University of Basel
Department of Mathematics and Computer Science

Distributed Programming and Internet (“DPI”)

Ali Ajorian (ali.ajorian@unibas.ch)

26. March 2025:

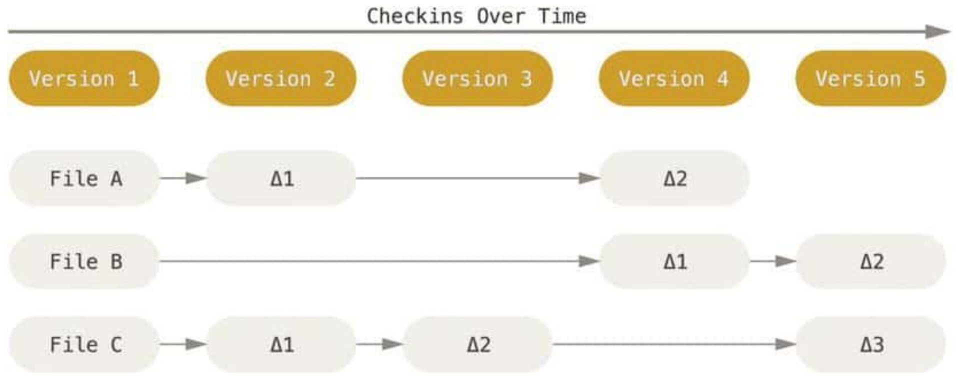
Git Internals

Version Controls

Version Control System (VCS)

A system for tracking changes made to digital assets over time

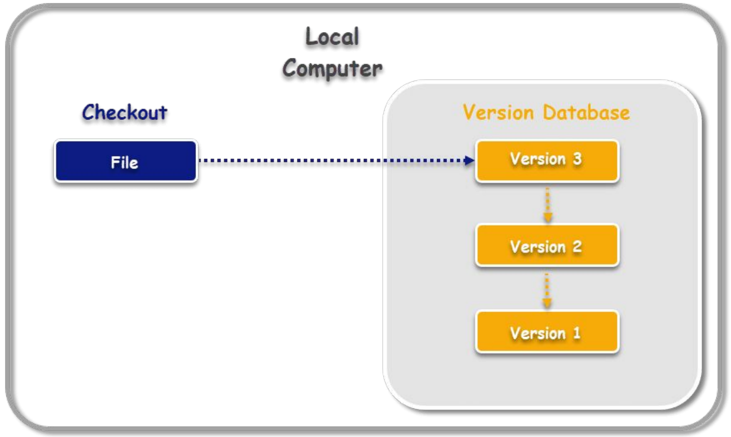
- **Most of them** store the **difference** between versions



https://www.toolsqa.com/git/what-is-git/

Local VCS

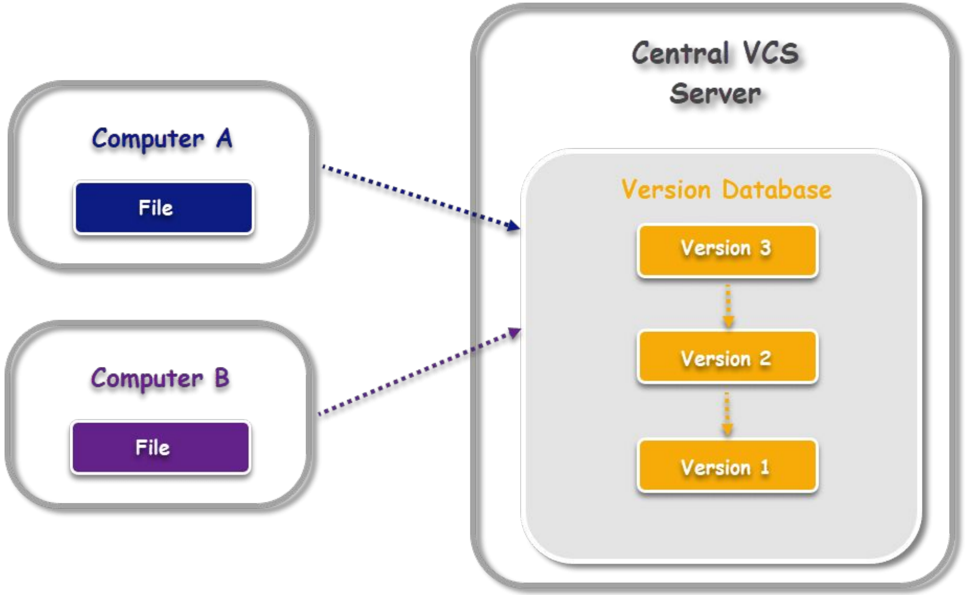
Different versions of a file on a computer



https://www.toolsqa.com/git/distributed-version-control-systems/

Remote Central VCS

Client-Server model



A “natural” model ?

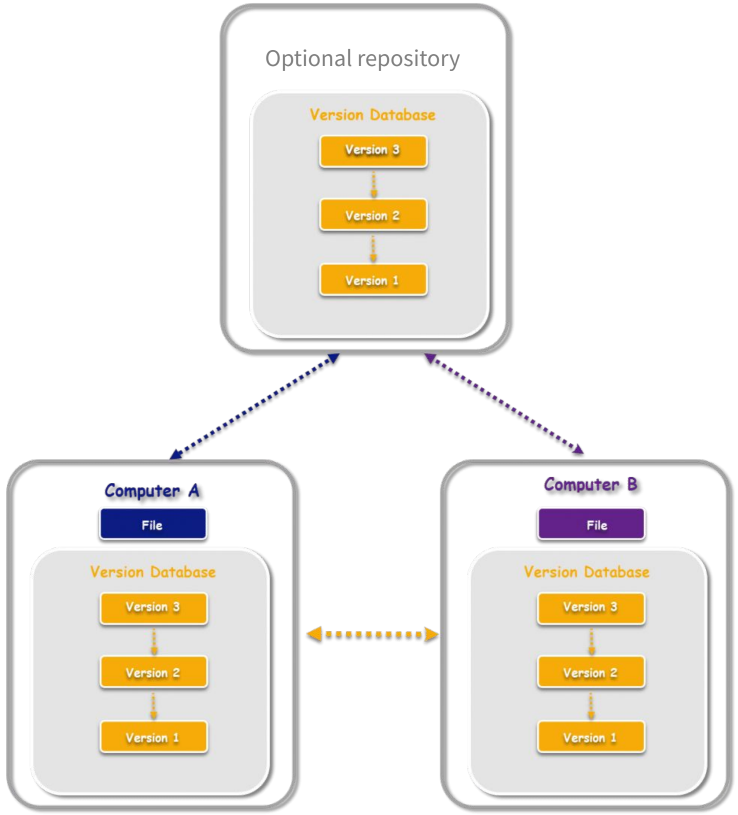
- One server keeps the authoritative versions
- **Easy synchronization** of content

https://www.toolsqa.com/git/distributed-version-control-systems/

Remote Distributed VCS

Peer-to-Peer model

- A local copy on each node



Do we really need an authority?

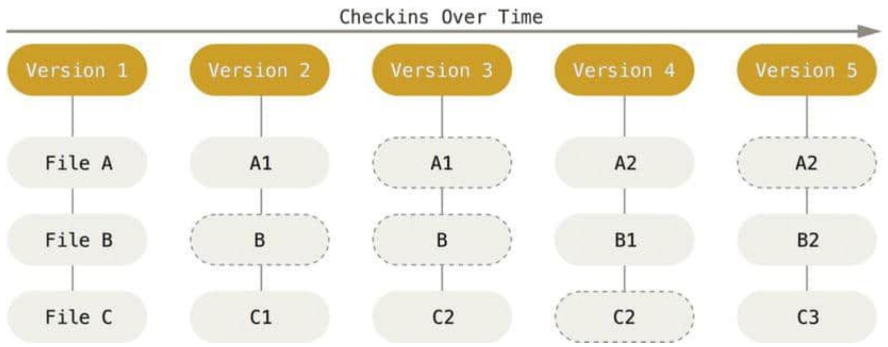
- Each peer keeps a replica
- Has more **complicated synchronization** (“synch algorithm”)

https://www.toolsqa.com/git/distributed-version-control-systems/

Git

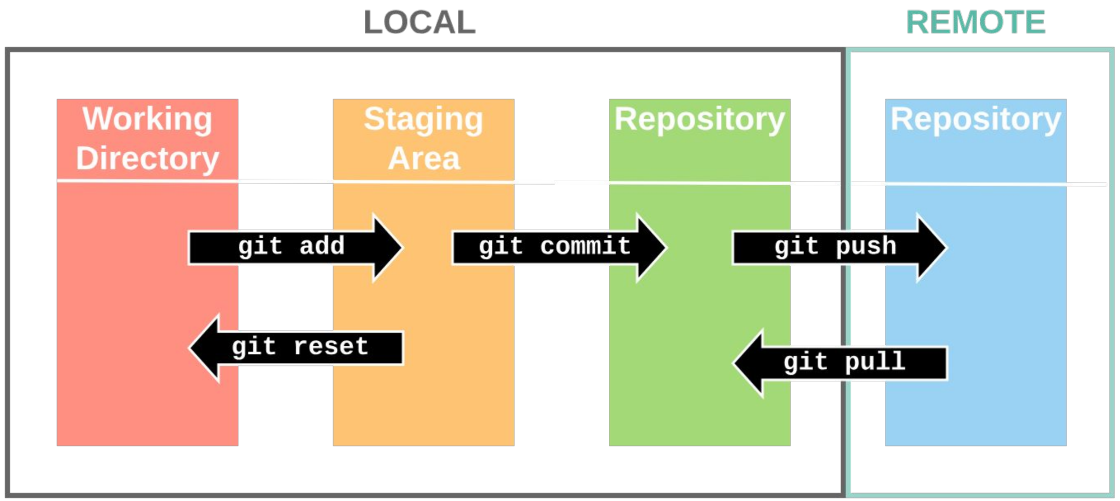
An open source *distributed* VCS

- Was developed by *Linus Torvalds*
- Complete **snapshots** instead of deltas
    - For optimization it uses **references** to unchanged files between versions



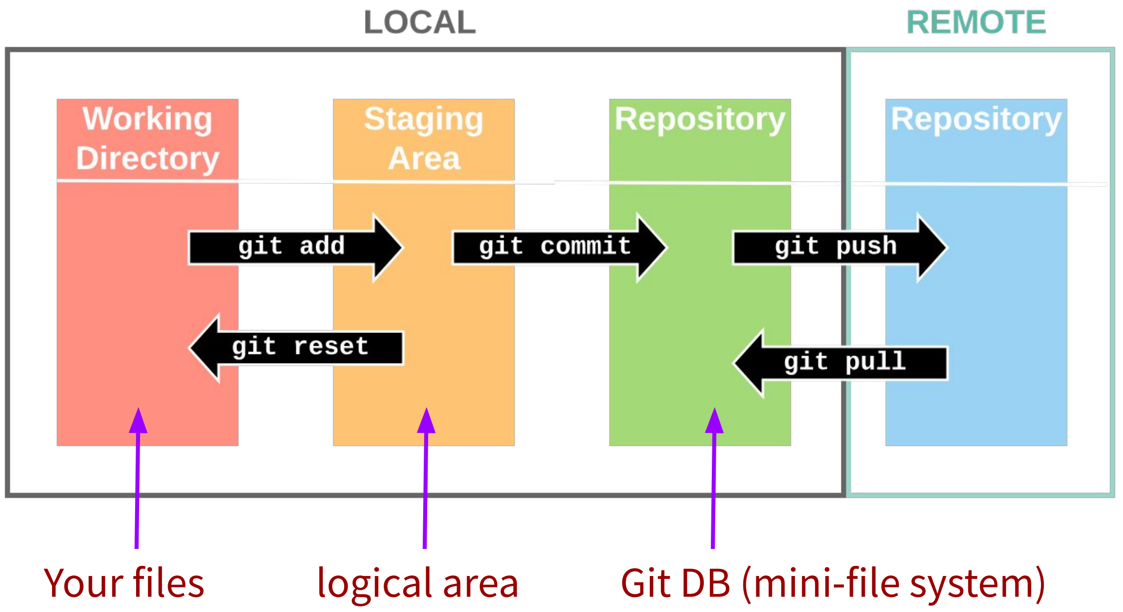
Git Life Cycle

Git is "offline-first": You work on your local copy, and later in the process you inform others about your work



Git Life Cycle

Git is “offline-first”: You work on your local copy, and later in the process you inform others about your work



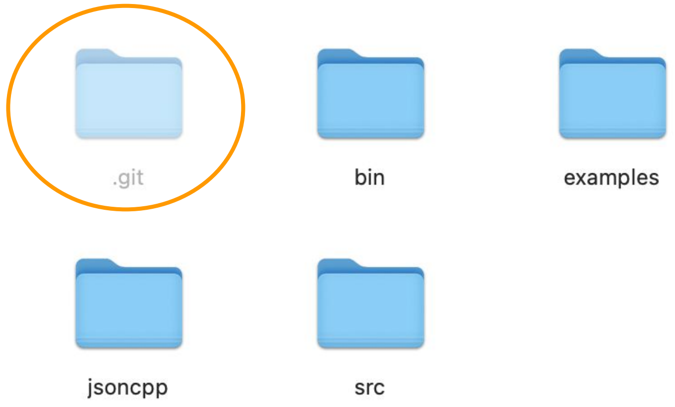
Git mini File System

An **append-only** File System (or database)

- Nothing is ever modified or deleted

git init repo_name

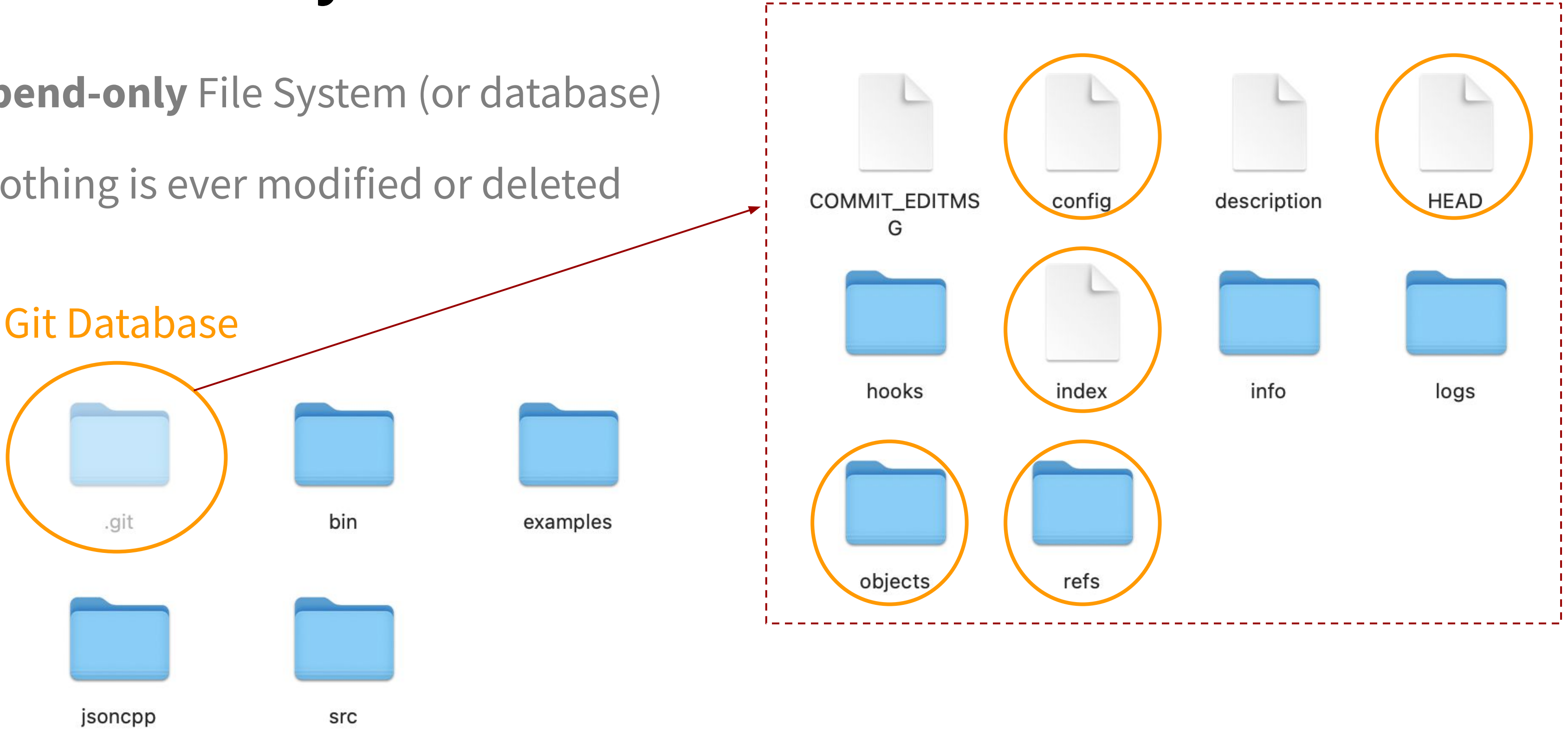
Git Database



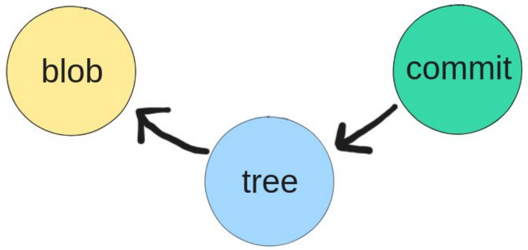
Git mini File System

An **append-only** File System (or database)

