

Version Control with Git

Xiaoxu Guan

High Performance Computing, LSU

November 11, 2015



(<https://www.atlassian.com/git/tutorials>)

Overview

- Why should we use a version control system?

Overview

- Why should we use a version control system?
- What is **Git**?

Overview

- Why should we use a version control system?
- What is **Git**?
- The setting up of **Git**:
 - Programmer's name, email address, etc.
 - Select some types of files to be controlled by **Git**;
 - Customize your **Git** working environment;
 - The setting up of a **Git** repository;

Overview

- Why should we use a version control system?
- What is **Git**?
- The setting up of **Git**:
 - Programmer's name, email address, etc.
 - Select some types of files to be controlled by **Git**;
 - Customize your **Git** working environment;
 - The setting up of a **Git** repository;
- The ideas and Workflow behind **Git**;
 - Working directory;
 - Cached files;
 - Commit changes to a repository;
 - Set up multiple branches and resolve the conflicts;
 - Work on remote servers;

Overview

- Why should we use a version control system?
- What is **Git**?
- The setting up of **Git**:
 - Programmer's name, email address, etc.
 - Select some types of files to be controlled by **Git**;
 - Customize your **Git** working environment;
 - The setting up of a **Git** repository;
- The ideas and Workflow behind **Git**:
 - Working directory;
 - Cached files;
 - Commit changes to a repository;
 - Set up multiple branches and resolve the conflicts;
 - Work on remote servers;
- **Summary and Further Reading**

Why should we use a version control system?

- Keep your files “forever” (a back-up strategy);
- Collaboration with your colleagues;
- Keep track of every change you made;
- Work on a large-scale program with many other teams;
- Test different ideas or algorithms without creating a new directory or repository;
- Enhance productivity of code development;
- Not only applicable to source code, but also other files;
- The tool that tracks and manages changes and different versions is called **Version Control System (VCS)**;

What is **Git**?

- **Git** is one of the VCSs [Subversion (SVN), CVS, Mercurial, Bazaar, . . .];
- Created by Linus Torvalds in April 2005 (motivated by maintenance of Linux kernel);
- A **distributed** VCS (compared to a **centralized** VCS);
- Allows many developers (say, hundreds) on the same project;
- Focus on **non-linear** code development;
- “**Delta**” techniques are used to run faster and efficiently;
- Ensure integrity and trust;
- Enforce “bookkeeping” and accountability;
- Support branched development;
- Be free (an open source VCS);

What is Git?

“The information manager from hell”!

- **Git** is one of the VCSs [Subversion (SVN), CVS, Mercurial, Bazaar, . . .];
- Created by Linus Torvalds in April 2005 (motivated by maintenance of Linux kernel);
- A **distributed** VCS (compared to a **centralized** VCS);
- Allows many developers (say, hundreds) on the same project;
- Focus on non-linear code development;
- “Delta” techniques are used to run faster and efficiently;
- Ensure integrity and trust;
- Enforce “bookkeeping” and accountability;
- Support branched development;
- Be free (an open source VCS);

What is **Git**?

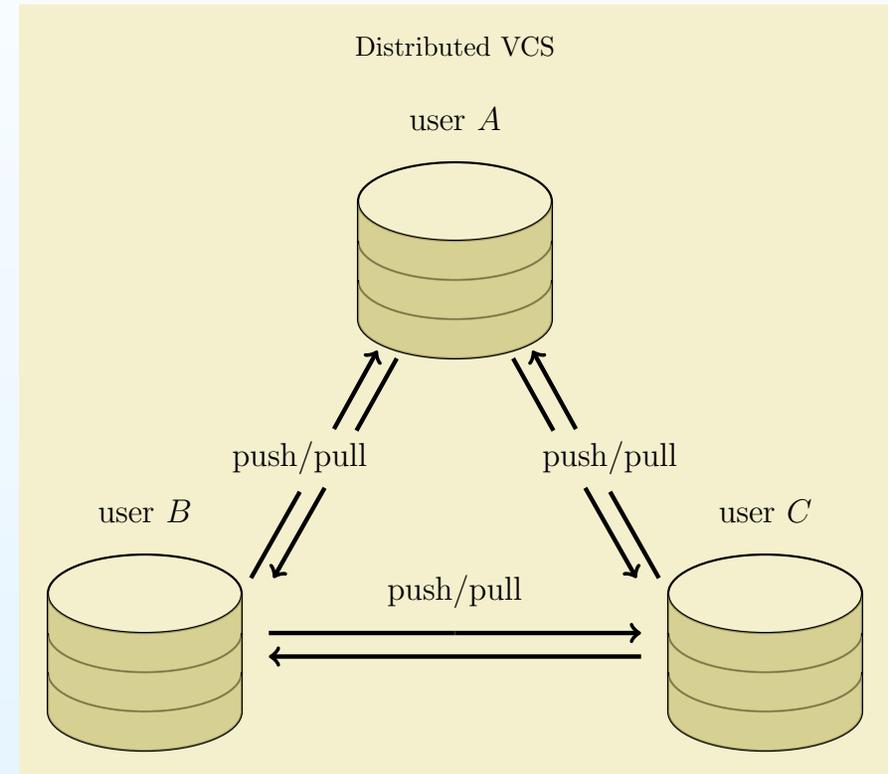
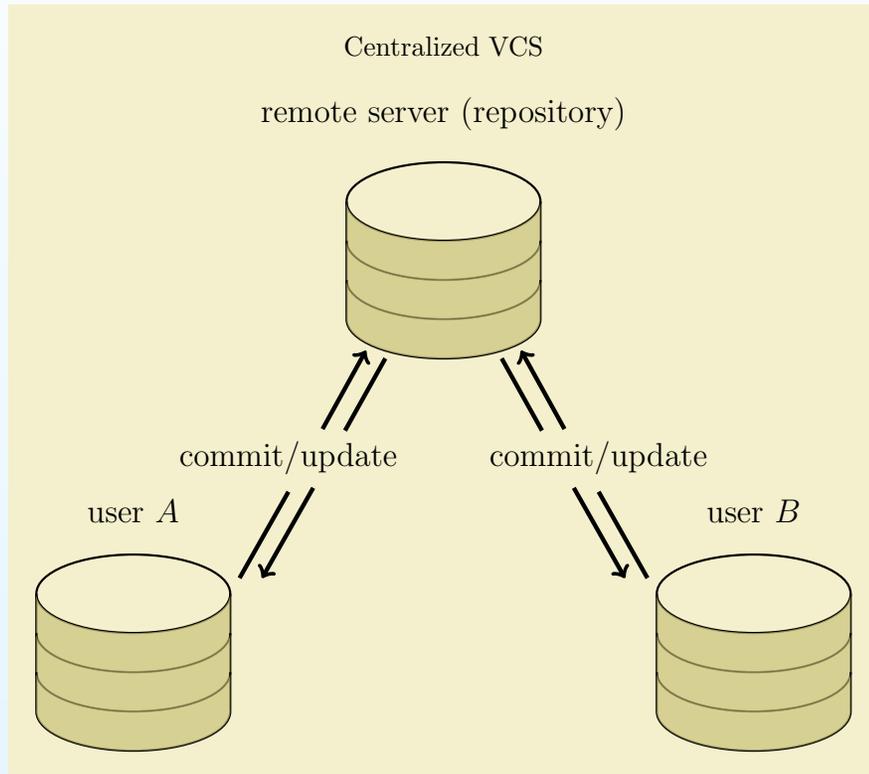
With this commit **Git** was born on April 7, 2005:

```
commit e83c5163316f89bfbde7d9ab23ca2e25604af29
Author: Linus Torvalds <torvalds@pp970.osdl.org>
Date: Thu Apr 7 15:13:13 2005 -0700
Initial revision of "git", the information manager from hell
```

— *Version Control with Git (J. Loeliger, O'reilly, 2009)*

What is Git?

Centralized VCS vs. Distributed VCS



Git is one of **distributed** version control systems.

Configure Git

- **Git** was installed on all the LSU **HPC** and **LONI** machines;
- What happens if we run **git** without any arguments?

Configure Git

- **Git** was installed on all the **LSU HPC** and **LONI** machines;
- What happens if we run **git** without any arguments?

```
[xiao Xu@smic1 ~]$ git
usage: git [--version] [--exec-path[=GIT_EXEC_PATH]] [--html-path]
        [-p|--paginate|--no-pager] [--no-replace-objects]
        [--bare] [--git-dir=GIT_DIR] [--work-tree=GIT_WORK_TREE]
        [--help] COMMAND [ARGS]
```

The most commonly used git commands are:

add	Add file contents to the index
bisect	Find by binary search the change that introduced a bug
branch	List, create, or delete branches
checkout	Checkout a branch or paths to the working tree
clone	Clone a repository into a new directory
commit	Record changes to the repository
diff	Show changes between commits, commit and working tree, etc
fetch	Download objects and refs from another repository
grep	Print lines matching a pattern
init	Create an empty git repository or reinitialize an existing one
log	Show commit logs
merge	Join two or more development histories together
mv	Move or rename a file, a directory, or a symlink
pull	Fetch from and merge with another repository or a local branch
push	Update remote refs along with associated objects

Configure Git

- Let **Git** know who you are and how to reach you:
- Two configuration files: `.gitconfig` and `.gitignore`
- `$ git config --global user.name "First Last"`
- `$ git config --global user.email "hello@world.org"`
- `$ git config --global core.editor "vi"`
- `$ cat ~/.gitconfig`

```
[user]
< Tab > name = Xiaoxu Guan
< Tab > email = xiaoxu.guan@gmail.com
[core]
< Tab > editor = vi
```

Configure Git

- We definitely don't want to keep all the files on your system under the control of **Git**;
- How can we tell **Git** to deliberately overlook certain types of files?
- Configure your `~/.gitignore` file;
- `$ cat ~/.gitignore`

```
## Files are ignored.
# generated by Fortran/C/C++ compilers.
*.o
# generated by Fortran compiler.
*.mod
# I name the executable files generated by
# Fortran/C/C++ compilers with an
# extension name .project.
*.project
```

Configure Git

```
...  
# (La)TeX  
*.log  
# (La)TeX  
*.aux  
# error messages at run time.  
*.err  
# temporary file in vim.  
*.swp  
...
```

Basically, that's all we need for the set up of **Git!**

Initialize a repository

- Clone an existing repository from other machine:
 - `git clone <repo>`
 - `git clone <repo> <directory>`
 - `$ git clone myuid@machine.name.org \`
 - `:/home/myuid/project_1`
 - **A local directory `project_1` was cloned from the remote server.**

Initialize a repository

- Clone an existing repository from other machine:
 - `git clone <repo>`
 - `git clone <repo> <directory>`
 - `$ git clone myuid@machine.name.org \`
 - `:/home/myuid/project_1`
 - A local directory `project_1` was cloned from the remote server.
- Start from scratch:
 - `$ git init`
 - `$ git init <directory>`
 - `$ pwd`
 - `$ /home/xiaoxu`
 - `$ mkdir project_1; cd project_1`

Initialize a repository

- Start from scratch:
 - ...
 - `$ git init` **OR**
 - `$ git init project_1`

Initialize a repository

- Start from scratch:
 - ...
 - `$ git init` **OR**
 - `$ git init project_1`
- In both cases, a “hidden” directory `.git` was created on `project_1`.

