

# Git 2 branch, conflitti

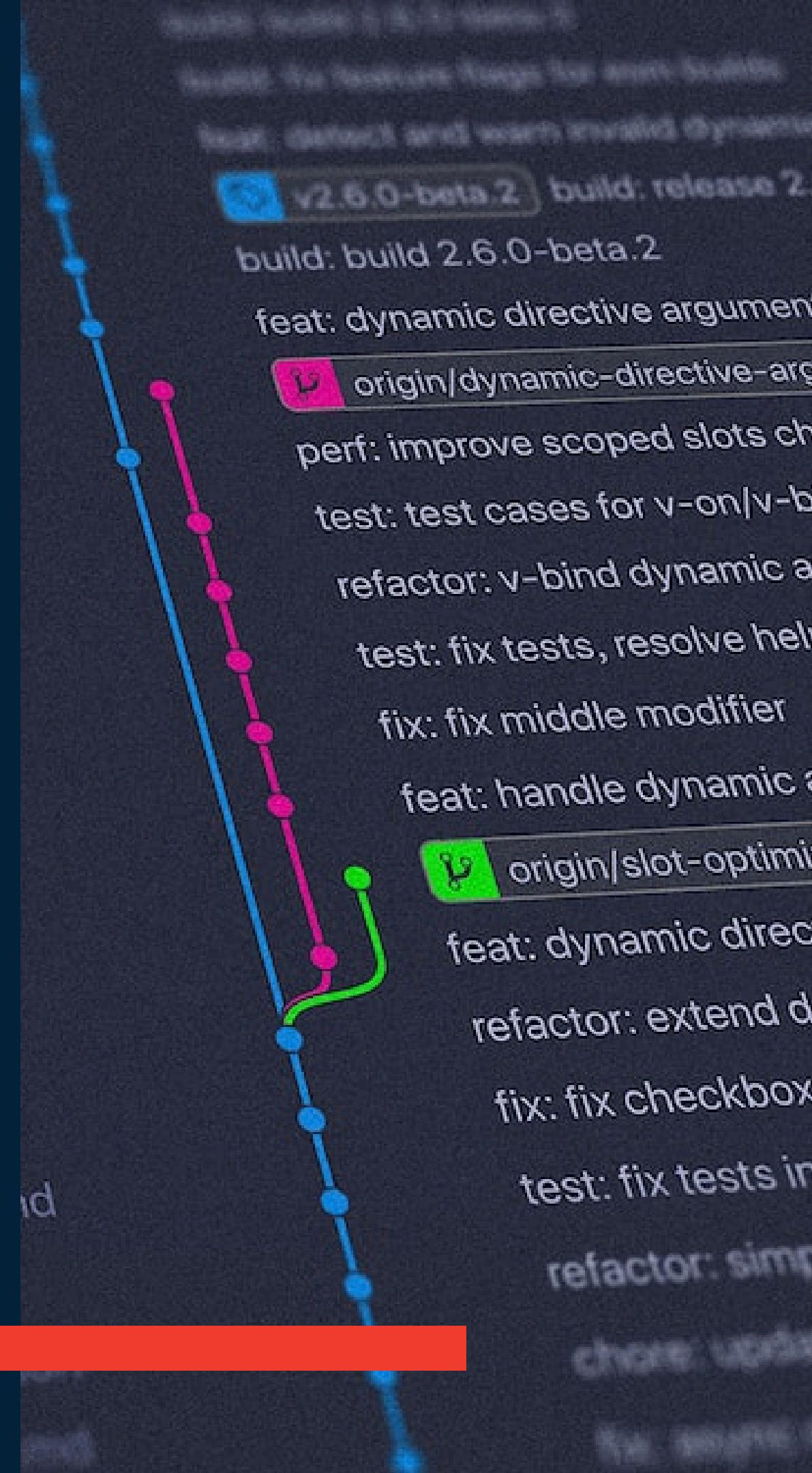
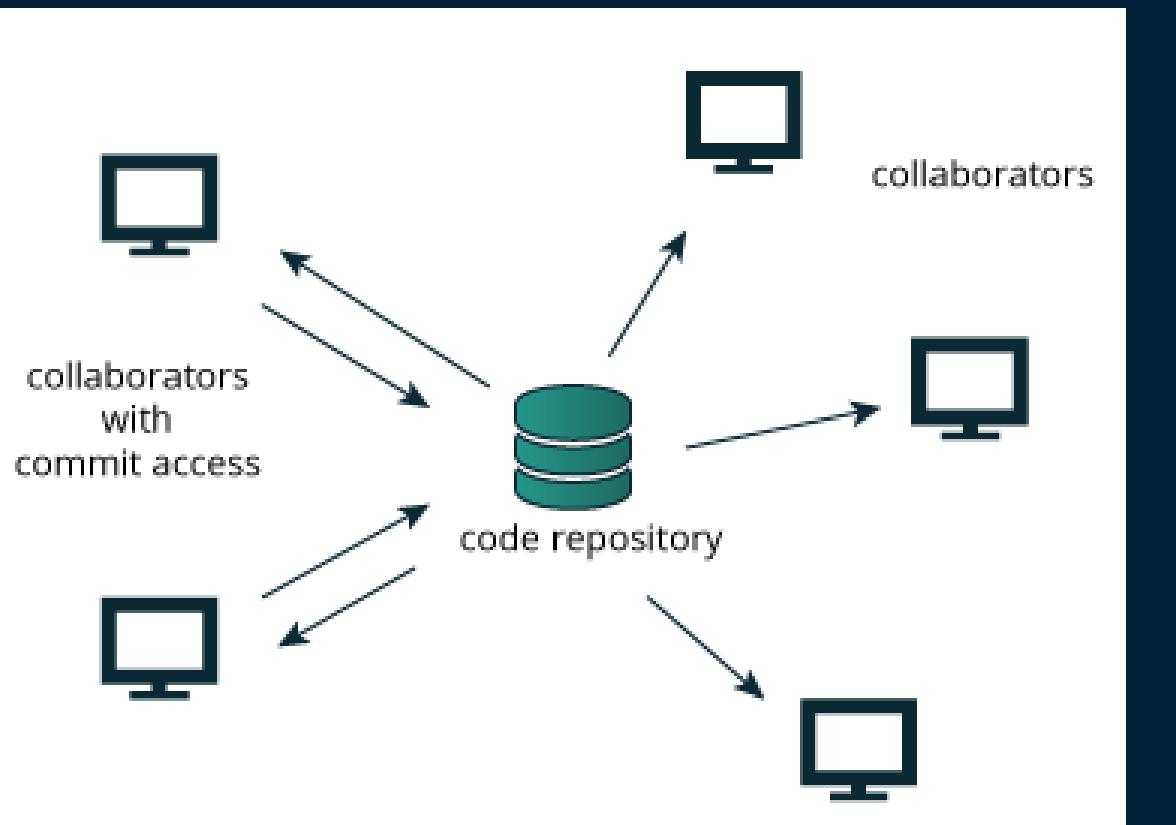
[risorse.vercel.app/lab/2025](https://risorse.vercel.app/lab/2025)

Alice Benatti, Mattia Graziani



# Cos'è git?

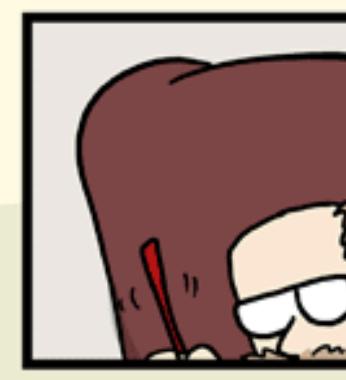
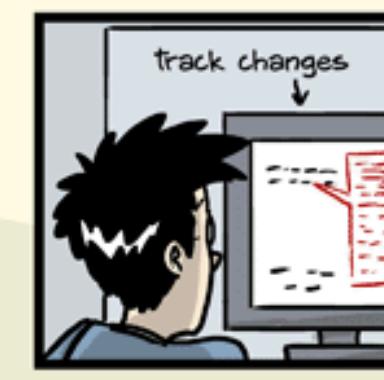
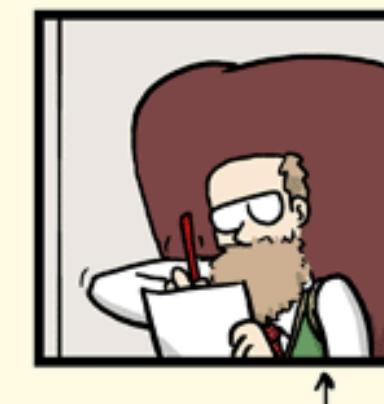
- ✓ • **sistema di versionamento** = tenere traccia delle versioni dei file, ottenendo una cronologia di tutte le modifiche
- **distribuito** = chiunque ha una copia completamente locale della *repository*



# Perché usarlo?

- ✓ • Cronologia trasparente su tutti i cambiamenti dei file
- ✓ • Ripristinare subito un qualsiasi **stato precedente**
- Collaborare a copie dinamiche diverse dello stesso progetto

"FINAL".doc



JORGE CHAM © 2012

# Installare git Windows

- via Command Line con *winget*  
`winget install --id Git.Git -e --source winget`
- scaricare dal [link del sito ufficiale](#)

[Click here to download](#) the latest (2.42.0) 64-bit version of Git for Windows. This is the most recent [maintained build](#). It was released [about 2 months ago](#), on 2023-08-30.

[Other Git for Windows downloads](#)



# Installare git Linux

con Ubuntu/Mint/Debian

```
$ sudo apt-get update  
$ sudo apt-get install git
```

con Fedora

```
$ yum install git (up to Fedora 21)  
$ dnf install git (Fedora 22 and later)
```

[[per altre distribuzioni clicca qui](#)]



# Installare git MacOS

Ci sono diversi package manager che permettono di installare facilmente git, scegli quello che preferisci:

- ***Homebrew***

Installa [homebrew](#) se non lo hai già, e lancia il comando:

```
$ brew install git
```

- ***MacPorts***

Installa [MacPorts](#) se non lo hai già, e lancia il comando:

```
$ sudo port install git
```

- ***Xcode***

Apple fornisce un pacchetto binario Git con [Xcode](#).