- Nothing is ever modified or deleted



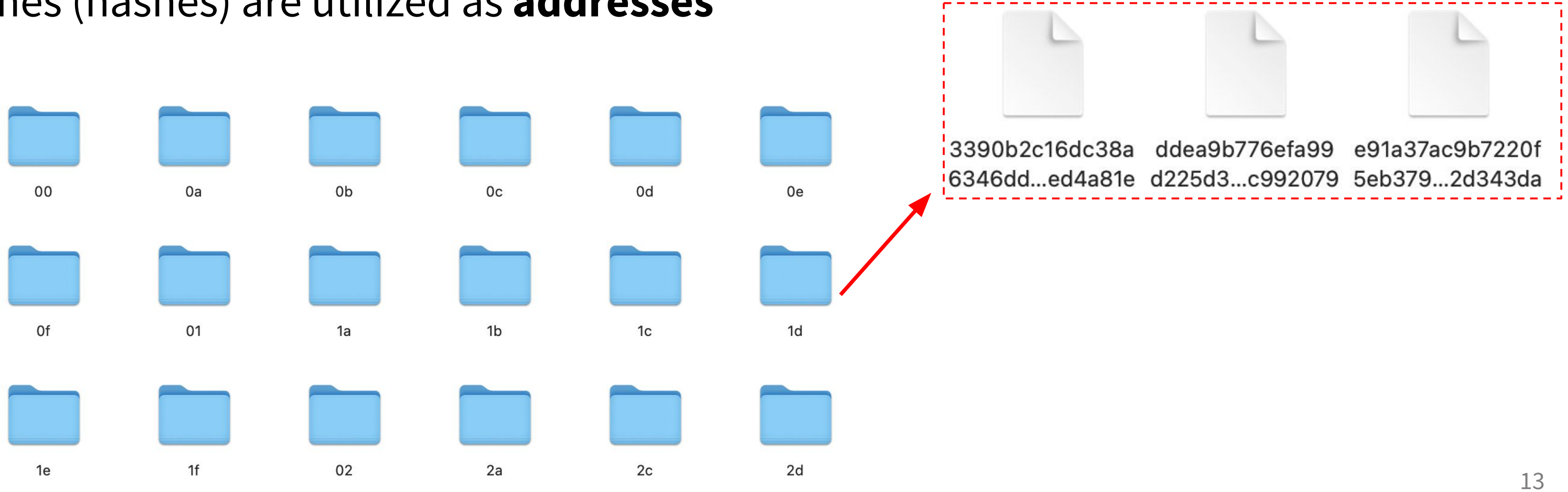
Git Objects



Git Objects

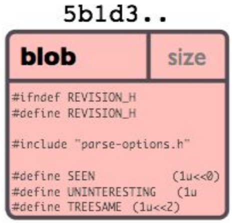
Git objects are files; their name is the SHA-1 hash value of the file's content (content addressable data with self certifying identifier)

- The first two letters identify the **directory**
- Names (hashes) are utilized as **addresses**


00
0a
0b
0c
0d
0e
0f
01
1a
1b
1c
1d
1e
1f
02
2a
2c
2d
3390b2c16dc38a 6346dd...ed4a81e
ddea9b776efa99 d225d3...c992079
e91a37ac9b7220f 5eb379...2d343da

The Three Most Important Git Objects

Blob (Binary Large OBjects): only content without metadata (only file contents)



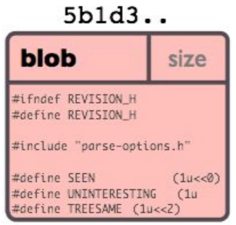
```
#ifndef REVISION_H
#define REVISION_H

#include "parse-options.h"

#define SEEN              (1u<<0)
#define UNINTERESTING   (1u
#define TREESAME  (1u<<2)
```

The Three Most Important Git Objects

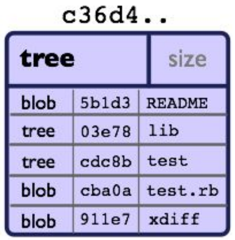
Blob (Binary Large OBjects): only content without metadata (only file contents)

Tree: a collection of **addresses** that reference various blobs and other trees

- Capability of assigning **names to objects**
- Represents directories (Subtrees provide subfolder mechanism)



```
#ifndef REVISION_H
#define REVISION_H

#include "parse-options.h"

#define SEEN                (1u<<0)
#define UNINTERESTING    (1u
#define TREESAME  (1u<<2)
```



blob	5b1d3	README
tree	03e78	lib
tree	cdc8b	test
blob	cba0a	test.rb
blob	911e7	xdiff

The Three Most Important Git Objects

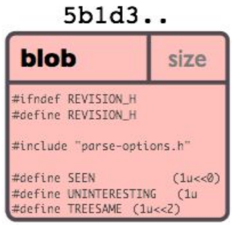
Blob (Binary Large OBjects): only content without metadata (only file contents)

Tree: a collection of **addresses** that reference various blobs and other trees

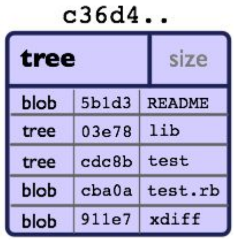
- Capability of assigning **names to objects**
- Represents directories (Subtrees provide subfolder mechanism)

Commit: a snapshot of current state (version)

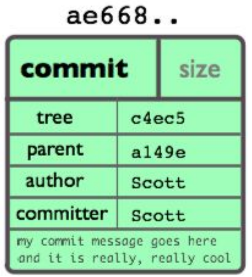
- A root tree (single **address**)
- **List of Parent** commits (**addresses**)
- Metadata: time, author, etc



```
#ifndef REVISION_H
#define REVISION_H

#include "parse-options.h"

#define SEEN          (1u<<0)
#define UNINTERESTING   (1u
#define TREESAME  (1u<<2)
```



blob	5b1d3	README
tree	03e78	lib
tree	cdc8b	test
blob	cba0a	test.rb
blob	911e7	xdiff



tree	c4ec5
parent	a149e
author	Scott
committer	Scott

my commit message goes here
and it is really, really cool

The Three Most Important Git Objects

Blob (Binary Large OBjects): only content without metadata (only file contents)

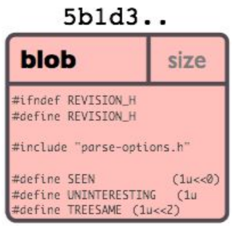
Tree: a collection of **addresses** that reference various blobs and other trees

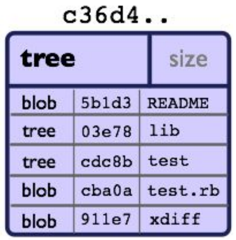
- Capability of assigning **names to objects**
- Represents directories (Subtrees provide subfolder mechanism)

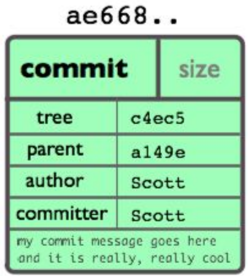
Commit: a snapshot of current state (version)

- A root tree (single **address**)
- **List of Parent** commits (**addresses**)
- Metadata: time, author, etc

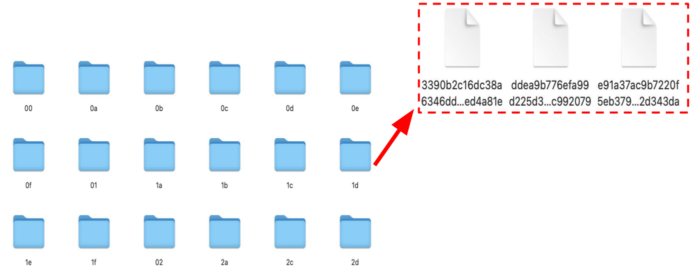
There is also another object type called **annotated tags**





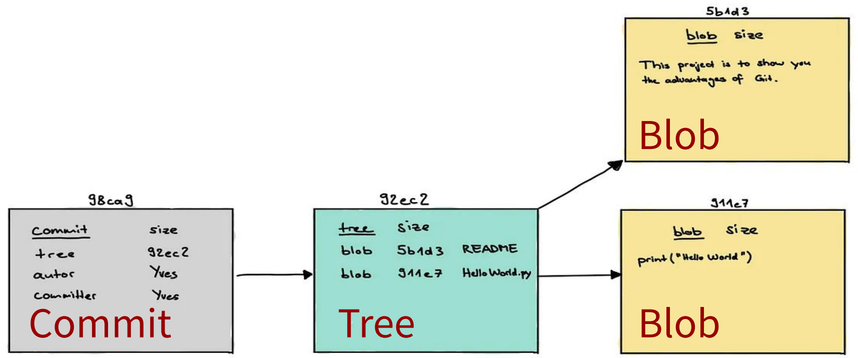


Append-only Structure


00
0a
0b
0c
0d
0e
0f
01
1a
1b
1c
1d
1e
1f
02
2a
2c
2d
3390b2c16dc38a 6346dd...ed4a81e
ddea9b776efa99 d225d3...c992079
e91a37ac9b7220f 5eb379...2d343da

Merkle tree or ***Hash tree*** structure

- Each node is labeled with a SHA hash of its content
- Each node contains the hash of its children (data integrity)
- Leaves contain our data

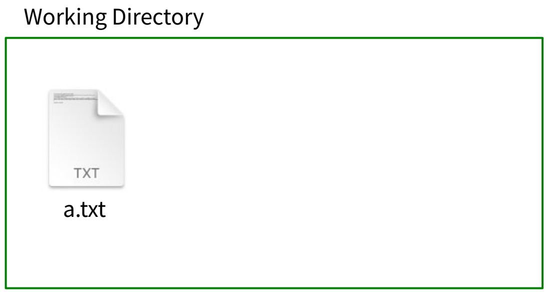


Append-only Structure (Example 1)

```
git init test

cd test

echo “hello world” > a.txt

git add a.txt

git commit -m “1st”
```

Append-only Structure (Example 1)

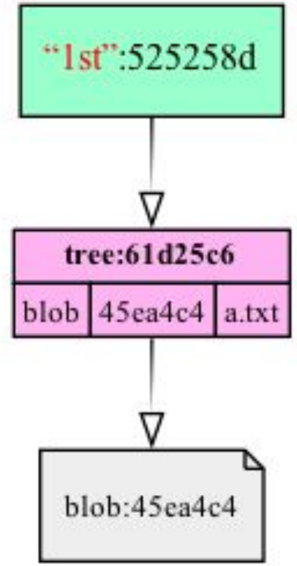
```
git init test

cd test

echo “hello world” > a.txt

git add a.txt

git commit -m “1st”
```



Append-only Structure (Example 1)

```
git init test
cd test
echo "hello world" > a.txt
git add a.txt
git commit -m "1st"
```

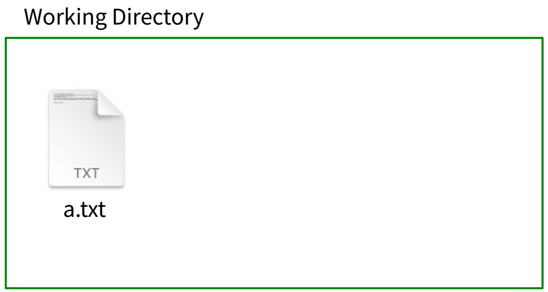
Working Directory



Append-only Structure (Example 1)

Modifying the content

```
git init test
cd test
echo “hello world” > a.txt
git add a.txt
git commit -m “1st”
echo “world\!” >> a.txt
git add a.txt
git commit -m “2nd”
```

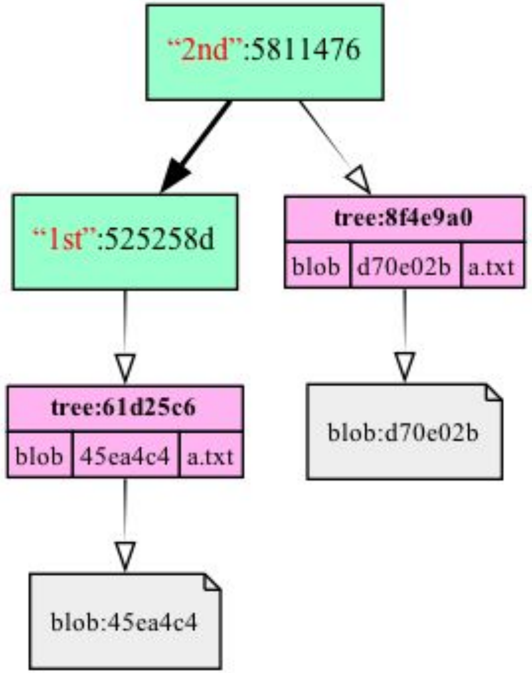


Append-only Structure (Example 1)

Modifying the content

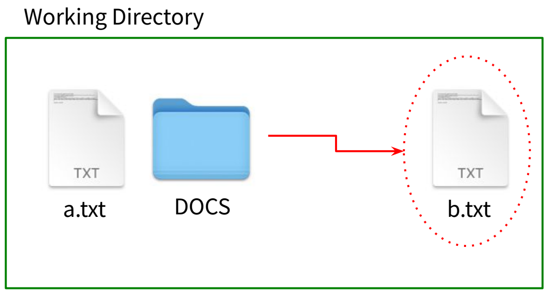
```
git init test
cd test
echo "hello world" > a.txt
git add a.txt
git commit -m "1st"
echo "world\!" >> a.txt
git add a.txt
git commit -m "2nd"
```

Working Directory



Append-only Structure (Example 2)

```
git init test
cd test
echo “hello world” > a.txt
mkdir DOCS
echo “This is a test” > DOCS/b.txt
git add a.txt DOCS/b.txt
git commit -m “1st”
```

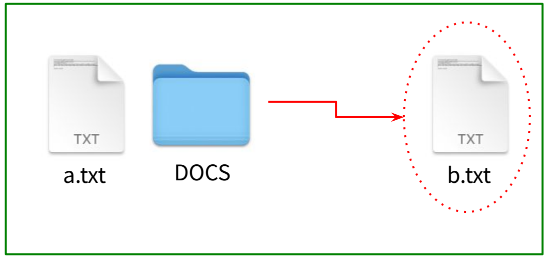


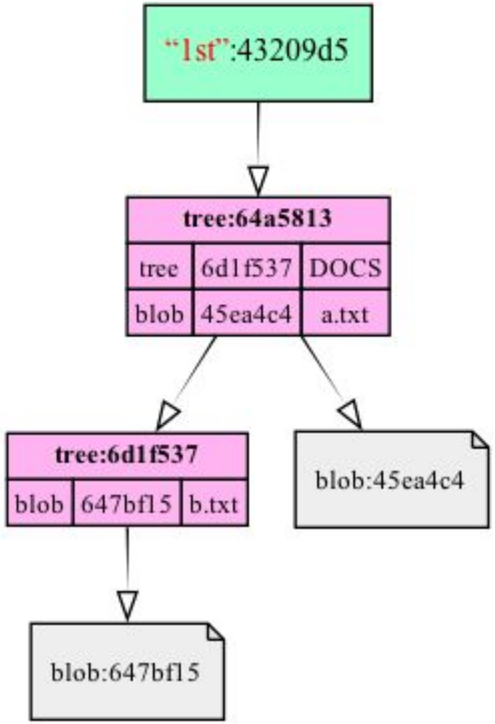
Append-only Structure (Example 2)

With one commit

```
git init test
cd test
echo “hello world” > a.txt
mkdir DOCS
echo “This is a test” > DOCS/b.txt
git add a.txt DOCS/b.txt
git commit -m “1st”
```

Working Directory

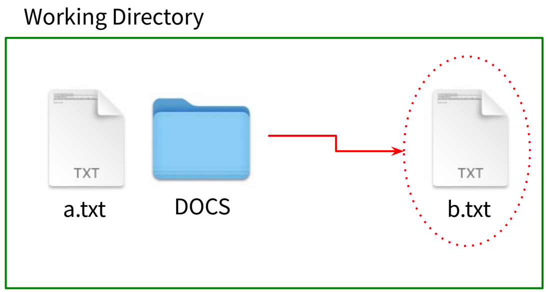


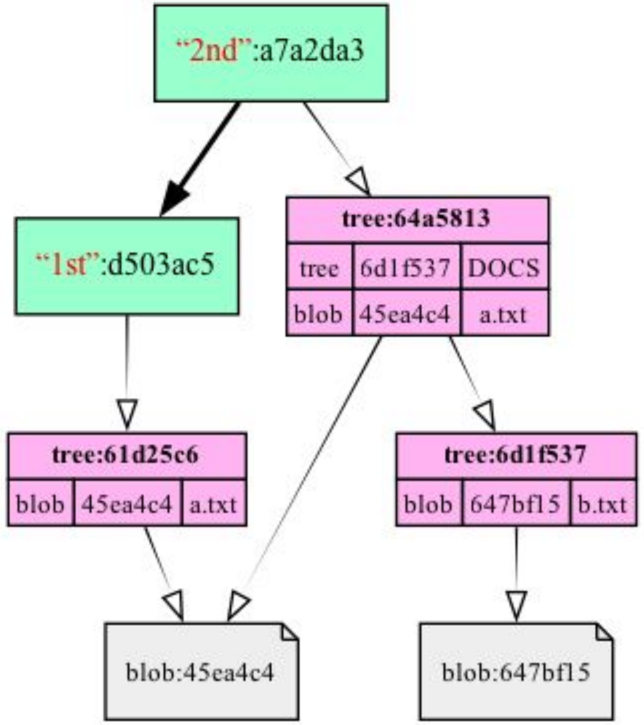


Append-only Structure (Example 2)

With two commits

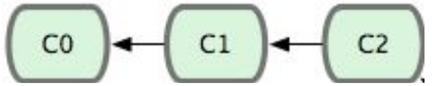
```
git init test
cd test
echo "hello world" > a.txt
git add a.txt
git commit -m "1st"
mkdir DOCS
echo "This is a test" > DOCS/b.txt
git add DOCS/b.txt
git commit -m "2nd"
```



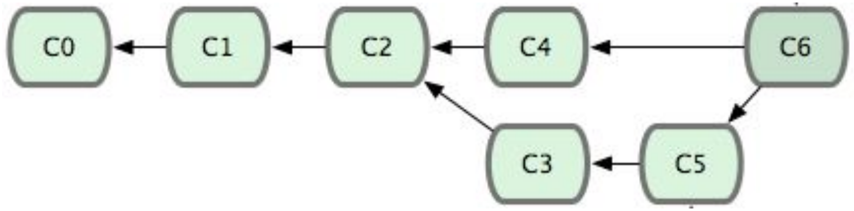


Sequential or Concurrent Versions (Commits)

When we have multiple versions we have different commits, each points to its parent(s) (the trees and blobs are not drawn)



Versions are not always **sequential**, can have **concurrent branches**



Each C_i represents a commit (version), focusing solely on the commits themselves and disregarding blobs and tree objects. Each commit points back to the version from which it was derived.

