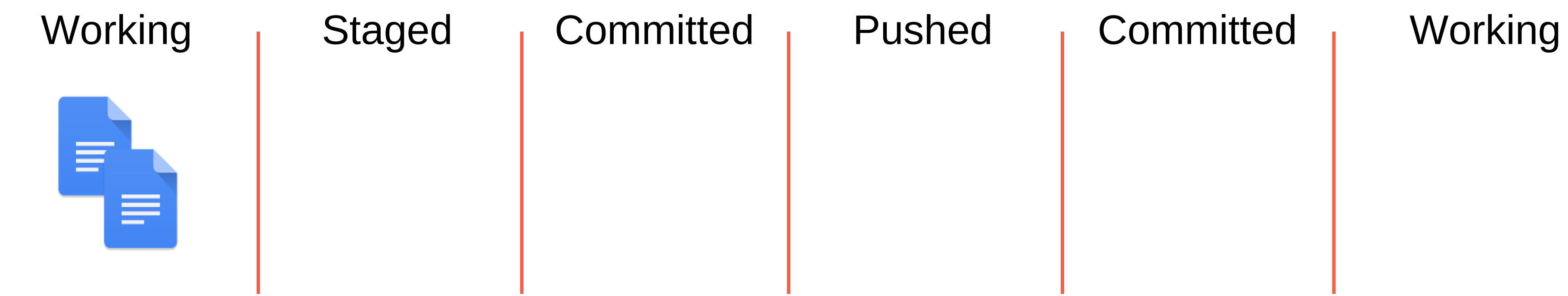
Summary

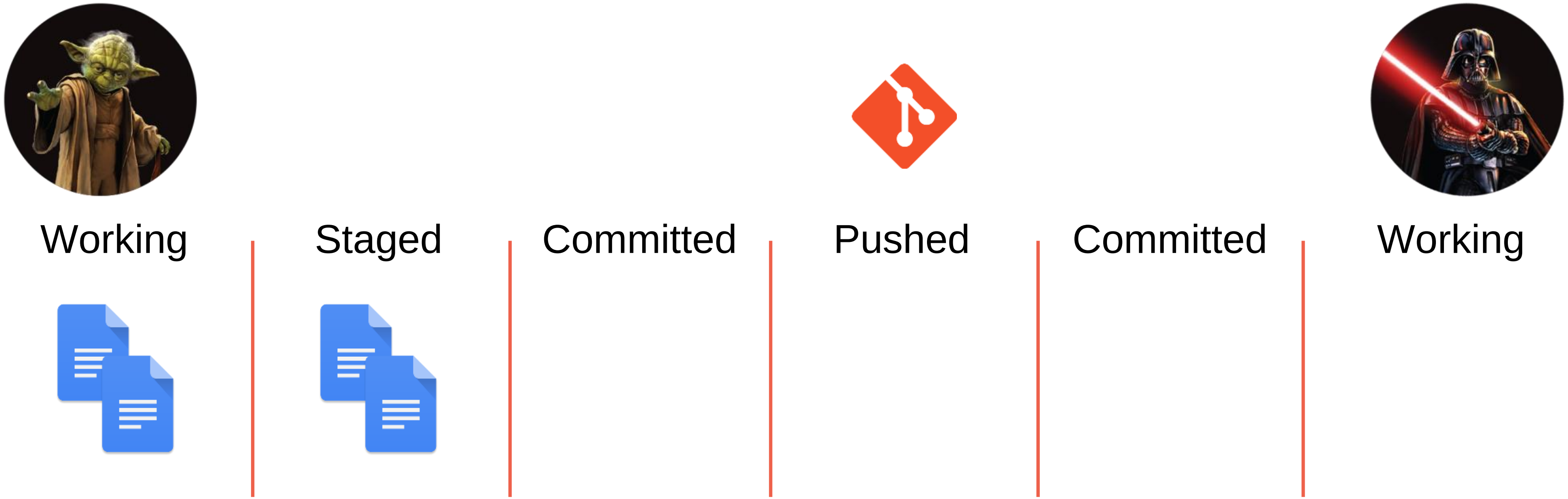
- Bare repositories are used to share code
- Shared repositories should **always** be bare
- Non-bare or **working** repositories are the ones that you develop in
- To start work you **clone** bare repository

Git parallel work process

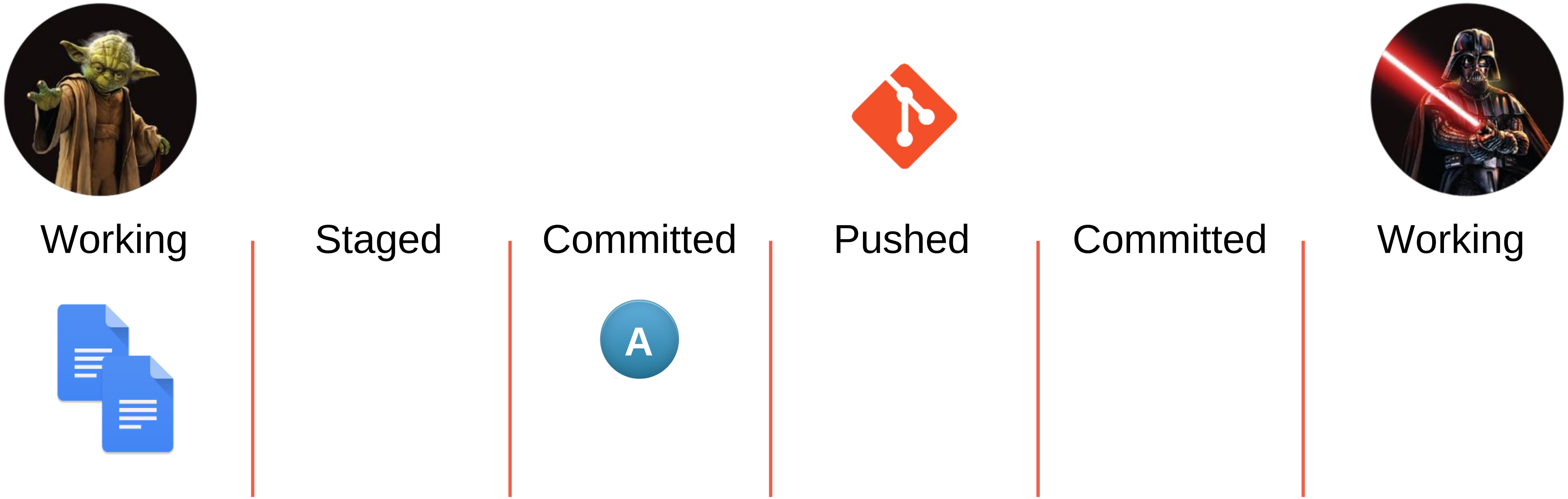

Isolate
Work
Update
Share


branch feature
checkout feature
edit
test
add
commit
merge master
test
checkout master
merge feature

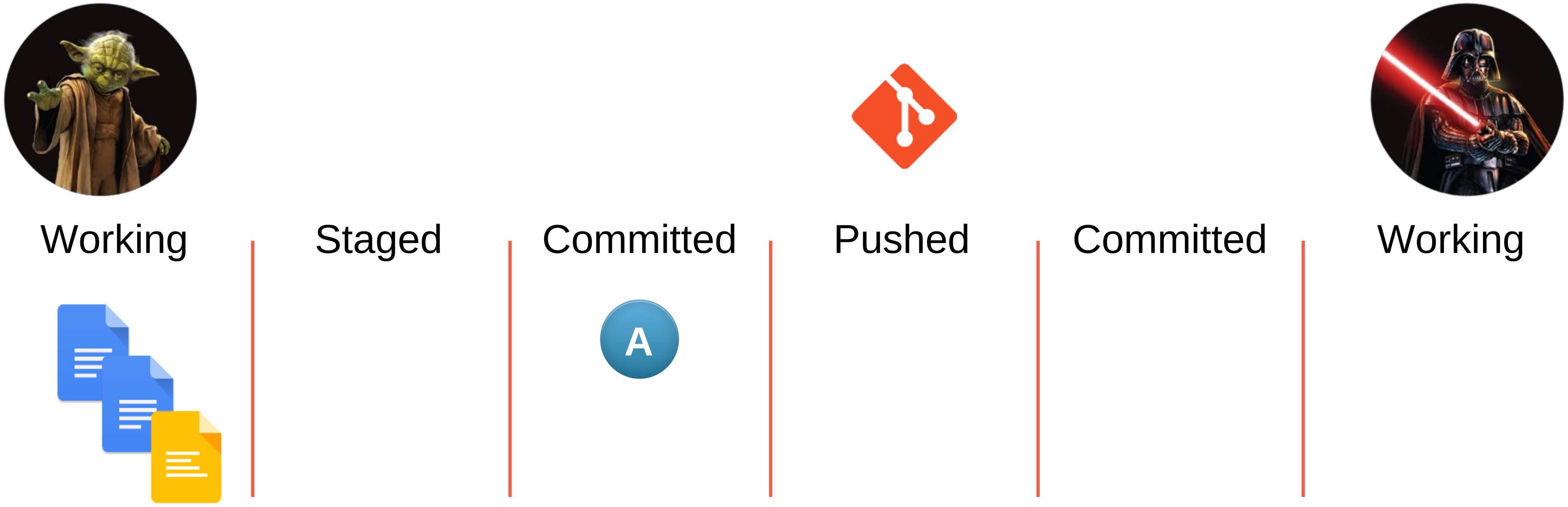

Working
Staged
Committed
Pushed
Committed
Working



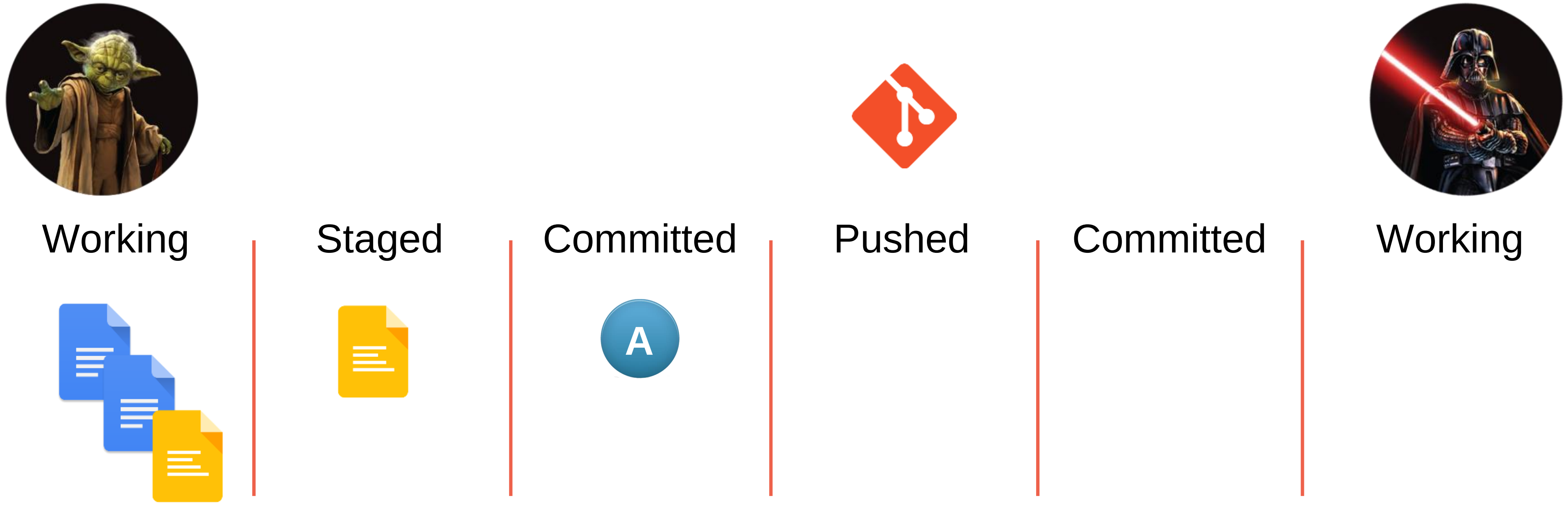
```
$ git add <file>
```



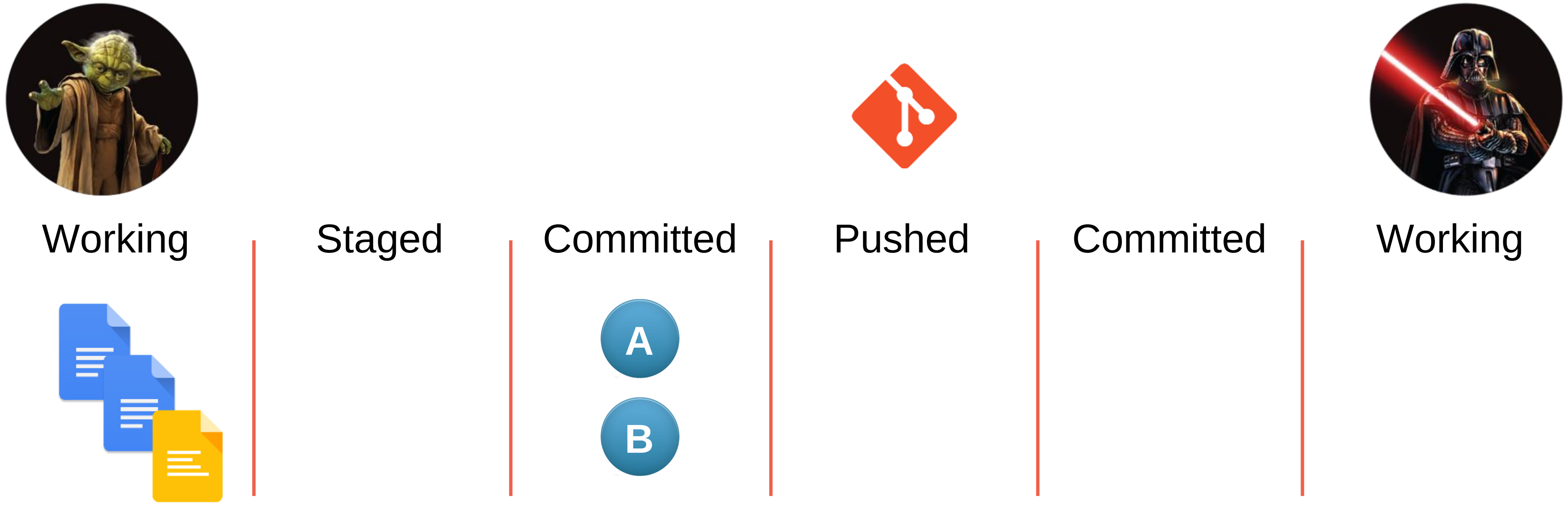
```
$ git add <file>
$ git commit -m "Initial commit"
```