[per ulteriori info seguite la [guida sul sito ufficiale](#)]



# Configurazione Profilo

Configurare *git* con i vostri dati:  
usate la mail che preferite, sarà poi associata con il vostro user.name alle varie operazioni  
che svolgerai con git.

```
$ git config --global user.name "Nome Cognome"  
$ git config --global user.email "nome.cognome@studio.unibo.it"  
$ git config --global init.defaultBranch main
```

# Reminder

```
$ git init [<directory>]
```

**Inizializzare una repository Git vuota in una nuova <directory>**

```
$ git add [<path>...]
```

**Aggiunge le modifiche specificate in <path> agli *staged files***

```
$ git commit -m "<msg>"
```

**Registra le modifiche agli *staged files* commentandole con un <msg>**

```
$ git status
```

**Mostriamo lo stato attuale della nostra repository**

```
$ git log [--graph]
```

**Mostra lo storico dei commit**

```
$ git diff
```

**Mostra le differenze presenti nei file rispetto all'ultimo commit**

```
$ git restore <path>
```

**Ripristina un file alla versione dell'ultimo commit**

```
$ git checkout
```

**Per muoversi da un commit all'altro**

```
$ git stash
```

**Mette da parte la *staging area* che non vogliamo committare ora**

# Warm up

Creiamo una nuova repository per questo laboratorio:

```
$ git init git2
```

Entriamo dentro la repo, creiamo un file vuoto README.md e committiamolo.

```
$ touch README.md
```

```
$ git add .
```

```
$ git commit -m "add: readme.md"
```



00



# Branch

*ramificazioni*

# git branch

Per lavorare su cose diverse basandosi sullo stesso codice mantenendo **più versioni separate** dello stesso progetto:

- quella principale;
- quella su cui stiamo sviluppando una funzione ancora sperimentale;
- quella su cui stiamo correggendo un problema...

**Non dobbiamo copiare tutta la directory della repo!**

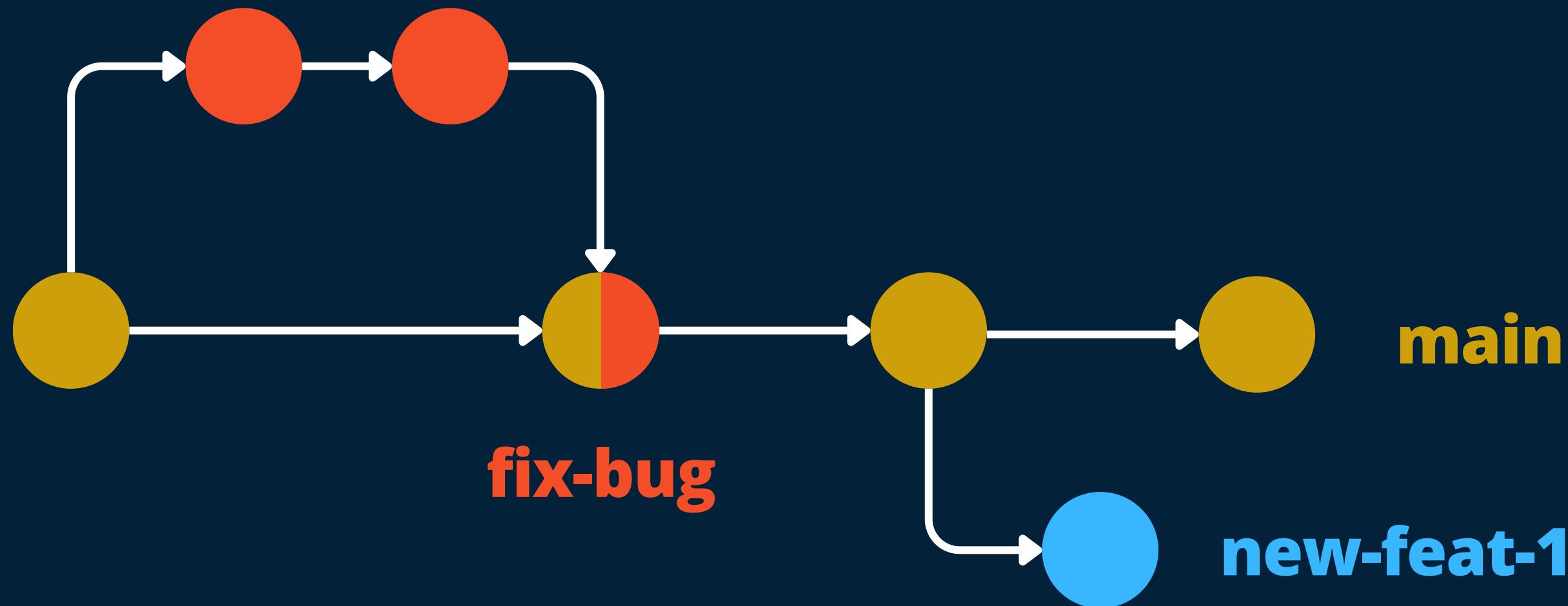
In *git*, ogni versione (*branch*, ramificazione) si alterna nella stessa cartella.

**Usiamo il seguente comando per mostrare un elenco delle branch esistenti nella repository**

```
$ git branch
```

# git branch

Esempio di una branch tipo



# git branch & git switch

01

Creiamo la branch “*mini-feat*”

```
$ git branch mini-feat
```

Spostiamoci sulla nuova branch

```
$ git switch mini-feat
```



# Creiamo un file

\$ nano mini.cpp

```
#include <iostream>
#include <vector>
#include <cstdlib>
#include <ctime>

int main() {
    std::srand(std::time(nullptr));
    std::vector<std::string> frasi = { "Apply, Develop, Maintain!",
"ADM mode: ON", "Se funziona... non toccarlo.", "Build riuscito (per
ora)."};
    std::cout << "==== ADM MINI ====\n";
    std::cout << frasi[std::rand() % frasi.size()] << "\n";
    std::cout << "Buon laboratorio!\n";
    return 0;
}
```

# 03

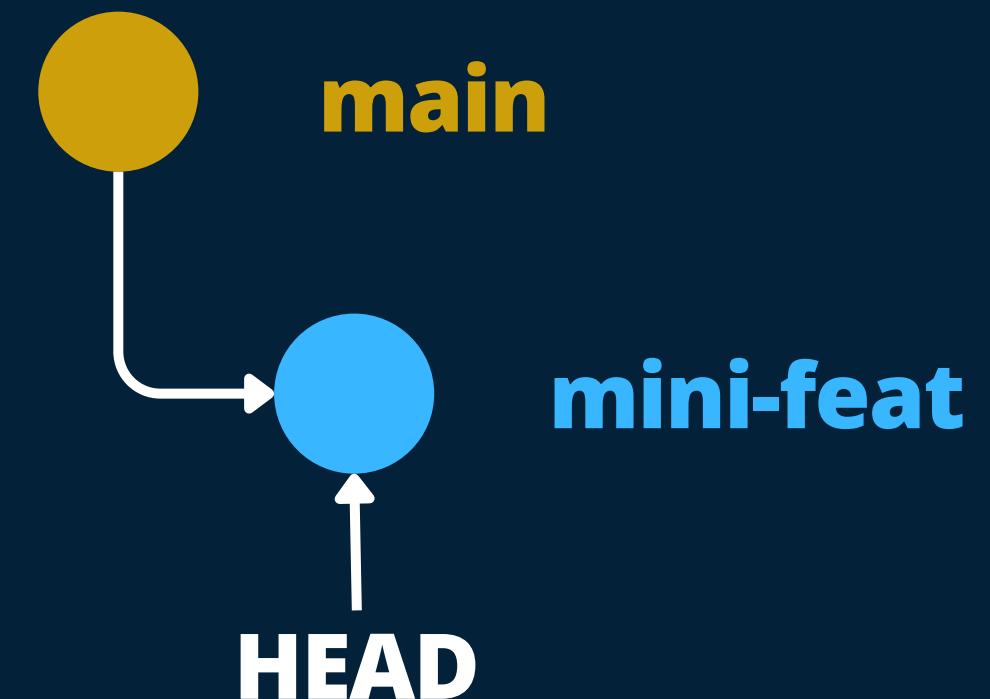
Siamo nella branch giusta? Controlliamo con:

```
$ git branch
```

Aggiungiamo alla staging area il file e committiamo

```
$ git add mini.cpp  
$ git commit -m "add: new mini feature"
```

Facendo *git log --graph --all* possiamo vedere una cosa del genere:



# 04

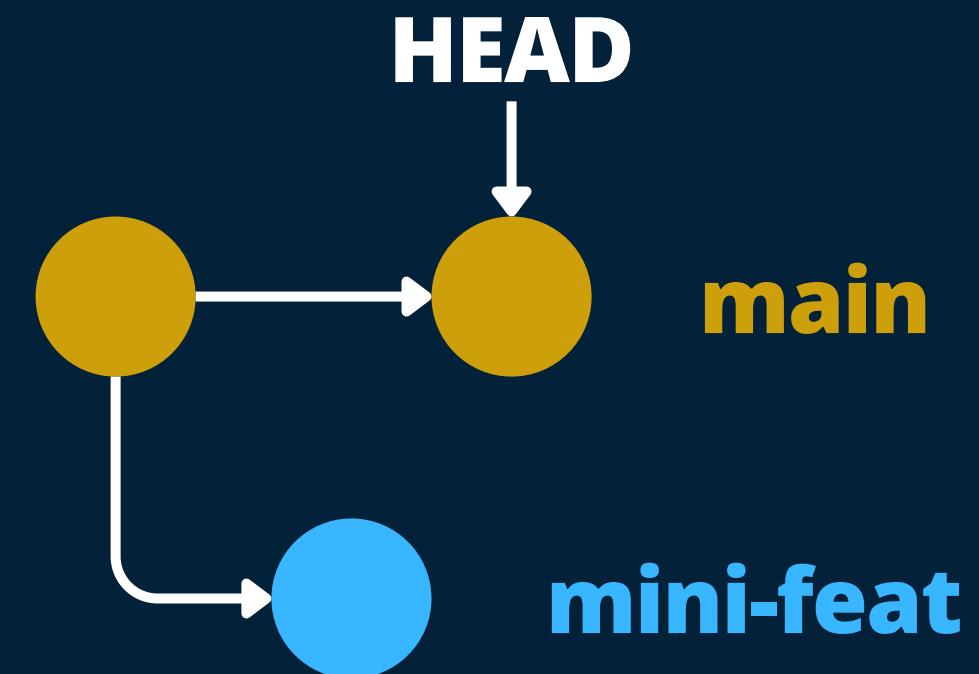
Vogliamo aggiungere un commit al main, quindi torniamo lì

```
$ git switch main
```

Modifichiamo README.md scrivendoci qualcosa e committiamolo.