Initialize a repository

- Start from scratch:
 - ...
 - `$ git init` **OR**
 - `$ git init project_1`
- In both cases, a “hidden” directory `.git` was created on `project_1`.
- `$ git status`
 - **fatal: Not a git repository (or any of the parent directories):
.git**
 - **write a simple source code** `a.f90`
 - `$ git status`

Initialize a repository

- `$ git status`

On branch master

Initial commit

Untracked files:

(use "git add <file>..." to include in what will be committed)

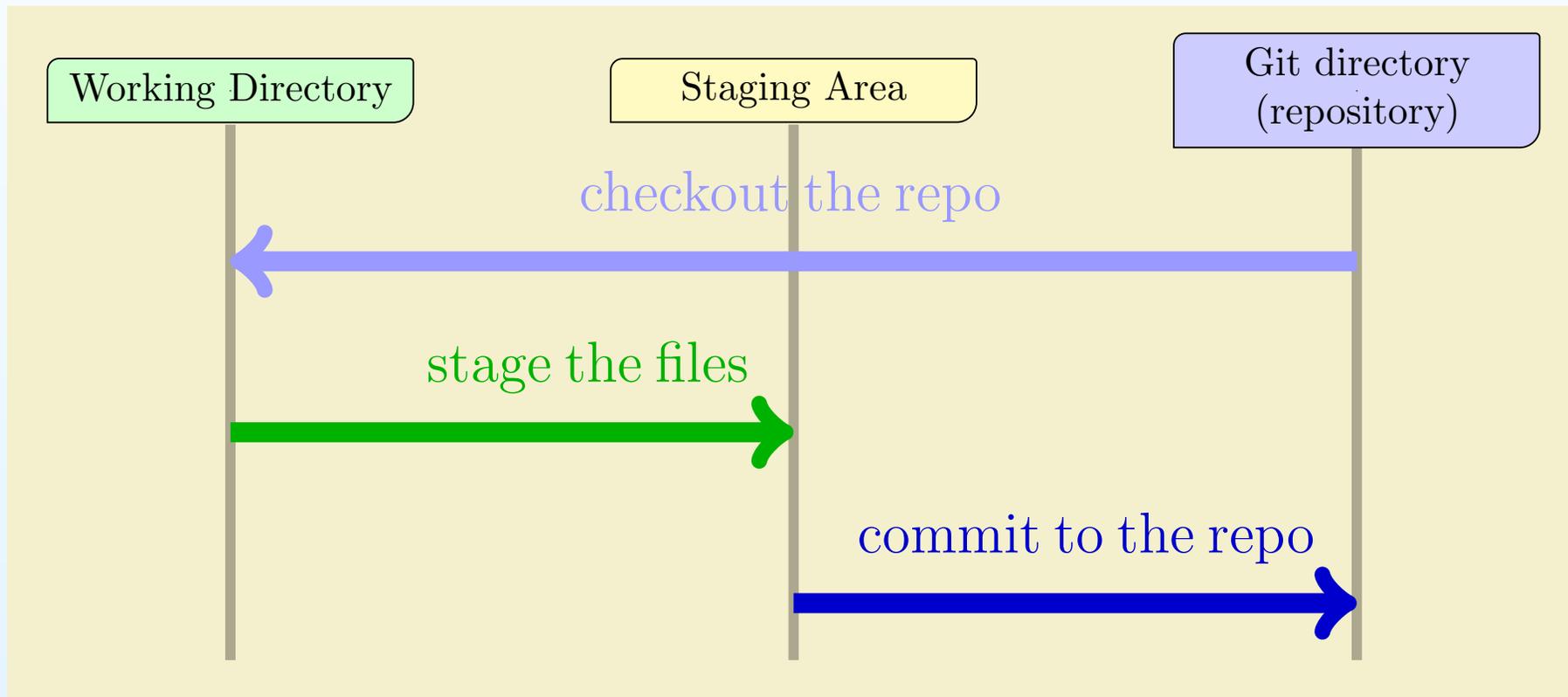
a.f90

nothing added to commit but untracked files present
(use "git add" to track)

- branch
- master
- git commit
- git add
- untracked and tracked files

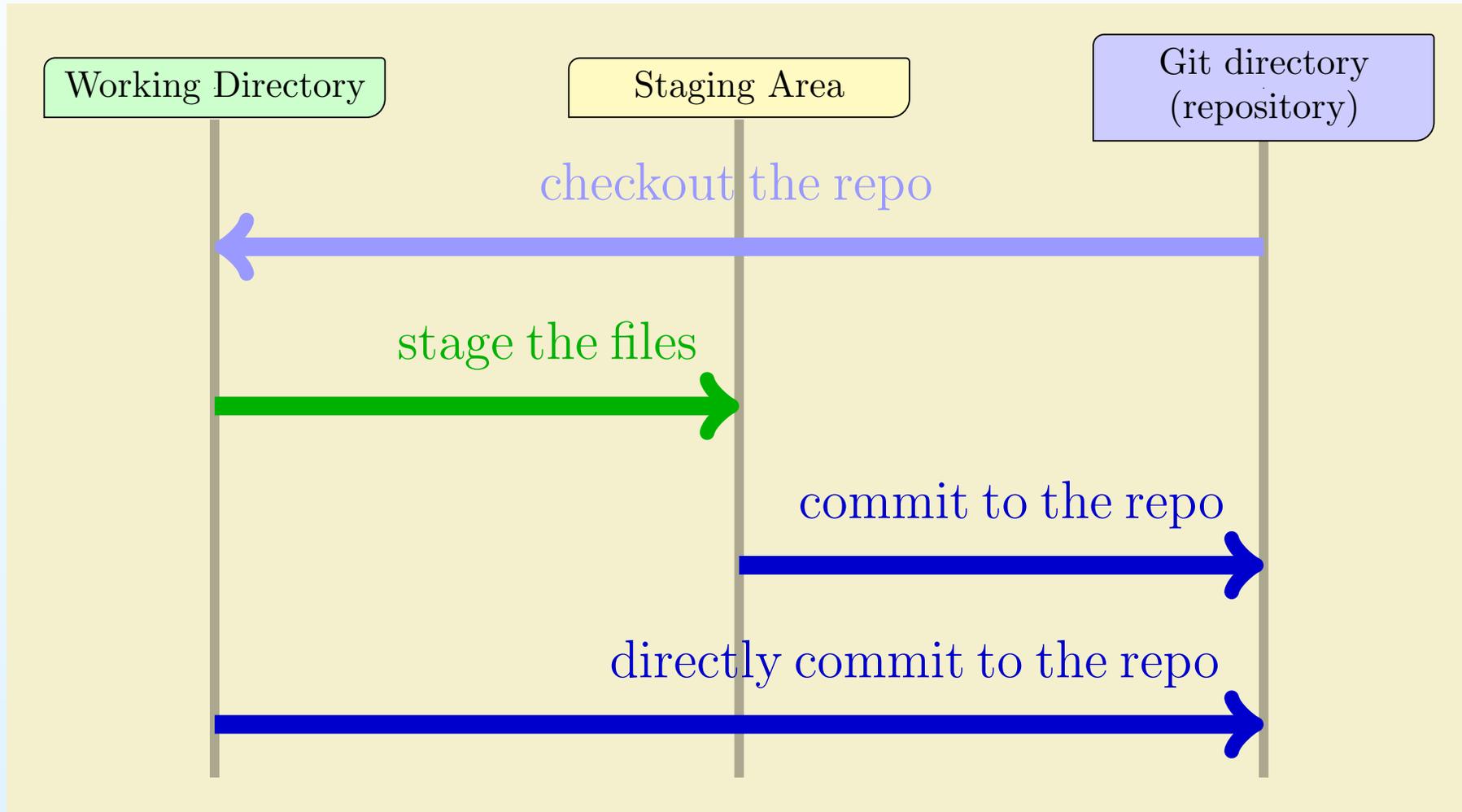
Workflow behind Git

- The **Three States**: **Working Directory**, **Staging Area**, and **Git Directory (repository)**.



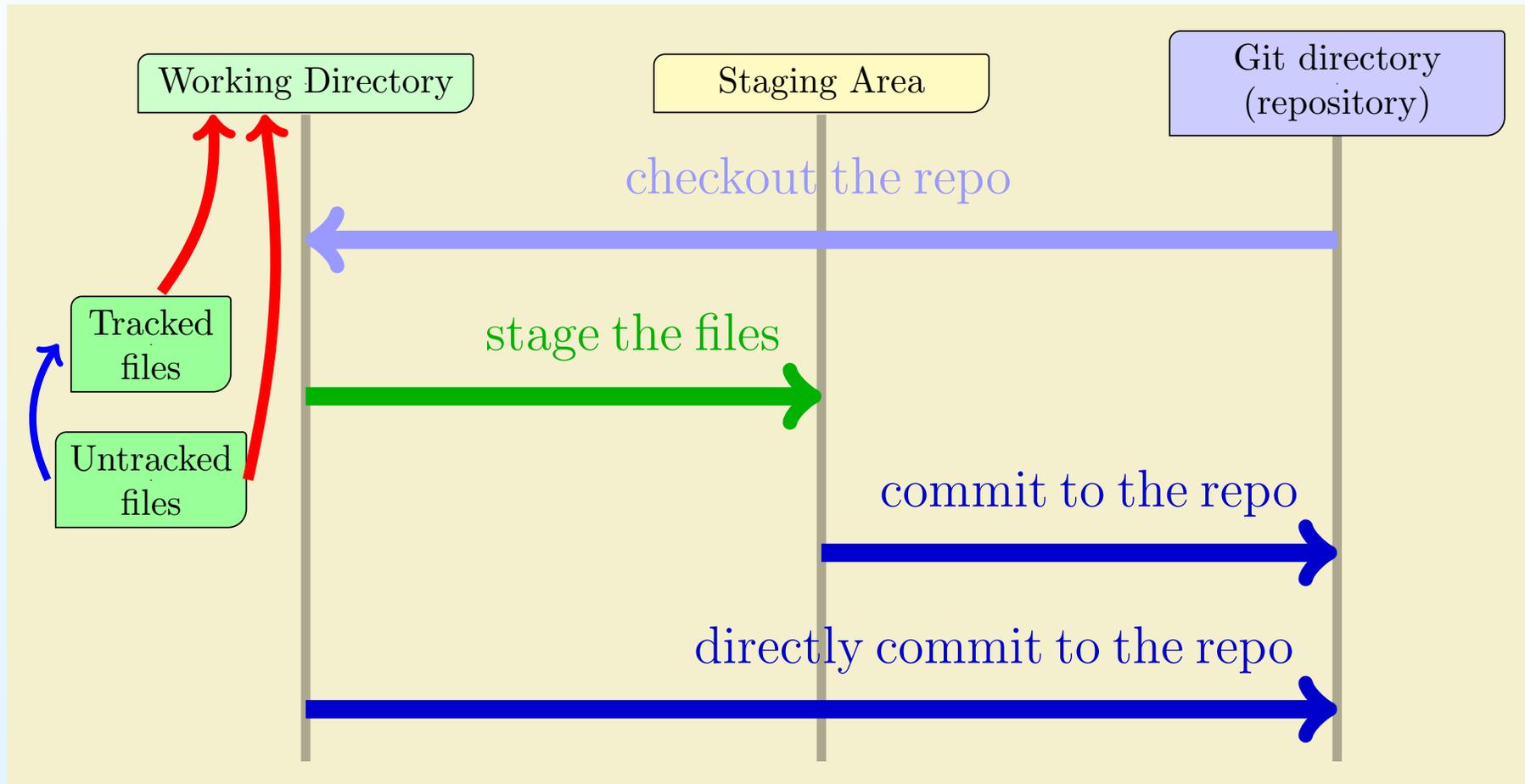
Workflow behind Git

- The **Three States**: **Working Directory**, **Staging Area**, and **Git Directory (repository)**.



Workflow behind Git

- The **Three States**: **Working Directory**, **Staging Area**, and **Git Directory (repository)**.



Workflow behind Git

- The **Three** States: **Working Directory**, **Staging Area**, and **Git Directory (repository)**.
- **Working directory**: A single copy of the **Git** repository. You can use and modify the files;
- **Staging area**: It is simply a file recording the information that you will do next time to commit the changes into the repo, and also known as the **Index**;
- **Staged files**: the files were modified and put on the stage but not committed to the repo yet;
- **Tracked and untracked files**: controlled by **Git** or not;
- **Committed files**: the files are stored in the **Git** repo;

All in One!