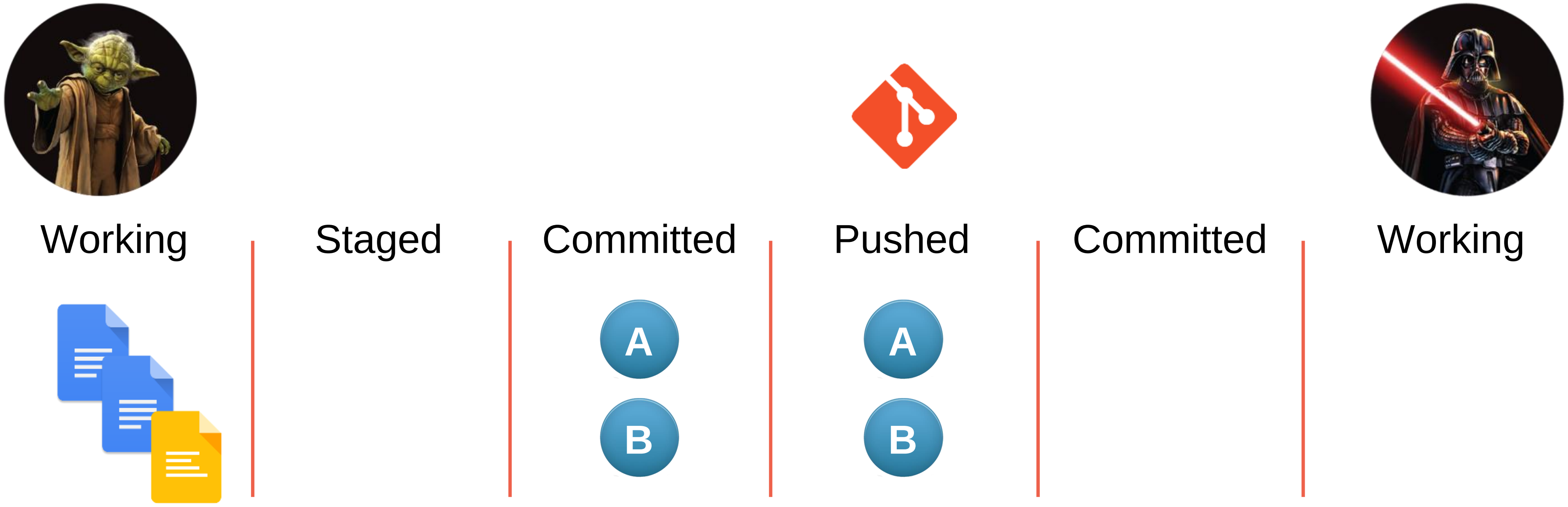
```
$ git add <file>
$ git commit -m "Initial commit"
```



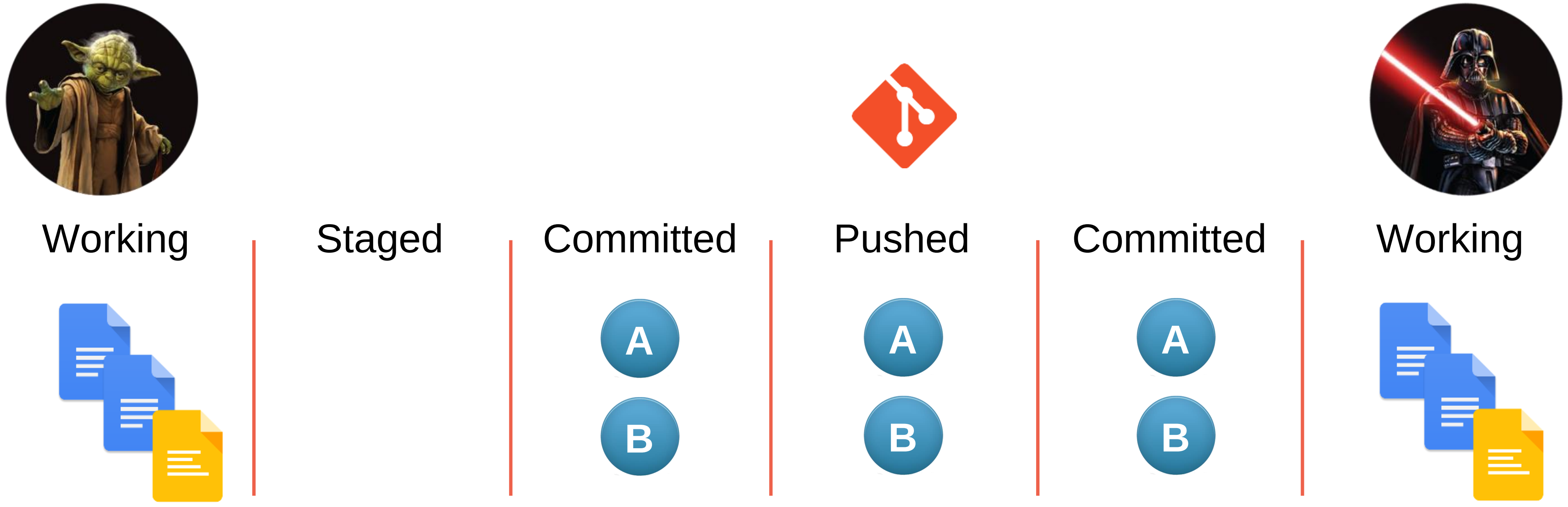
```
$ git add <file>
$ git commit -m "Initial commit"
$ git add <file>
```



```
$ git add <file>
$ git commit -m "Initial commit"
$ git add <file>
$ git commit -m "Added new file"
```



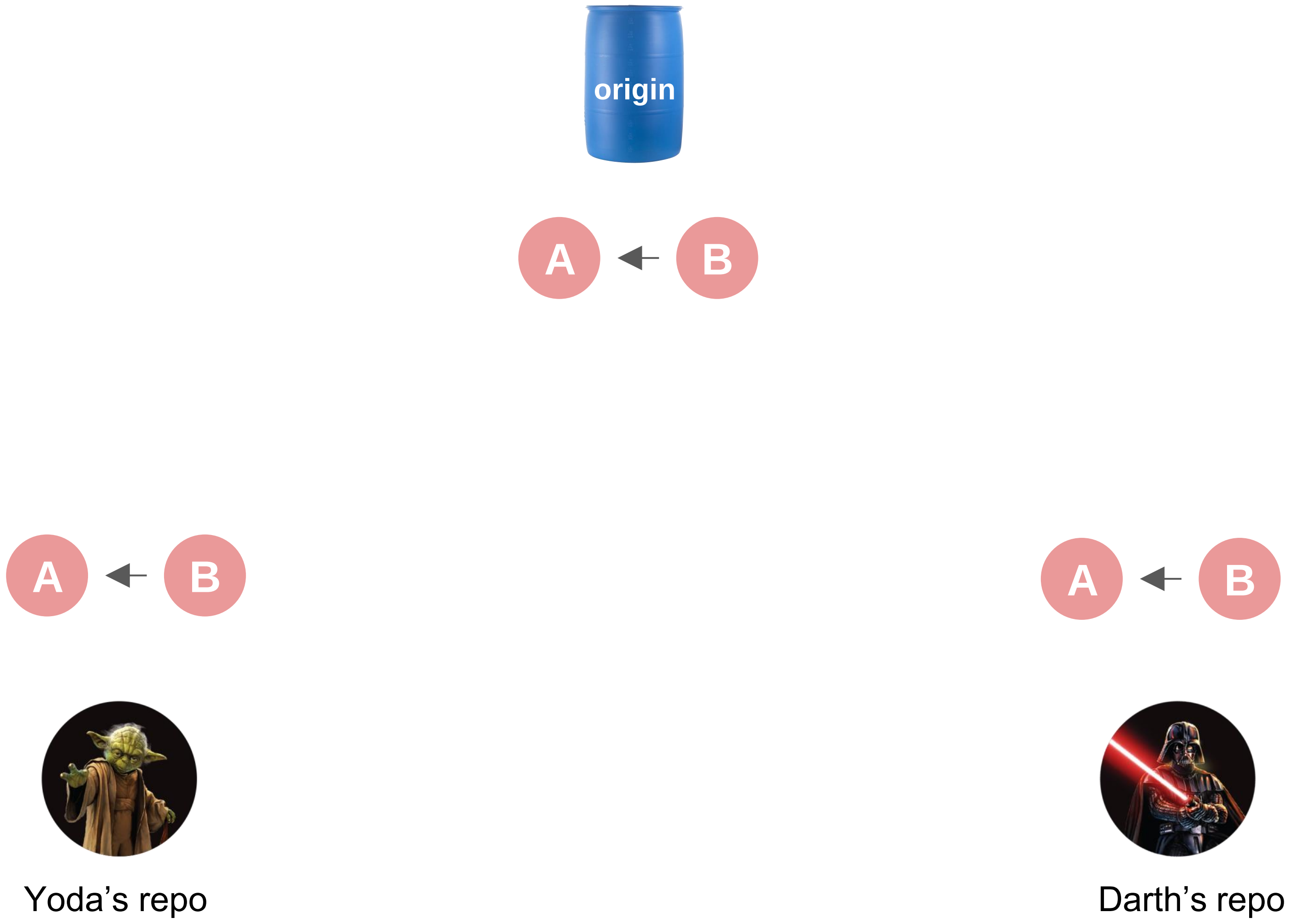
```
$ git add <file>
$ git commit -m “Initial commit”
$ git add <file>
$ git commit -m “Added new file”
$ git push
```