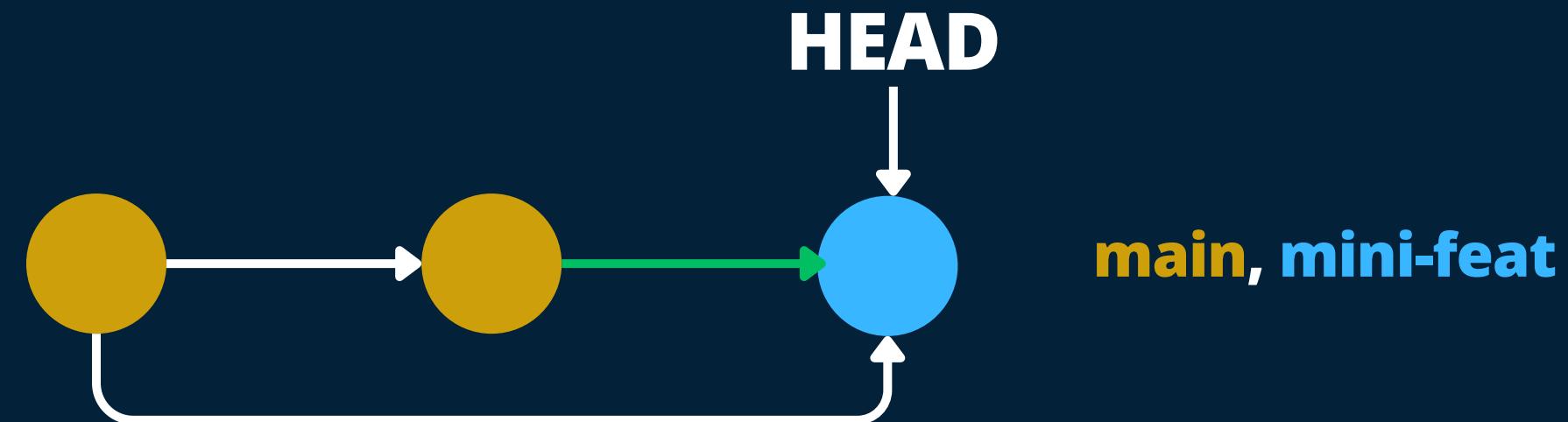
```
$ git add README.md  
$ git commit -m "chore: edit README.md"
```

Facendo adesso *git log --graph --all* possiamo vedere una situazione del genere:



# git merge

unisce i commit di due branch

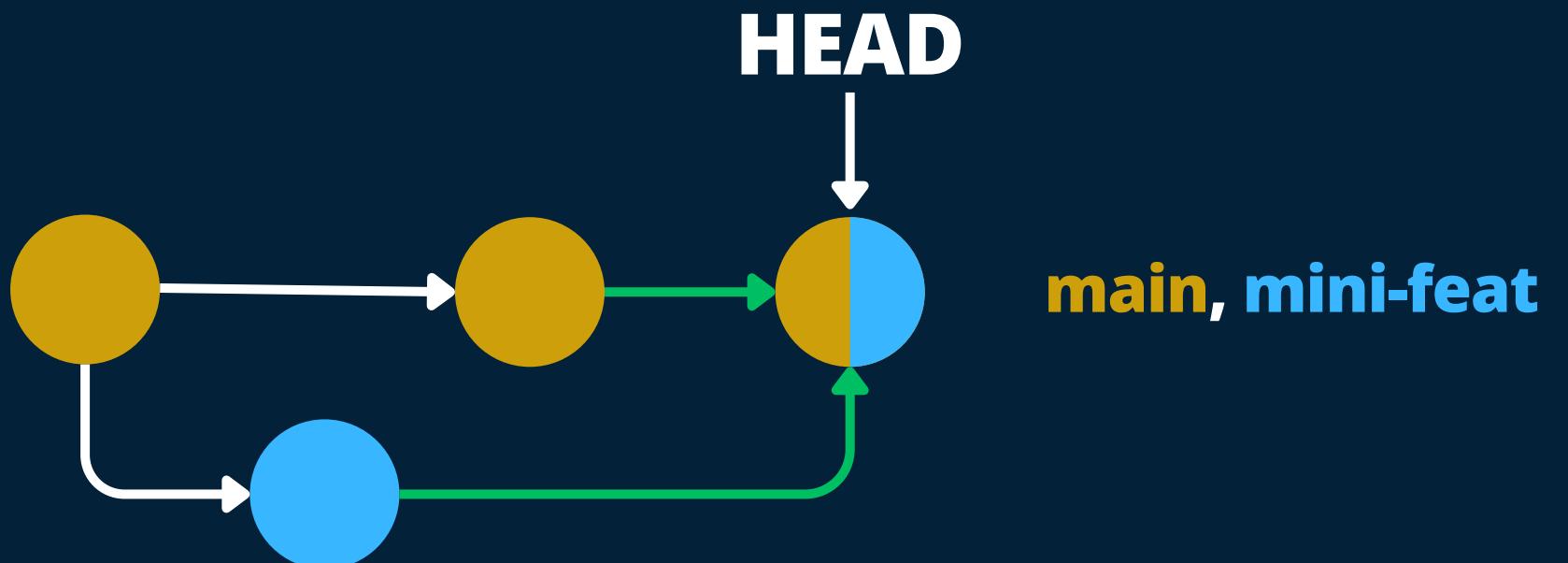
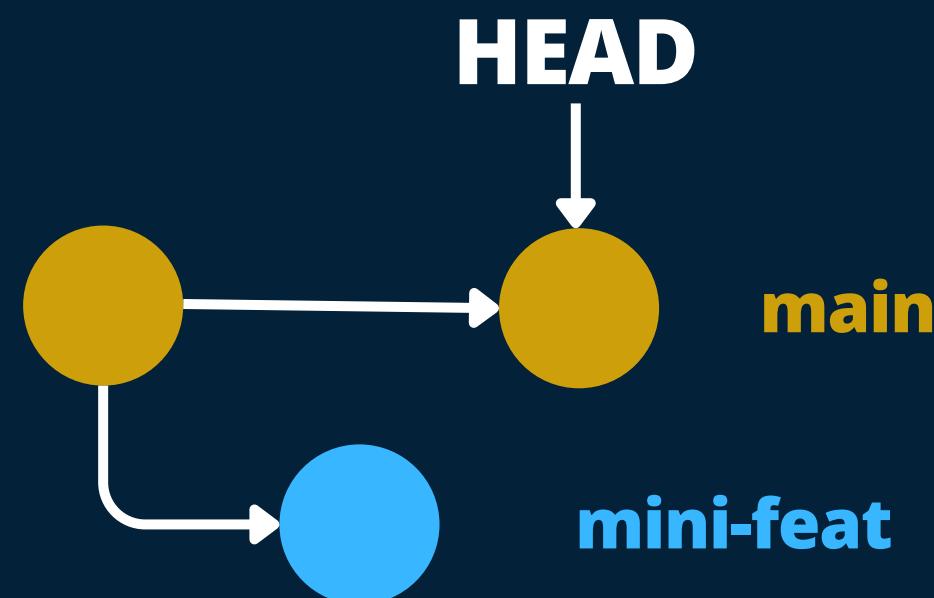


Se è possibile viene fatto il *fast-forward*, ovvero viene spostata la branch **main** allo stesso commit di **mini-feat**.

Questo è possibile solo se gli unici commit di differenza tra le due branch sono sulla branch che si sta mergiendo (nel nostro caso **mini-feat**)

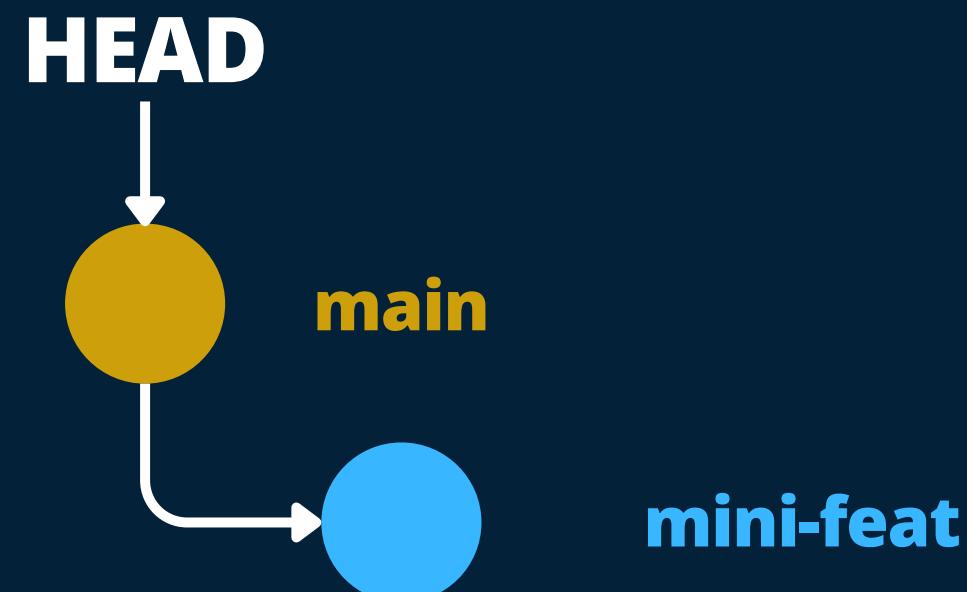
# git merge

Se non è possibile fare *fast-forward*, viene creato un commit di merge.



# git merge

È possibile forzare il commit di merge con: `--no-ff`



# git merge

**unisce i commit di due branch**

05

```
$ git merge <branch_da_mergiare>
```

Per fare *merge* della nostra branch “mini-feat” **dobbiamo spostarci sulla branch di destinazione** [“main” nel nostro caso] e da lì eseguire il comando.

