

Git & GitHub



■ ¿Qué es Git y para qué sirve?

- Un sistema de control de versiones
- Un software de apoyo para desarrollar
- Control sobre la evolución de un proyecto
- Control sobre el desarrollo colaborativo
- Desarrollo en paralelo de funcionalidades
- Estructuración y mantenimiento de versiones



■ ¿Cómo se instala?

- En Windows, usando [el instalador](#)
- En [macOs](#), usando [homebrew](#), [MacPorts](#) o instalando [XCode](#)
- En Linux, usando el [instalador de paquetes](#) de tu distribución



■ Configurando Git

git config

```
# Mostrar el valor del parámetro de configuración <param>
git config <param>

# Establece el valor <value> en el parámetro de configuración <param>
git config <param> <value>

# Establece el valor para cualquier proyecto Git en este ordenador
git config --global <param> <value>

# Ejemplos:

# Establecer nuestro nombre
git config --global user.name "Homer J. Simpson"

# Establecer nuestro e-mail (no tiene por qué ser el de GitHub)
git config --global user.email "Homer J. Simpson"
```



■ Nuestro primer repo

git init

```
# Creamos una carpeta para nuestro proyecto
mkdir star-wars

# Accedemos a la carpeta del proyecto
cd star-wars

# Inicializamos nuestro repo
git init --initial-branch=main
```

Un repo de git es una carpeta de nuestro ordenador en la que hemos ejecutado el comando `git init`. Este comando crea una carpeta oculta llamada `.git` donde git almacena toda su info.





Las tres zonas

Working Copy

- Es la carpeta de nuestro proyecto.
- Podemos ver lo que hay en su interior con el explorador de archivos.
- También se le conoce como *Working Tree* o *Unstaged Files*.

Staging Area

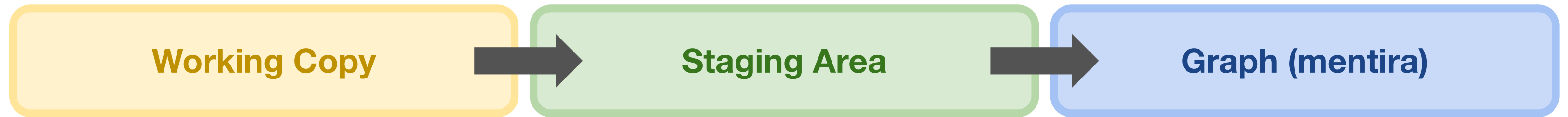
- Es “el lugar” donde pondremos los cambios que queremos guardar una versión.
- Sólo podemos ver lo que hay en su interior usando algún comando de *git*.
- También se conoce como *index* o *cache*.

Graph (mentira)

- Es un “lugar virtual” donde se almacenan los cambios.
- Sólo podemos ver lo que hay en su interior usando algún comando de *git*.
- Su nombre real es *repositorio*, pero este nombre nos puede confundir.



■ Flujo de trabajo habitual



Escribimos código (hacemos cambios) usando nuestra herramienta favorita.

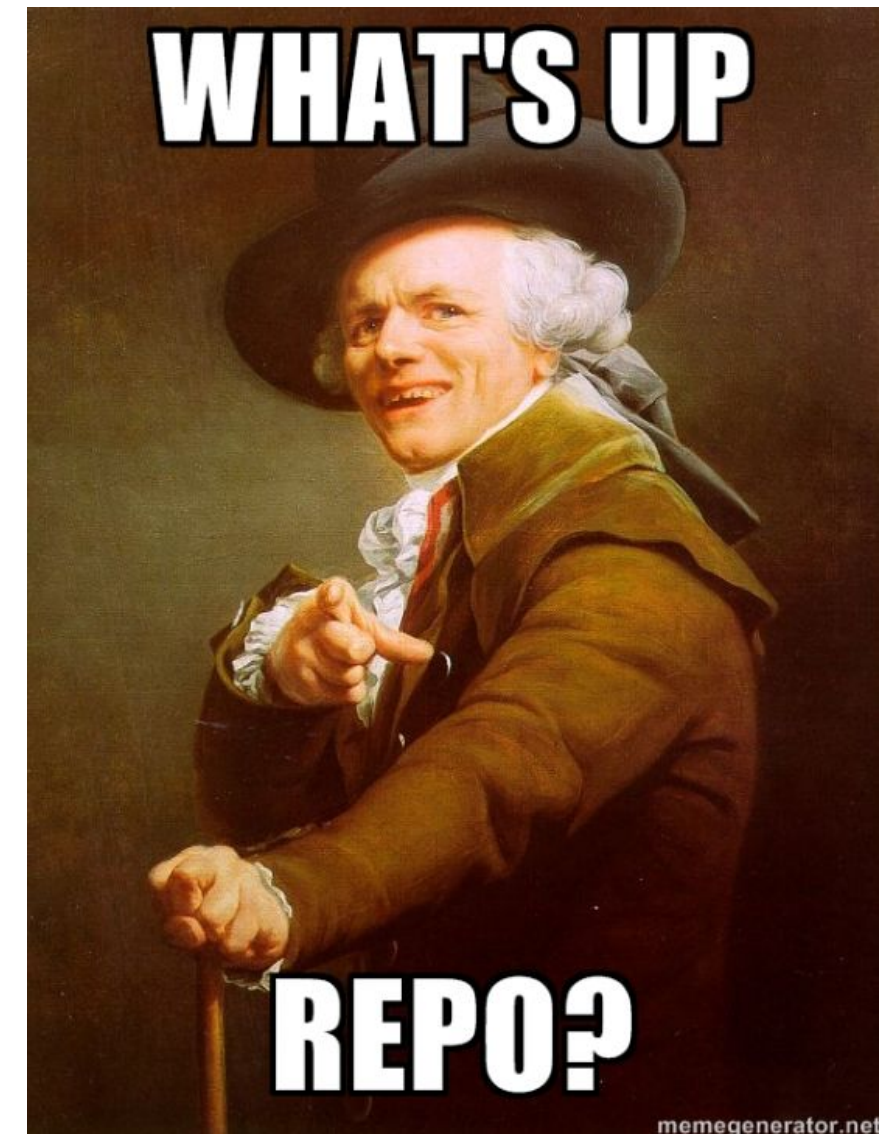
De todo el código que hemos escrito, seleccionamos los archivos cambiados de los que queremos guardar una versión.