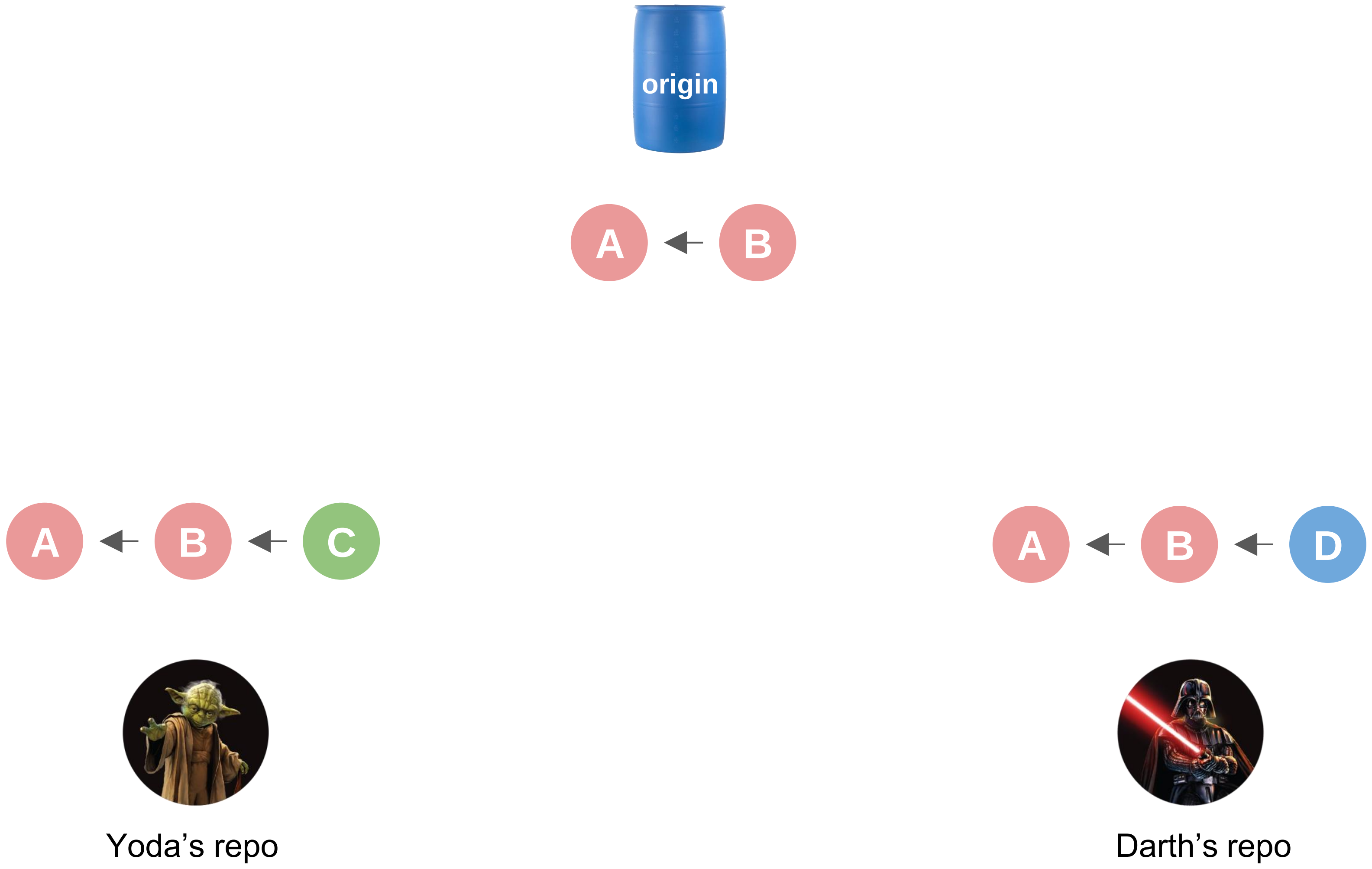


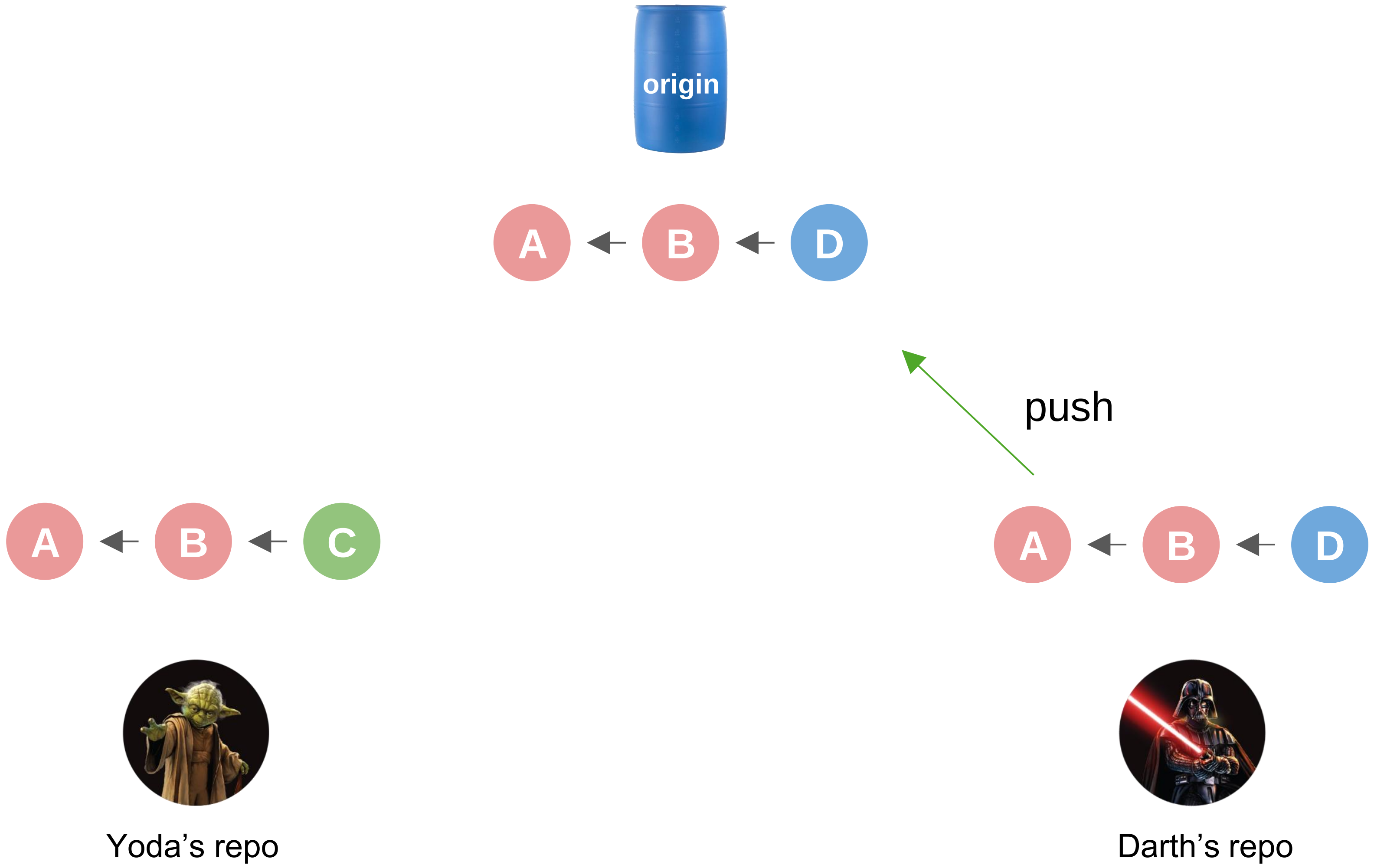
```
$ git add <file>
$ git commit -m “Initial commit”
$ git add <file>
$ git commit -m “Added new file”
$ git push
```