Per spostarci come facciamo?

# git merge

unisce i commit di due branch

05

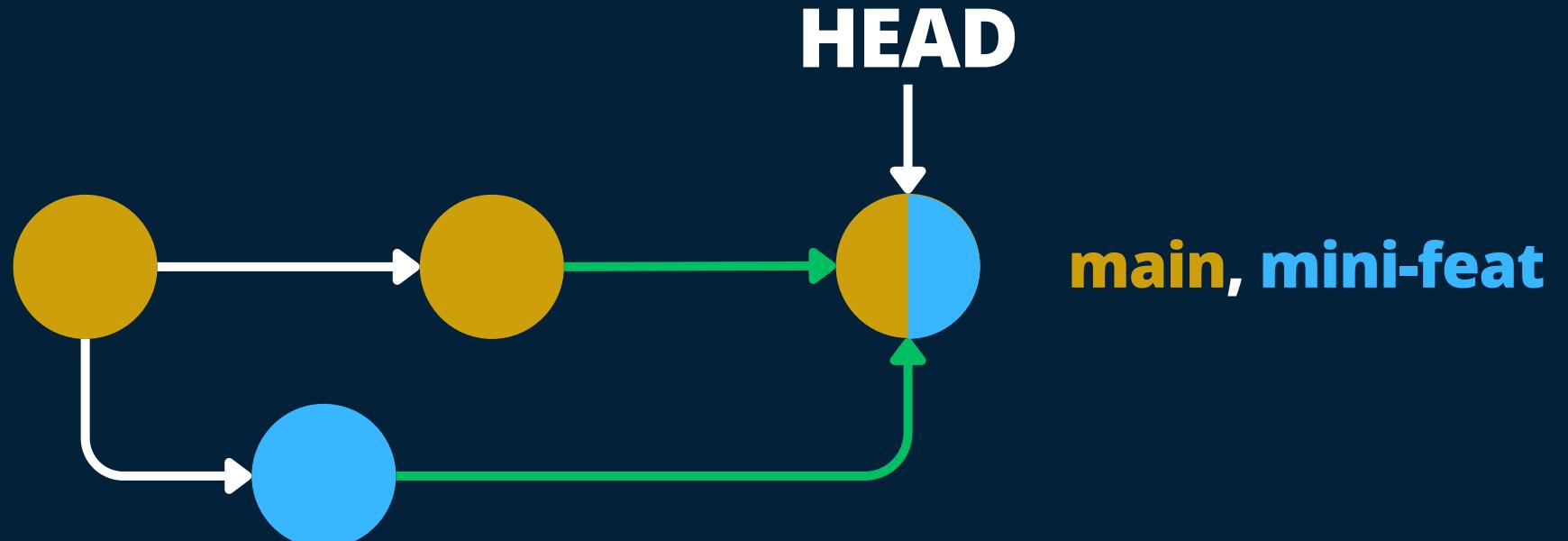
```
$ git merge <branch_da_mergiare>
```

Per fare *merge* della nostra branch “mini-feat” **dobbiamo spostarci sulla branch di destinazione** [“main” nel nostro caso] e da lì eseguire il comando.

Per spostarci come facciamo?

```
$ git switch main
```

Verifichiamo con `$ git log --graph --all`



# 06

Creiamo una nuova branch “docs”

```
$ git branch docs
```

Spostiamoci sulla nuova branch

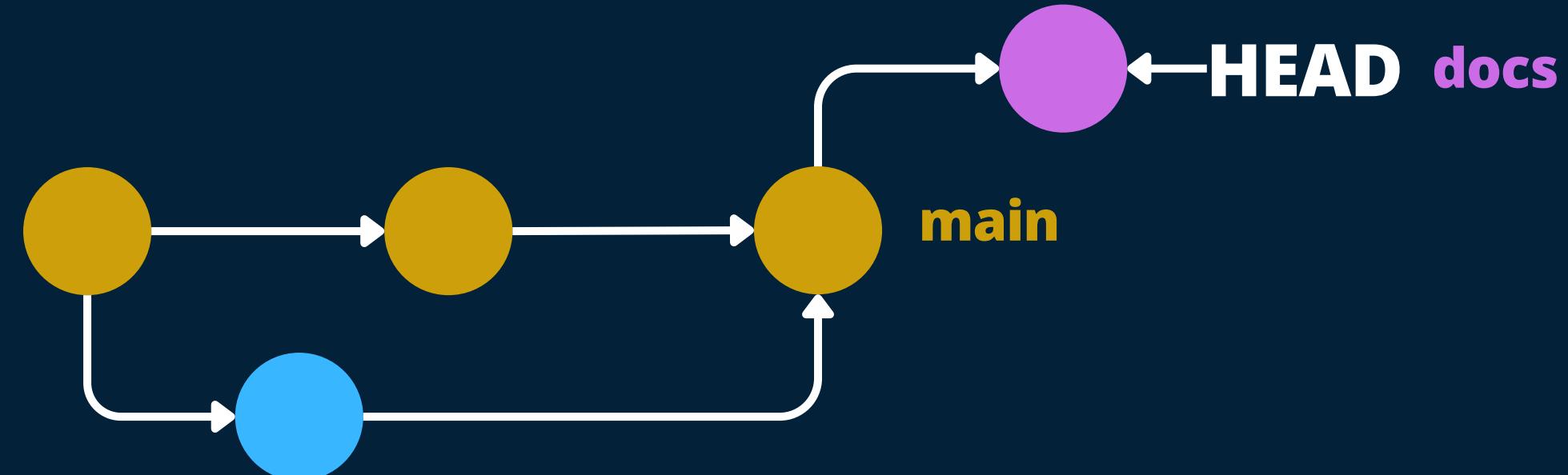
```
$ git switch docs
```

Creiamo un nuovo file e inseriamo qualcosa

```
$ nano TODO.md
```

Aggiungiamo alla staging area il file e committiamo

```
$ git add TODO.md  
$ git commit -m "add: new docs"
```



# 0'7

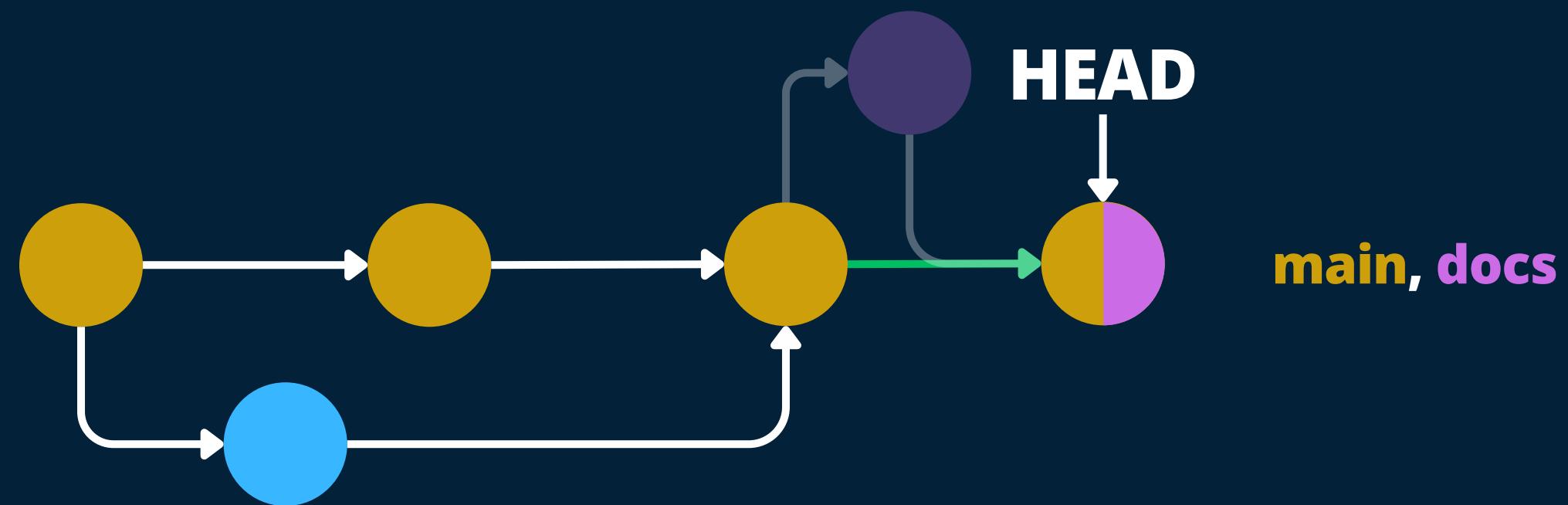
Torniamo sul branch **main** per vedere il merge con *fast-forward*

```
$ git switch main
```

Mergiamo

```
$ git merge docs
```

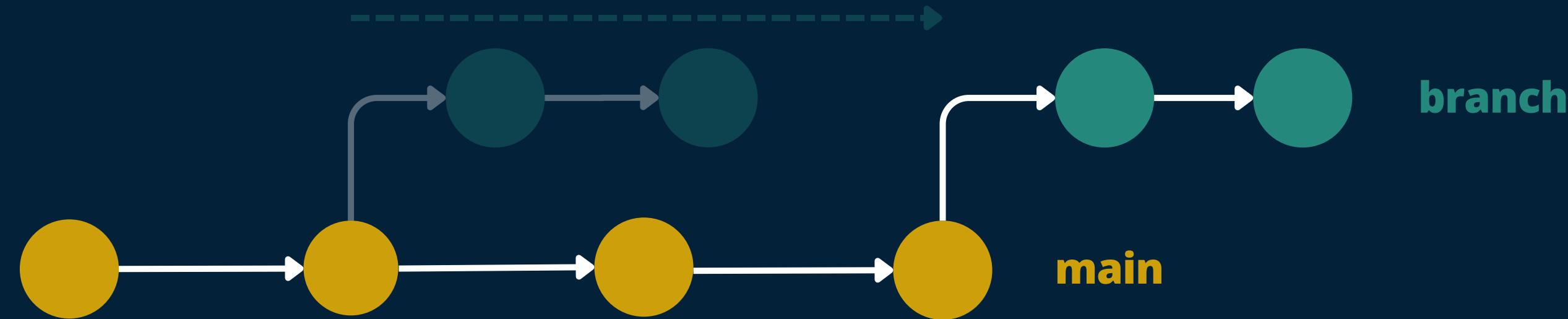
Vediamo con *git log --graph --all* la situazione attuale



# git rebase

**Sposto l'intera timeline della branch per farla partire dall'ultimo commit della branch di origine**

```
$ git rebase <branch_di_origine>
```



Dalla branch da spostare eseguiamo il comando.