Initialize a repository

- `$ git status`

On branch master

Initial commit

Untracked files:

(use "git add <file>..." to include in what will be committed)

`a.f90`

nothing added to commit but untracked files present
(use "git add" to track)

- It shows the paths that have differences (1) between the **working directory** and **staged** files, the paths that have differences (2) between the **staged** files and **committed repo**, and (3) the paths for **untracked** files;

Workflow behind Git

- **Untracked** files:
- **Git** detected there is a file called a.f90, but it cannot find any previous snapshots (or committed files);
- `$ git add a.f90`
- `$ git add .`
- **# Stage everything (be careful with this).**
- `$ git status`
- What do we see now? (Untracked file → Tracked file; Changes to be committed: ...)
- This means that **Git** successfully put this file to the staging area;
- **Staging area** looks like a **buffer** between the working directory and the repo;

Workflow behind Git

- Now we can commit it into the repo;
- `$ git commit # commit the changes from staging area to the repo (an editor pops up);`
- `$ git commit -m "<descriptive message>" # the same as the above, but no editor explicitly involved;`
- We need to let **Git** (and us) know that what kind of changes are done in the commit;
- `$ git commit -a -m "<descriptive message>" # Do it in one step;`
- Commit all the changes in the working directory to the repo;
Only for the tracked files;

Git file and commit management

- List the commit history:
 - `$ git log`

Git file and commit management

- List the commit history:
 - `$ git log`
- Search a particular pattern in all messages:
 - `$ git log --grep="<pattern>"`

Git file and commit management

- List the commit history:
 - `$ git log`
- Search a particular pattern in all messages:
 - `$ git log --grep="<pattern>"`
- Search a particular author in all commits:
 - `$ git log --author="<pattern>"`

Git file and commit management

- List the commit history:
 - `$ git log`
- Search a particular pattern in all messages:
 - `$ git log --grep="<pattern>"`
- Search a particular author in all commits:
 - `$ git log --author="<pattern>"`
- Search the history for a particular file:
 - `$ git log <filename>`
 - `$ git --graph --decorate --oneline`

Git file and commit management

- List the commit history:
 - `$ git log`
- Search a particular pattern in all messages:
 - `$ git log --grep="<pattern>"`
- Search a particular author in all commits:
 - `$ git log --author="<pattern>"`
- Search the history for a particular file:
 - `$ git log <filename>`
 - `$ git --graph --decorate --oneline`
- List and view the changes (differences) introduced in each commit:
 - `$ git log -p -2`
Only list the last 2 commits;

Git file and commit management

- **Q1.** How to do list all the **tracked files** in the current working directory?
 - A simple answer: `$ git ls-files`

Git file and commit management

- **Q1.** How to do list all the **tracked files** in the current working directory?
 - A simple answer: `$ git ls-files`
- **Q2.** How to do list all the **untracked files** in the current working directory?
 - A simple solution: `$ git status -u`

Git file and commit management

- **Q1.** How to do list all the **tracked files** in the current working directory?
 - A simple answer: `$ git ls-files`
- **Q2.** How to do list all the **untracked files** in the current working directory?
 - A simple solution: `$ git status -u`
- **Q3.** How can I list all the commits in the current branch?
 - `$ git log` # shows the commit history;

Git file and commit management

- **Q1.** How to do list all the **tracked files** in the current working directory?
 - A simple answer: `$ git ls-files`
- **Q2.** How to do list all the **untracked files** in the current working directory?
 - A simple solution: `$ git status -u`
- **Q3.** How can I list all the commits in the current branch?
 - `$ git log` # shows the commit history;
- **Q4.** How can I **remove** files from Git?
 - `$ git rm myfile.f90` # this **not only** removes the file from the repo, **but** also removes the file from the local working directory;
 - `$ git rm --cached myfile.f90` # this time it **only** removes the file from the repo, but without deleting the file from the local working directory;

Git file and commit management

- **Q5.** How can I **recover** the file I **deleted** by `git rm`? The answer to this question may not be simple!
 - `$ git log --diff-filter=D --summary`
 # prints all the commits that deleted files. Or if you know the deleted filename,
 - `$ git rev-list -n 1 HEAD -- b.f90`
 # prints the only commit tag (a 40-digit hexadecimal SHA-1 code and unique commit ID) that deleted the file;
 - `0b7b587b46b19a4903b1ad35942dbed965fbddae`
 # this's the commit tag;
 - `$ git checkout "commit_tag^" b.f90`

Git file and commit management

- **Q5.** How can I **recover** the file I **deleted** by `git rm`? The answer to this question may not be simple!
 - `$ git log --diff-filter=D --summary`
prints all the commits that deleted files. Or if you know the deleted filename,
 - `$ git rev-list -n 1 HEAD -- b.f90`
prints the only commit tag (a 40-digit hexadecimal SHA-1 code and unique commit ID) that deleted the file;
 - `0b7b587b46b19a4903b1ad35942dbed965fbddae`
this's the commit tag;
 - `$ git checkout "commit_tag^" b.f90`
- Don't panic because **Git** has you covered!

Git file and commit management

- The **git checkout** command: to checkout some **files**, **commits**, and **branches**;

Git file and commit management

- The **git checkout** command: to checkout some **files**, **commits**, and **branches**;
- `$ git checkout master`
`# returns the master branch.`

Git file and commit management

- The **git checkout** command: to checkout some **files, commits, and branches**;
- `$ git checkout master`
`# returns the master branch.`
- `$ git checkout <commit_tag>`
`# returns to the exact status of commit_tag under the condition;`

Git file and commit management

- The **git checkout** command: to checkout some **files, commits, and branches**;
- `$ git checkout master`
returns the `master` branch.
- `$ git checkout <commit_tag>`
returns to the exact status of `commit_tag` under the condition;
- Local working directory can only keep one status of the repo;

Git file and commit management

- The **git checkout** command: to checkout some **files, commits, and branches**;
- `$ git checkout master`
returns the `master` branch.
- `$ git checkout <commit_tag>`
returns to the exact status of `commit_tag` under the condition;
- Local working directory can only keep one status of the repo;
- If we want to go back to the one status of the previous commits, for the safety reasons we'd better commit the current changes to the repo;

Git file and commit management

- The **git checkout** command: to checkout some **files, commits, and branches**;
- `$ git checkout master`
returns the `master` branch.
- `$ git checkout <commit_tag>`
returns to the exact status of `commit_tag` under the condition;
- Local working directory can only keep one status of the repo;
- If we want to go back to the one status of the previous commits, for the safety reasons we'd better commit the current changes to the repo;
- Otherwise, you will see something like **HEAD detached at `commit_tag`**;

Git file and commit management

- Let's assume that we committed all the current changes to the repo; and we want to **go back** to the one status of the **previous** commits in the repo's history;

Git file and commit management

- Let's assume that we committed all the current changes to the repo; and we want to **go back** to the one status of the **previous** commits in the repo's history;
- `$ git log --oneline`
prints a short list for the commit history;

Git file and commit management

- Let's assume that we committed all the current changes to the repo; and we want to **go back** to the one status of the **previous** commits in the repo's history;
- `$ git log --oneline`
prints a short list for the commit history;
- `$ git log --since=4.weeks`
prints a list for the commit history in the last 4 weeks;

Git file and commit management

- Let's assume that we committed all the current changes to the repo; and we want to **go back** to the one status of the **previous** commits in the repo's history;
- `$ git log --oneline`
prints a short list for the commit history;
- `$ git log --since=4.weeks`
prints a list for the commit history in the last 4 weeks;
- `$ git checkout <commit_tag>`
returns to the exact status of `commit_tag`;

Git file and commit management

- Let's assume that we committed all the current changes to the repo; and we want to **go back** to the one status of the **previous** commits in the repo's history;
- `$ git log --oneline`
prints a short list for the commit history;
- `$ git log --since=4.weeks`
prints a list for the commit history in the last 4 weeks;
- `$ git checkout <commit_tag>`
returns to the exact status of `commit_tag`;
- `$ git checkout <commit_tag> a.f90`
In this case, I'm only interested in one of the particular files,
`a.f90` in the `commit_tag` without checking out the entire
previous commit;

Git file and commit management

A more advanced topic!

- Once we **committed** all the changes to the repo, **Git** will never lose them!

Git file and commit management

A more advanced topic!

- Once we **committed** all the changes to the repo, **Git** will never lose them!
- Changing the **git** history (**be careful** again);

Git file and commit management

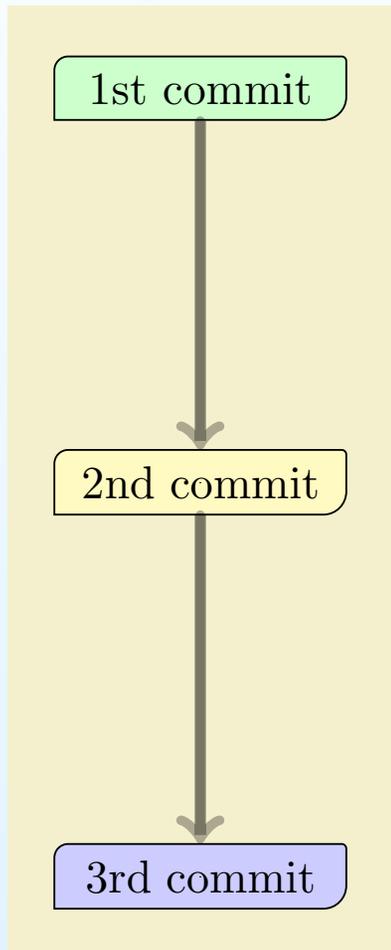
A more advanced topic!

- Once we **committed** all the changes to the repo, **Git** will never lose them!
- Changing the **git** history (**be careful** again);
- `$ git commit --amend`
 - Instead of committing as a new snapshot in the repo, it combines the current changes in the staging area to the last commit (the most recent one).

Git file and commit management

A more advanced topic!

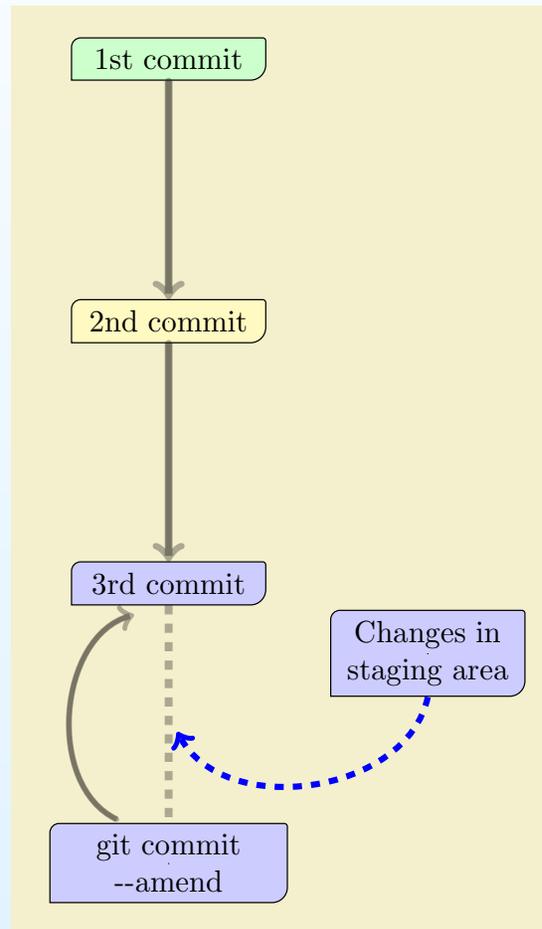
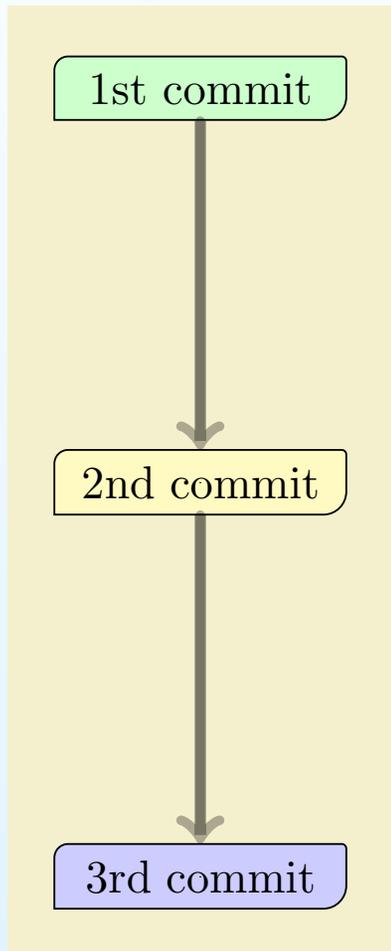
- `$ git commit --amend`



Git file and commit management

A more advanced topic!