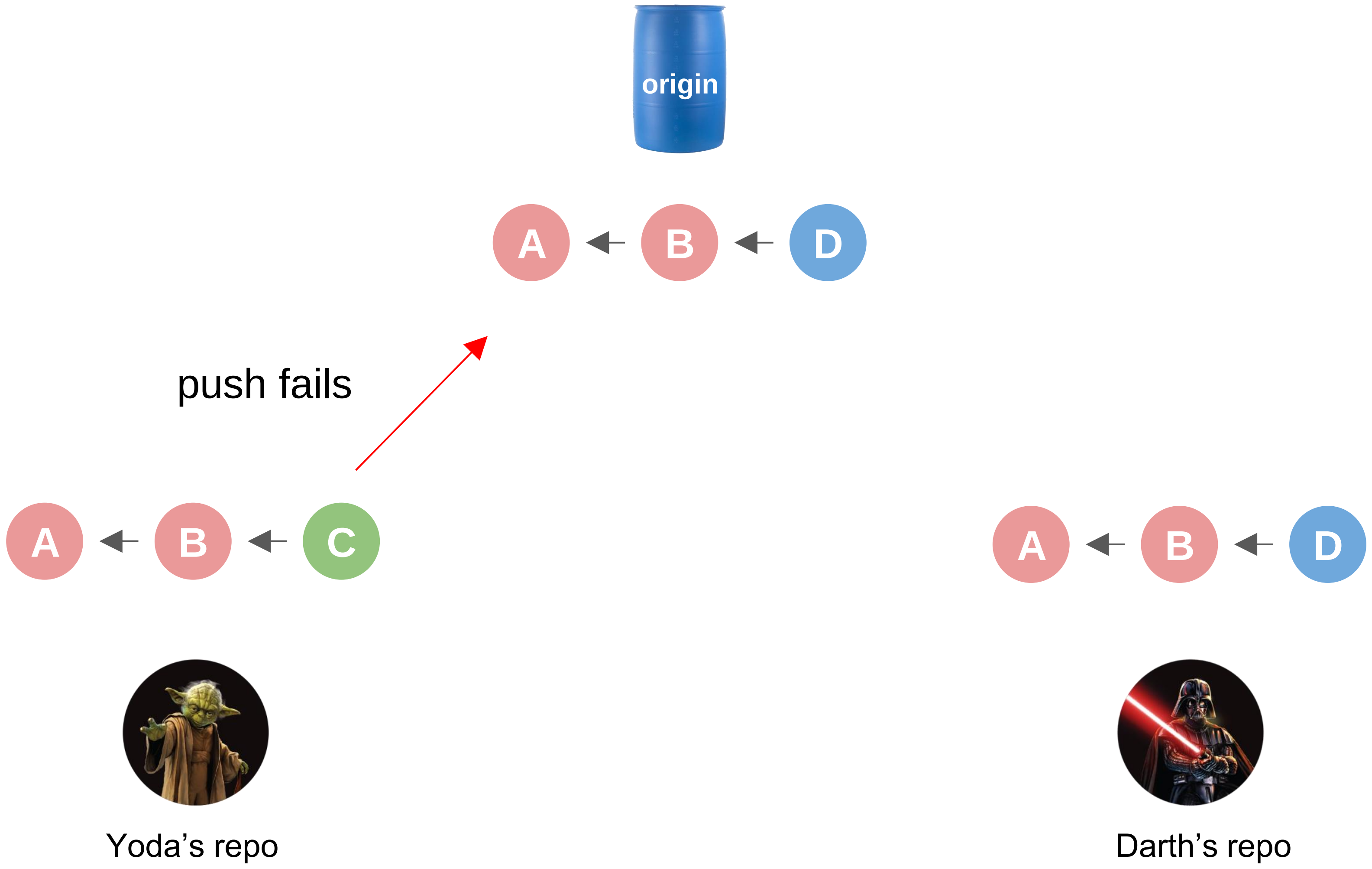
```
$ git pull
```

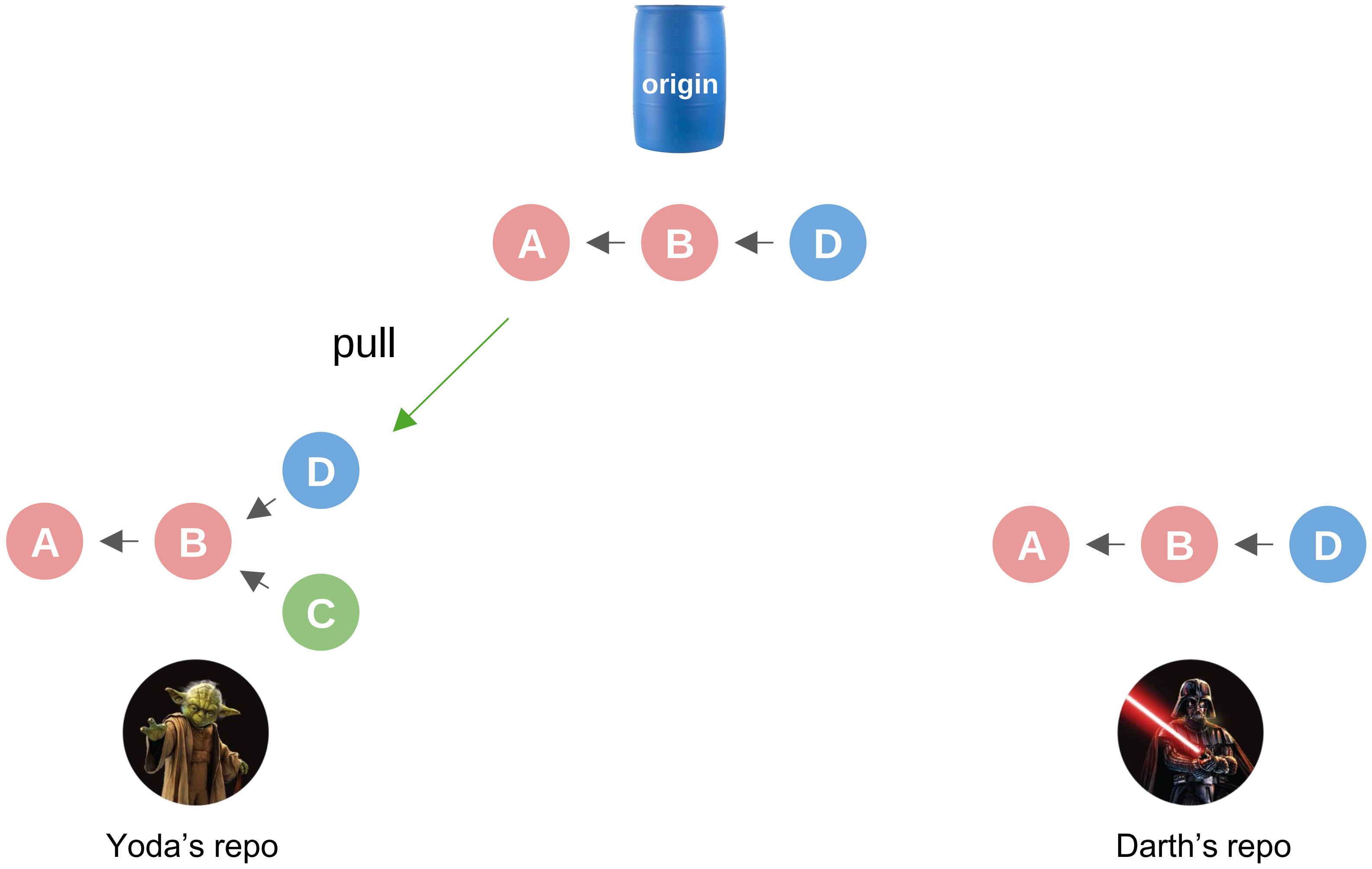
Parallel change

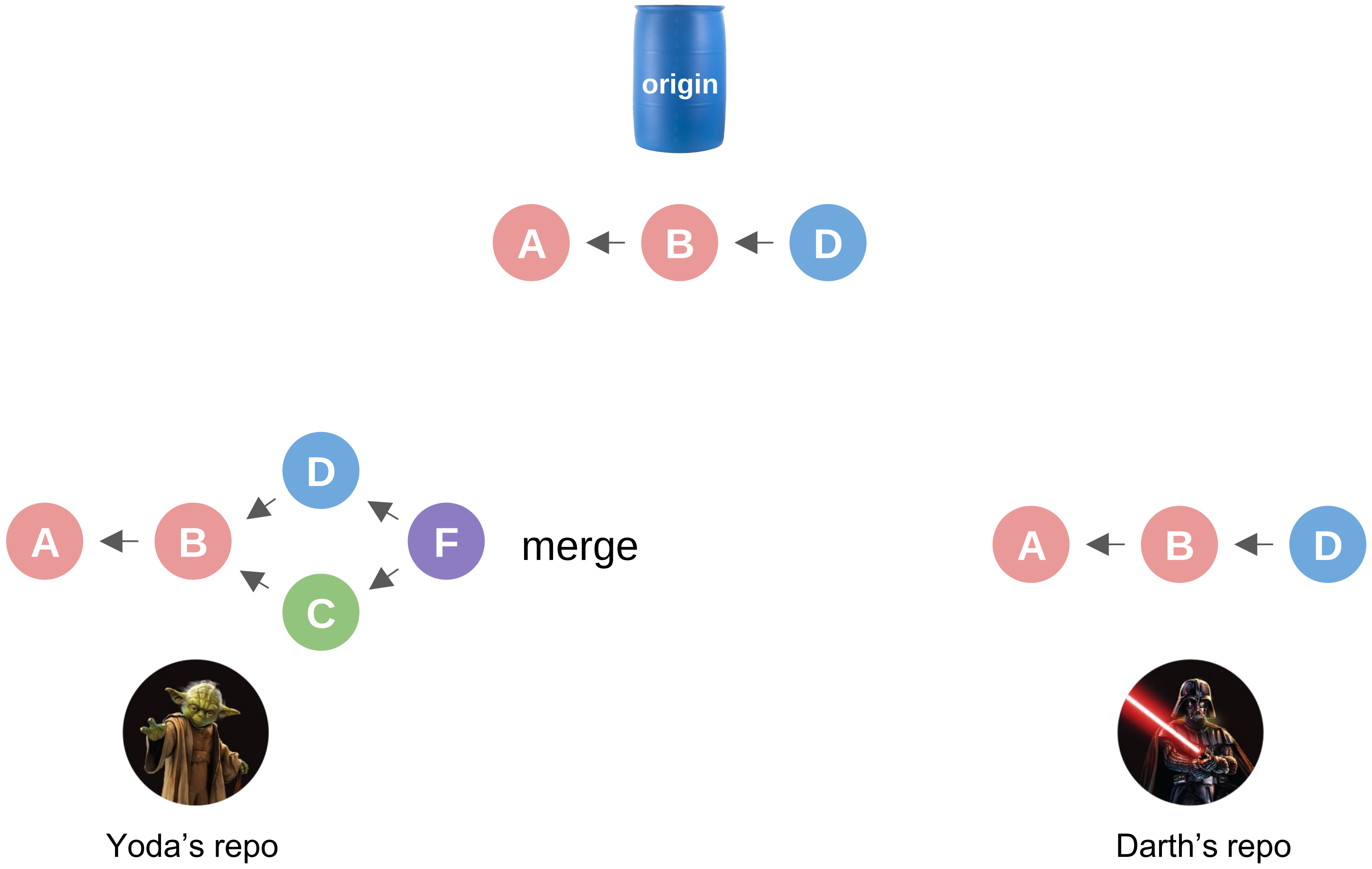

origin
A
B
A
B
A
B
Yoda’s repo
Darth’s repo

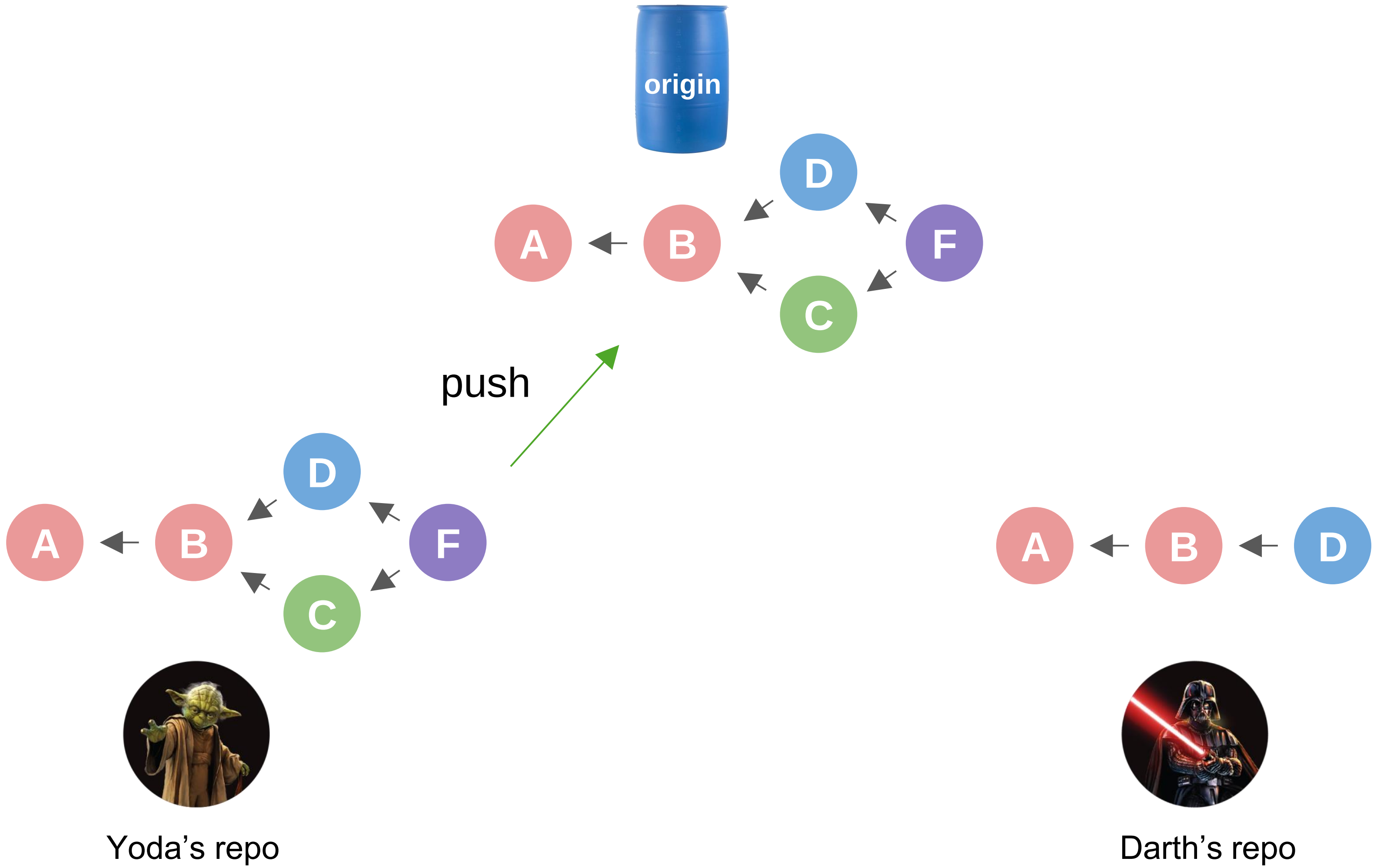

origin
A
B
A
B
C
A
B
D
Yoda’s repo
Darth’s repo

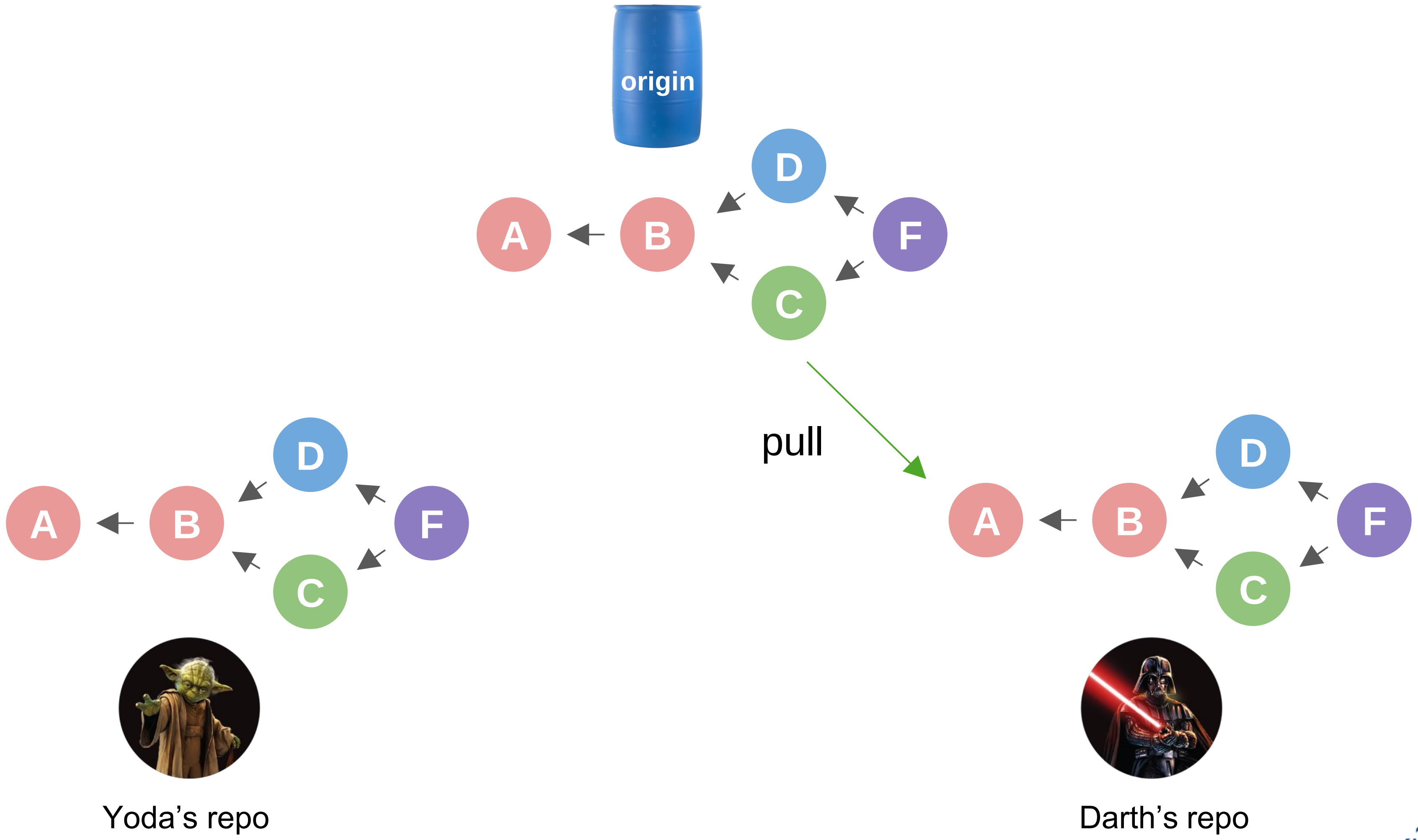

origin
A
B
D
push
A
B
C
A
B
D
Yoda’s repo
Darth’s repo


origin
A
B
D
push fails
A
B
C
A
B
D
Yoda’s repo
Darth’s repo


origin
A
B
D
pull
D
A
B
C
Yoda's repo
A
B
D
Darth's repo


origin
A
B
D
A
B
D
C
F
merge
A
B
D
Yoda’s repo
Darth’s repo


origin
A
B
D
C
F
push
A
B
D
C
F
A
B
D
Yoda’s repo
Darth’s repo


origin
A
B
D
C
F
pull
A
B
D
C
F
Yoda’s repo
A
B
D
C
F
Darth’s repo

Summary

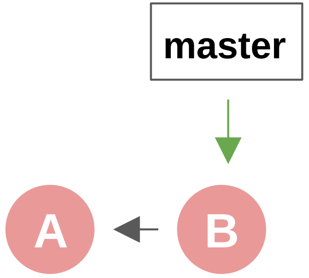
- When there were new changes in the branch, push will fail
- pull will try to merge changes and create a “merge commit”
- After merge is completed, you can push your changes
- Commit can have two or more parents - “octopus merge”