Observing Objects

Self Study

To list all commits: *git rev-list --all*

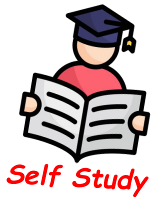
To list all objects: *git rev-list --all --objects*

To see type of an object: *git cat-file -t hash_value*

To see content of an object: *git cat-file -p hash_value*

To see all commits of a branch: *git log [branch_name | --all] [--oneline] [--graph] [--decorate]*

Creating Objects (Plumbing commands)



To create a blob from a.txt file [and **store** it on git database]: *git hash-object a.txt [-w]*

- From string: *git hash-object --stdin <<< string* or *echo string | git hash-object --stdin*

To create a tree object: *echo "100644 blob BLOB_HASH\ta.txt" | git mktree*

- 100644 (regular file), 100755 (executable), 040000 (subtree).
- Empty tree is universal: *4b825dc642cb6eb9a060e54bf8d69288fbee4904*

To create a commit object: *git commit-tree <tree-hash> [-p <parent-hash>] [-m "message"]*

- A commit object with empty tree:

  git commit-tree 4b825dc642cb6eb9a060e54bf8d69288fbee4904 -m "Empty tree commit"

Pointers

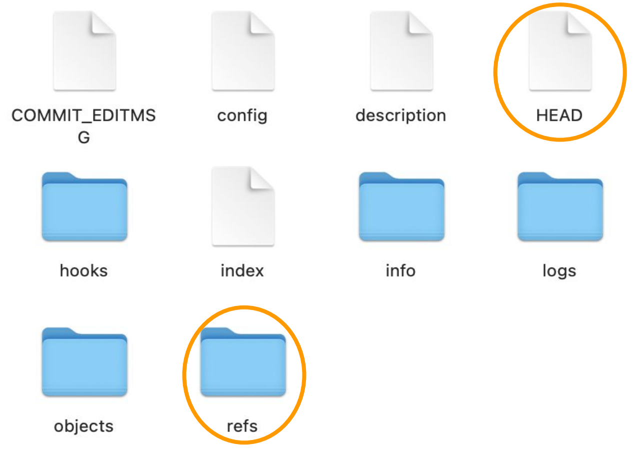
Pointers

Similar to C/C++, each *commit object* can be referenced by **pointers**.

- A pointer is a file containing the **address** of the commit object (SHA hash) directly or indirectly.

There are three types of pointers:

1. **HEAD:** keeps track of the active commit
2. **Lightweight Tags**: static pointers
3. **Dynamic pointers (branches)**: keep track of the latest snapshot of a branch



Pointers

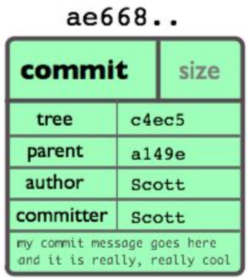
Similar to C/C++, each *commit object* can be referenced by **pointers**.

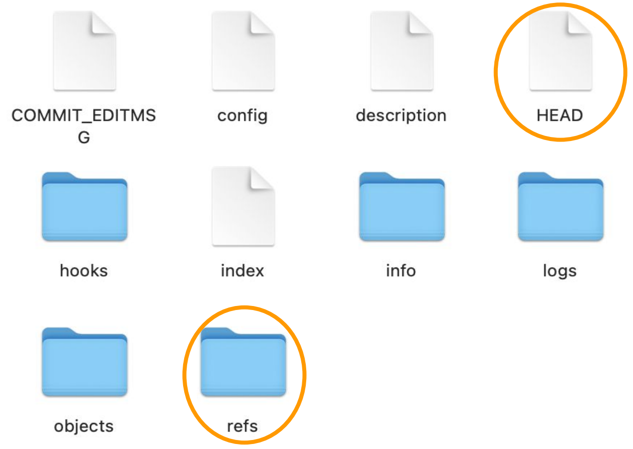
- A pointer is a file containing the **address** of the commit object (SHA hash) directly or indirectly.

There are three types of pointers:

1. **HEAD:** keeps track of the active commit
2. **Lightweight Tags**: static pointers
3. **Dynamic pointers (branches)**: keep track of the latest snapshot of a branch

These pointers are needed to track addresses of different snapshots

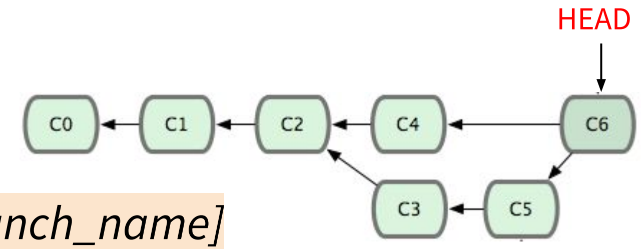




HEAD

HEAD points to the current active snapshot, i.e **the commit object with which working directory is synchronized**. The pointer is stored on *.git/HEAD* containing a *SHA* hash.

- **Directly**: *2780059eb33aff3561fd84147c2d8ce528d9667b*
- **Indirectly**: *ref: refs/heads/branch_name*

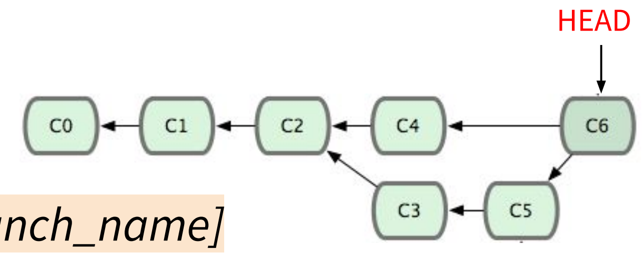


To change active commit use: *git checkout [hash | branch_name]*

HEAD

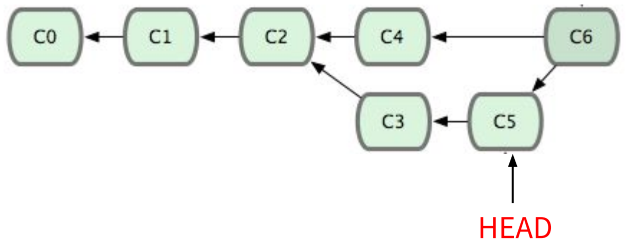
HEAD points to the current active snapshot, i.e **the commit object with which working directory is synchronized**. The pointer is stored on *.git/HEAD* containing a *SHA* hash.

- **Directly**: *2780059eb33aff3561fd84147c2d8ce528d9667b*
- **Indirectly**: *ref: refs/heads/branch_name*

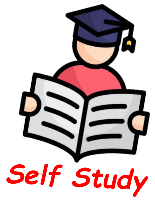


To change active commit use: *git checkout [hash | branch_name]*

- *Example: git checkout c5*



Tags (Static Pointers)



Git tags are a way to mark specific points in your repository's history as important, often used for releases or milestones.

1. **Lightweight Tags** are simple static pointers

- To define a new: *git tag [tag_name | branch_name] hash*

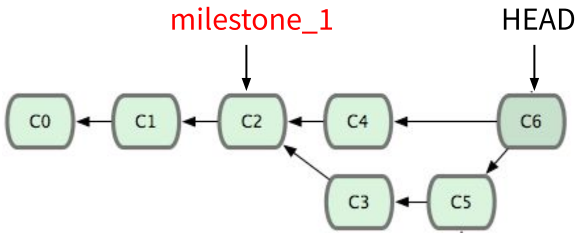
Tags (Static Pointers)

Self Study

Git tags are a way to mark specific points in your repository's history as important, often used for releases or milestones.

1. Lightweight Tags are simple static pointers

- To define a new: *git tag [tag_name | branch_name] hash*



Example: *git tag milestone_1 c2*

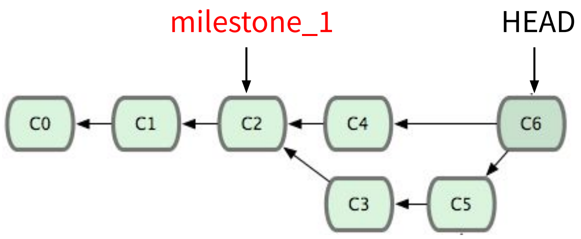
Tags (Static Pointers)

Self Study

Git tags are a way to mark specific points in your repository's history as important, often used for releases or milestones.

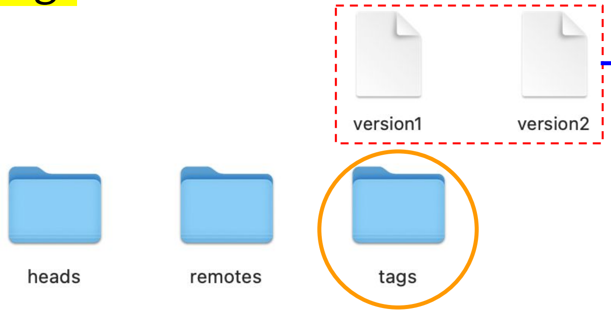
1. Lightweight Tags are simple static pointers

- To define a new: *git tag [tag_name | branch_name] hash*



Example: *git tag milestone_1 c2*

Are stored in .git/refs/tags

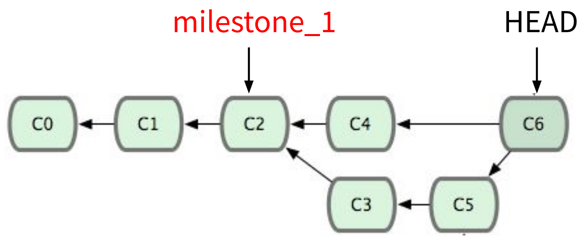


cec93120270…

Tags (Static Pointers)

Self Study

Git tags are a way to mark specific points in your repository's history as important, often used for releases or milestones.



Example: *git tag milestone_1 c2*

1. Lightweight Tags are simple static pointers

- To define a new: *git tag [tag_name | branch_name] hash*

Are stored in .git/refs/tags

2. Annotated Tags: includes metadata such as the tagger's name, email, and date. It is on object not a pointer.

- To define a new: *git tag -a [tag_name | branch_name] -m message*

Branches (Dynamic Pointers)

Starting from a specific version (or snapshot), you can create multiple branches to work on different features or changes simultaneously.

In Git, branches are *dynamic pointers* that reference the **most recent commit** in a particular line of development.

- Git has a default *main/master* branch

To define: *git branch branch_name [hash | branch_name]*

Branches (Dynamic Pointers)

Starting from a specific version (or snapshot), you can create multiple branches to work on different features or changes simultaneously.

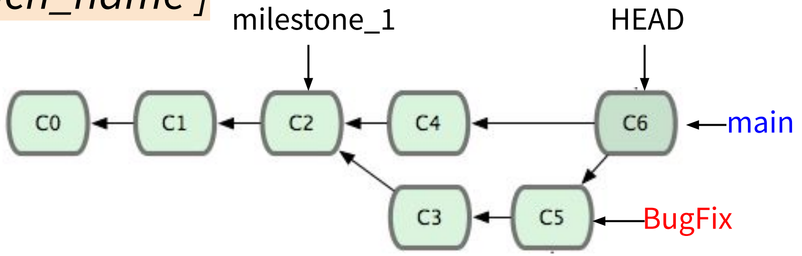
In Git, branches are *dynamic pointers* that reference the **most recent commit** in a particular line of development.

- Git has a default *main/master* branch

To define: *git branch branch_name [hash | branch_name]*

Example: git branch BugFix c5

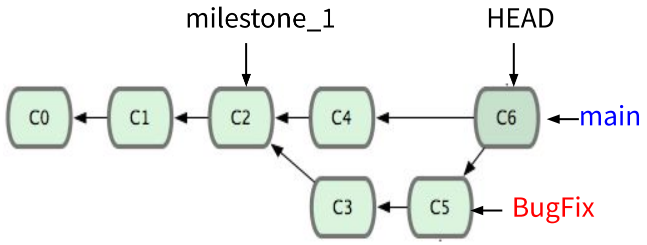
Use -f option to force git to update a branch



https://book.the-turing-way.org/reproducible-research/vcs/vcs-workflow-branches

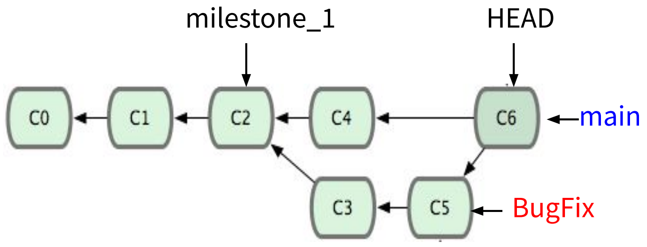
Branches (Dynamic Pointers)

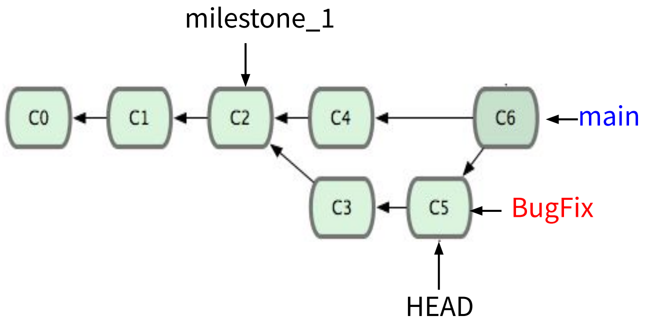
A branch is a *dynamic pointer* because when you add a new commit to a branch, the pointer points to the new commit.



Branches (Dynamic Pointers)

A branch is a *dynamic pointer* because when you add a new commit to a branch, the pointer points to the new commit.



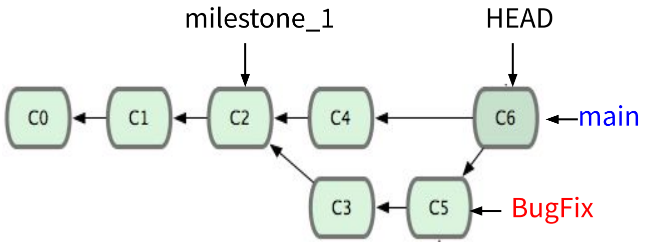


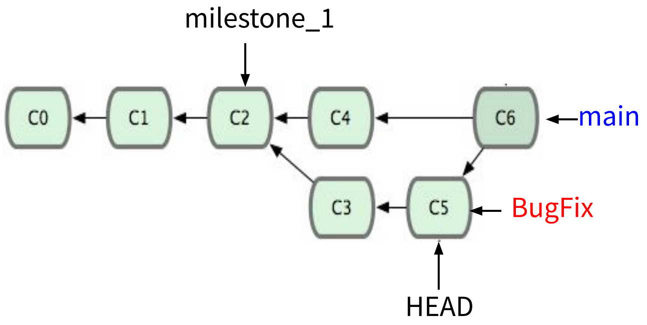
git checkout BugFix

https://book.the-turing-way.org/reproducible-research/vcs/vcs-workflow-branches

Branches (Dynamic Pointers)

A branch is a *dynamic pointer* because when you add a new commit to a branch, the pointer points to the new commit.