# 08

**Ricreiamo una situazione per usare rebase**

Creiamo una nuova branch

```
$ git switch -c test
```

Realizziamo un file test.cpp vuoto

```
$ touch test.cpp
```

Committiamolo, dovremmo essere in questa situazione:



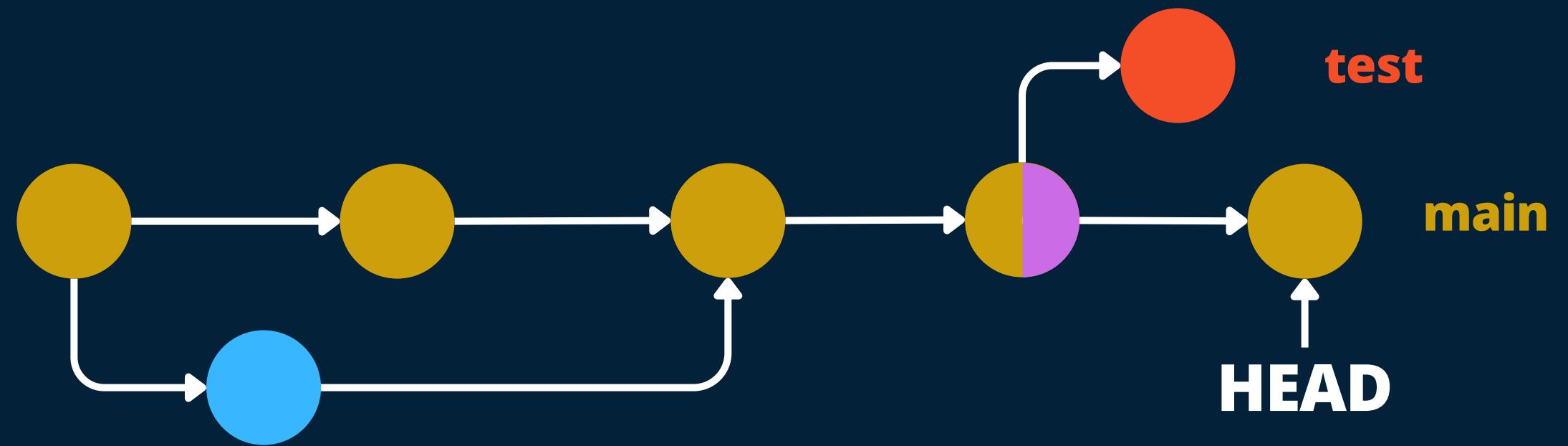
# 09

Torniamo sul main e modifichiamo il README.md

```
$ git switch main
```

```
$ nano README.md
```

... committiamo ...

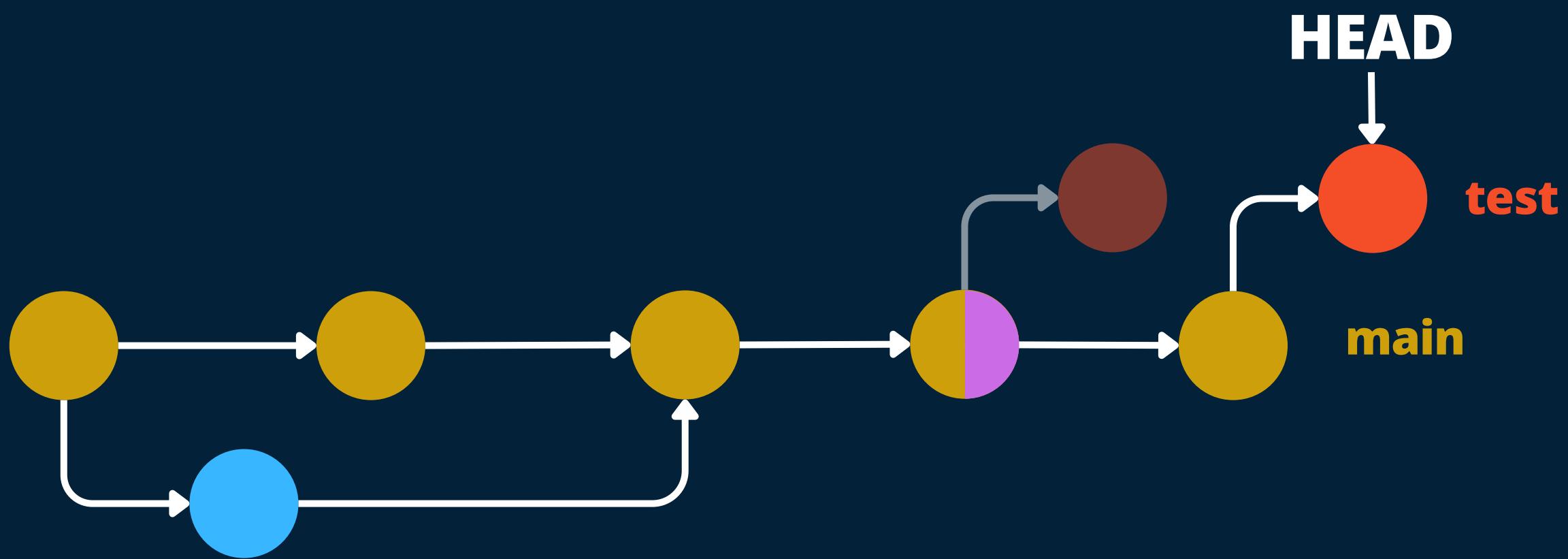


Torniamo sulla branch **test** e facciamo il rebase

```
$ git switch test
```

```
$ git rebase main
```

Committiamo



# git show

**Possiamo mostrare dettagli su uno o più commit a scelta dando un <object> univoco di riferimento**

```
$ git show [<hash_code_commit>]
```

Un modo di riferirsi a un <object> è un prefisso univoco del codice sha1 del commit.

Se non si specifica nulla, viene usato *HEAD* (il commit corrente).

Nel nostro caso, proviamo ad usarlo su un commit.

# git reset

Riporta la **branch** allo stato del **commit specificato**

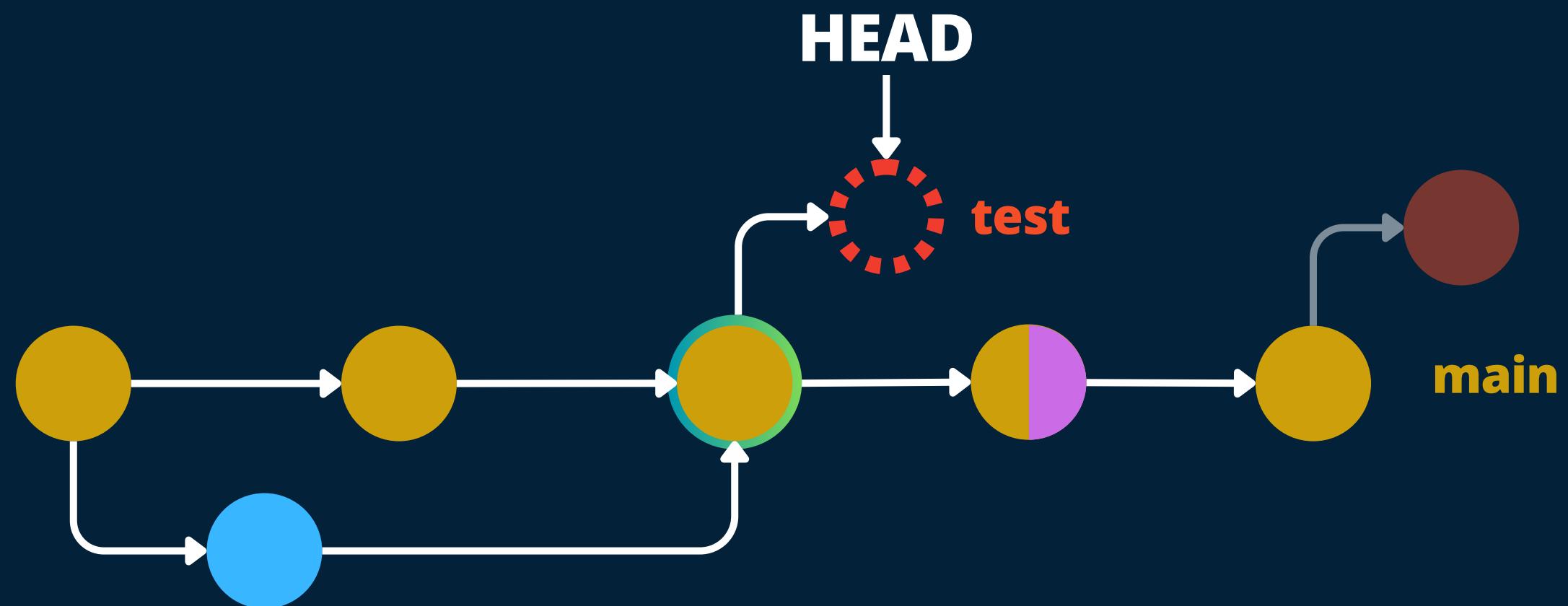
```
$ git reset <hash_code_commit>
```

Dalla branch attuale, elimina i commit fino ad *<hash\_code\_commit>* (escluso).