Manual conflict resolve

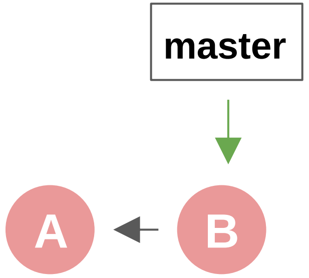
- When automatic merge fails, you have to resolve conflict manually
- Use `git add` to mark resolution
- Then you can commit as usual

Remote branches

Developer and Remote

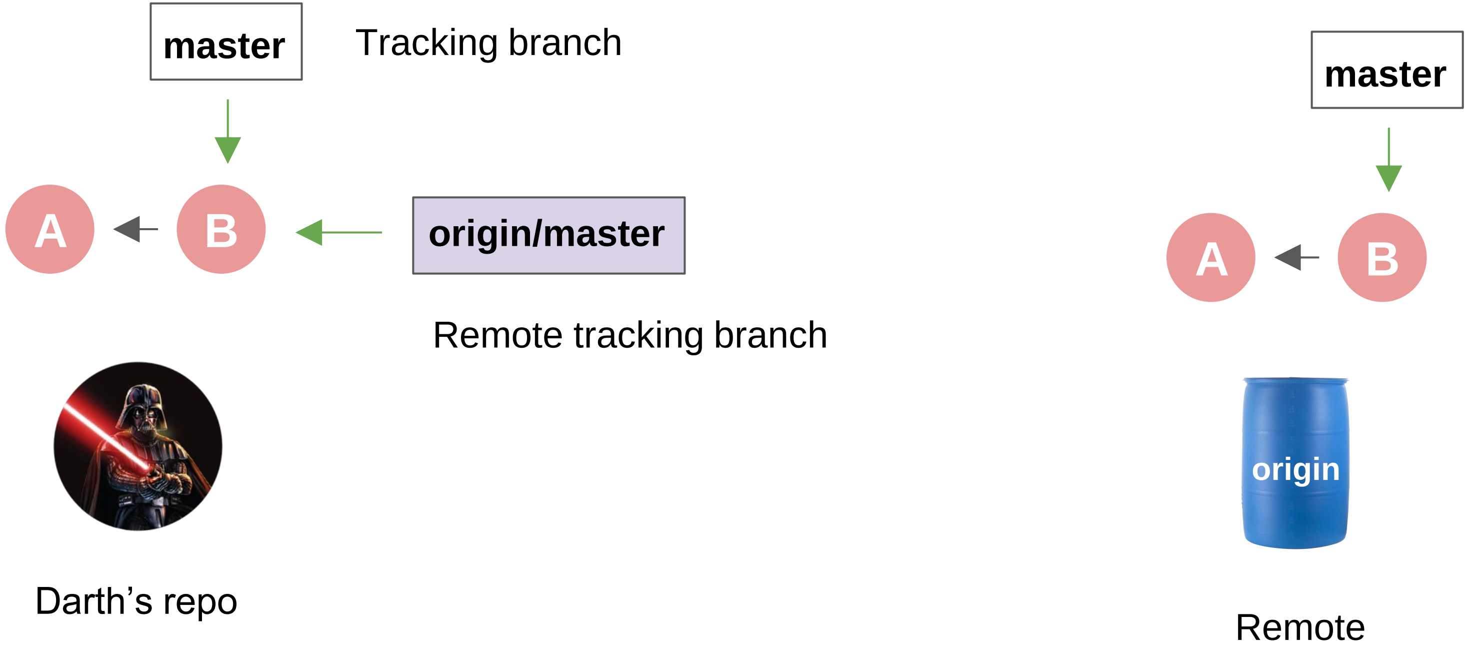


Darth’s repo



Remote

Tracking state of Remote

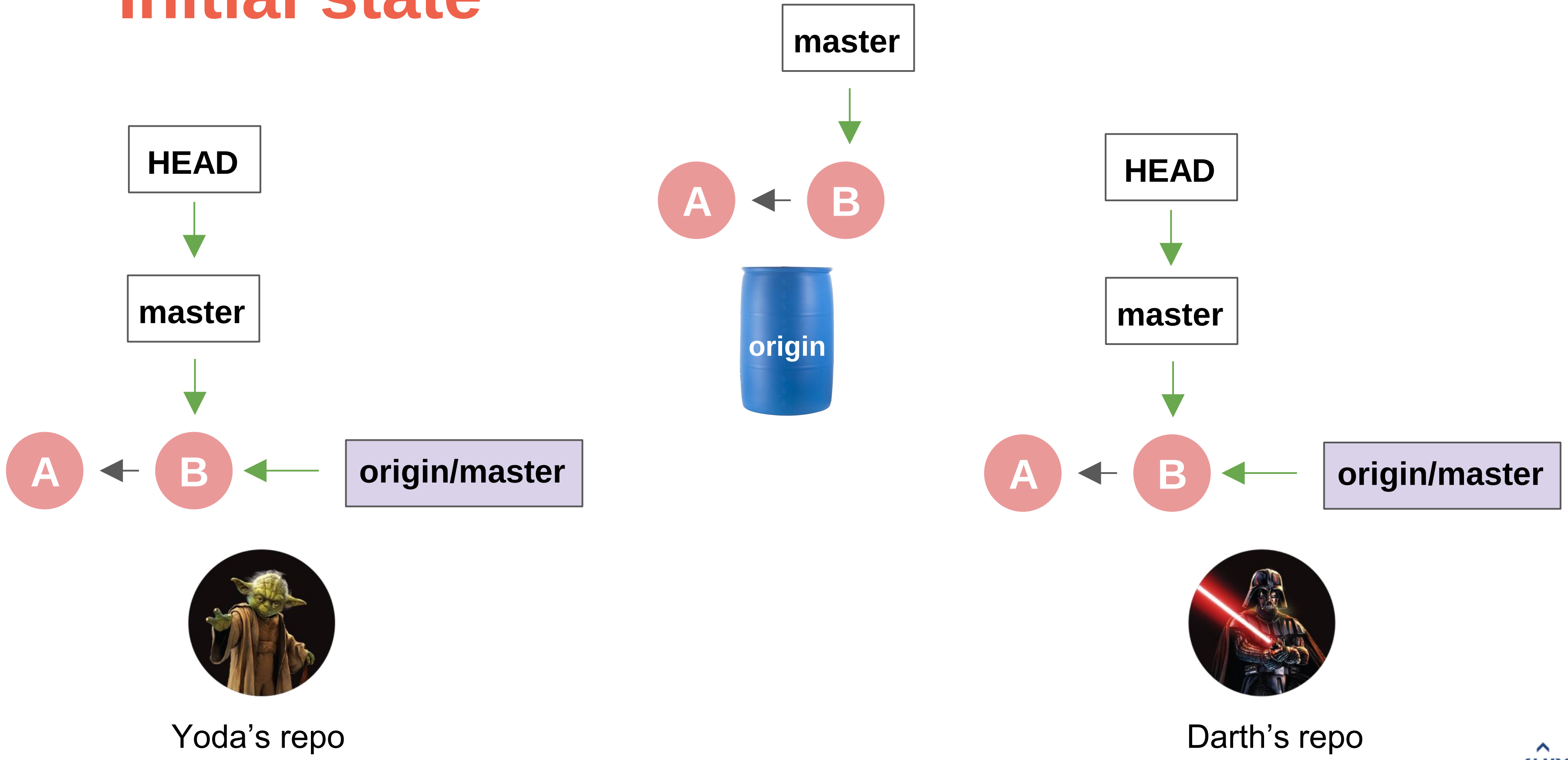


Commands

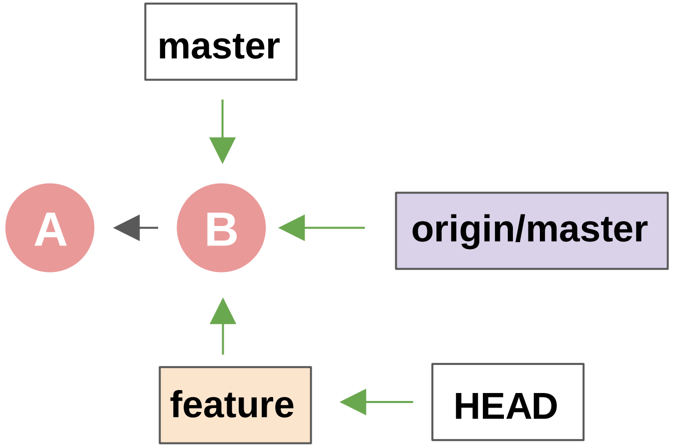
Command	Description
`git branch <name>`	Create new branch pointing to current commit
`git push <remote> <name>`	Push branch to remote
`git push -u <remote> <name>`	Push branch to remote and make it tracking
`git branch -a -vv`	Show very detailed info about branches
`git push origin --delete <branch>`	Delete a remote branch

Sharing branches

Initial state



Yoda is making a new branch

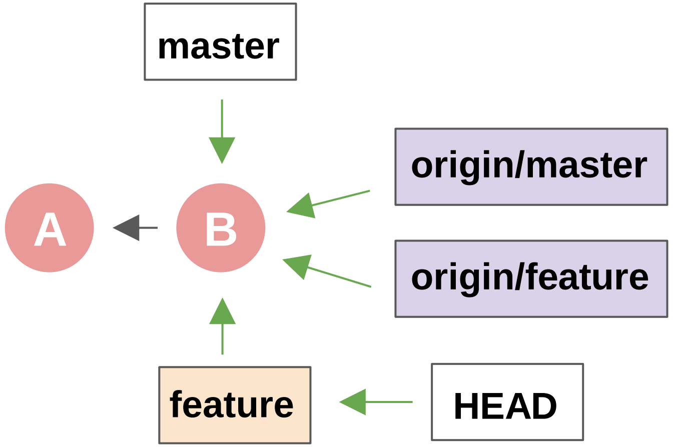


```
$ git checkout -b feature
```

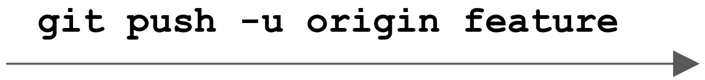
Yoda’s repo

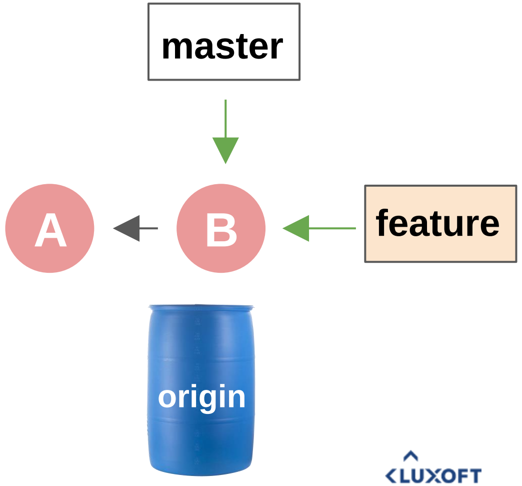
Yoda is pushing branch to remote

```
$ git checkout -b feature
$ git push -u origin feature
```