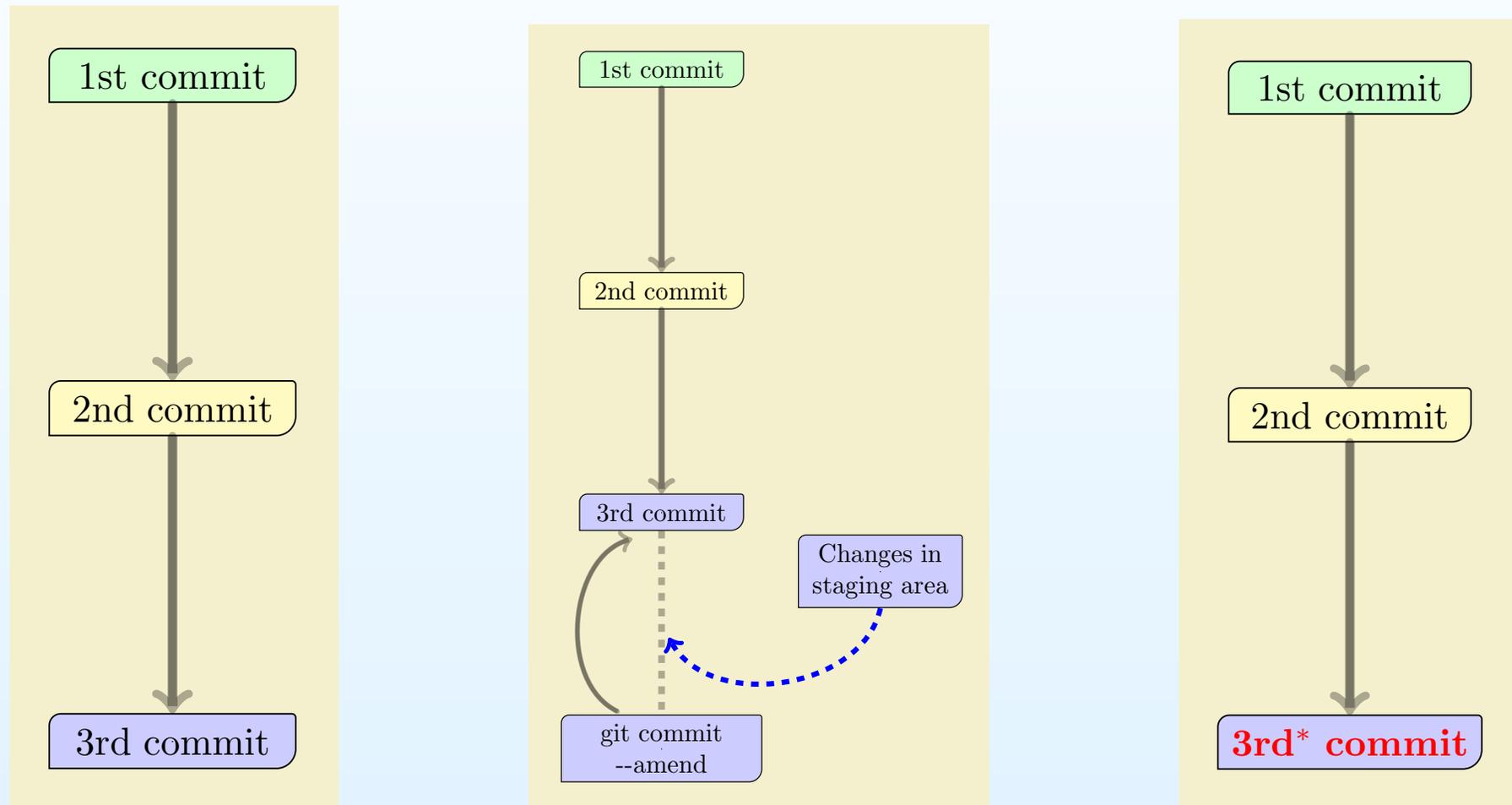
- `$ git commit --amend`



Git file and commit management

A more advanced topic!

- `$ git commit --amend #` overwrites the previous one.



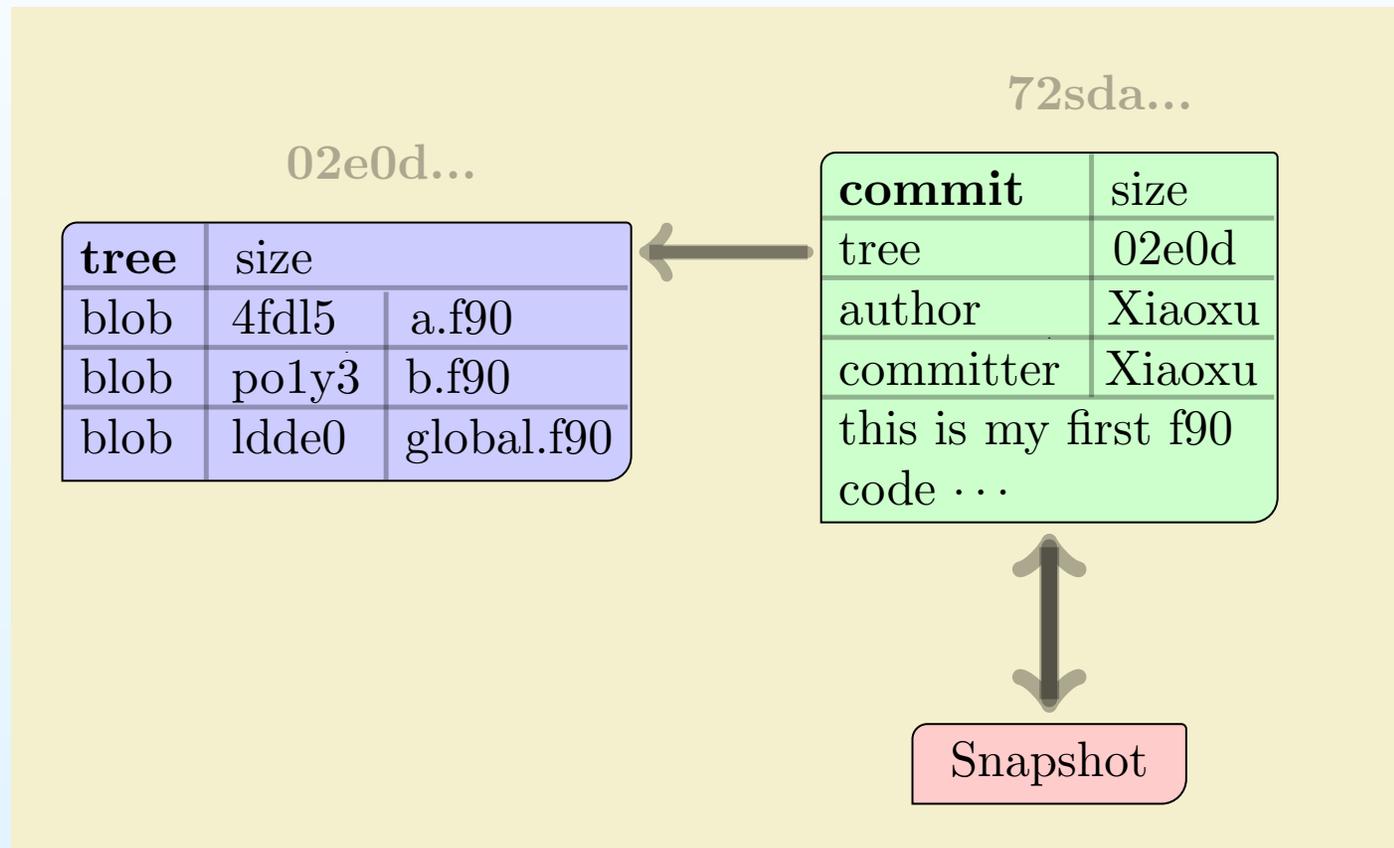
Git Branching

• Why do we need branches and what is a Git branch?

- We want to do many different things at the same time on the same working directory/repo;
- **Git** encourages using branches — one of the features focusing the non-linear development;
- Multiple branches in a working directory/repo;
- Test different ideas or algorithms, ...
- Don't be confused with sub-directory;
- Create and merge branches;
- The default branch is called `master`;
- The power of **Git** largely relies on the concept of **branch** and the way how **Git** manipulates them;
- A “**killer** feature” in **Git**;

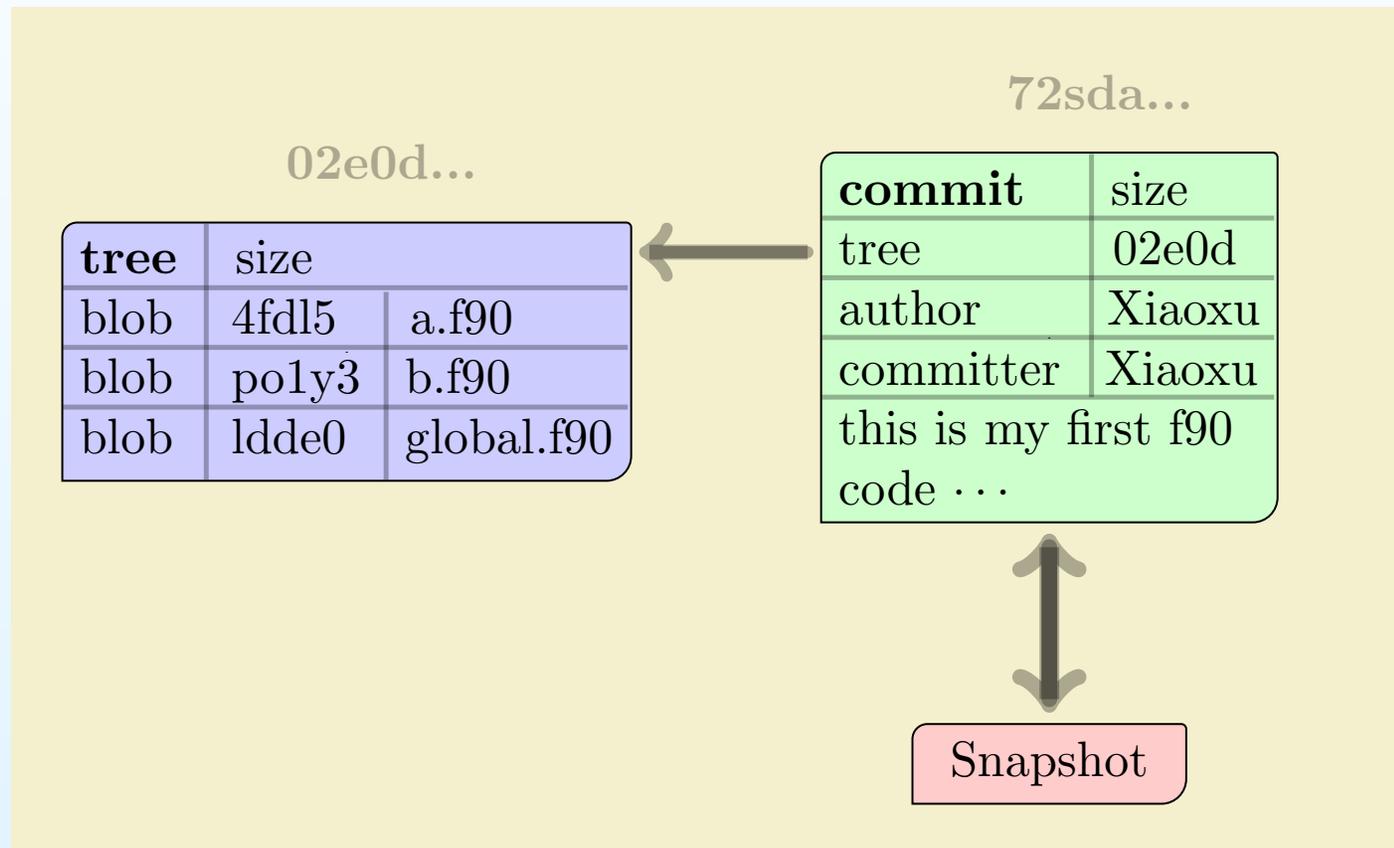
Git Branching

- How does Git store data?