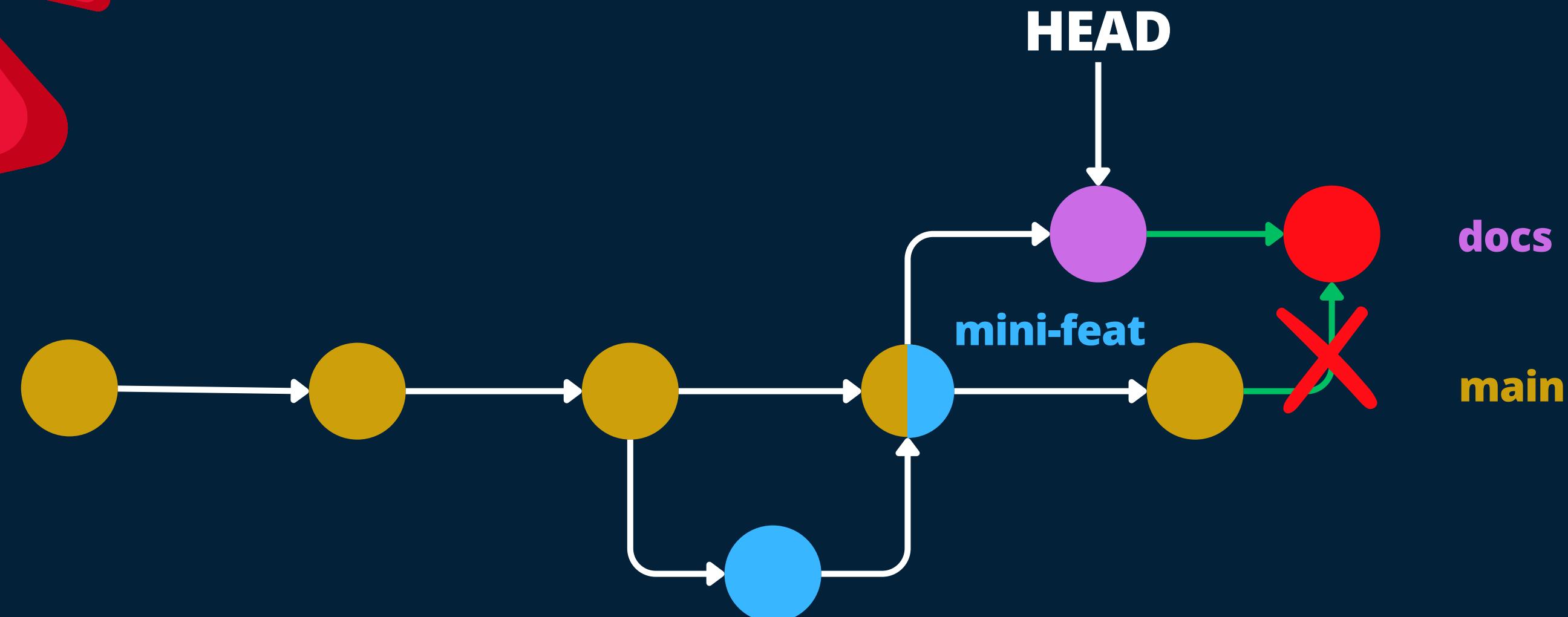
Le modifiche effettuate sui commit cancellati, vengono **raggruppate** in un'unica modifica riportate con stato del file **modified**.

Proviamo a fare reset! Scegliamo prima un **commit**

```
$ git reset <hash_code>
```



ma torniamo alla configurazione che avevamo prima



**Se è possibile evitiamo di mergiare main su branch secondarie.**

**Casi particolari in cui è “accettabile“:**

- main ha molti commit avanti rispetto alla branch che si vuole mergiare
- è una branch remota su cui ci stanno lavorando altri



# .gitignore

**file di git in cui specificare pattern (estensioni) da ignorare**

Crea un file “.gitignore” se non già esistente

```
$ nano .gitignore
```

Nel nostro esempio è consigliabile usare un *.gitignore* di questo tipo:

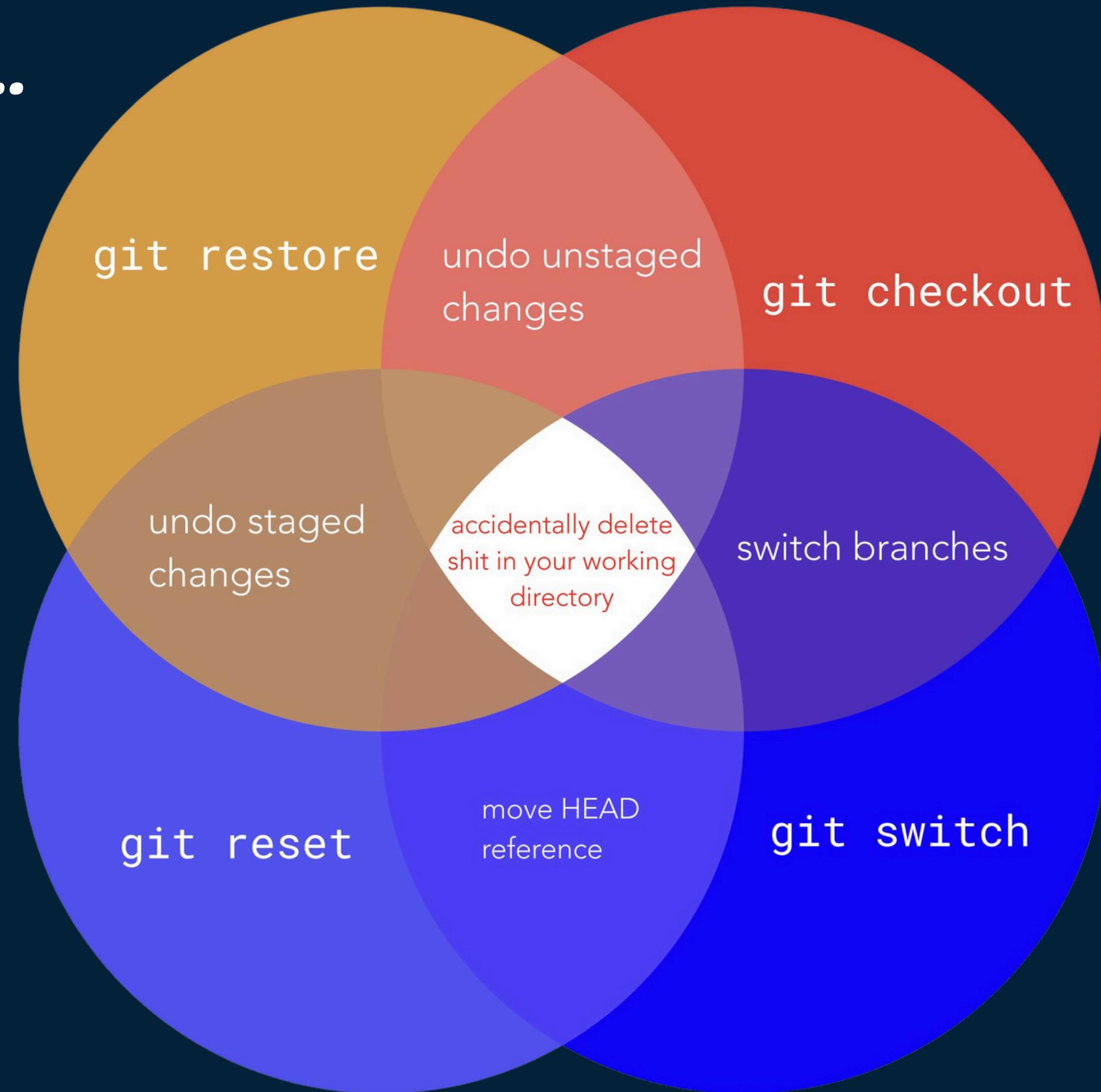
```
# commento  
*.out
```

In questo modo git ignorerà tutti i file con estensione “.out“:

- non li tracerà
- non considererà alcuna modifica/creazione se già esistente il *.gitignore*

Aspettiamo a committare

# Riassumendo...



# Conflitti

Abbiamo visto che si può lavorare su più branch in parallelo.

Al momento del merge/rebase di queste branch si possono presentare conflitti.

Il motivo è che si possono avere più versioni di uno stesso file

- git non sa quale delle due è quella giusta da salvare

È necessario risolverli per completare le operazioni di merge/rebase che si stavano eseguendo.

# Risolvere i conflitti

Spesso risolvere i conflitti è la parte più difficile e delicata di un progetto.

Facciamo un po' di pratica per sapere come approcciare a situazioni peggiori future.

Lo affronteremo in parallelo in due modi:

1. command line
2. vscode

Sempre su branch test

# Modifichiamo

\$ nano test.cpp



```
#include <cstdlib>
#include <iostream>

int main() {
    if (std::system("g++ main.cpp -o adm_program") != 0) {
        std::cout << "ERR: Compilazione fallita\n";
        return 1;
    }
    if (std::system("./adm_program") != 0) {
        std::cout << "ERR: Errore in esecuzione\n";
        return 1;
    }
    std::cout << "Test OK\n";
    return 0;
}
```

... eseguiamolo e committiamo

# Complichiamo un po'...

13

Torniamo su main e allineiamolo alla versione del file test.cpp della branch test

```
$ git switch main
```

Facciamolo con restore

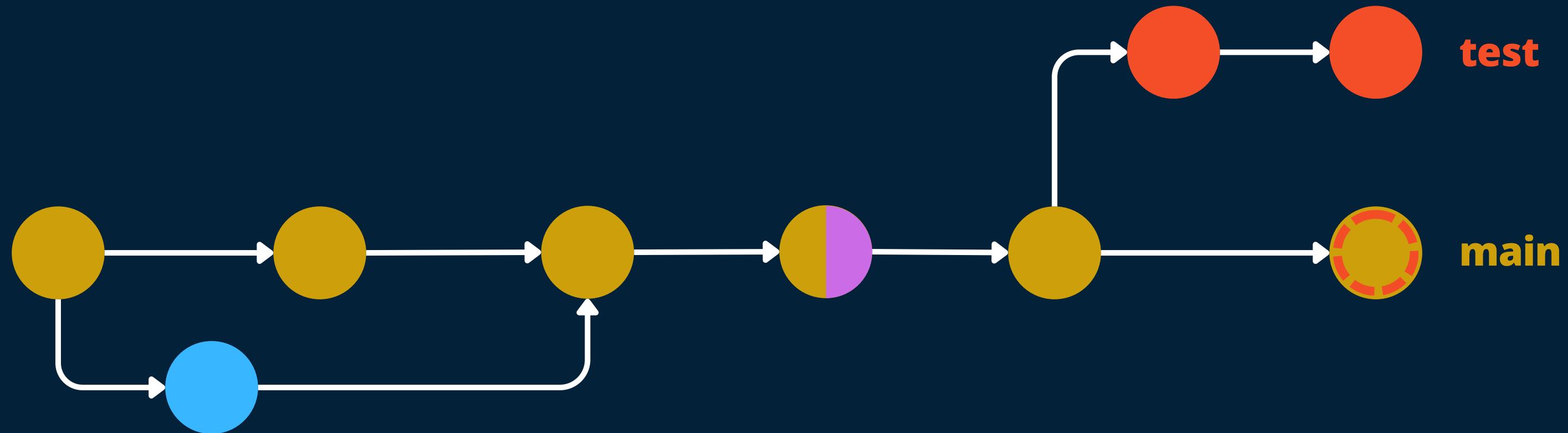
```
$ git restore --source=<hash-commit> test.cpp .gitignore
```