Yoda's repo

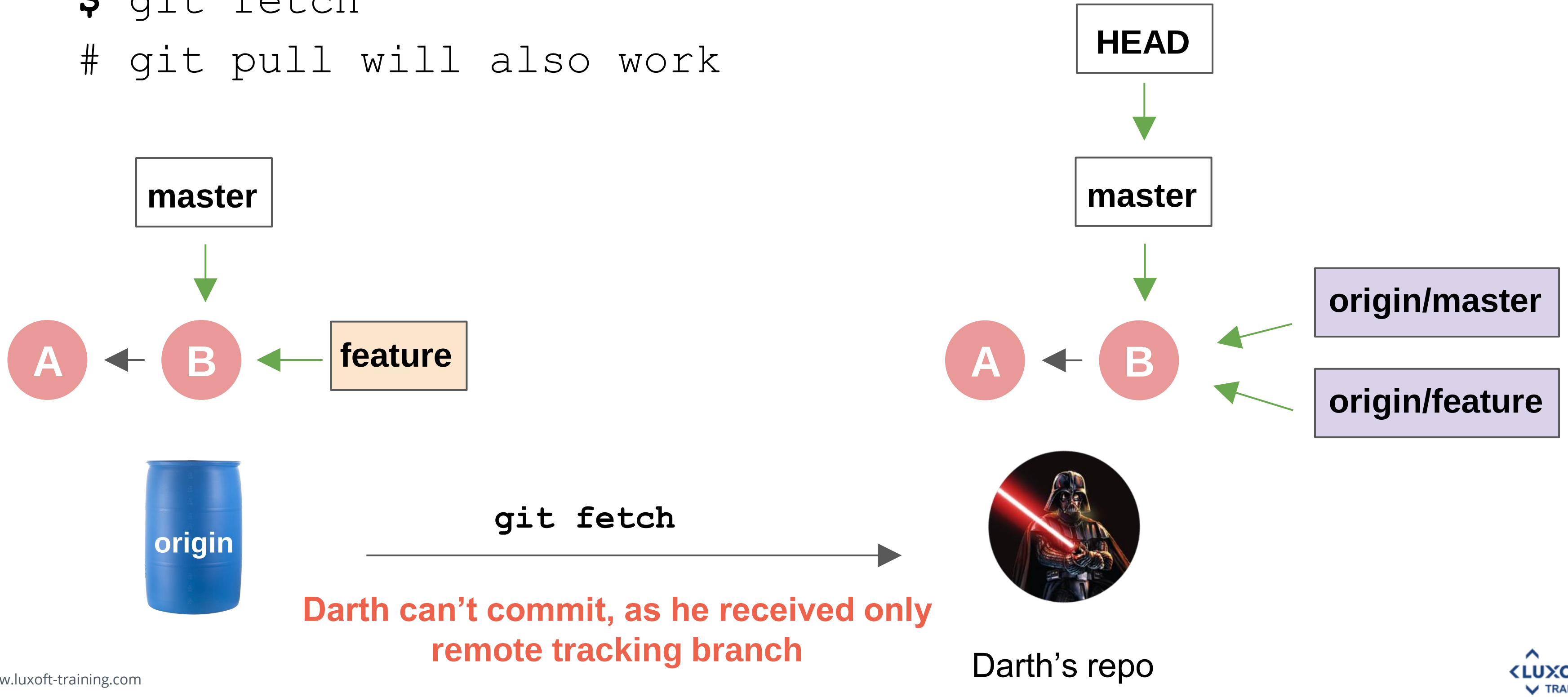




-u flag makes **feature track origin/feature**

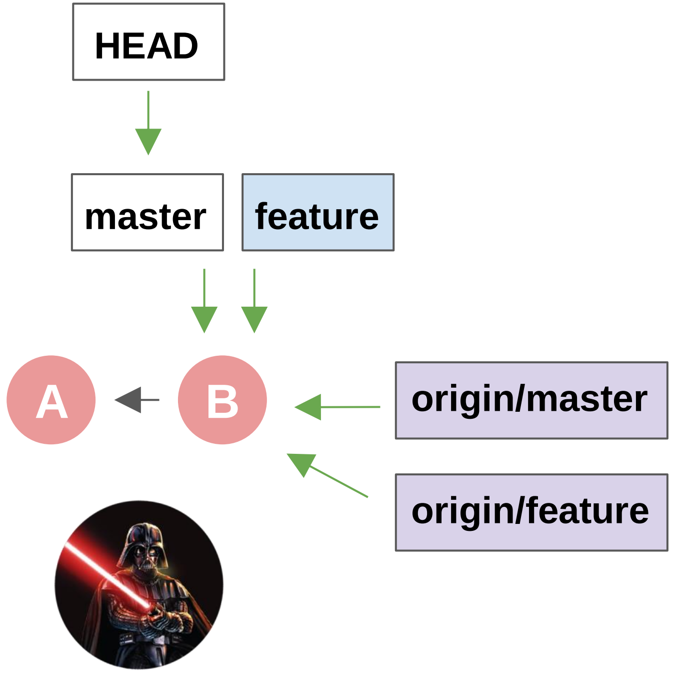
With next fetch Darth gets the new branch

```
$ git fetch
# git pull will also work
```



Darth can't commit, as he received only remote tracking branch

Darth checks out the branch to start work



Darth's repo

```
$ git checkout feature
```

The command automatically creates a branch as a tracking one

Custom log

Custom format of log

```
$ git log --pretty=format:'%Cred%h%Creset <%an> %C(#a2d6f5)%cr%Creset'
```

Pattern	Description
`%h`	Abbreviated hash
`%d`	References' names (decorate)
`%s`	Subject (message)
`%an`	Author name
`%cr`	Commit date (relative)

Aliases

Add aliases to project

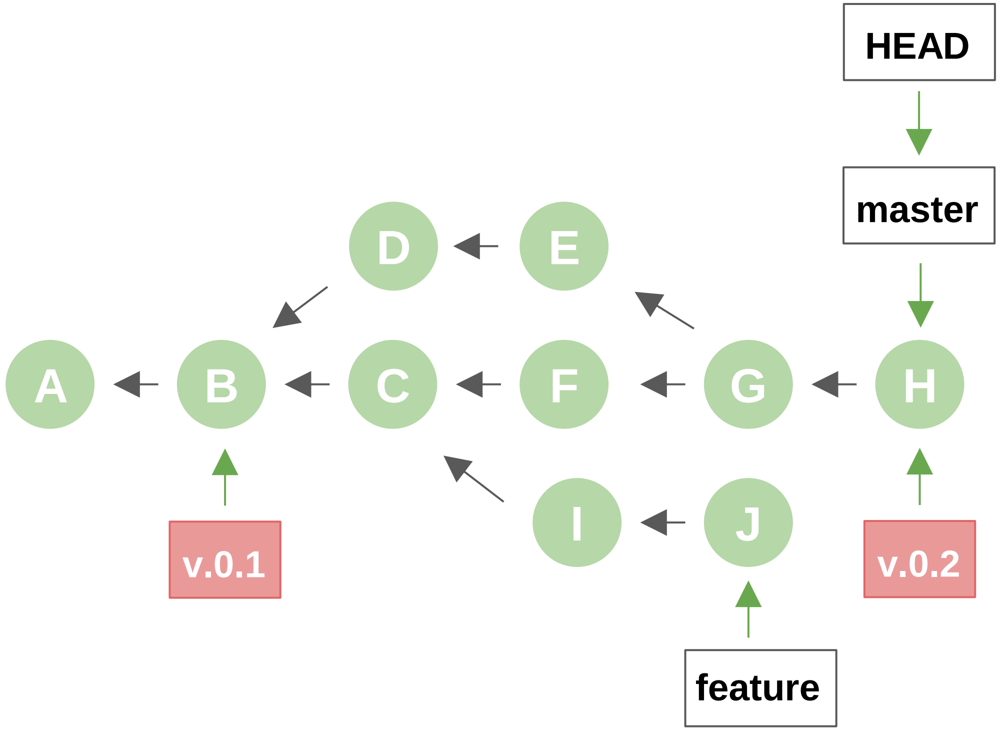
```
$ git config --global alias.co checkout
$ git config --get-regexp alias
```

Practice. Lab 2

Specific commands

Tagging

Tag specific points in history as important



Create tag:

```
$ git tag v.0.2
```

Push tag to server:

```
$ git push --tags
```

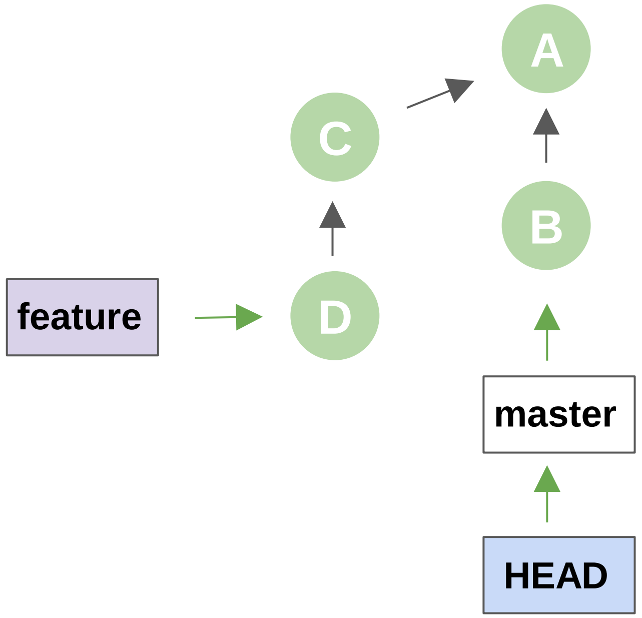
List your tags:

```
$ git tag
v.0.1
v.0.2
```

Rebase

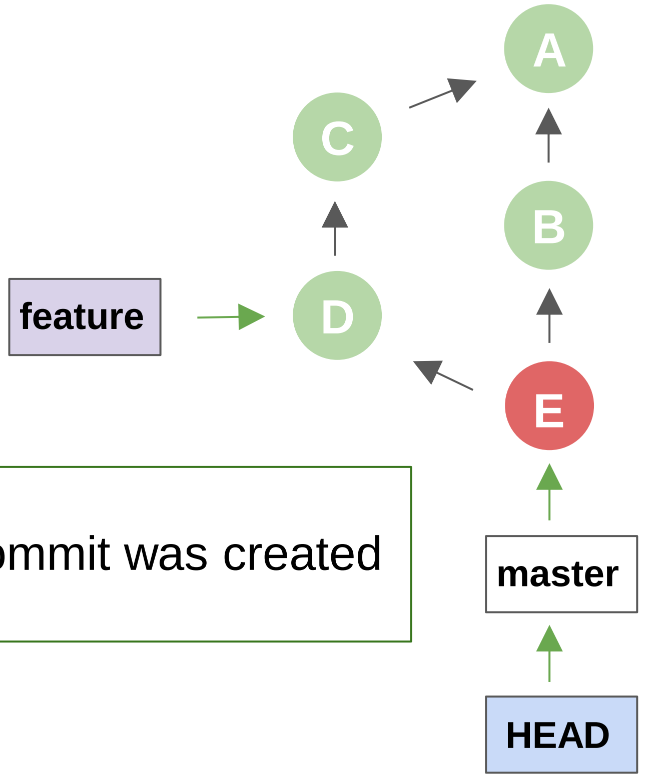
Integrate changes from one branch into another one

```
# starting from master on A

$ git checkout -b feature
$ git commit -m "C"
$ git commit -m "D"

$ git checkout master
$ git commit -m "B"
```



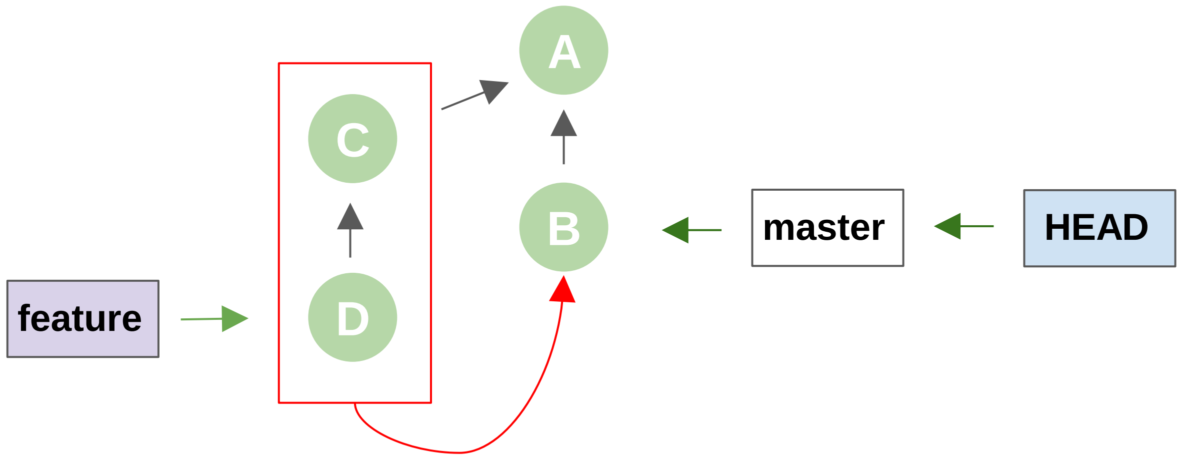
Merge recalling

```
$ git merge feature
```