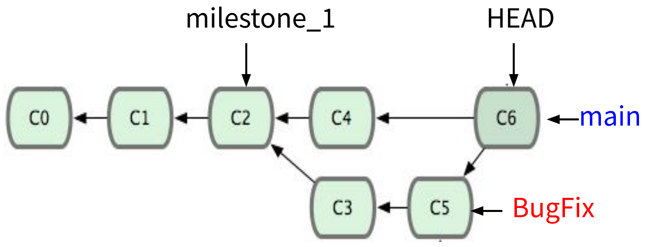


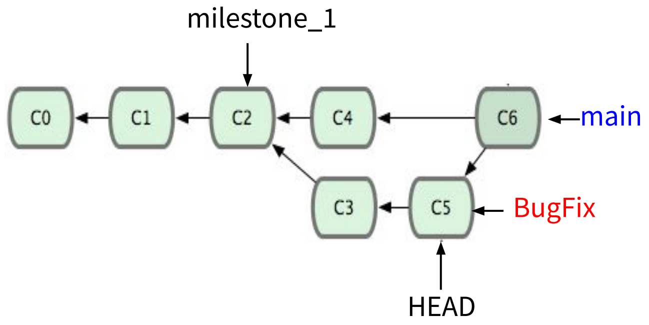


```
git checkout BugFix
git commit -m “c7”
```

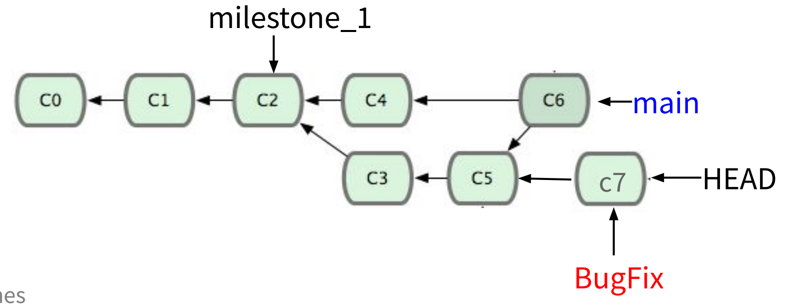
Branches (Dynamic Pointers)

A branch is a *dynamic pointer* because when you add a new commit to a branch, the pointer points to the new commit.



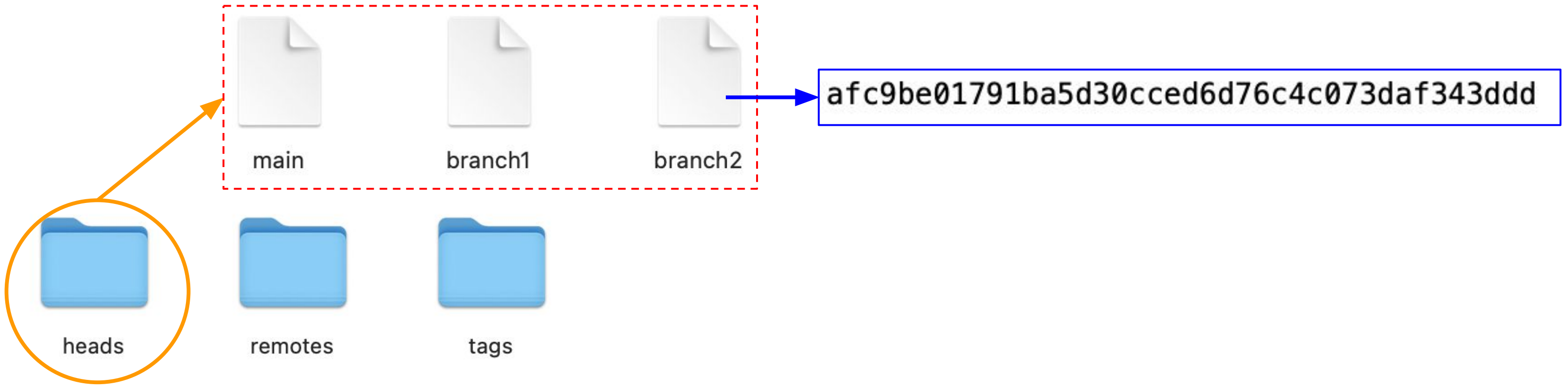


```
git checkout BugFix
git commit -m “c7”
```



Branches (Dynamic Pointers)

Branches are stored in files under .git/refs/heads/ or .git/refs/remotes/, etc, with the file name corresponding to the branch name.



Integration of Changes

Merge and Rebase

Merge

Merges objects of two branches: current working branch and a target branch

- It updates working directory, staging area (index) and commit history
- It is probable to create a merge commit

Syntax: *git merge [target branch]*

Types:

- **Fast-Forward**: A simple pointer update, no commit needed
- **Three-Way**: If both branches have diverged, git creates a commit object automatically

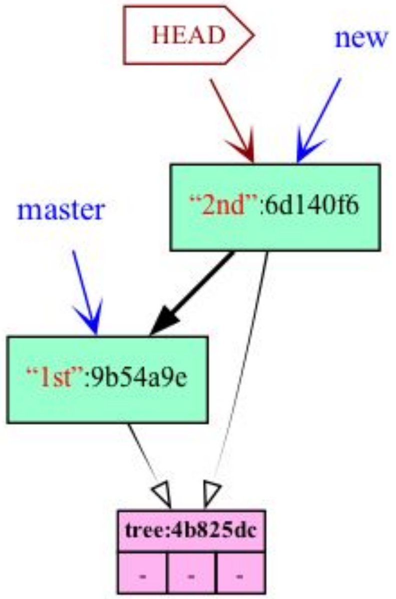
Fast-Forward Merge (Example)

```
git init test

cd test

git commit --allow-empty -m “1st”

git checkout -b new

git commit --allow-empty -m “2nd”
```

--allow-empty let you create an empty commit!

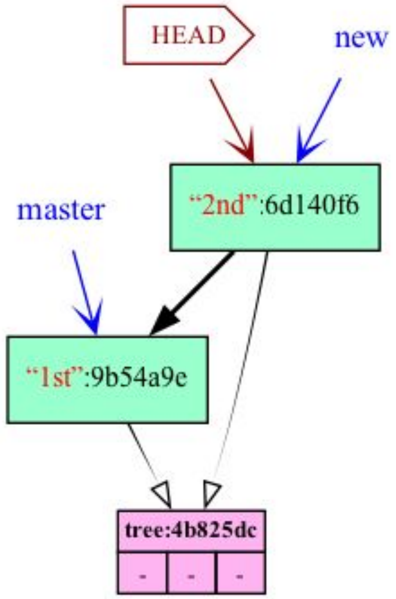
Fast-Forward Merge (Example)

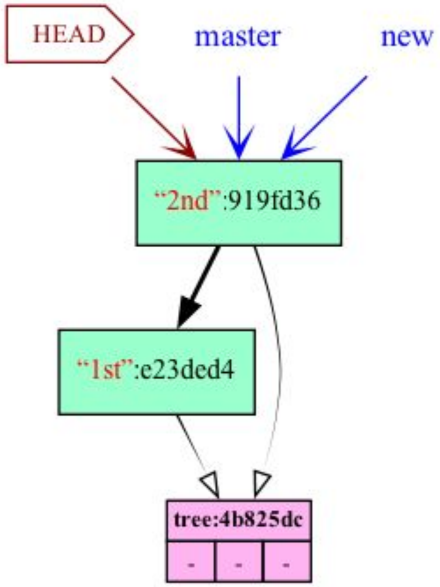
```
git init test

cd test

git commit --allow-empty -m "1st"

git checkout -b new

git commit --allow-empty -m "2nd"
```



Fast-Forward Merge (Example)

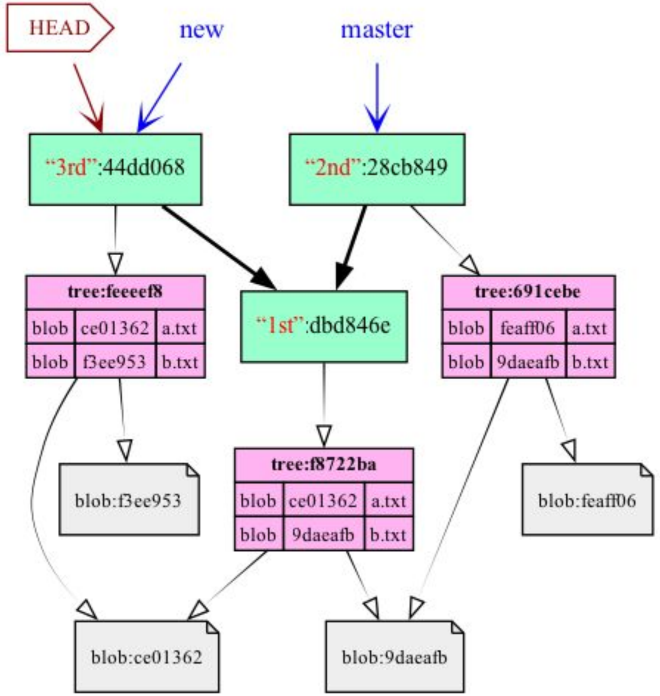
```
git init test
cd test
git commit --allow-empty -m “1st”
git checkout -b new
git commit --allow-empty -m “2nd”
git checkout master
git merge new
```





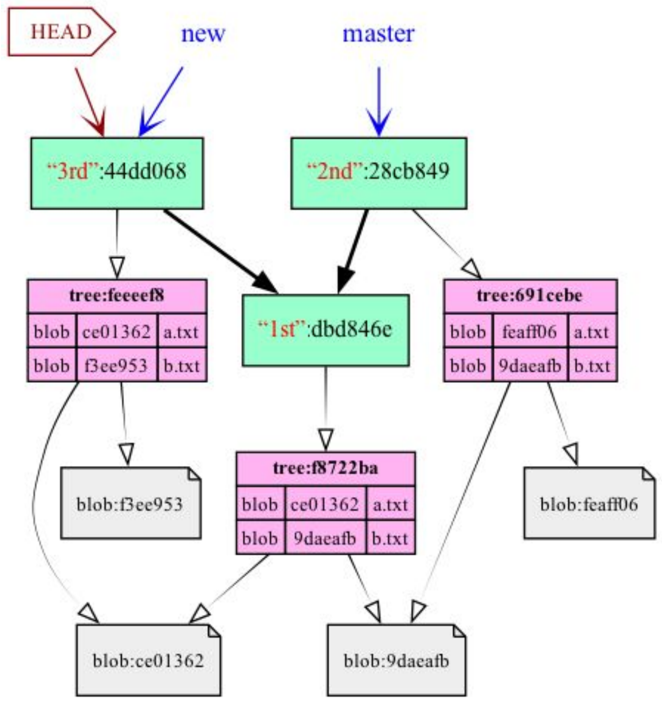
Three-Way Merge (Example)

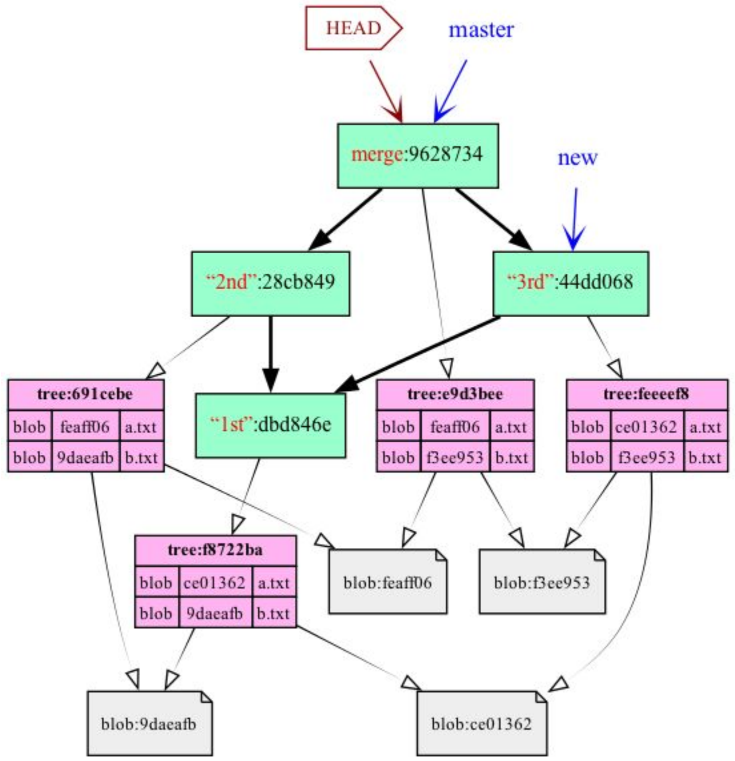
```
git init test
cd test
echo hello > a.txt
echo test > b.txt
git add a.txt b.txt
git commit -m “1st”
git branch new
echo ”hello world” > a.txt
git add  a.txt
git commit -m “2nd”
git checkout new
echo ”testing” > b.txt
git add  b.txt
git commit -m “3rd”
```



Three-Way Merge (Example)

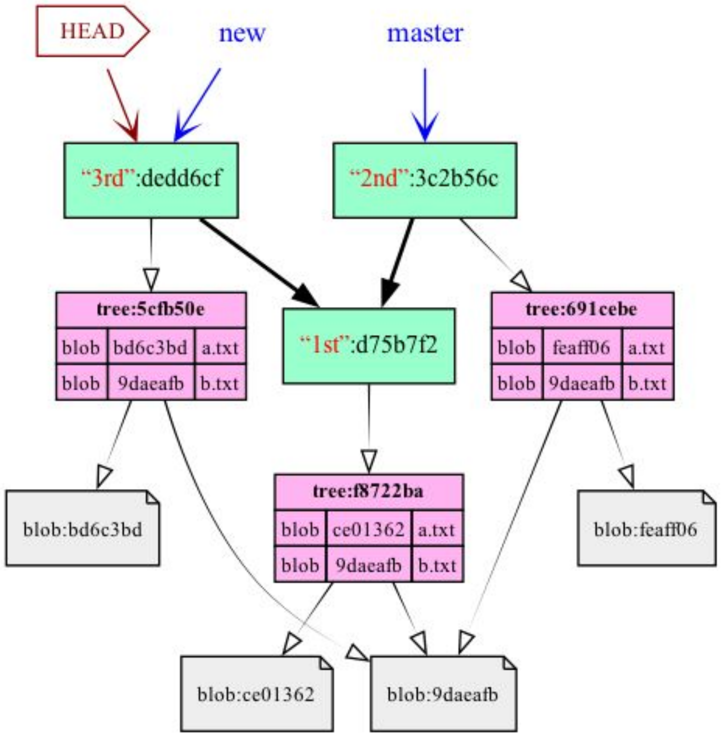
```
git init test
cd test
echo hello > a.txt
echo test > b.txt
git add a.txt b.txt
git commit -m “1st”
git branch new
echo ”hello world” > a.txt
git add  a.txt
git commit -m “2nd”
git checkout new
echo ”testing” > b.txt
git add  b.txt
git commit -m “3rd”
git checkout master
git merge new
```





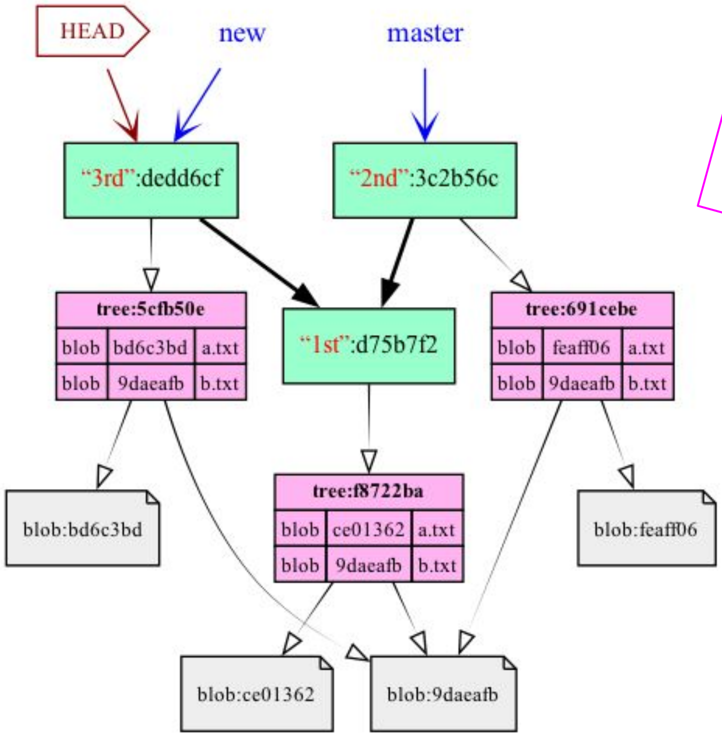
Conflict (Example)

```
git init test
cd test
echo hello > a.txt
echo test > b.txt
git add a.txt b.txt
git commit -m “1st”
git branch new
echo ”hello world” > a.txt
git add  a.txt
git commit -m “2nd”
git checkout new
echo ”hello class” > a.txt
git add  a.txt
git commit -m “3rd”
```



Conflict (Example)

```
git init test
cd test
echo hello > a.txt
echo test > b.txt
git add a.txt b.txt
git commit -m “1st”
git branch new
echo ”hello world” > a.txt
git add  a.txt
git commit -m “2nd”
git checkout new
echo ”hello class” > a.txt
git add  a.txt
git commit -m “3rd”
git checkout master
git merge new
```



Error: I’m confused! Which blob should I use as the data for a.txt?

Rebase

Self Study

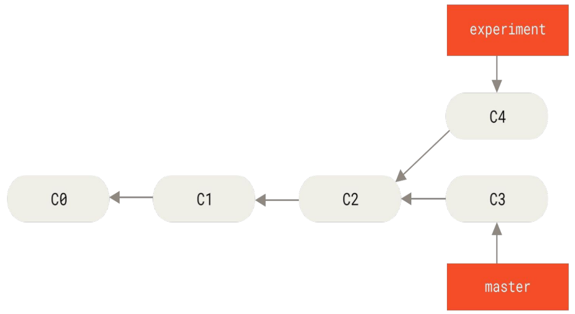
Instead of merging, you can take all the changes that were committed on one branch and replay them on a different branch.