Committiamo

# Com'è messa la repo?

14

Autiamoci con `git log --graph --all` per capire lo stato attuale della repo



# Continuiamo a incasinare

15

Andiamo sul branch test e modifichiamo queste righe nel file **test.cpp**

```
$ git switch test
```

```
#include <cstdlib>
#include <iostream>

int main() {
    if (std::system("g++ mini.cpp -o adm_program.out")
!= 0) {
        std::cout << "ERR: Compilazione fallita\n";
        return 1;
    }
    if (std::system("./adm_program.out") != 0) {
        [...]
```

Tracciamo le modifiche

# Non ne abbiamo più per nessuno

Creiamo una nuova branch **time** a partire da **main** in cui facciamo altre modifiche al file test.cpp

```
$ git switch main
```

```
$ git switch -c time
```

```
$ nano test.cpp
```

```
#include <ctime>
#include <iostream>

int main() {
    std::time_t t = std::time(nullptr);
    std::cout << "Current time: " << std::ctime(&t);
    return 0;
}
```

... eseguiamo e committiamo anche questa versione

Torniamo su main

```
$ git switch main
```

Aggiungiamo queste righe a **test.cpp**

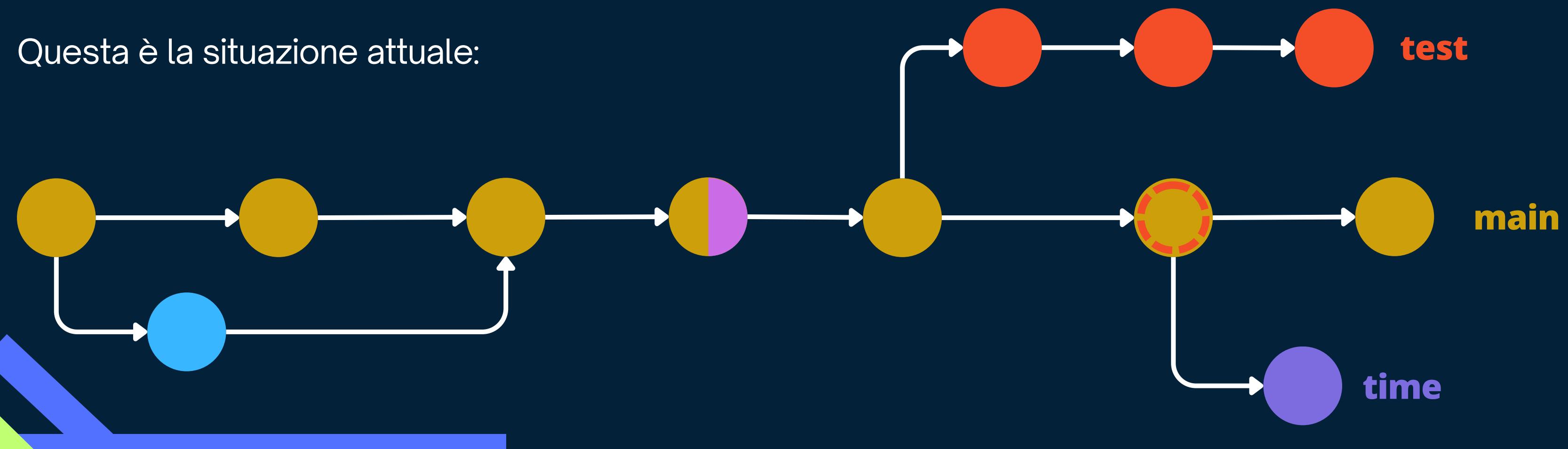
```
[...]  
    std::cout << "Test OK\n";  
  
    int x, y;  
    std::cout << "Inserisci due numeri da sommare: ";  
    std::cin >> x >> y;  
    std::cout << "La somma dei due numeri è: " << (x + y) << "\n";  
  
    return 0;  
}
```

Eseguiamo e committiamo!

# Ora basta, dobbiamo risolvere

Abbiamo simulato una piccola situazione reale, in cui ci sono più versioni dello stesso file in svariate branch. Cerchiamo di fare una versione finale che unisca le varie differenze.

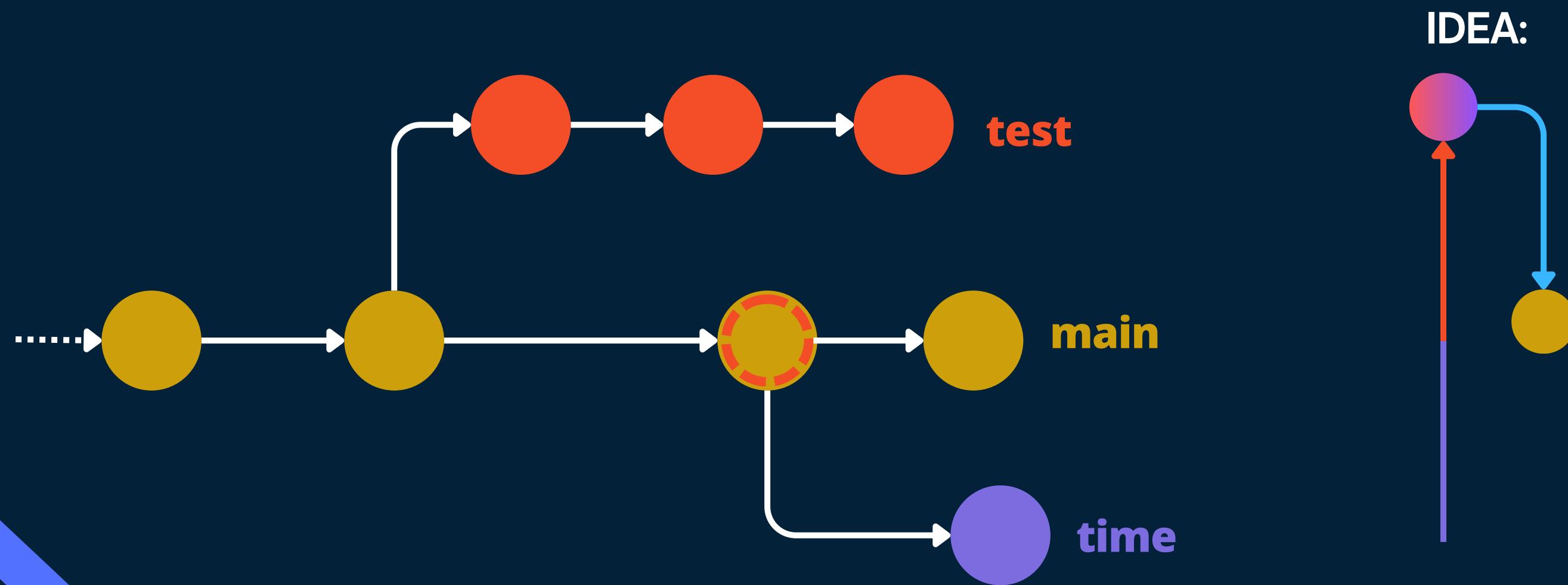
Questa è la situazione attuale:



# Come risolviamo?

20

Abbiamo simulato una piccola situazione reale, in cui ci sono più versioni dello stesso file in svariate branch. Cerchiamo di fare una versione finale che unisca le varie differenze.



# MERGIAMOOOO

21

Andiamo sul branch **test** e mergiamo il branch **time**

```
$ git switch test
```

```
$ git merge time
```



**ABBIAMO DEI CONFLITTI!!!!**

# Conflitti command line in generale

1. per vedere i file che sono in conflitto usiamo *git status*

Dovremmo vedere una cosa del genere:

```
$ git status  
[...]  
Unmerged path:  
    both added: <nome_file_conflitto>
```

2. modifichiamo i file in conflitto con un editor di testo a scelta, git avrà aggiunto delle righe dove si trovano i conflitti in cui specifica quali parti vengono da quale versione:

```
<<<<<< HEAD  
Modifiche correnti  
=====  
Modifiche “incoming”  
>>>>> nome_branch
```

# Conflitti command line in generale

A questo punto resta a noi scegliere quali righe tenere/modificare/eliminare.

Il file dovrà risultare senza le linee aggiunte da git :

```
<<<<< HEAD
```

```
Modifiche correnti
```

```
=====
```

```
Modifiche “incoming”
```

```
>>>>> nome_branch
```

*Conflitti presenti*

*Conflitti risolti*

```
Modifiche “incoming”
```

3. aggiungiamo alla staging area il file
4. in base all'azione che stavamo facendo eseguiamo:

Nel nostro caso usiamo: `$ git merge --continue`

`$ git rebase --continue`

# Conflitti vscode in generale

Nell'editor dei conflitti visualizziamo il file nelle due versioni presenti che generano il conflitto

The screenshot shows the VS Code Merge Editor interface. At the top, it says "Merging: file.md ! X". Below that, there are two panes: "Incoming" (branch 9b40130) and "Current" (branch b46f770). In the Incoming pane, the first line "Prova questo file carino" is highlighted in green, and the third line "Si vero molto bello" is highlighted in yellow. In the Current pane, the first line is also green, and the second line "Non so, mi sembra poco utile" is highlighted in yellow. A yellow box highlights the conflict area between the two lines. Below the panes, the "Result" section shows the merged file: "Prova questo file carino" (green) and "No Changes Accepted" (yellow). A white arrow points from the bottom right towards a "Complete Merge" button.