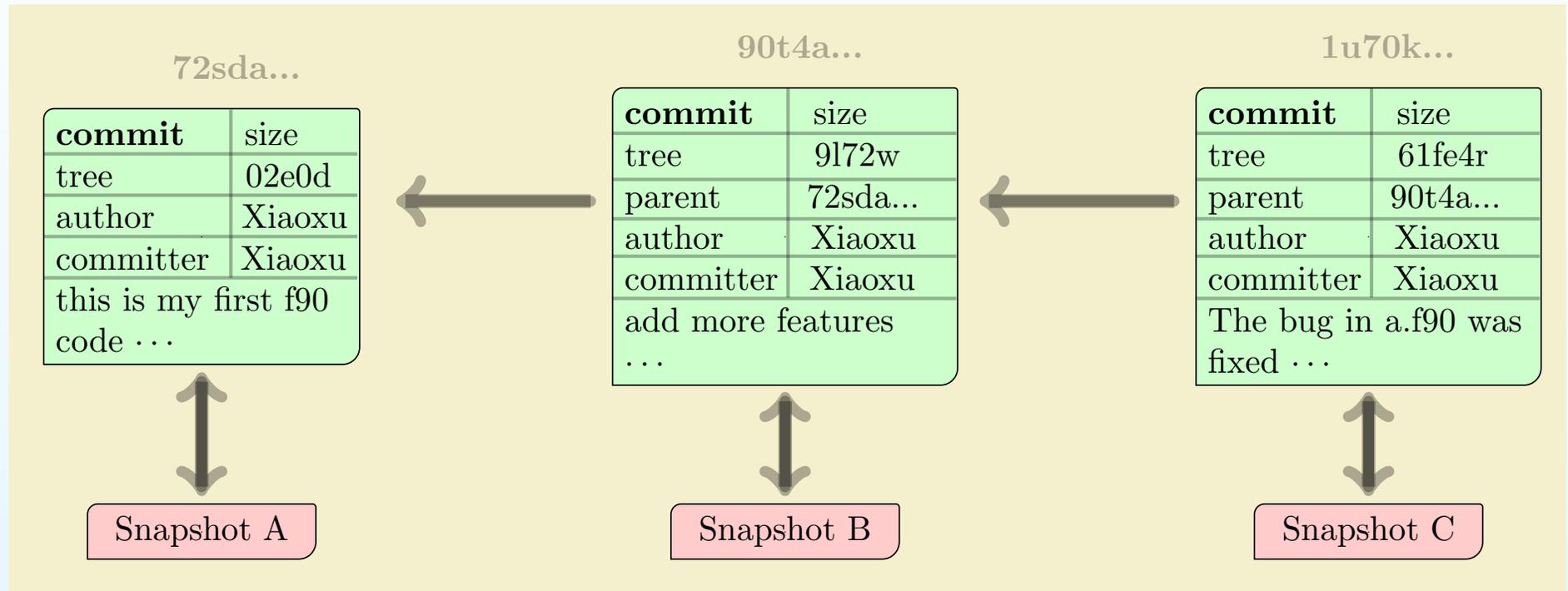
Git Branching

- How does Git store data?
- **Data** includes the file data and meta data;



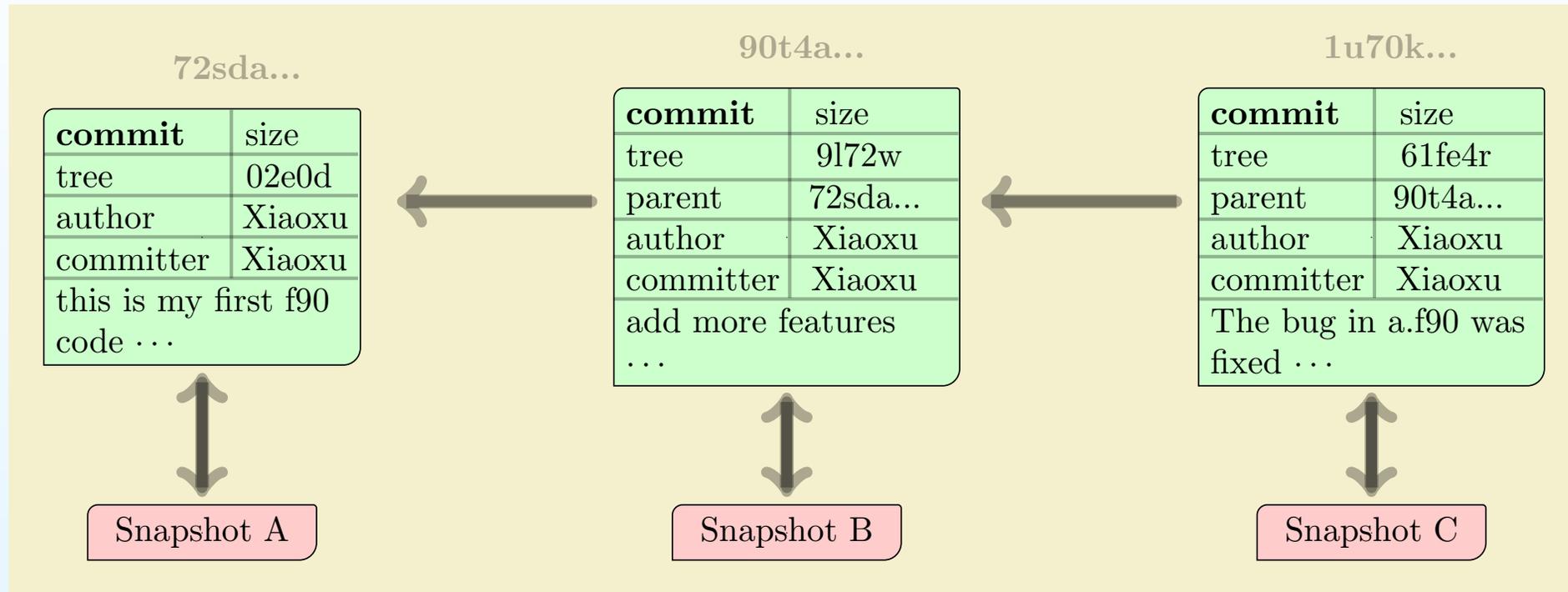
Git Branching

- After several commits, it may look like this:



Git Branching

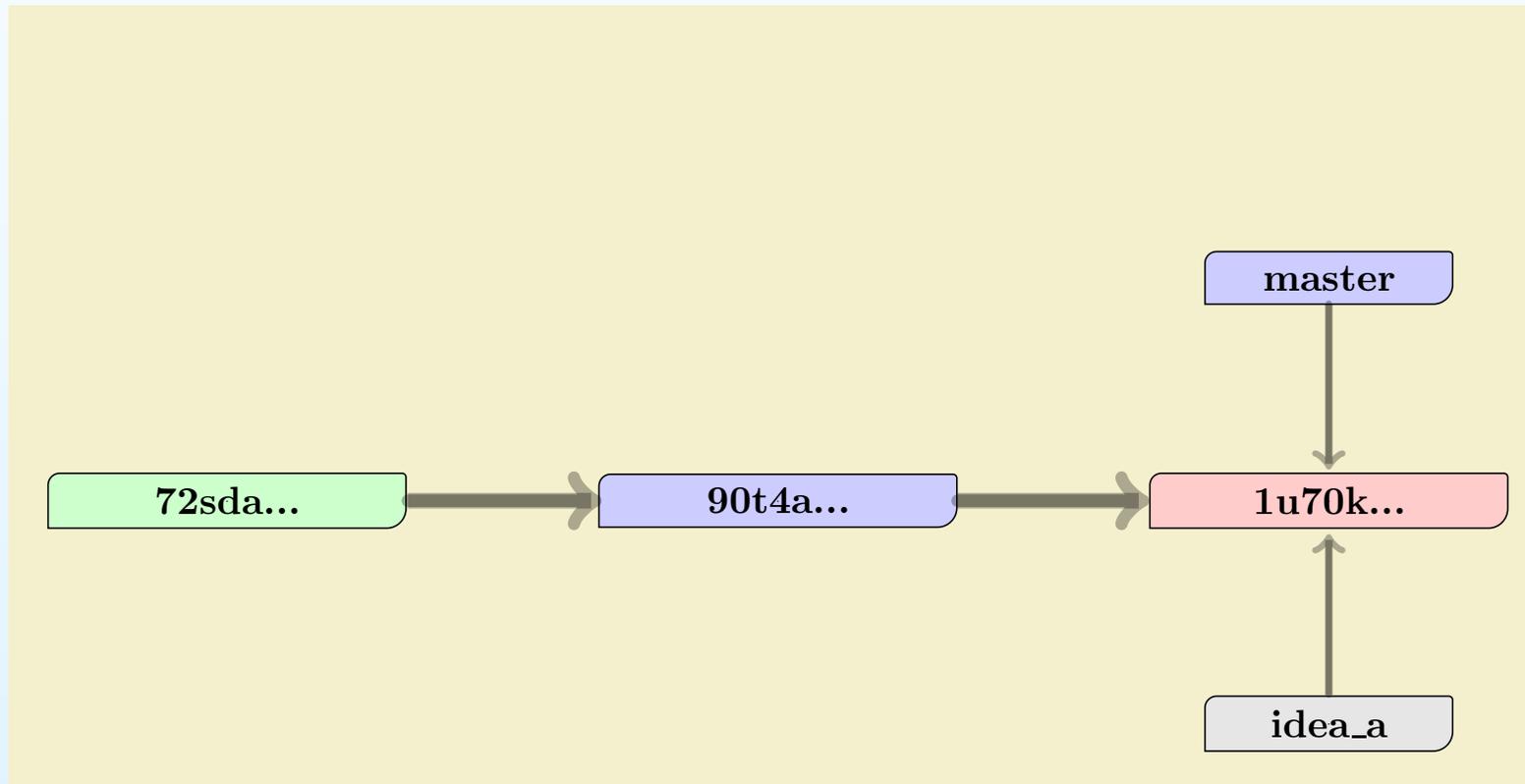
- After several commits, it may look like this:



- A **Git branch** is a lightweight movable **pointer** to one of the commits. Whenever we make a new commit, the pointer moves forward.

Git Branching

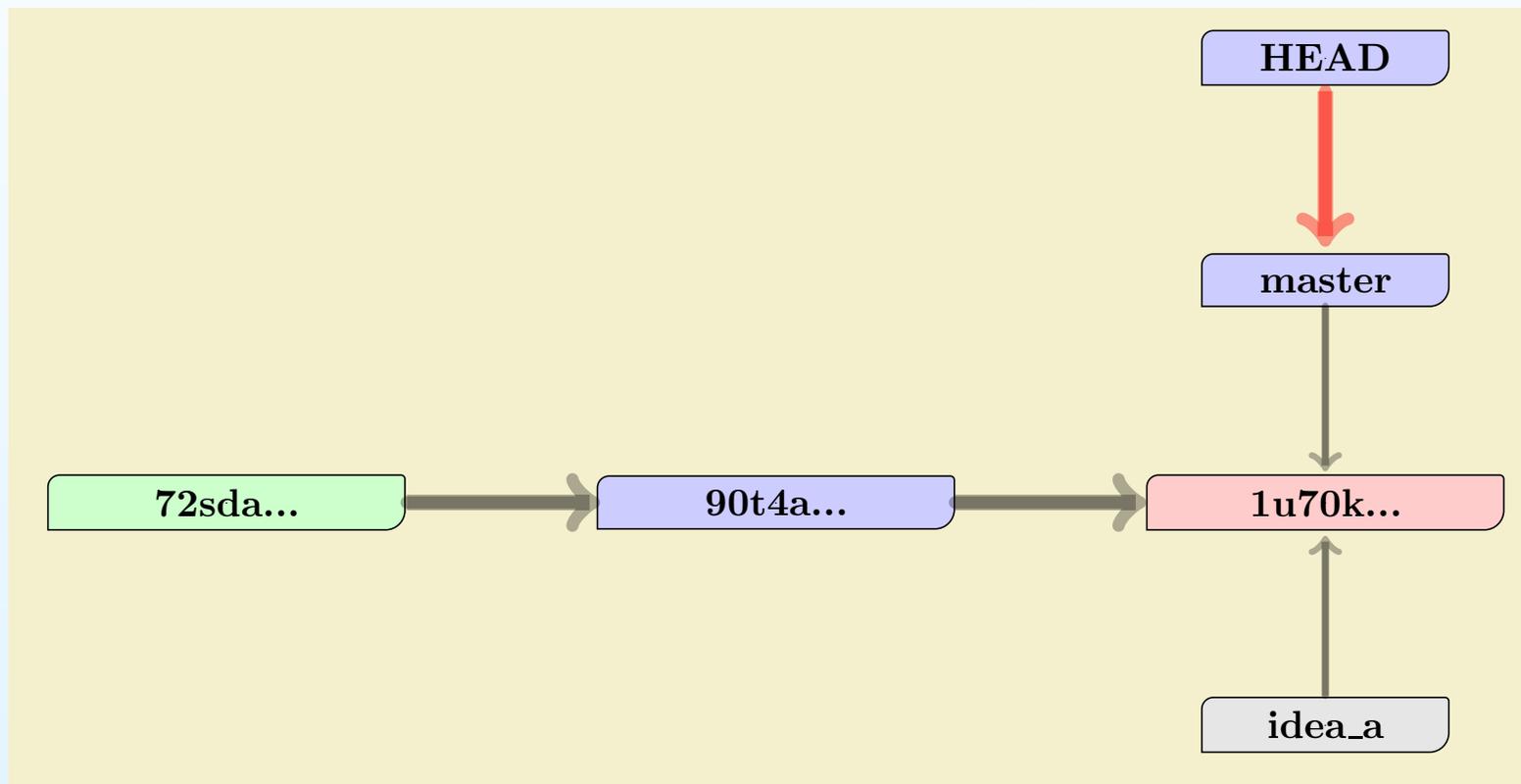
- Let's create a new branch:
 - \$ git branch # which branch am I on?
 - \$ git branch idea_a # creates a branch called # "idea_a";