- Rewrites history and linearizes it without any merge commit.
- It results in commit objects becoming dangling
- Syntax: `git rebase [target branch]`

Rebase vs. Merge

Self Study

The final integrated content in the working directory remains the same, but rebasing creates a cleaner, linear history.

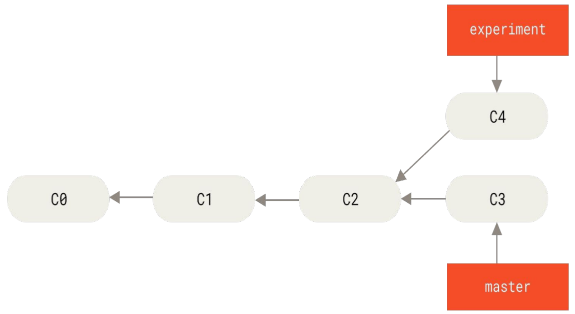


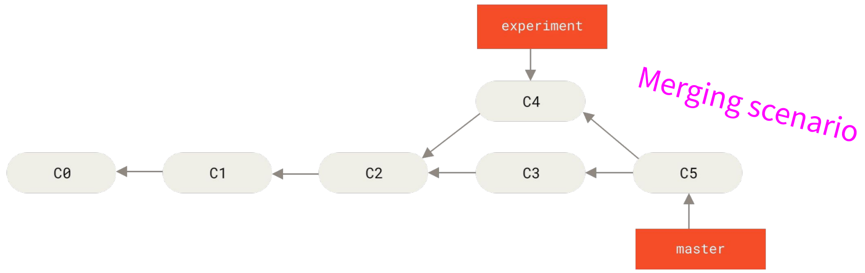
https://git-scm.com/book/en/v2/Git-Branching-Rebasing

Rebase vs. Merge

Self Study

The final integrated content in the working directory remains the same, but rebasing creates a cleaner, linear history.



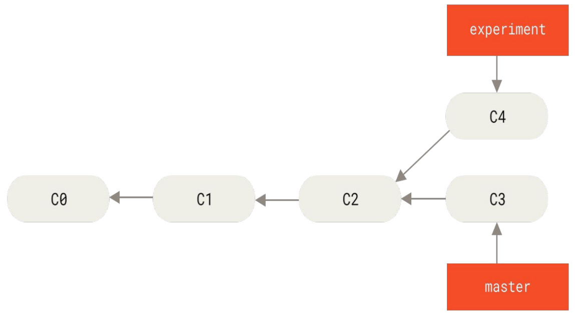


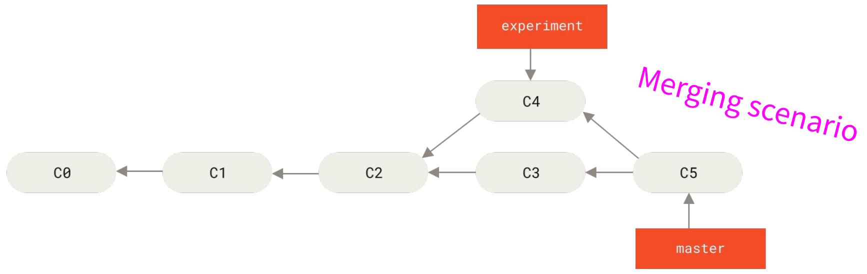
https://git-scm.com/book/en/v2/Git-Branching-Rebasing

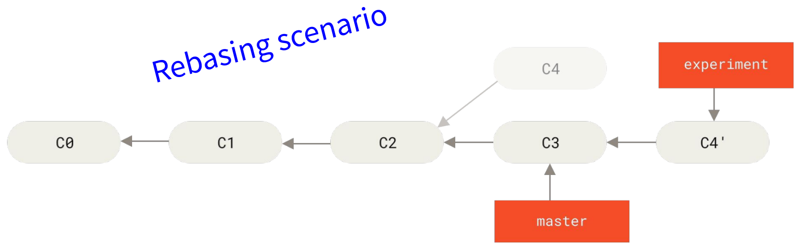
Rebase vs. Merge

Self Study

The final integrated content in the working directory remains the same, but rebasing creates a cleaner, linear history.





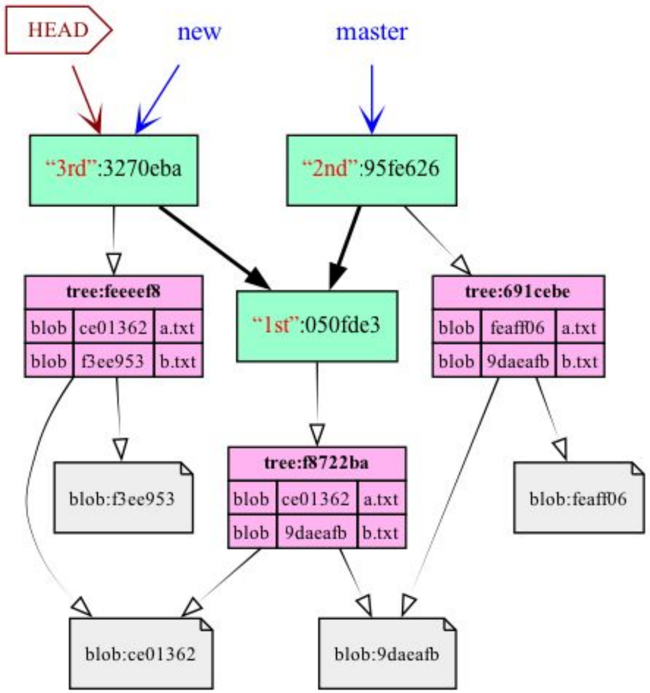


https://git-scm.com/book/en/v2/Git-Branching-Rebasing

Rebase (Example)

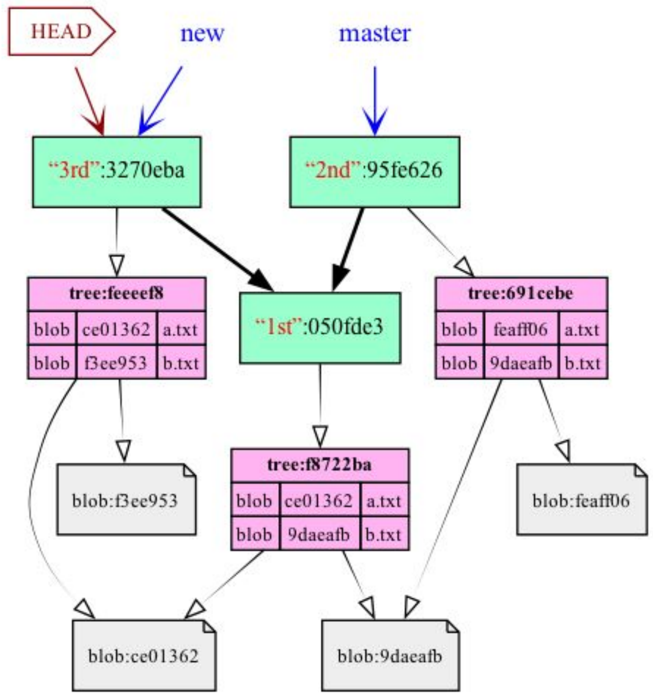
Self Study

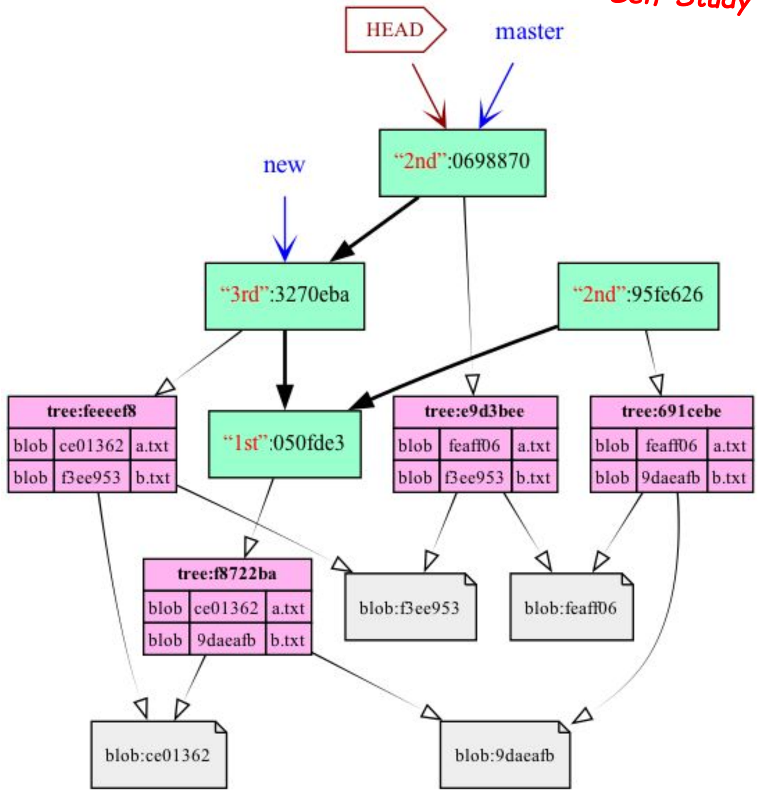
```
git init test
cd test
echo hello > a.txt
echo test > b.txt
git add a.txt b.txt
git commit -m "1st"
git branch new
echo "hello world" > a.txt
git add  a.txt
git commit -m "2nd"
git checkout new
echo "testing" > b.txt
git add  b.txt
git commit -m "3rd"
```



Rebase (Example)

```
git init test
cd test
echo hello > a.txt
echo test > b.txt
git add a.txt b.txt
git commit -m "1st"
git branch new
echo "hello world" > a.txt
git add  a.txt
git commit -m "2nd"
git checkout new
echo "testing" > b.txt
git add  b.txt
git commit -m "3rd"
git checkout master
git rebase new
```



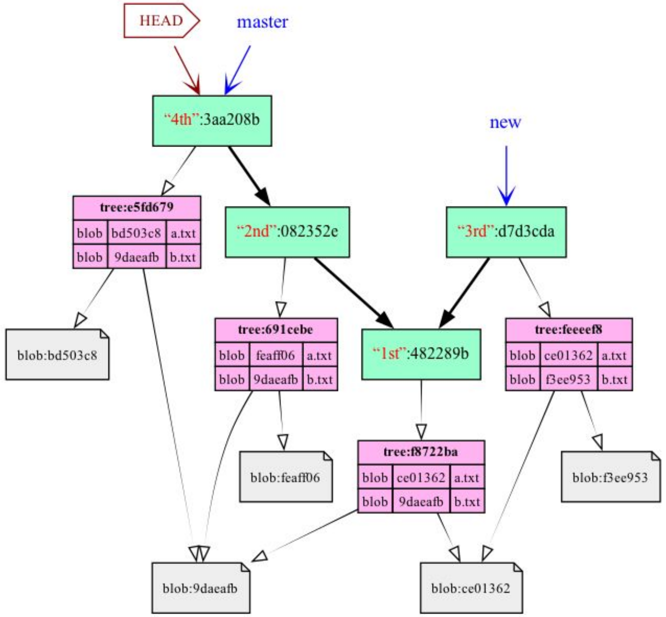


Self Study

Rebase (Example 2)

Self Study

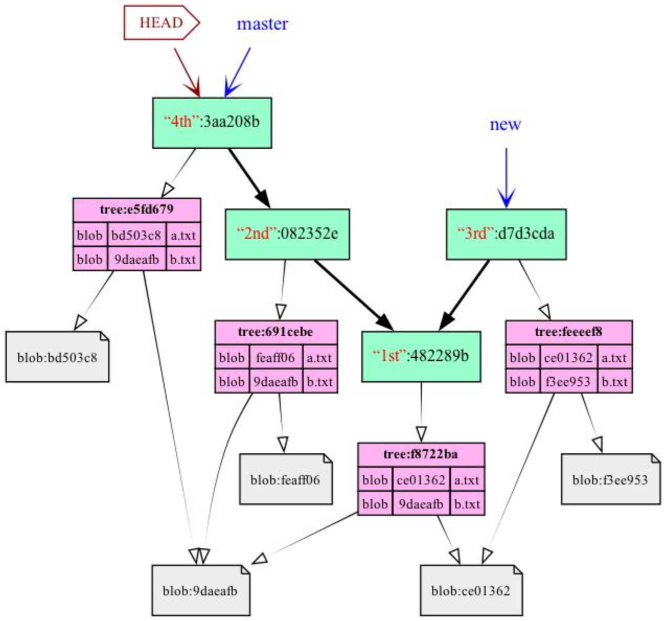
```
git init test
cd test
echo hello > a.txt
echo test > b.txt
git add a.txt b.txt
git commit -m “1st”
git branch new
echo ”hello world” > a.txt
git add  a.txt
git commit -m “2nd”
git checkout new
echo ”testing” > b.txt
git add  b.txt
git commit -m “3rd”
git checkout master
echo “Done” >> a.txt
git add a.txt
git commit -m “4th”
```

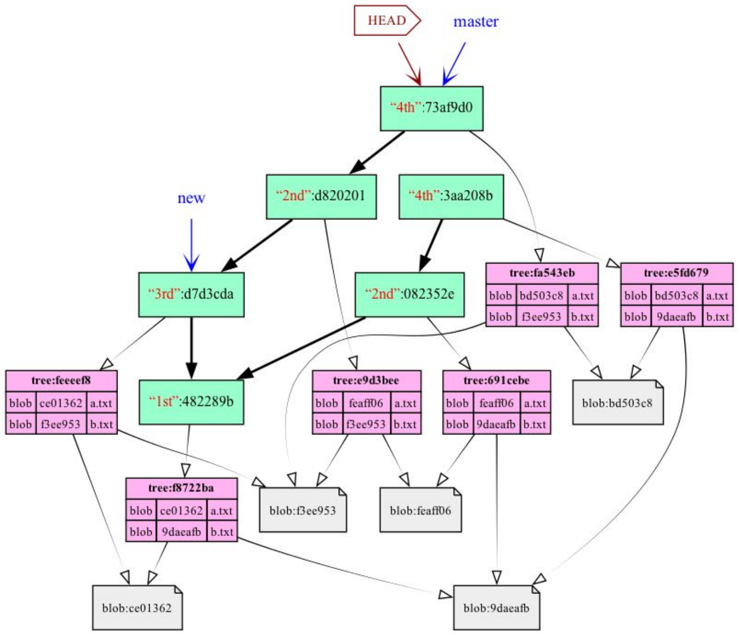


Rebase (Example 2)

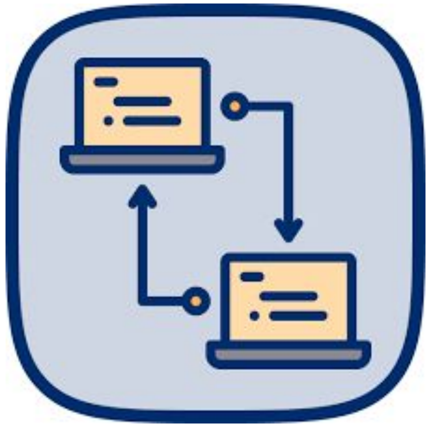
Self Study

```
git init test
cd test
echo hello > a.txt
echo test > b.txt
git add a.txt b.txt
git commit -m “1st”
git branch new
echo ”hello world” > a.txt
git add  a.txt
git commit -m “2nd”
git checkout new
echo ”testing” > b.txt
git add  b.txt
git commit -m “3rd”
git checkout master
echo “Done” >> a.txt
git add a.txt
git commit -m “4th”
git rebase new
```





Remotes



Remote Repositories

A remote repository is a repository connected to your local repository

- Typically accessible over the internet or a local network (GitHub, GitLab, etc)
- Acts as a bridge for sharing, synchronizing changes and backup

Remote Repositories

A remote repository is a repository connected to your local repository

- Typically accessible over the internet or a local network (GitHub, GitLab, etc)
- Acts as a bridge for sharing, synchronizing changes and backup

Common operations:

- *git clone*: Downloads a copy of the remote repository to your local machine
- *git push* : Uploads your local changes to the remote repository
- *git pull* : Fetches and integrates changes from the remote repo into your local copy
- *git fetch* : Retrieves updates from the remote repo without automatically merging

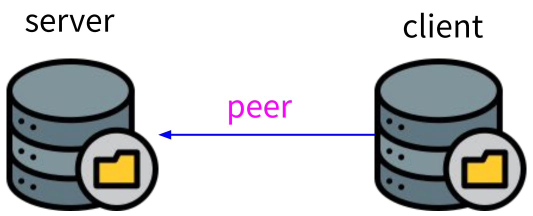
Example

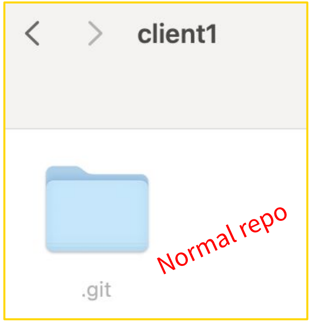
```
git init client

git init  --bare server

cd client

git remote add peer ../server

git remote -v
```