Merging: file.md ! X

file.md

Incoming φ 9b40130 • nuova\_branch

1 Prova questo file carino

Accept Incoming | Accept Combination (Incoming First) | Ignore

2

3 Si vero molto bello

Current φ b46f770 • main

1 Prova questo file carino

Accept Current | Accept Combination (Current First) | Ignore

2 Non so, mi sembra poco utile

Result file.md

1 Prova questo file carino You, 1 second ago • Uncommitted changes

No Changes Accepted

1 Conflict Remaining

Complete Merge

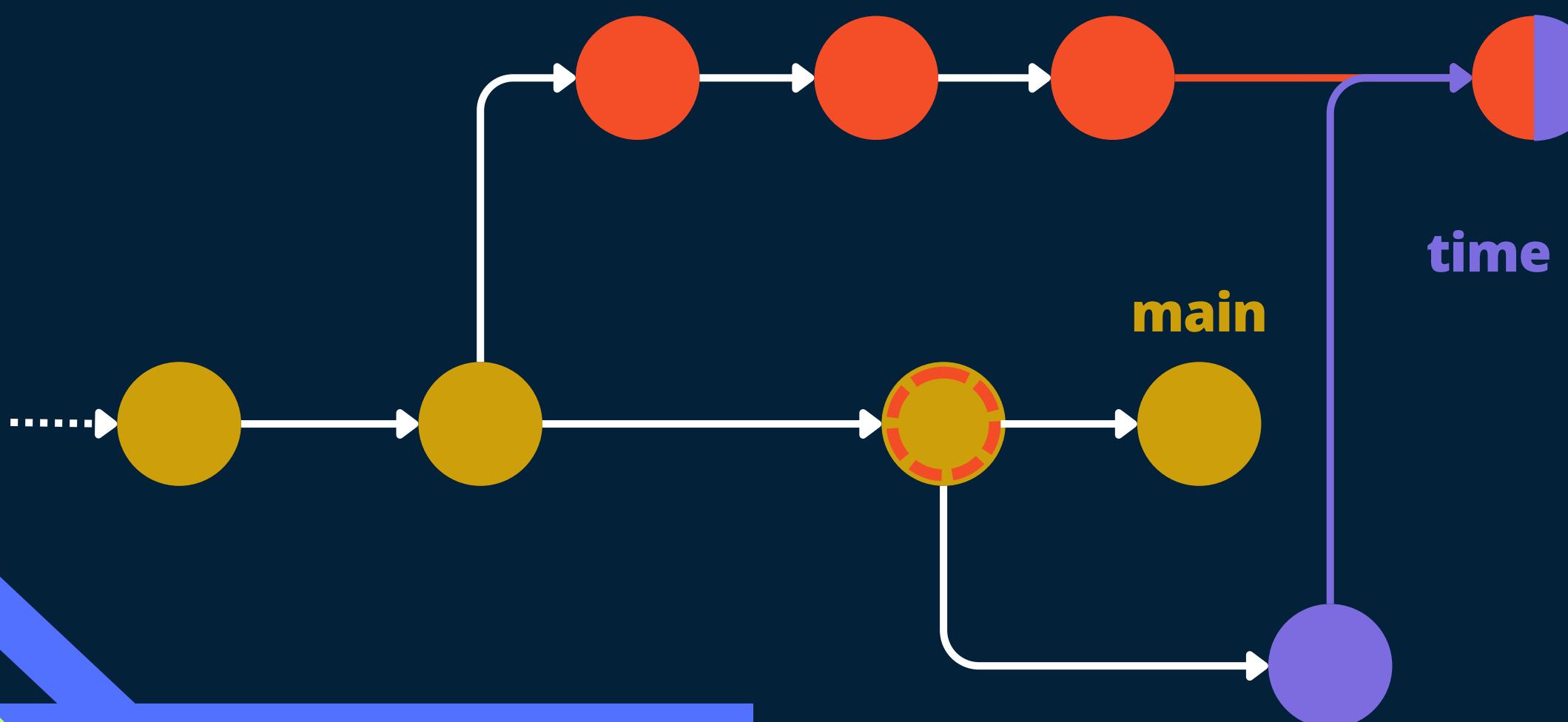
Versione finale che si salverà in  
base alla risoluzione del conflitto

# Primo passo verso la fine

22

Dopo aver finito il merge siamo in questa situazione

Resta soltanto l'ultimo merge, come lo affrontiamo?



2 OPZIONI (entrambe con conflitti):

- test**
- rebase e merge
  - merge diretto

# Proviamo con il rebase

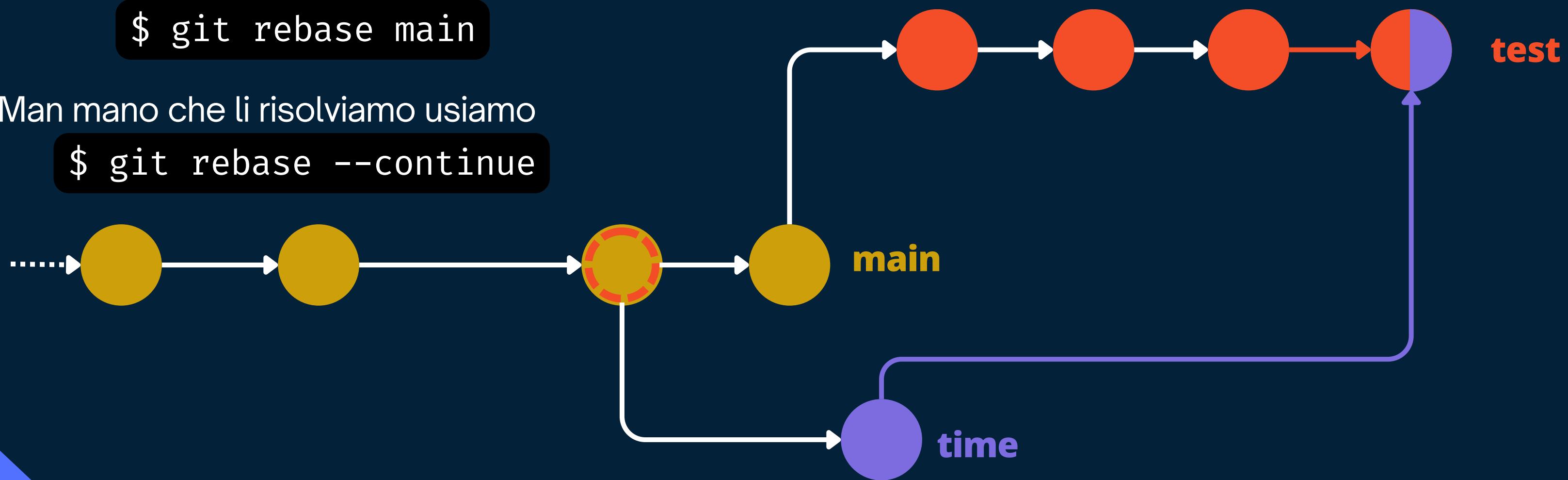
23

In casi come il nostro, facciamo rebase su un commit con modifiche che includono il nostro codice, dovremo quindi risolvere i conflitti per i commit della nostra branch, così abbiamo proprio un **cambio di base** dei commit.

```
$ git rebase main
```

Man mano che li risolviamo usiamo

```
$ git rebase --continue
```



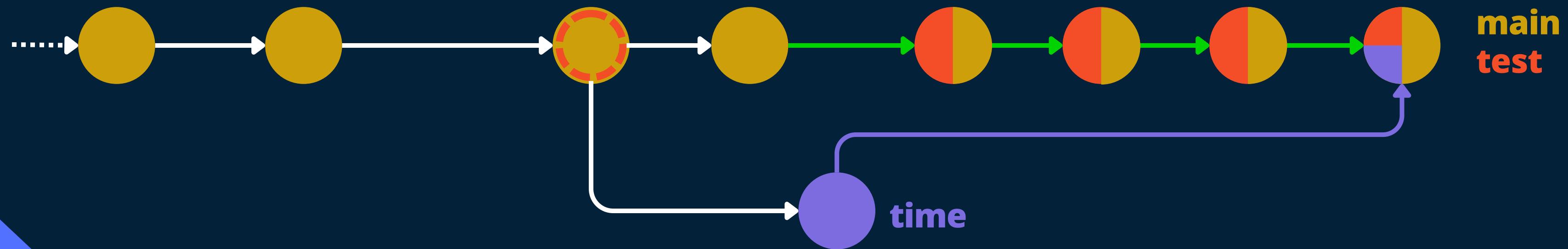
Se arriviamo a questo punto ci mancherà solo mettere test su main

# Ce l'abbiamo fatta!!!

24

Ora abbiamo un progetto completamente mergiato su main con tutte le modifiche che volevamo!

Abbiamo affrontato varie casistiche di operazioni che capitano quotidianamente nello sviluppo di progetti in collaborazione come vedremo nel prossimo incontro!



# Flag speciali pt2

Scopri tu cosa fanno di magico ✨

```
$ git mergetool
```

```
$ git checkout -b
```

```
$ git rebase --exec "<cmd>"
```

```
$ git merge --squash
```

```
$ git reset --keep
```

```
$ git bisect
```

# Flag speciali

dello scorso incontro

- \$ git add -p [<path>] aggiunge alla staging area solo alcune parti di un file specificato
- \$ git add -u aggiunge alla staging area SOLO i file già tracciati
- \$ git commit -am <msg\_commit> committa aggiungendo tutti i tracked files
- \$ git commit --ammend modifica l'ultimo commit, permettendoti di aggiungere o togliere file e/o cambiare il messaggio senza crearne uno nuovo
- \$ git log --grep='<string>' mostra solo i commit il cui messaggio contiene <string>
- \$ git stash -u include anche i file non tracciati

>ADM  
staff

# GITHUB 1

ISSUE  
PULL REQUEST  
CONFLITTI

# GITHUB 2

CI/CD  
ACTIONS  
GIT LFS

LABORATORI FRA PARI  
AA 2025-2026

27/11 16:00-18:00

AULA E2  
via mura Anteo Zamboni 2B

04/12 16:00-18:00

AULA E2  
via mura Anteo Zamboni 2B



PORTA LA TUA  
CURIOSITA'





## **Com'è andata?**

Da questo semestre abbiamo aggiornato i laboratori, cercando di puntare di più sulla pratica e affrontare “i classici problemi” che si incontrano nell’uso quotidiano.

**Lasciaci un feedback! Per noi è importante**