Guardamos los cambios que se han puesto en Staging Area

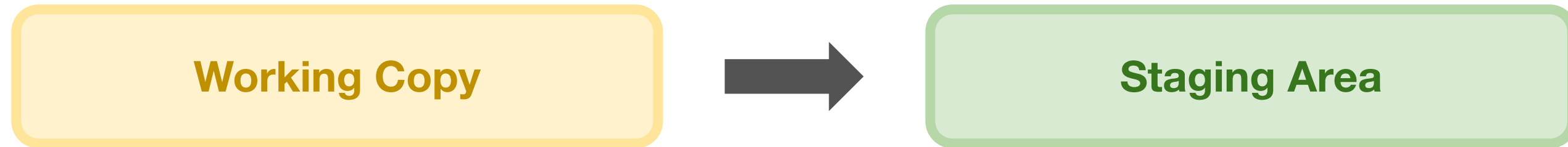


■ git status

- El comando `git status` nos resume el estado de nuestro.
- Nos dice qué archivos no tienen ninguna versión guardada.
- Qué archivos han cambiado desde su última versión guardada.
- Qué archivos están en el staging área y cuáles no.



■ git add: poniendo cambios en el **Staging Area**



git add

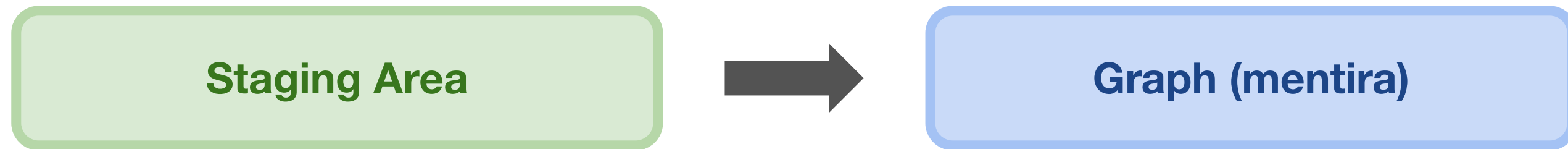
```
# Añade al Staging Area los cambios del archivo <file>
git add <file>

# Añade al Staging Area todos los archivos con cambios dentro de la carpeta <folder>
git add <folder>

# Añade al Staging Area todos los archivos con cambios que terminen por .md
git add *.md
```



■ git commit: del **Staging Area** al **Graph**



git commit

```
# Hace un commit con los cambios que hay en Staging Area
# Antes, abrirá un editor de texto para que podamos introducir el mensaje
git commit

# Hace un commit con los cambios que hay en Staging Area con el mensaje indicado
git commit -m "Mensaje del commit"
```