Git Branching

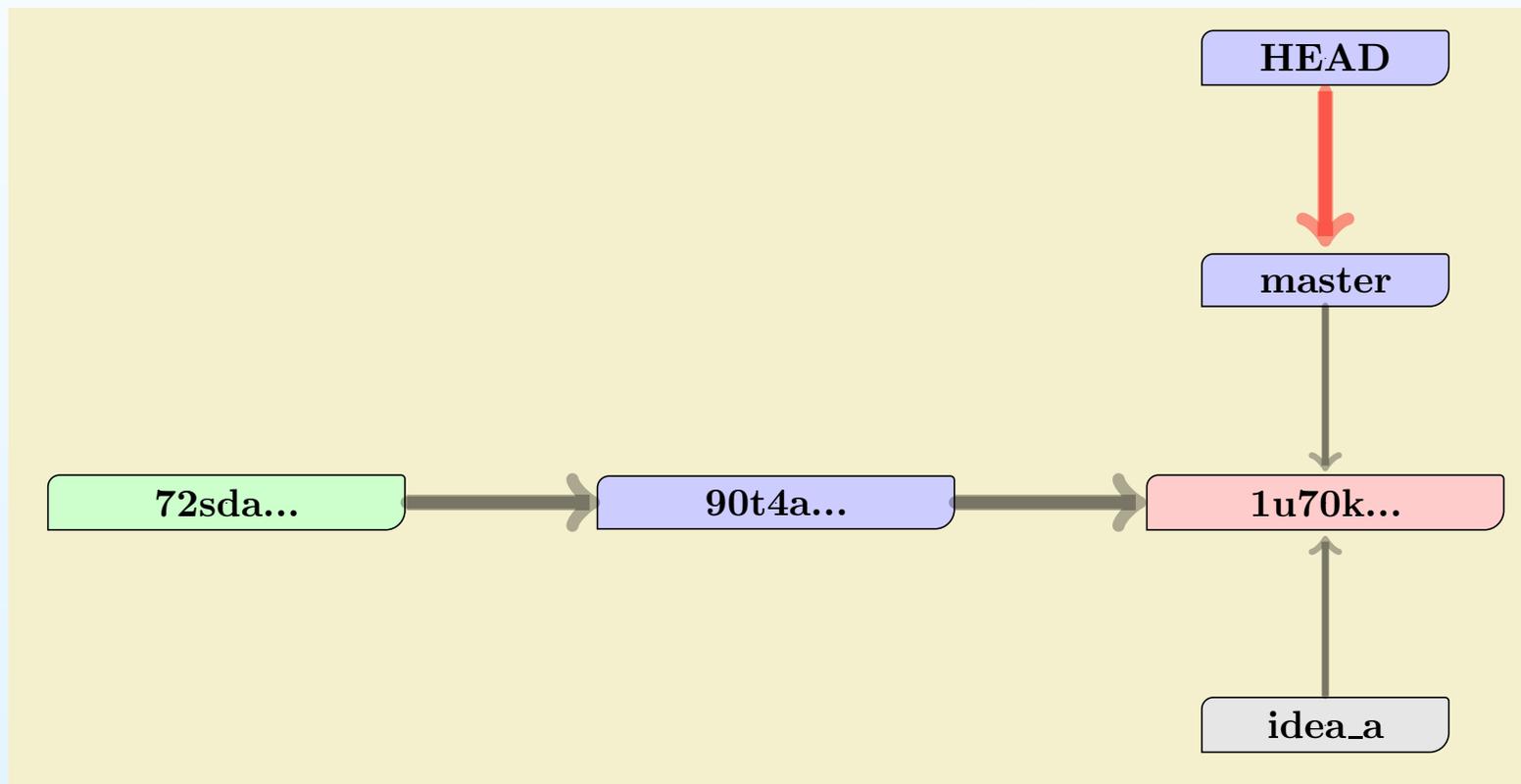
- Let's create a new branch:

- `$ git branch # which branch am I on?`
- `$ git branch idea_a # creates a branch called # "idea_a";`



Git Branching

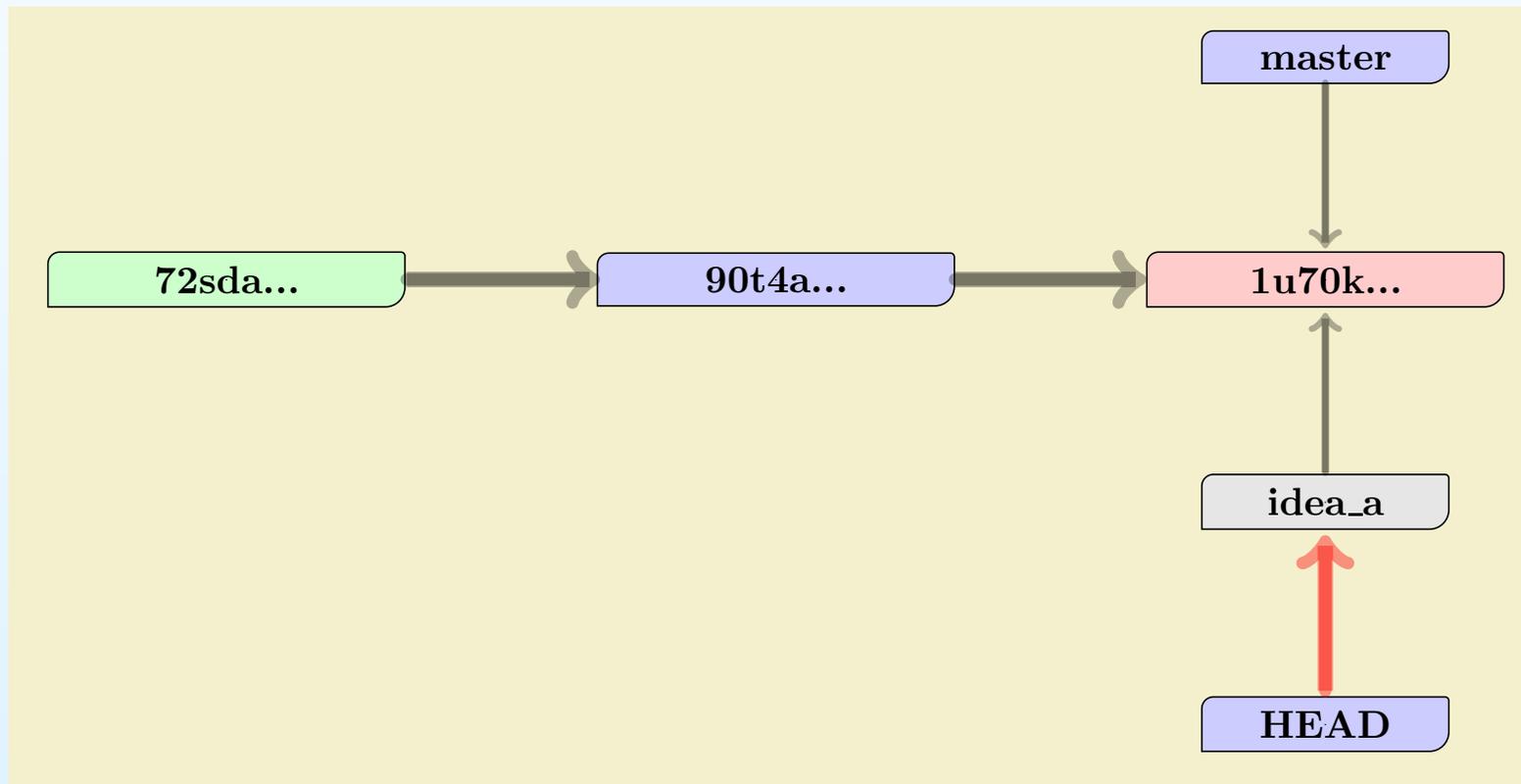
- Let's create a new branch:
 - \$ git branch # which branch am I on?
 - \$ git branch idea_a # creates a branch called # "idea_a";



- **Git HEAD** is a special pointer that points to which branch you are on;

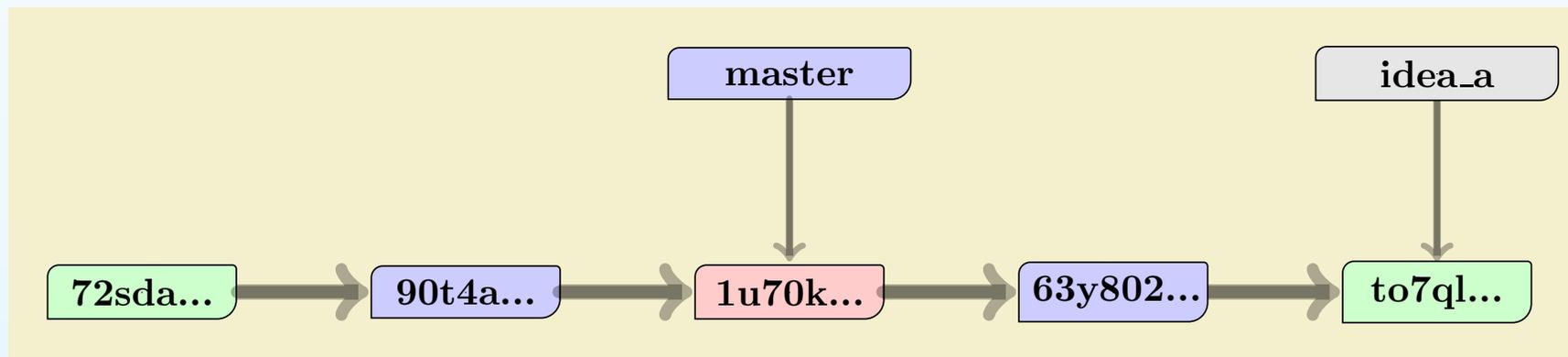
Git Branching

- Switch to another branch:
 - `$ git checkout idea_a # change to the branch idea_a;`
 - We need to commit the changes on the previous branch before changing to a new branch;



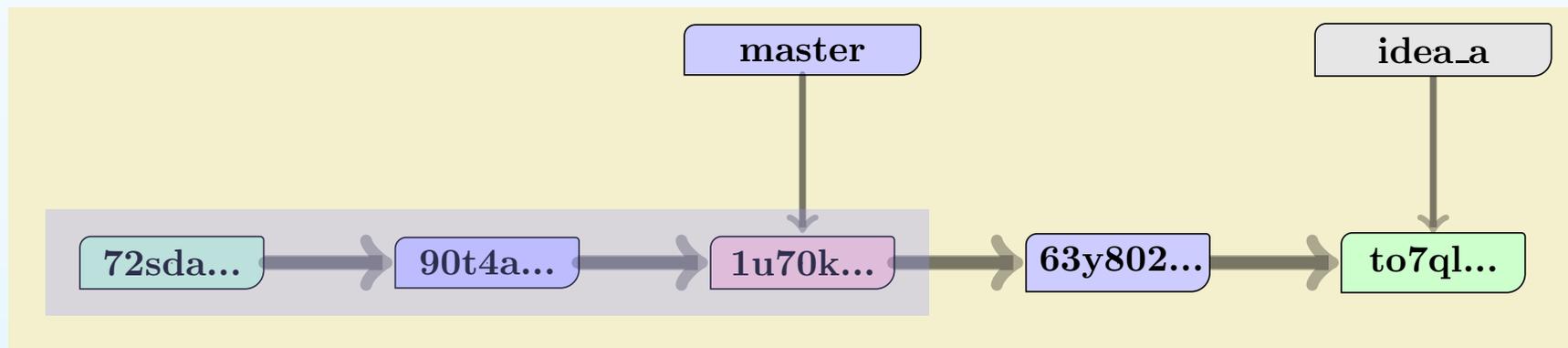
Git Branching

- Create and switch to another branch at the same time:
 - `$ git checkout -b idea_a # create & change to the branch idea_a;`
 - Let's work on the branch `idea_a` for now. After several commits, the entire branch may look like this:



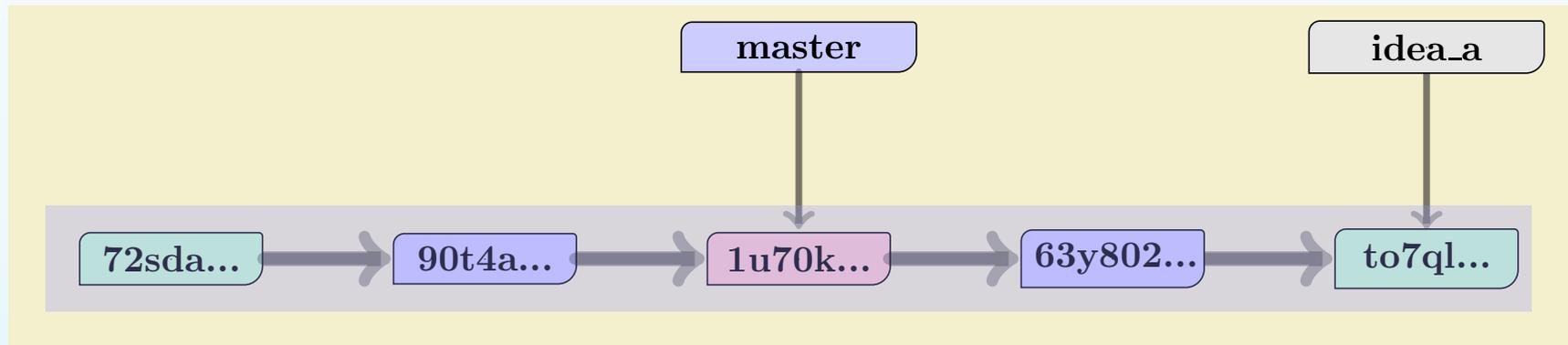
Git Branching

- Create and switch to another branch at the same time:
 - `$ git checkout -b idea_a # create & change to the branch idea_a;`
 - Let's work on the branch `idea_a` for now. After several commits the entire branches may look like this:



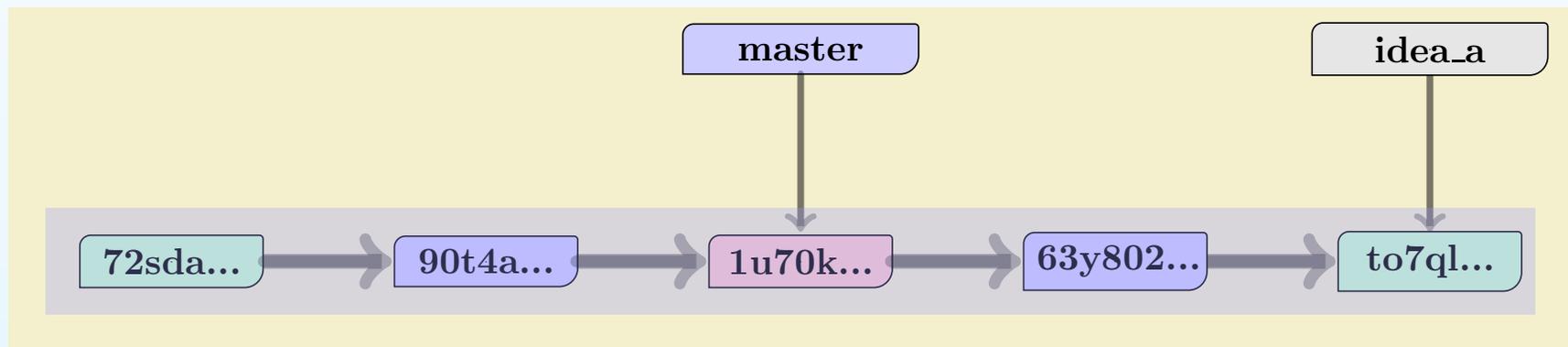
Git Branching

- Create and switch to another branch at the same time:
 - `$ git checkout -b idea_a # create & change to the branch idea_a;`
 - Let's work on the branch `idea_a` for now. After several commits the entire branches may look like this:



Git Branching

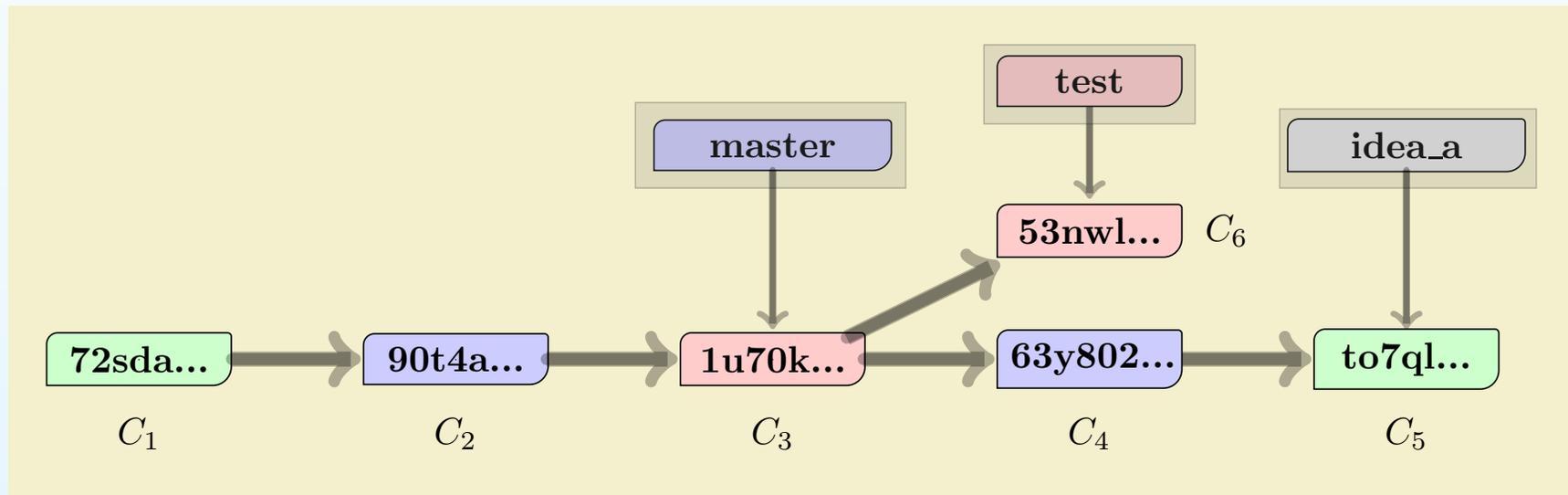
- Create and switch to another branch at the same time:
 - `$ git checkout -b idea_a # create & change to the branch idea_a;`
 - Let's work on the branch `idea_a` for now. After several commits the entire branches may look like this:



- In general, different branches take different paths of commits. This allows us to test different things (ideas, algorithms, etc.) on the same directory without interfering the other files;

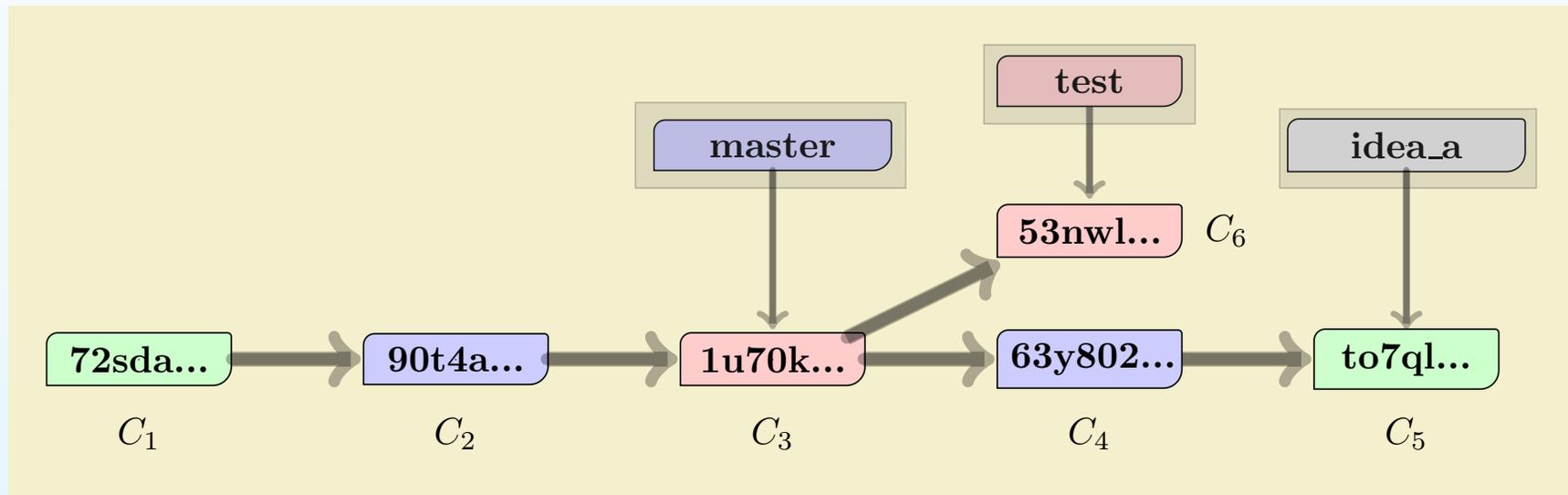
Git Branching

- Create the third branch and merge it with `master`:
 - Let's say I have worked on the branch `idea_a` for a while, and I have to go back to the `master` branch to fix bugs;
 - Create a `test` branch to debug the code;



Git Branching

- Create the third branch and merge it with `master`:
 - Let's say I have worked on the branch `idea_a` for a while, and I have to go back to the `master` branch to fix bugs;
 - Create a `test` branch to debug the code;



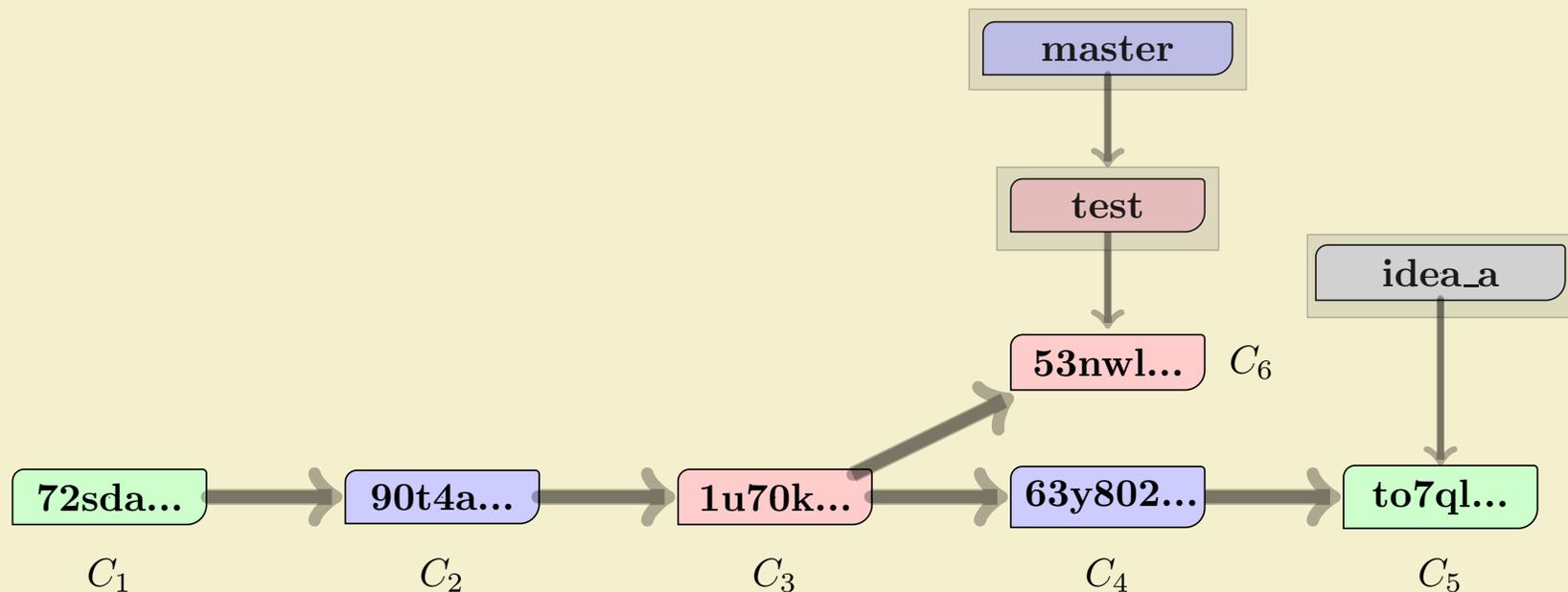
- Once I have fixed the bugs, I want to merge the branch `test` to the branch `master`;