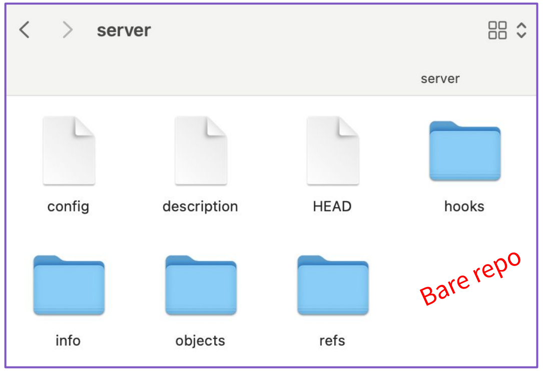
A bare repository is a repository without a working directory

Example

```
git init client
git init  --bare server
cd client
git remote add peer ../server
git remote -v
```

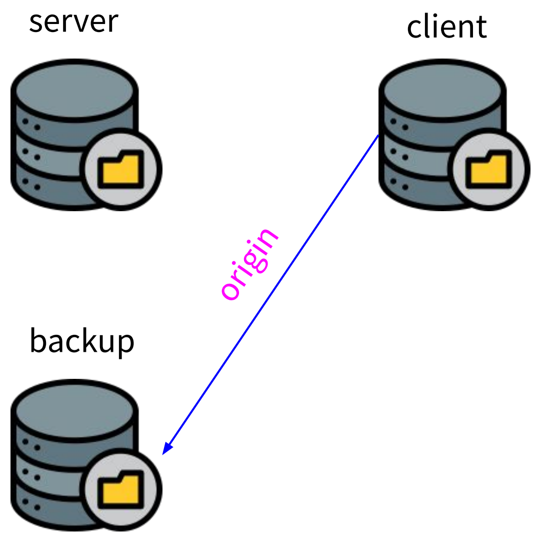






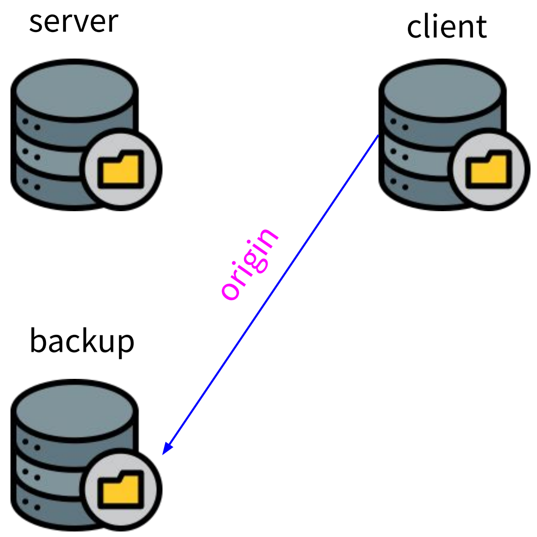
Example

```
git init client

git init  --bare server

cd client

git remote add peer ../server

git remote -v


git remote remove peer

cd ..

git init backup

cd client

git remote add origin ../backup

git remote -v
```



Example

```
git init client
git init  --bare server
cd client
git remote add peer ../server
git remote -v


git remote remove peer
cd ..
git init backup
cd client
git remote add origin ../backup
git remote -v
```



It is possible to connect to multiple remote repositories at the same time.

Push

The push command is used to upload changes from a local **branch** to the corresponding branch on a **remote** repository.

- It uploads objects (blobs, trees, commits) and updates pointers of a target branch.
- If the remote branch doesn't exist and you have permission, it will be created.
- If there is a conflict, Git will reject your push command

Push

The push command is used to upload changes from a local **branch** to the corresponding branch on a **remote** repository.

- It uploads objects (blobs, trees, commits) and updates pointers of a target branch.
- If the remote branch doesn't exist and you have permission, it will be created.
- If there is a conflict, Git will reject your push command

Syntax: *git push [remote_name] [local_branch_name]*

- When pushing a new branch for the first time, you can use the *-u* or *--set-upstream* option to set the upstream branch (push and pull commands can be run without specifying the remote and branch name.)

Pull

Pull command downloads (fetches) objects and updates pointers (merge branches) from a remote **branch** on a **remote** repository to the corresponding local branch.

- It downloads objects (blobs, trees, commits) and updates pointers
- If the local branch doesn't exist it will be created.
- It automatically **merges** local branch with remote branch
- If there is a conflict, you must resolve the conflict

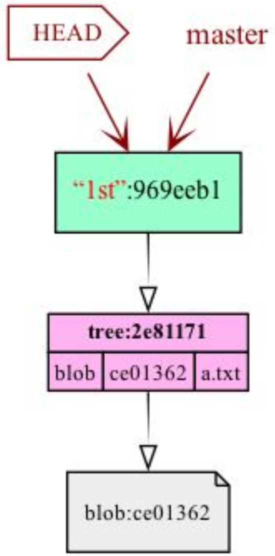
Syntax: *git pull [remote_name] [local_branch_name]*

Example

```
git init --bare server
git init client1
git init client2
cd client1
git remote add origin ../server
cd ../client2
git remote add origin ../server
echo hello > a.txt
git add a.txt
git commit -m "1st"
```

client1

client2



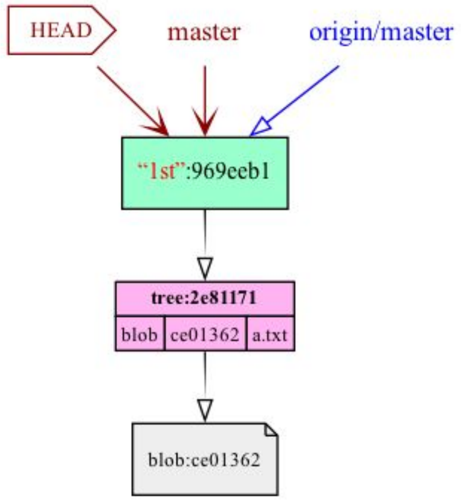
server

Example

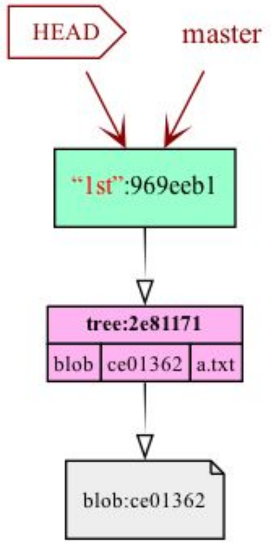
```
git init --bare server
git init client1
git init client2
cd client1
git remote add origin ../server
cd ../client2
git remote add origin ../server
echo hello > a.txt
git add a.txt
git commit -m “1st”
git push origin master
```

client1

client2



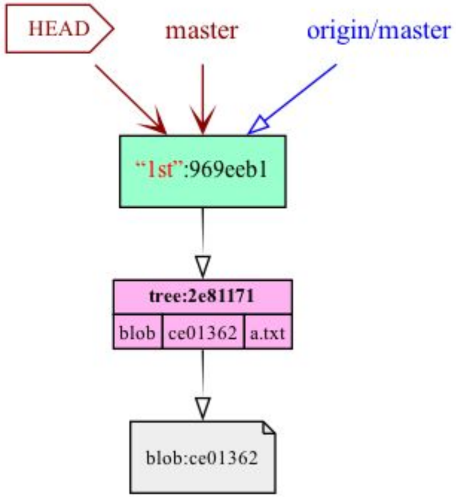
server



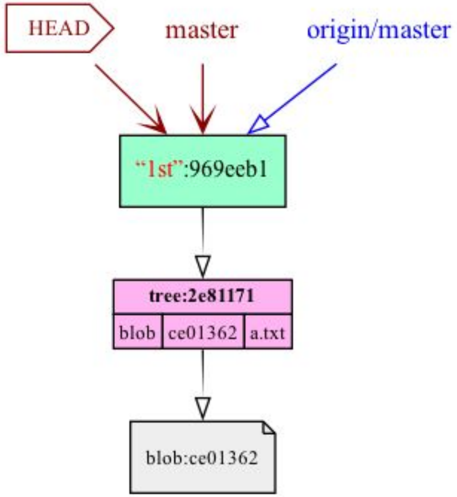
Example

```
git init --bare server
git init client1
git init client2
cd client1
git remote add origin ../server
cd ../client2
git remote add origin ../server
echo hello > a.txt
git add a.txt
git commit -m “1st”
git push origin master
cd ../client1
git pull origin master
```

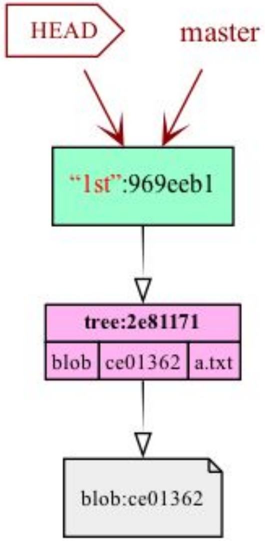
client1



client2



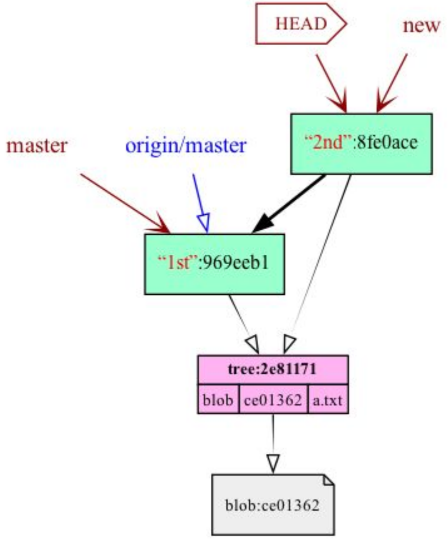
server



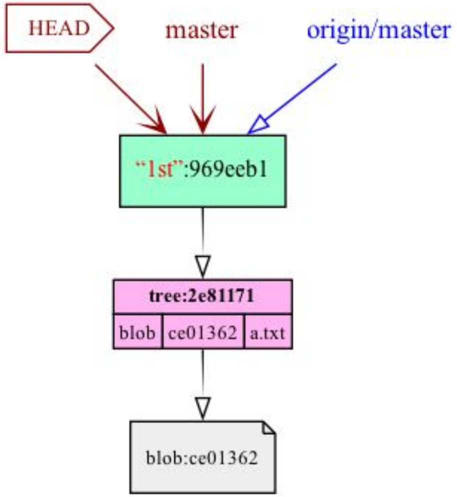
Example

```
git init --bare server
git init client1
git init client2
cd client1
git remote add origin ../server
cd ../client2
git remote add origin ../server
echo hello > a.txt
git add a.txt
git commit -m "1st"
git push origin master
cd ../client1
git pull origin master
git checkout -b new
git commit --allow-empty -m "2nd"
```

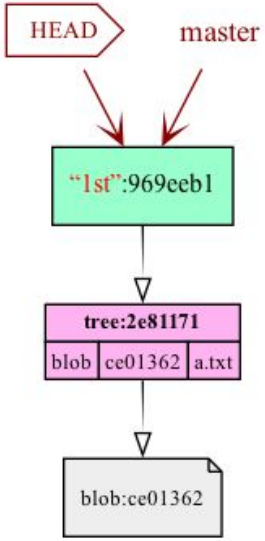
client1



client2



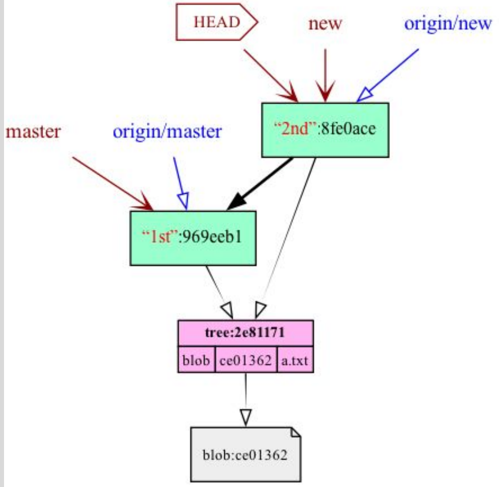
server



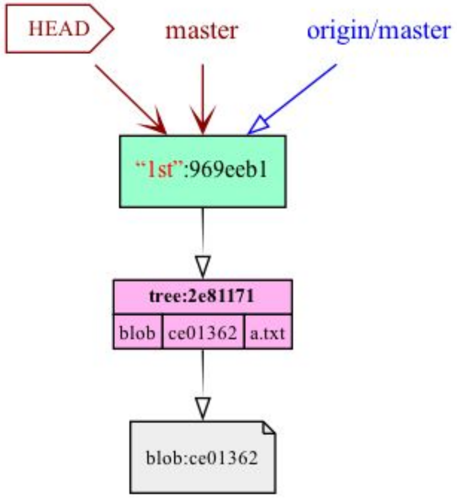
Example

```
git init --bare server
git init client1
git init client2
cd client1
git remote add origin ../server
cd ../client2
git remote add origin ../server
echo hello > a.txt
git add a.txt
git commit -m "1st"
git push origin master
cd ../client1
git pull origin master
git checkout -b new
git commit --allow-empty -m "2nd"
git push origin new
```

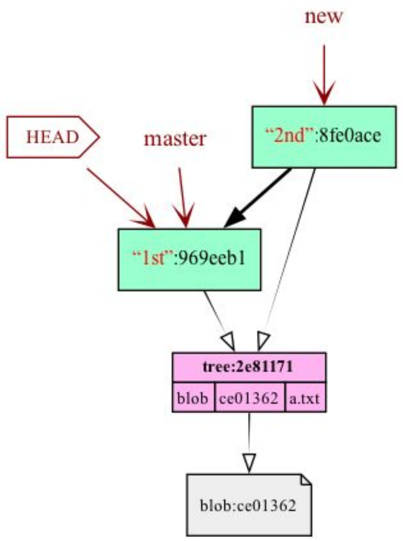
client1



client2



server



Fetch vs Pull

Fetch downloads updates (commits, branches, tags, etc.) from a remote repository

- It doesn’t apply changes to your working directory (i.e it neither checkout to remote branch nor merge it with your active local branch)
- Syntax: *git fetch [remote_name]*

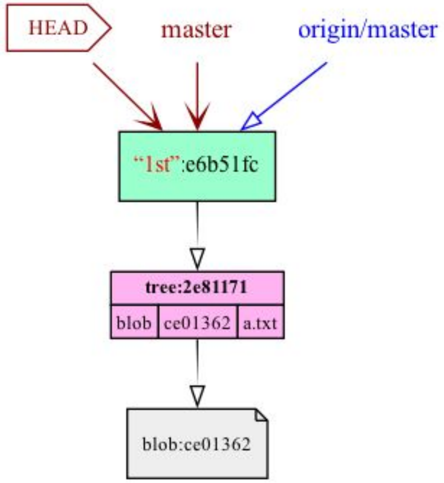
Pull = fetch + merge

Example

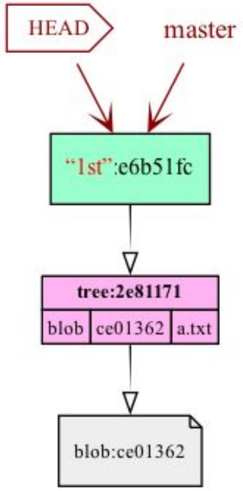
```
git init --bare server
git init client1
git init client2
cd client1
git remote add origin ../server
cd ../client2
git remote add origin ../server
echo hello > a.txt
git add a.txt
git commit -m “1st”
git push origin master
```

client1

client2



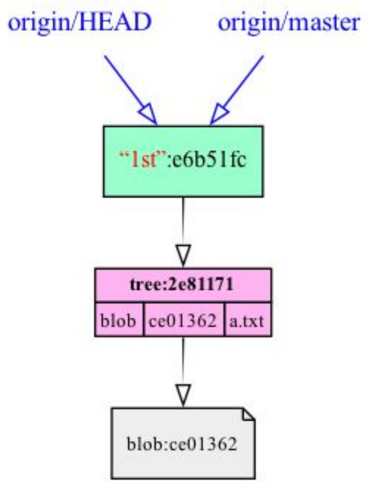
server



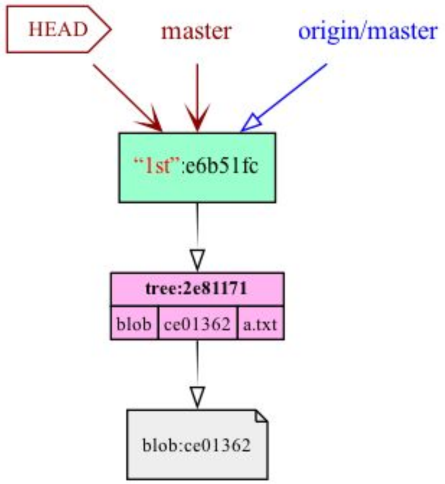
Example

```
git init --bare server
git init client1
git init client2
cd client1
git remote add origin ../server
cd ../client2
git remote add origin ../server
echo hello > a.txt
git add a.txt
git commit -m "1st"
git push origin master
cd ../client1
git fetch origin
```