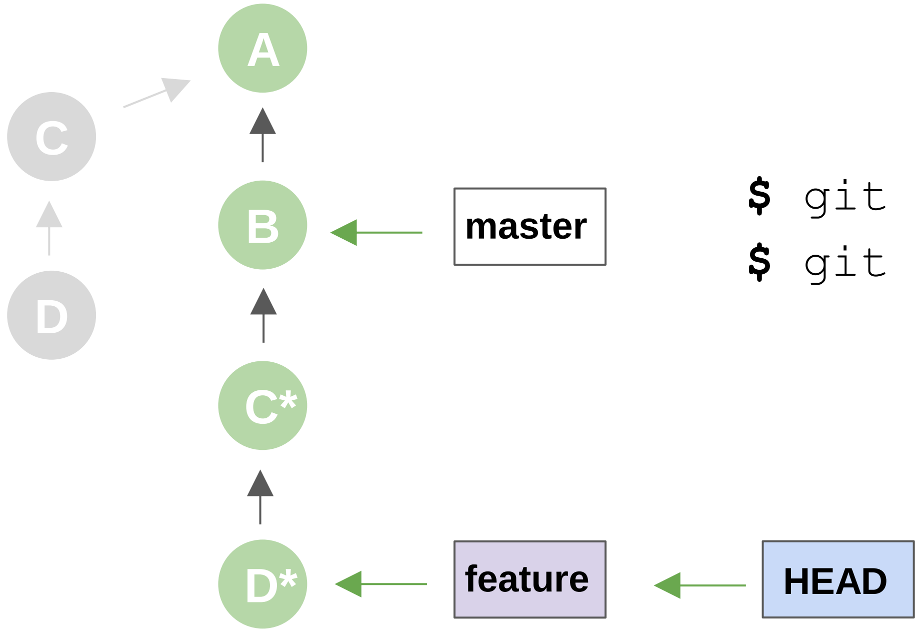
Merge commit was created

Rebase solves the same problem as merge



Replace the work to the new base

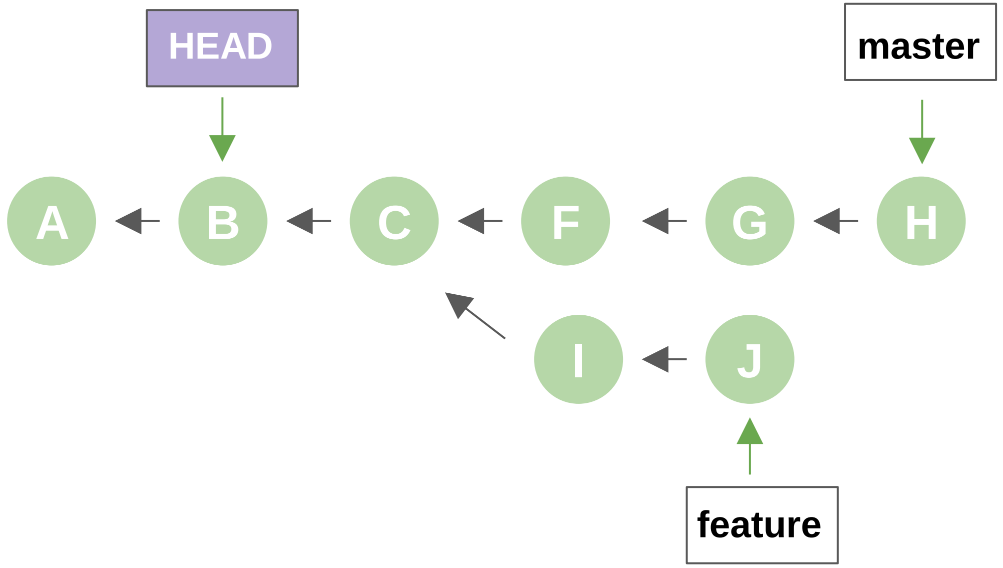
Rebase - replace the work to the new base



```
$ git checkout feature
$ git rebase master
```

Detached HEAD

Detached HEAD state



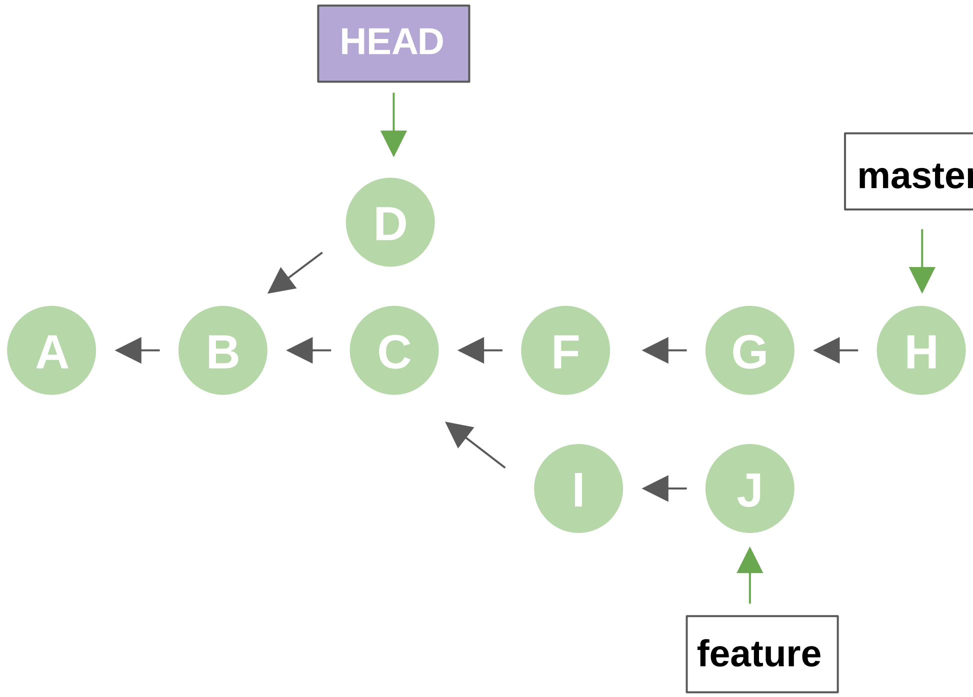
Starting from master on H

Checkout to commit B:

```
$ git checkout [sha_of_B]
```

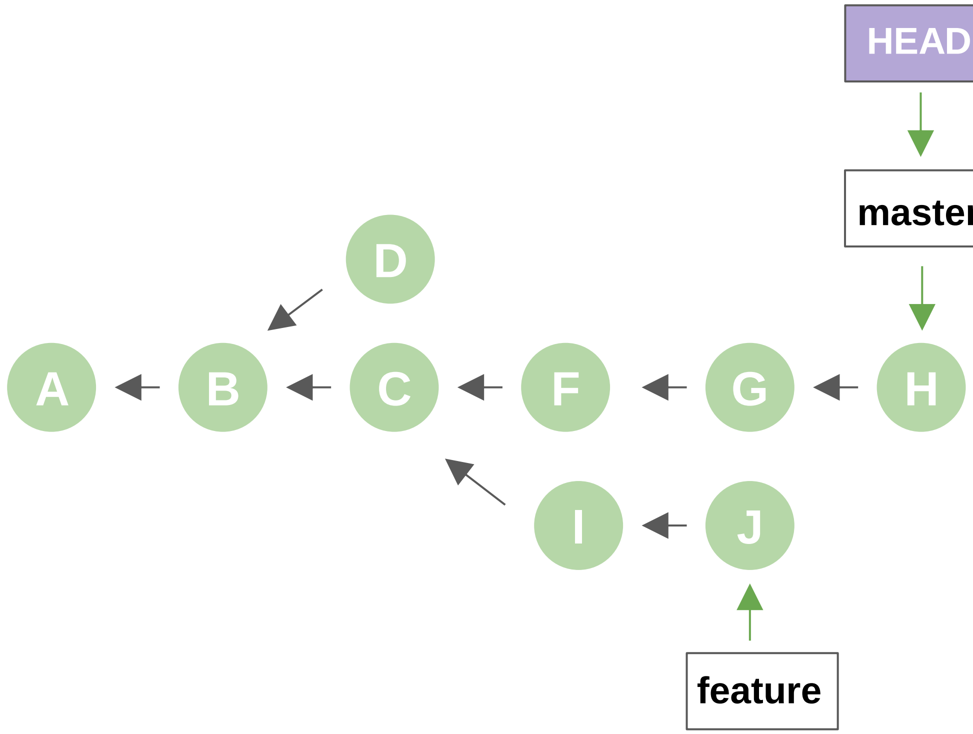
When a specific *commit* is checked out instead of a *branch* - is what's called a "detached HEAD"

Commit made in detached HEAD state



```
$ git commit -m "D"
```

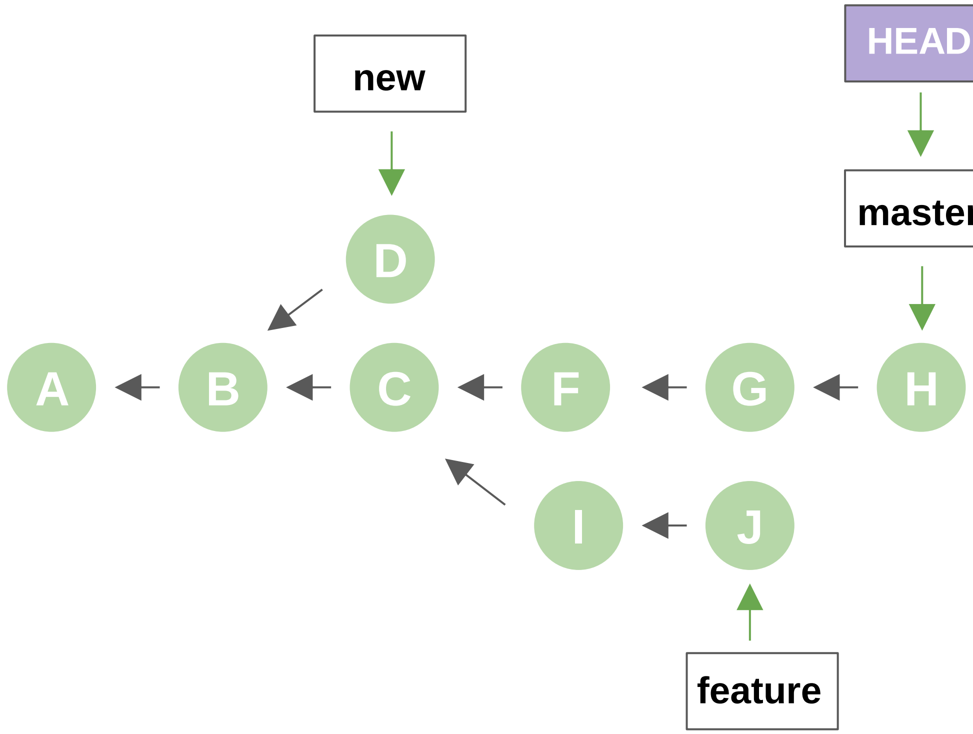
Commit made in detached HEAD state



```
$ git checkout master
```

We lost changes made in commit D, as they do NOT belong to any branch

Save commit made in detached HEAD state



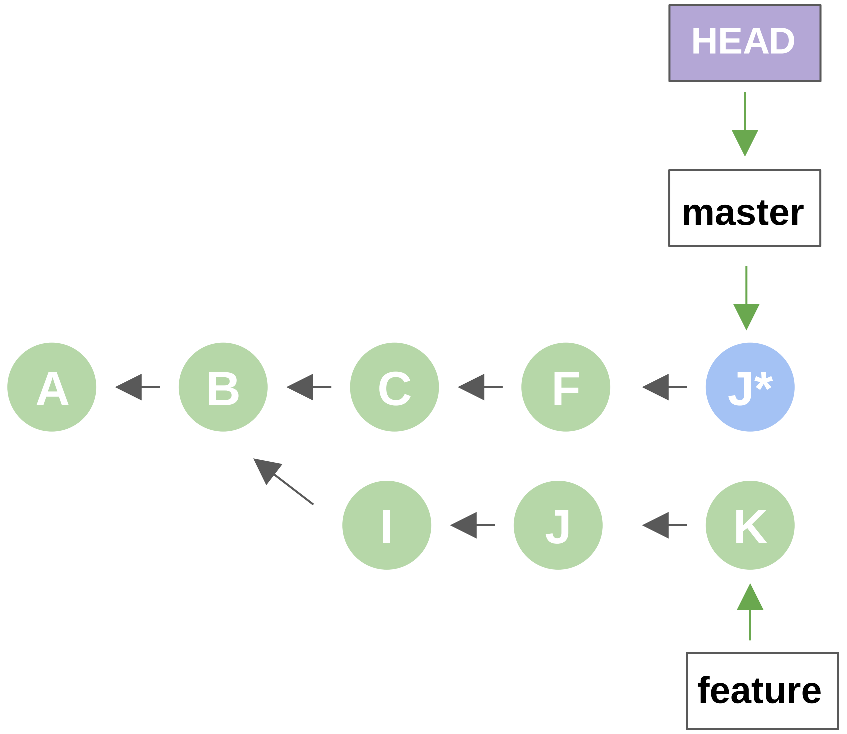
Starting from HEAD on D

```
$ git checkout -b new
$ git checkout master
```

To save changes made in commit D, create new branch (pointer to that commit)

Cherry-pick

Cherry-picking commit



Starting from master on F

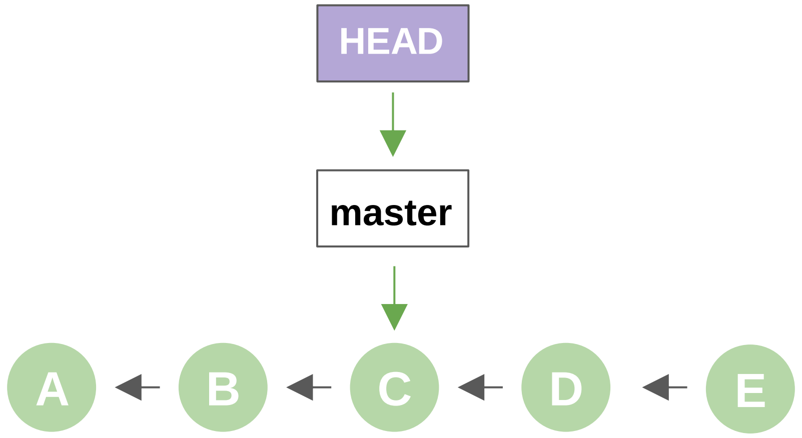
Cherry-pick commit J to master:

```
$ git cherry-pick [sha_of_J]
```