Git Branching

- `$ git checkout master`

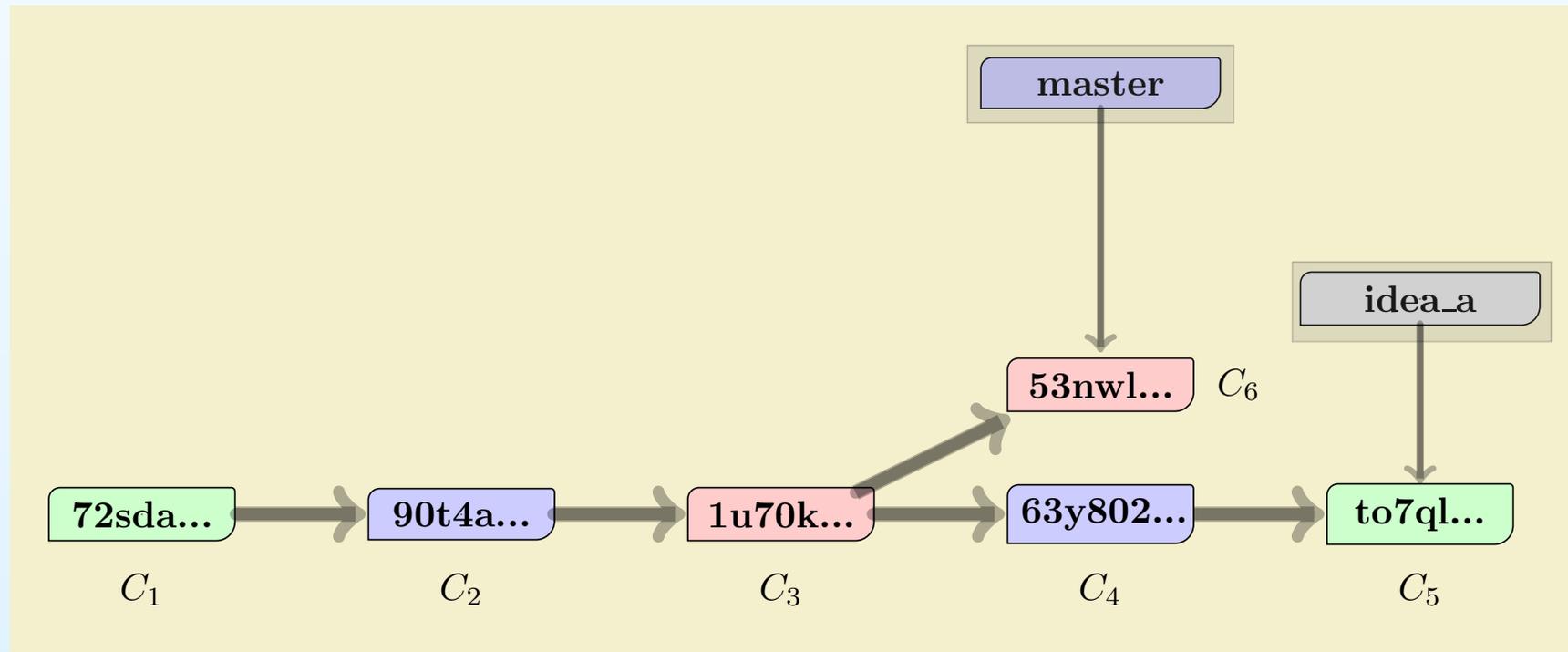
Git Branching

- \$ git checkout master
- \$ git merge test
 - We see "... Fast forward ...";
 - We want to merge an upstream branch to the `master`, so the pointer `HEAD` needs to move forward from its current position;



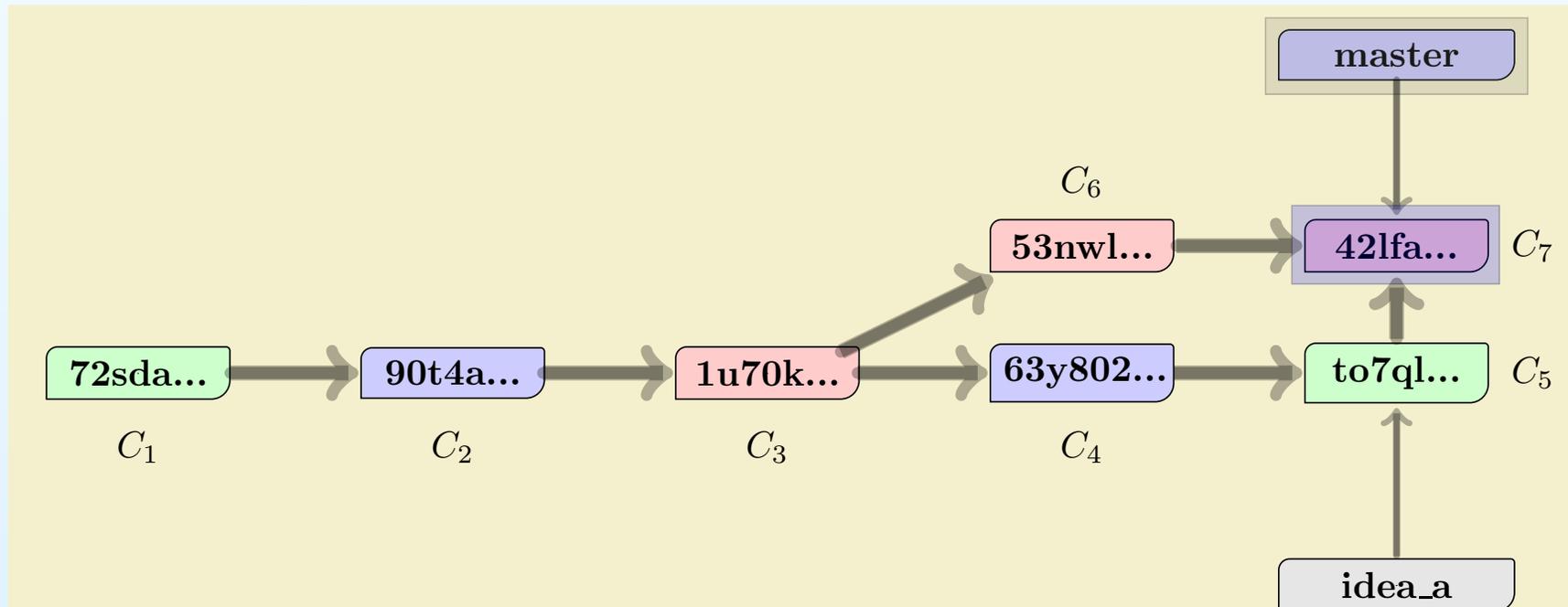
Git Branching

- I don't need the branch `test` anymore and want to remove it;
- `$ git branch -d test # Delete a branch;`
- `$ git branch # Now we have two branches;`
 - The pointer `HEAD` points to the `master` branch;
 - `master: C1, C2, C3, and C6`
 - `idea_a: C1, C2, C3, C4, and C5`



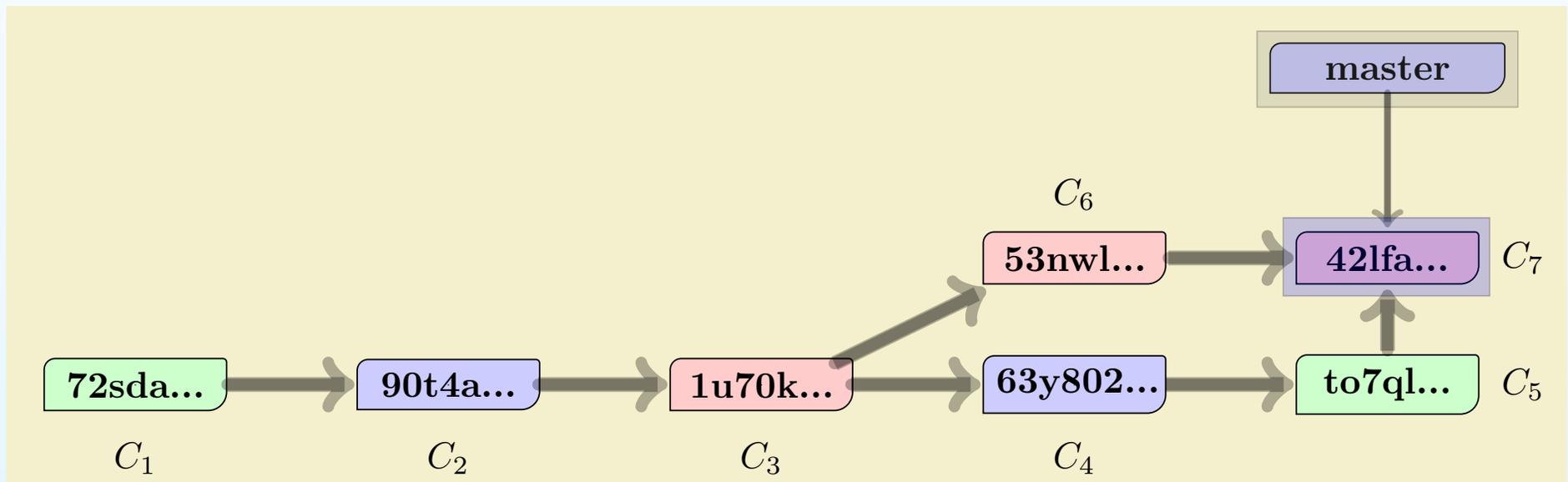
Git Branching

- Now I want to merge the `idea_a` into the `master` branch;
- `$ git checkout master # Back to the master branch;`
- `$ git merge idea_a # Merge idea_a into the master;`
 - It makes no difference with the merging of the `test` to the `master`;
 - However, a new commit (C_7) will be automatically created without any extra command involved;



Git Branching

- Therefore, C_7 has two parents (C_5 and C_6);
- Now we can safely remove the branch `idea_a` by
`$ git branch -d idea_a`



- So far, it seems very good. However, what about the potential merging **conflicts**?

Git Branching

- Sometimes we might have issues with merging branches;
- If we modify the same code or same parts of different branches, **Git** will complain about **conflicts**. It tells you which files are in conflict;
- If this happens, the merging process only attempted to merge the files that are **not** in conflict (merged/unmerged);
- The **developer** needs to fix the conflict issues before merging; we have to decide how to **keep/modify** the files in conflict; that's our choice, instead of **Git**;
- An example;

Collaborating through Git

- Where would you host the code or data for your team with multiple developers?
- Git repositories on servers:



<https://bitbucket.org>



<https://github.com>



...

<https://www.fogcreek.com/kiln>

- Note the different user policies;

Collaborating through Git

- For instance, **Git** on [Bitbucket](#) or [GitHub](#);
- Third-party servers that support **private/public** accounts;
- Let's say you received an invitation from [Bitbucket](#):
- `$ git clone
git@bitbucket.org:xiaoxu_guan/helium-fedvr.git`
- **Clone** a copy from a machine to another machine:
- `$ git clone
guan@stampede.tacc.utexas.edu:
/home1/01046/guan/Helium-FEDVR-2014
Helium-FEDVR-2014`
- The entire development history will be included in your local directory;

Collaborating through Git

- `$ git remote`
- `$ git remote -v`

```
origin guan@git.example.com:
/home/guan/Helium-FEDVR-2014 (fetch)
origin guan@git.example.com:
/home/guan/Helium-FEDVR-2014 (push)
```
- **Git supports multiple remotes;**
 - `$ git remote add second`

```
git://github.com/bob/project_w.git
```
 - Now we have two remotes that we can “`fetch`” from and “`push`” to each remote server;

Collaborating through Git

- How to fetch and pull from a remote server?
- `$ git fetch [server_name]`
- `$ git fetch origin` # Depends on which remote server
you want to fetch from;
- Remember that **Git** fetches any new work from the server to your local directory from the last time you have cloned or fetched;
- It's safe to fetch any new files/data from the server, no matter what you have been working on;
- `git fetch` does not automatically merge the new data/files with anything you have been working on;
- The developer has to **manually** merge the new files with your local files;

A summary

```
[xiaoxu@smic1 ~]$ git
usage: git [--version] [--exec-path[=GIT_EXEC_PATH]] [--html-path]
        [-pl--paginatel--no-pager] [--no-replace-objects]
        [--bare] [--git-dir=GIT_DIR] [--work-tree=GIT_WORK_TREE]
        [--help] COMMAND [ARGS]
```

The most commonly used git commands are:

- ✓ add Add file contents to the index
- ✓ bisect Find by binary search the change that introduced a bug
- ✓ branch List, create, or delete branches
- ✓ checkout Checkout a branch or paths to the working tree
- ✓ clone Clone a repository into a new directory
- ✓ commit Record changes to the repository
- diff Show changes between commits, commit and working tree, etc
- ✓ fetch Download objects and refs from another repository
- grep Print lines matching a pattern
- ✓ init Create an empty git repository or reinitialize an existing one
- ✓ log Show commit logs
- ✓ merge Join two or more development histories together
- mv Move or rename a file, a directory, or a symlink
- pull Fetch from and merge with another repository or a local branch
- push Update remote refs along with associated objects

Further Reading

- *Pro Git*, S. Chacon, Apress (2010)
- *Version Control with Git*, J. Loeliger (O'reilly, 2009)
- Official online documentation
<https://git-scm.com/doc>
- A very good Git tutorial from Atlassian
<https://www.atlassian.com/git/tutorials>

Questions?

`sys-help@loni.org`