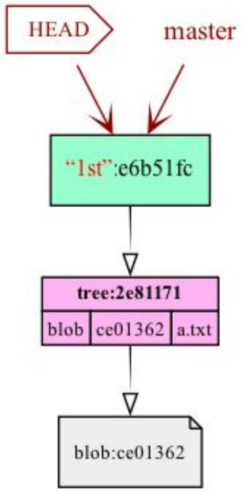
client1



client2



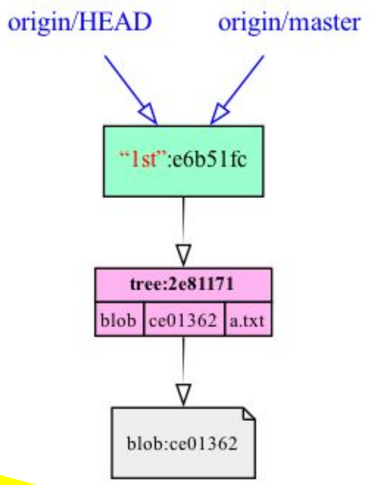
server



Example

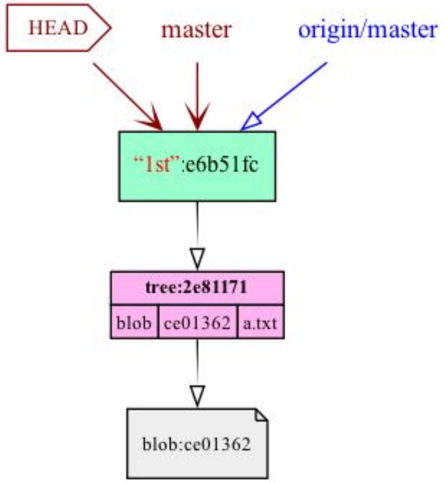
```
git init --bare server
git init client1
git init client2
cd client1
git remote add origin ../server
cd ../client2
git remote add origin ../server
echo hello > a.txt
git add a.txt
git commit -m "1st"
git push origin master
cd ../client1
git fetch origin
```

client1

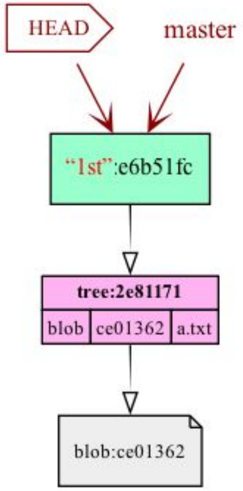


No update in working directory

client2



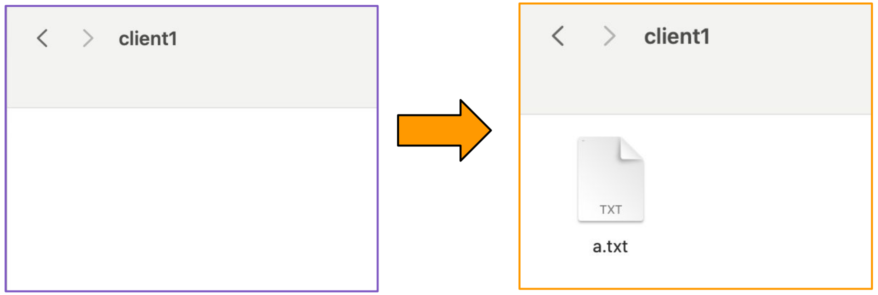
server



Example

```
git init --bare server
git init client1
git init client2
cd client1
git remote add origin ../server
cd ../client2
git remote add origin ../server
echo hello > a.txt
git add a.txt
git commit -m “1st”
git push origin master
cd ../client1
git fetch origin
git checkout origin/master
```

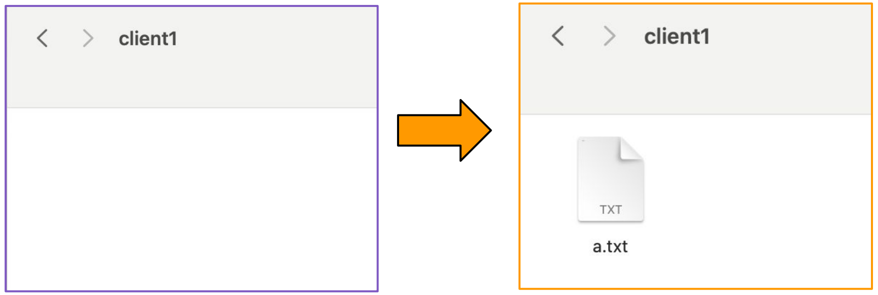
You can check out the branches stored in origin just like you would with any local branches.



Example

```
git init --bare server
git init client1
git init client2
cd client1
git remote add origin ../server
cd ../client2
git remote add origin ../server
echo hello > a.txt
git add a.txt
git commit -m “1st”
git push origin master
cd ../client1
git fetch origin
git checkout origin/master
git checkout master
git merge origin/master
```

You can check out the branches stored in origin just like you would with any local branches.



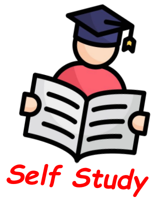
To incorporate changes for future developments, you should merge this branch with your local branch.

Thank you for your attention …

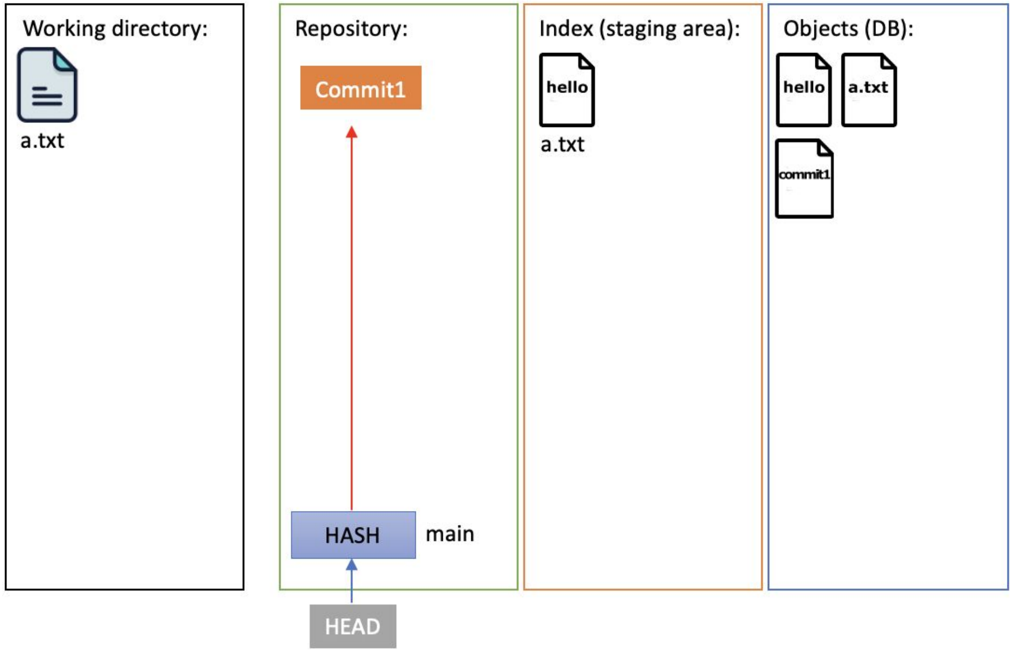
Self Study

Practice

Adding a File



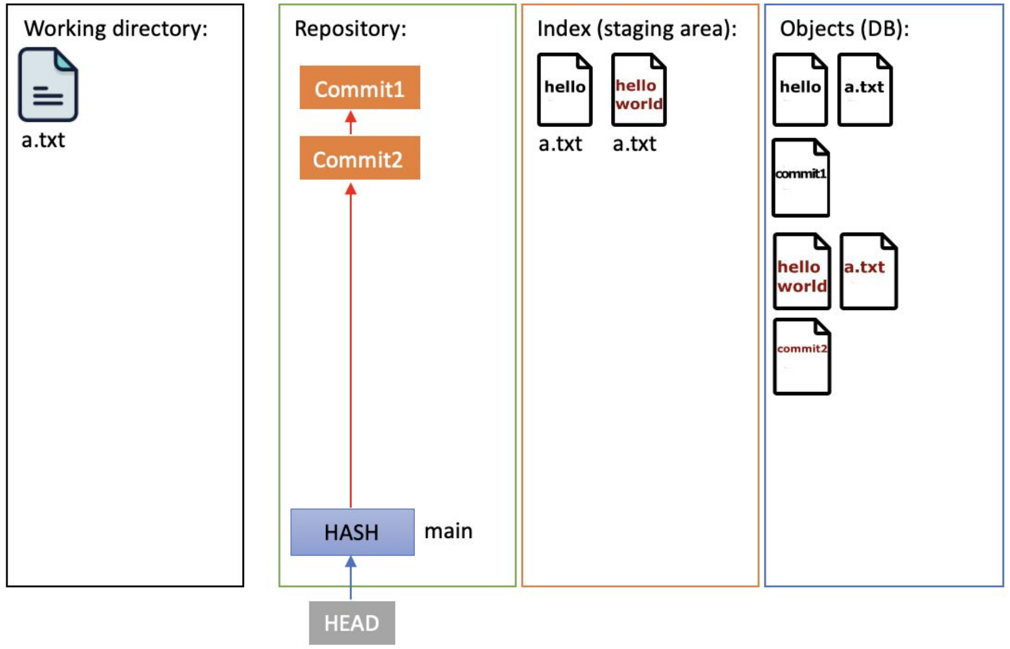
```
git init test

cd test

echo hello > a.txt

git add a.txt

git commit -m "commit 1"
```



Modifying a File

Self Study

```
git init test
cd test
echo hello > a.txt
git add a.txt
git commit -m "commit 1"
echo world >> a.txt
git add a.txt
git commit -m "commit 2"
git cat-file --batch-all-objects --batch-check
```



To see the content of an object: *git cat-file -p HASH*
Show All the Objects: *git cat-file --batch-all-objects --batch-check*

Exploring Git internals

Self Study

Branch: A pointer in file!

- *git branch feature1*

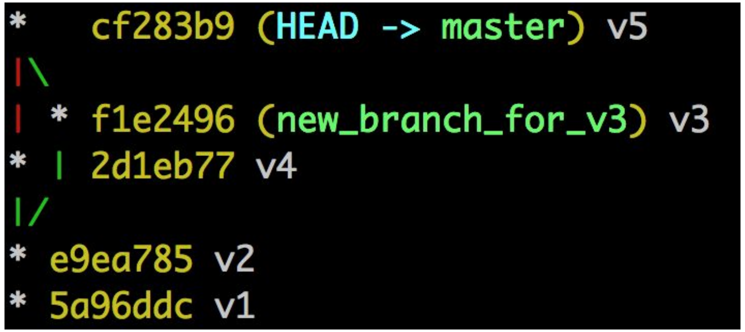
  Equivalent to: *echo COMMIT_HASH > .git/refs/heads/feature1*

- *git checkout feature1*

  Equivalent to: modify *HEAD to ref: refs/heads/feature1*

Use *git log* command to see commit pointers

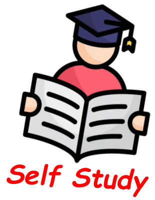
Ex: git --oneline --graph
Ex: to format: git --format="%an: %s"



```
*   cf283b9 (HEAD -> master) v5
|\
| * f1e2496 (new_branch_for_v3) v3
* | 2d1eb77 v4
|/
* e9ea785 v2
* 5a96ddc v1
```

https://www.benkanouse.com/the-git-graph/

Multiple repositories



We can create different repositories and connect them using *git remote add* command

- We can pull or push the entire **objects** and **pointers** of remote repository
- Does pull/push hold the graph structure ?
    - Of Course yes. Because it only gets or sends some files

Multiple repositories

Self Study

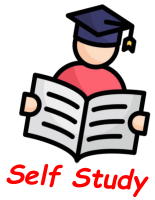
```
git init test1
cd test1
echo hello > a.txt
git add a.txt
git commit -m “commit 1”
git branch newfeature
git checkout newfeature
echo world >> a.txt
git add a.txt
git commit -m “commit 2”
git log
git cat-file --batch-all-objects --batch-check
```

```
cd ..
git init test2
git remote add test1 ../test1
git pull test1
git log
git cat-file --batch-all-objects --batch-check
echo HASH > .git/refs/heads/main
```

Self Study

Plumbing commands

Git plumbing commands

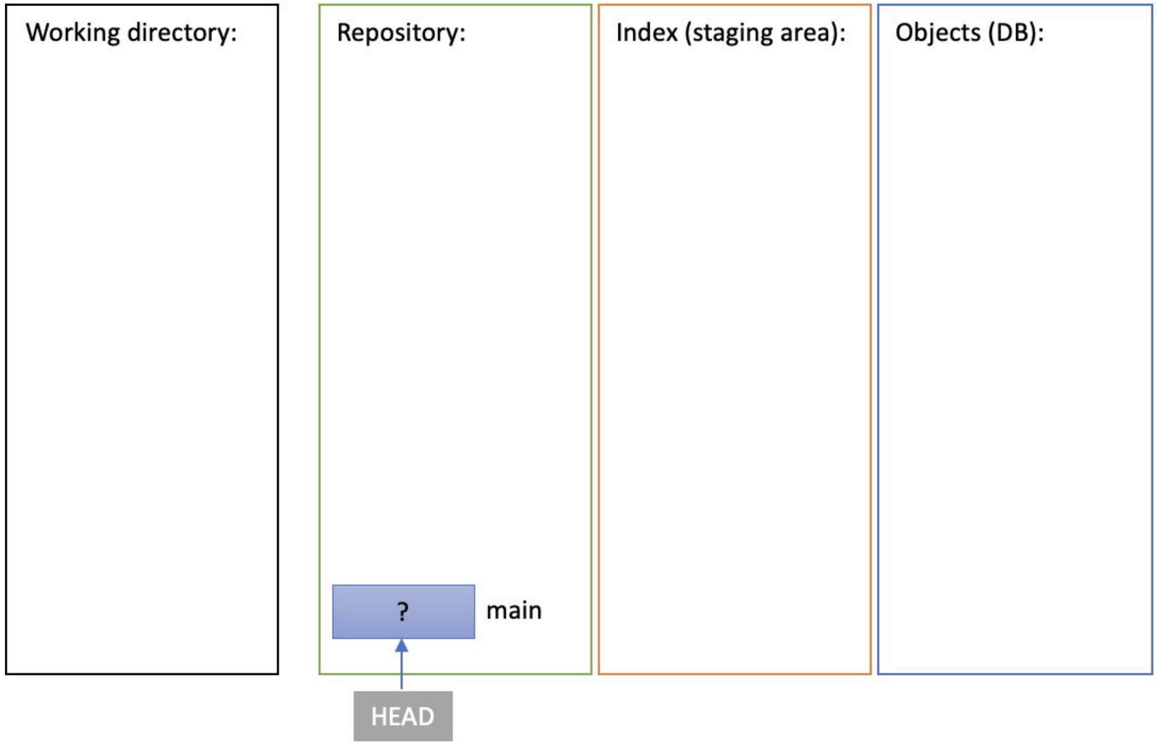


Instead of high level commands (push, pull, etc.) we do things manually

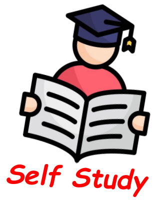
- Step by step instructions how to create a commit
- Adding a second commit: exploring the effect on the mini-fs
- Third commit: Shaping the object graph by hand

Initialize a repository

```
mkdir test

cd test

mkdir .git

mkdir .git/objects

mkdir .git/refs

mkdir .git/refs/heads

echo ref: refs/heads/main > .git/HEAD

git status
```



Preparing the First Commit

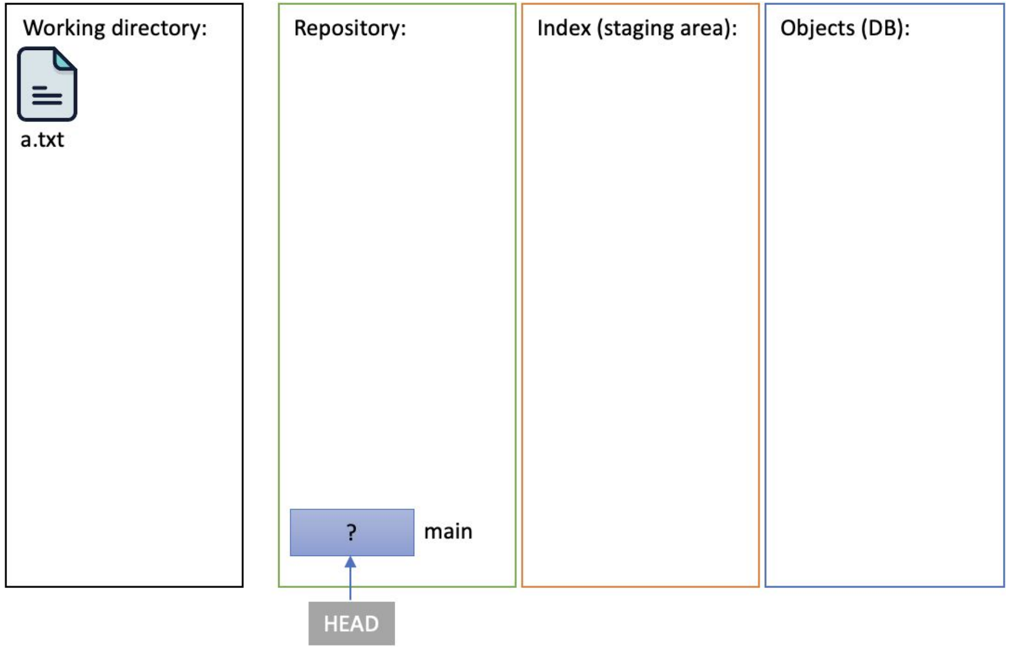


Create a file a.txt with the content ‘hello’ and commit it with message ‘commit 1’