`git cherry-pick` takes a commit from somewhere else and "plays it back" wherever you are right now

Reflog

Data loss



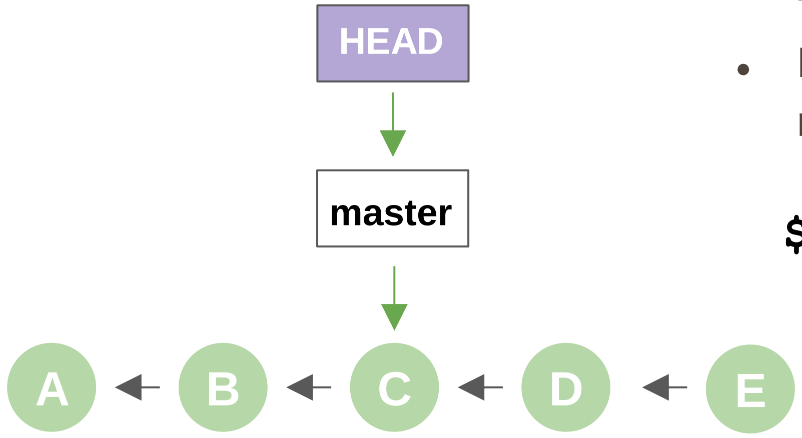
Starting from master on E

Hard-reset master branch:

```
$ git reset --hard HEAD~2
```

Commits D and E are lost

Recover lost commits



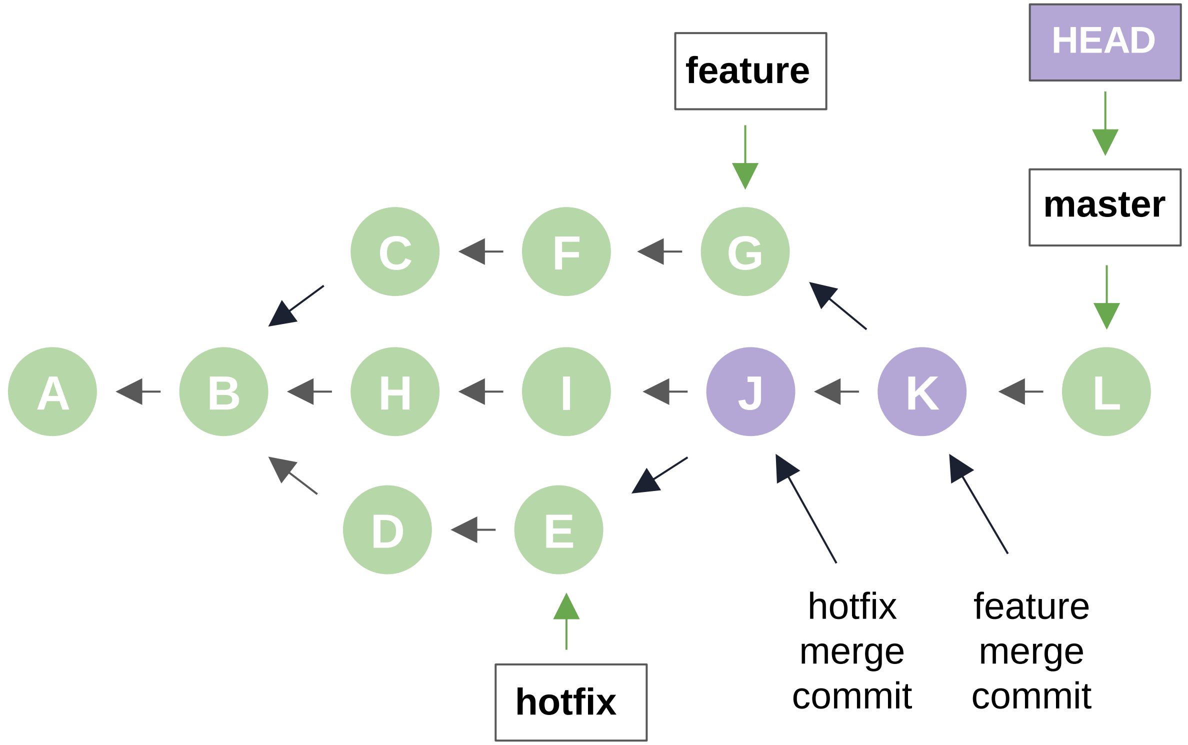
- Git silently records what your HEAD is every time you change it
- Each time you commit or change branches, the reflog is updated

```
$ git reflog
```

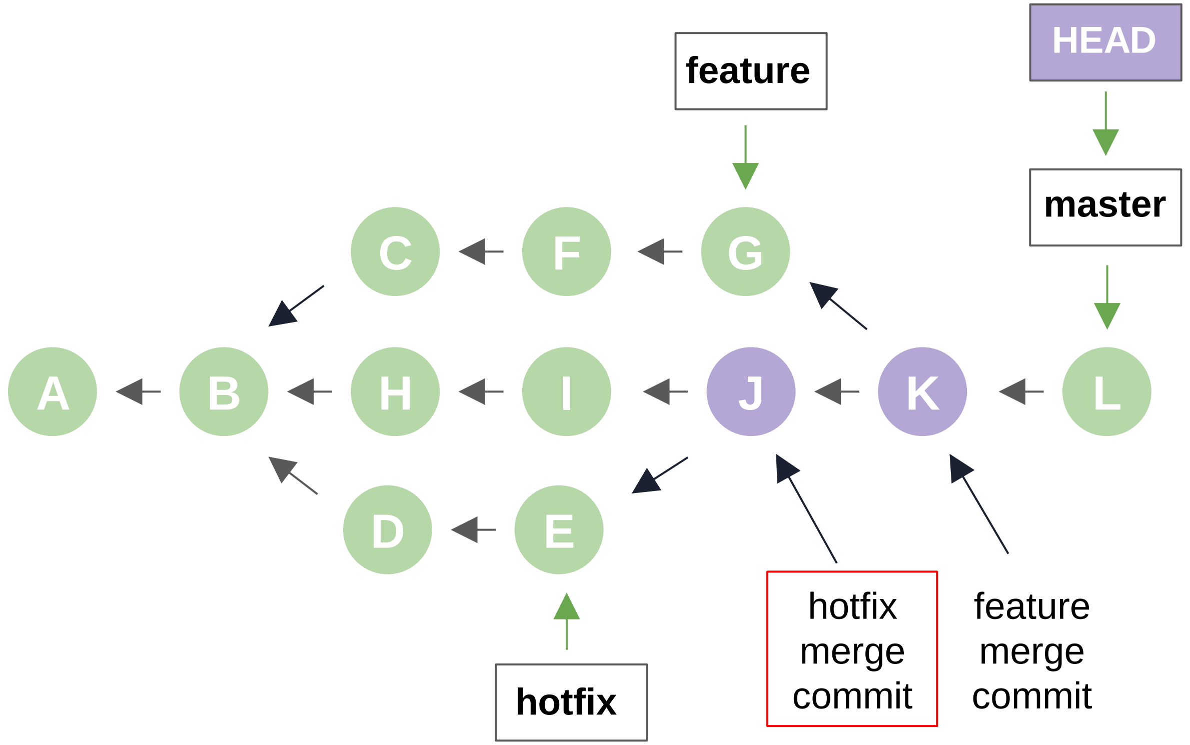
The **reflog** is an ordered list of the commits that HEAD has pointed to: it's undo history for your repo.

Revert

Undoing merges



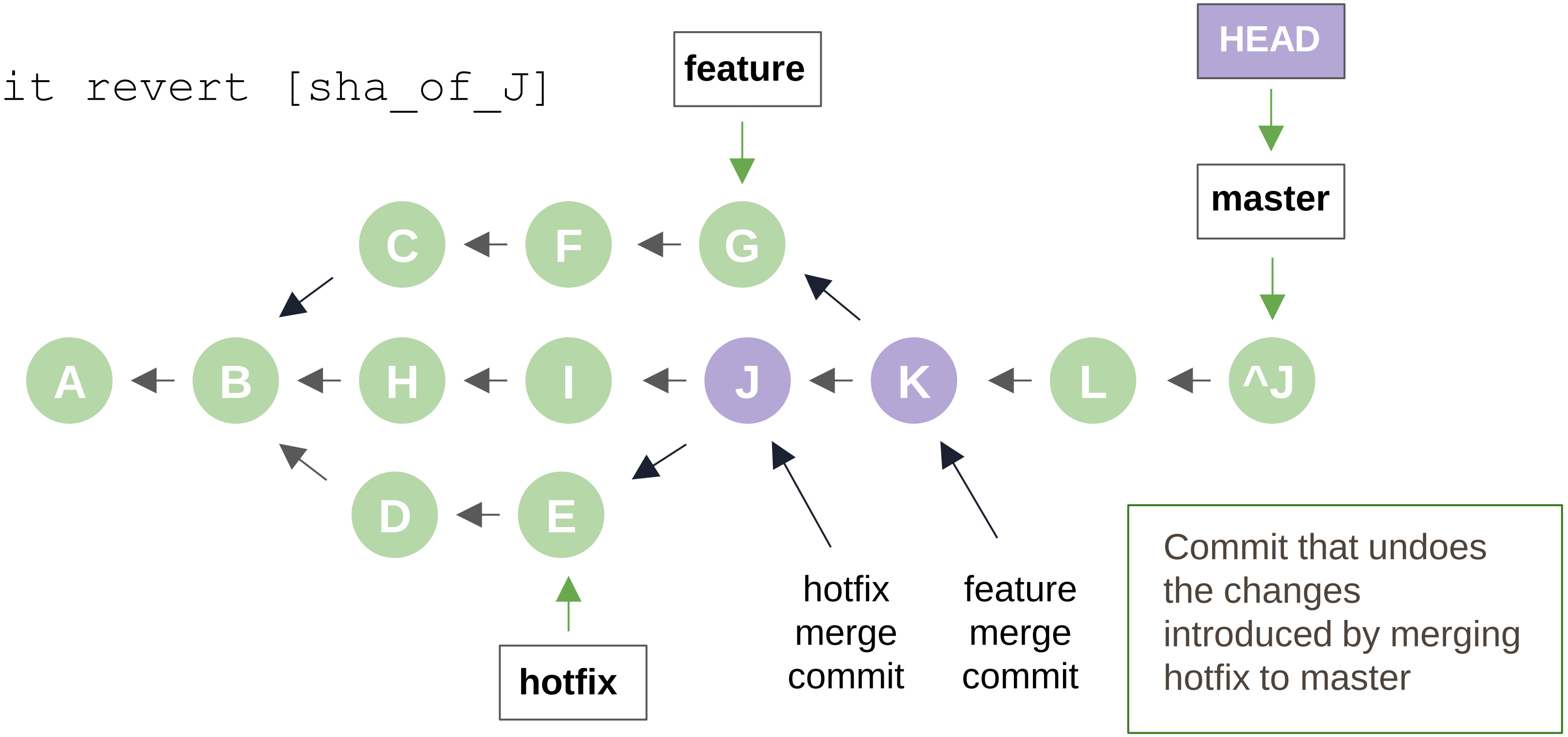
Undoing merges



We need to revert commit J

Undoing merges

```
$ git revert [sha_of_J]
```



Commit that undoes the changes introduced by merging hotfix to master

Stash

Stashing your work

- While working on your project, you want to **switch branches** to work on something else.
- But, you **don’t want to do a commit** of half-done work just so you can get back to this point later.
- The answer to this issue is the **`git stash`** command.

Stashing your work

Stashing **takes the dirty state of your working directory** – that is, your modified tracked files and staged changes – and **saves it on a stack of unfinished changes** that you can reapply at any time.

Code example

```
     // working in feature branch...
$ git stash
$ git checkout hotfix
      // working in hotfix branch...
$ git commit -am "Hotfix added"
$ git checkout feature
$ git stash pop
$ git commit -am "Feature added"
```

Commands

Command	Description
`git stash save`	Save your local modifications to a new stash.
`git stash show`	Show the changes recorded in the stash as a diff between the stashed state and its original parent.
`git stash list`	List the stashes that you currently have.
`git stash pop`	Remove a single stashed state from the stash list and apply it on top of the current working tree state.

Practice. Lab 3

Used materials

Used materials

- Вебинар Git Bootcamp - всё про Git и эффективную работу с кодом (Juriy Bura)

  https://www.youtube.com/playlist?list=PLQlWzK5tU-gAHvPwiABQD80lXCEpBIYmS
- Git documentation https://git-scm.com/docs/
- [Linux.conf.au 2013] - Git For Ages 4 And Up

  https://www.youtube.com/watch?v=1ffBJ4sVUb4
- Git from the inside out https://www.youtube.com/watch?v=fCtZWGhQBvo

Thank you

LUXOFT TRAINING

LXFT LISTED NYSE