1. Create your data

Self Study

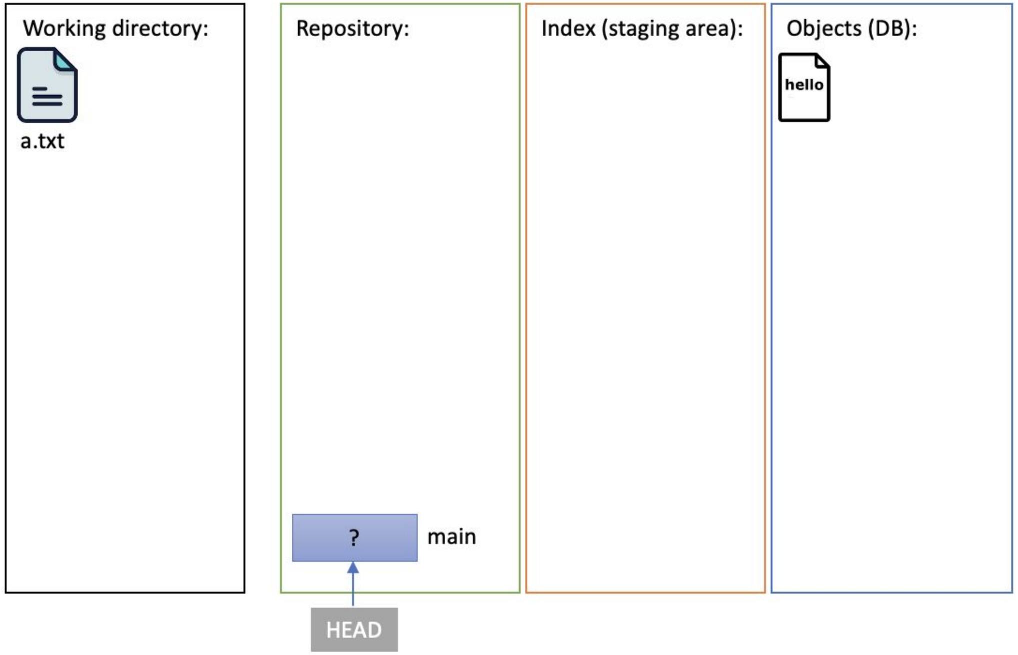
```
echo hello > a.txt
```



2. Import your data

Self Study

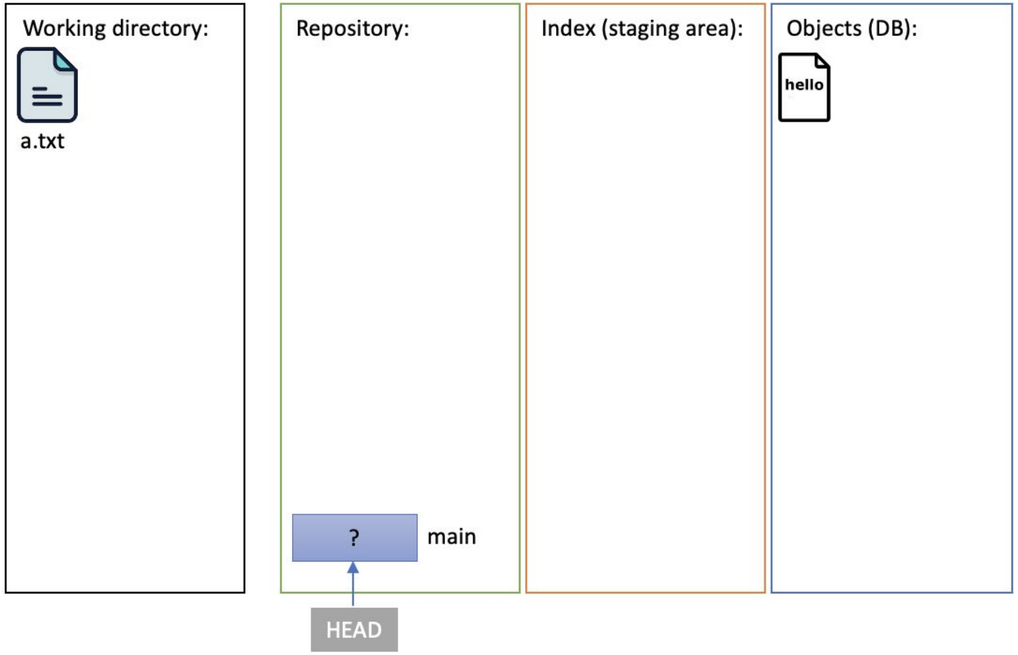
```
echo hello > a.txt

cat a.txt | git hash-object --stdin -w
```



(Check your directory)

Self Study

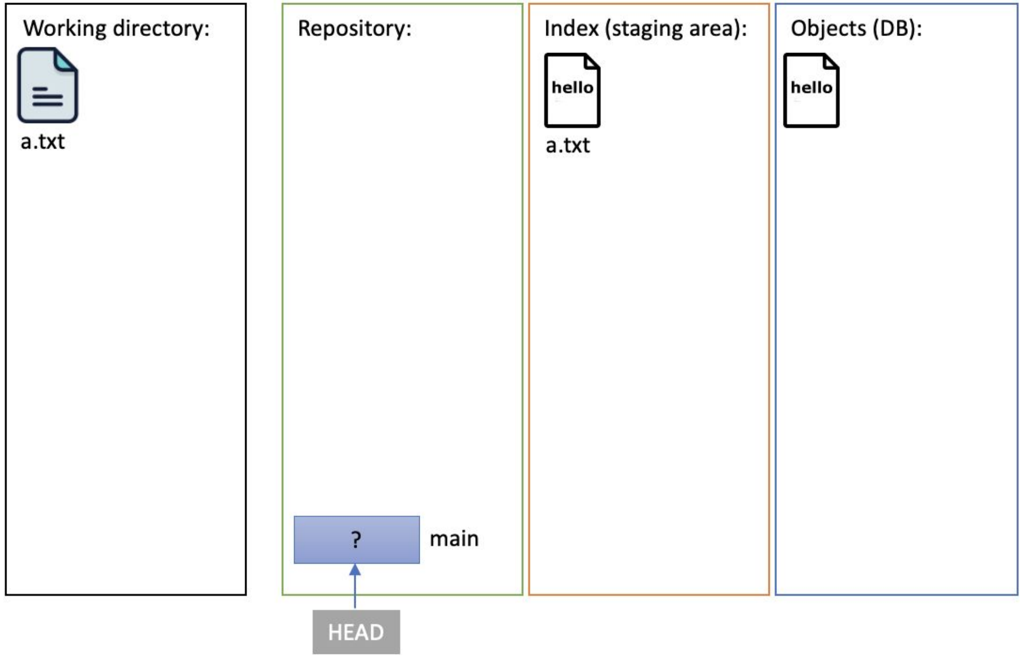
```
echo hello > a.txt

cat a.txt | git hash-object --stdin -w

git cat-file --batch-all-objects --batch-check
```



3. Index your data

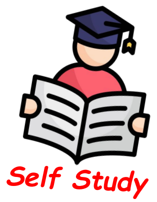
Self Study

```
echo hello > a.txt

cat a.txt | git hash-object --stdin -w

git cat-file --batch-all-objects --batch-check

git update-index --add --cacheinfo 100644 HASH a.txt
```



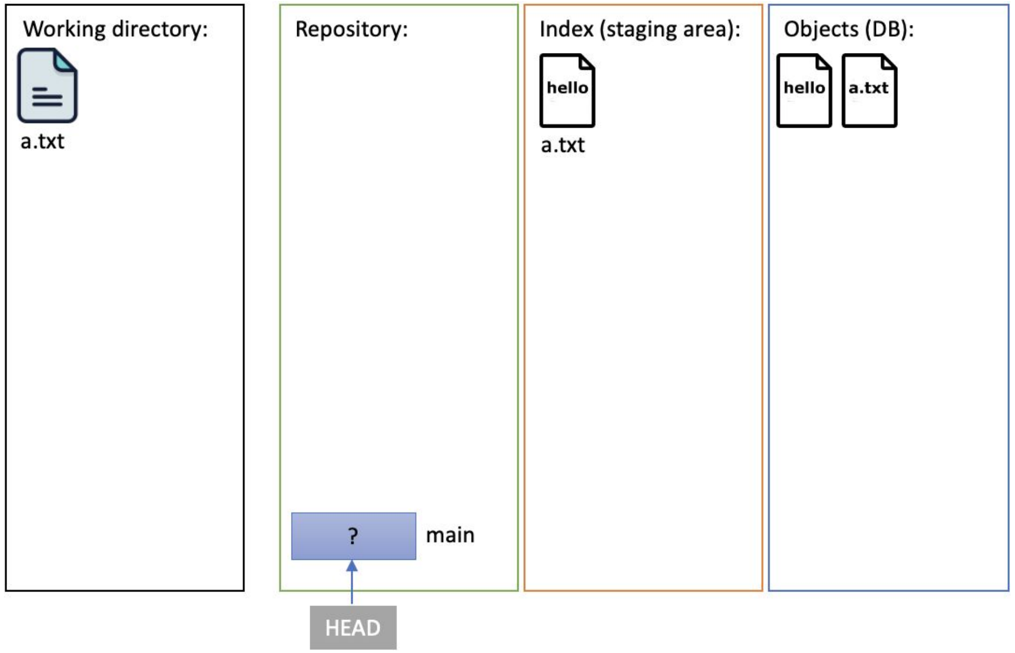
To remove from index: *git update-index --force-remove a.txt*

4. Group your data



```
echo hello > a.txt
cat a.txt | git hash-object --stdin -w
git cat-file --batch-all-objects --batch-check
git update-index --add --cacheinfo 100644 HASH a.txt
git write-tree
```

For a commit we need a tree

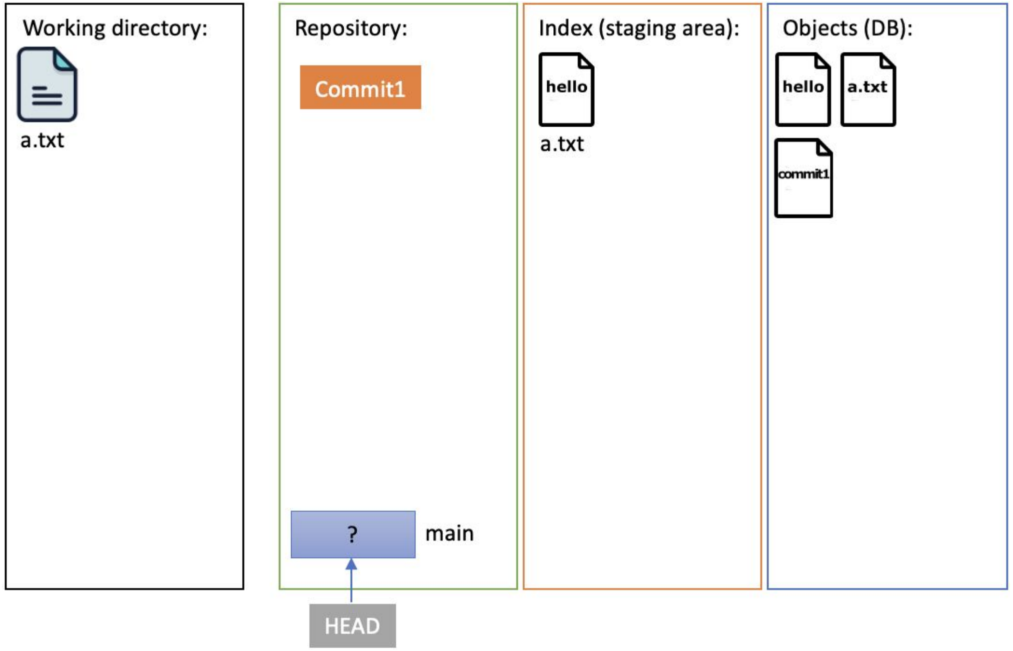


5. Snapshot your data (this IS the commit!)

Self Study

```
echo hello > a.txt

cat a.txt | git hash-object --stdin -w

git cat-file --batch-all-objects --batch-check

git update-index --add --cacheinfo 100644 HASH a.txt

git write-tree

git commit-tree TREE_HASH -m "commit 1"
```

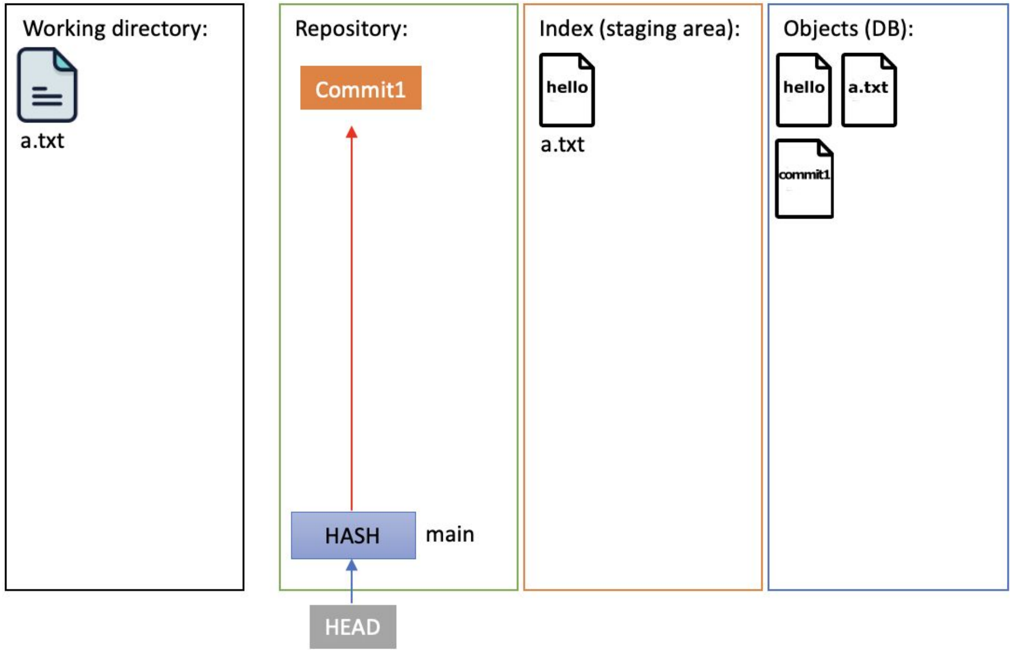
For a commit with parents: -p H1 -p H2, …



6. Help others to find your commit

Self Study

```
echo hello > a.txt
cat a.txt | git hash-object --stdin -w
git cat-file --batch-all-objects --batch-check
git update-index --add --cacheinfo 100644 HASH a.txt
git write-tree
git commit-tree TREE_HASH -m "commit 1"
echo COMMIT1_HASH > .git/refs/heads/main
```



Second Commit: modification

Self Study

```
echo hello > a.txt
cat a.txt | git hash-object --stdin -w
git cat-file --batch-all-objects --batch-check
git update-index --add --cacheinfo 100644 HASH a.txt
git write-tree
git commit-tree TREE_HASH -m “commit 1”
echo COMMIT1_HASH > .git/refs/heads/main
```

```
echo world >> a.txt
cat a.txt | git hash-object --stdin -w
git update-index --add --cacheinfo 100644 HASH a.txt
git write-tree
git commit-tree TREE_HASH -m “commit 2” -p COMMIT1_HASH
git status
git log
```

What is the problem?

Second Commit: moving pointer

Self Study

```
echo hello > a.txt
cat a.txt | git hash-object --stdin -w
git cat-file --batch-all-objects --batch-check
git update-index --add --cacheinfo 100644 HASH a.txt
git write-tree
git commit-tree TREE_HASH -m “commit 1”
echo COMMIT1_HASH > .git/refs/heads/main
```

```
echo world >> a.txt
cat a.txt | git hash-object --stdin -w
git update-index --add --cacheinfo 100644 HASH a.txt
git write-tree
git commit-tree TREE_HASH -m “commit 2” -p COMMIT1_HASH
git status
git log
echo COMMIT1_HASH > .git/refs/heads/main
git log
```

Third Commit: multiple parents

Self Study

```
echo hello > a.txt
cat a.txt | git hash-object --stdin -w
git cat-file --batch-all-objects --batch-check
git update-index --add --cacheinfo 100644 HASH a.txt
git write-tree
git commit-tree TREE_HASH -m “commit 1”
echo COMMIT1_HASH > .git/refs/heads/main
```

```
echo world >> a.txt
cat a.txt | git hash-object --stdin -w
git update-index --add --cacheinfo 100644 HASH a.txt
git write-tree
git commit-tree TREE_HASH -m “commit 2” -p COMMIT1_HASH
git status
git log
echo COMMIT1_HASH > .git/refs/heads/main
git log
git commit-tree TREE_HASH -m “commit 3” -p COMMIT1_HASH -p COMMIT2_HASH
echo COMMIT3_HASH > .git/refs/heads/main
git log --graph --oneline
```

Self Study

Log considers active branch

```
echo COMMIT1_HASH > .git/refs/heads/branch1
echo COMMIT2_HASH > .git/refs/heads/branch2
git checkout branch1
git log --graph --oneline
git checkout branch2
git log --graph --oneline
git log --all --graph --oneline
```