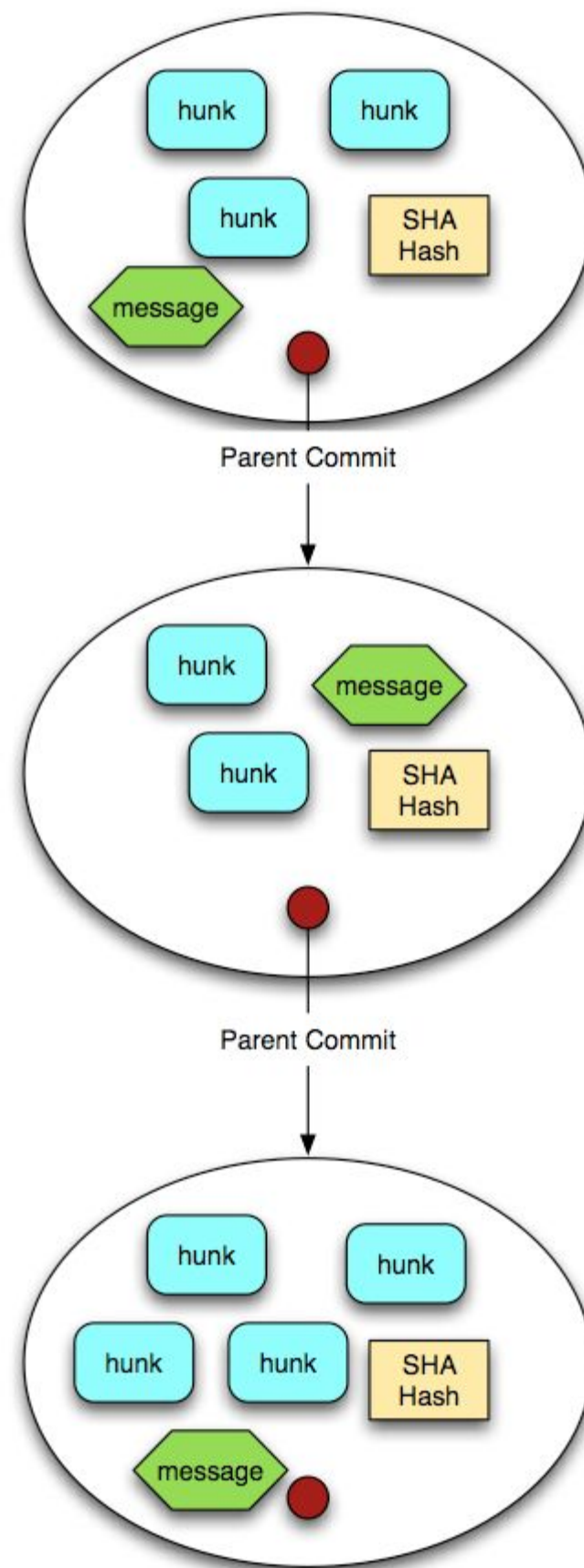


■ ¿Qué es un commit?

Un **commit** es un **paquete** que contiene:

- Uno o más hunks (cambios en archivos)
- Un mensaje que describe qué cambios
- Un hash SHA para identificar el commit (un identificador único)
- La fecha y hora de creación
- El nombre y e-mail del autor
- **Un enlace con su "commit padre"**





■ ¿Qué pasa en **Graph** cuando hago un commit?







Git Simulator v.1.3.73 🧐



```
git add *.js
```





Git Simulator v.1.3.73



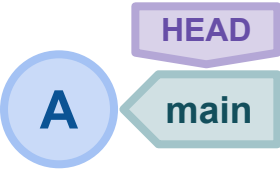
Git Simulator v.1.3.73 🧐

```
git commit -m "Mi primer commit"
```



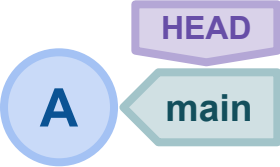


Git Simulator v.1.3.73



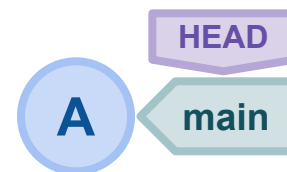


Git Simulator v.1.3.73



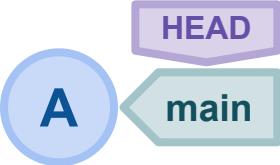
Git Simulator v.1.3.73 🧐

```
git add *.css
```







Git Simulator v.1.3.73



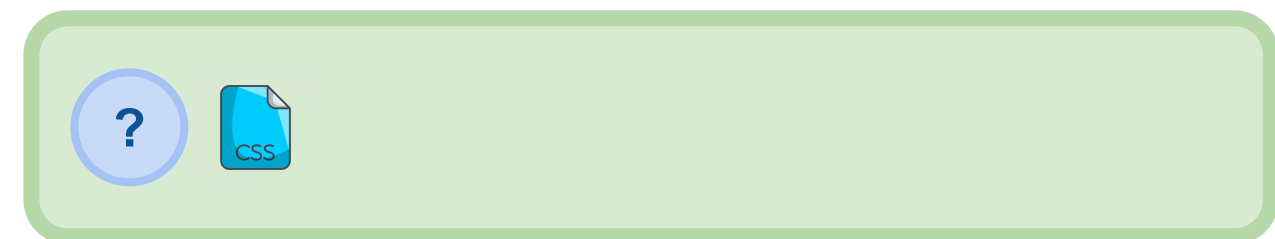
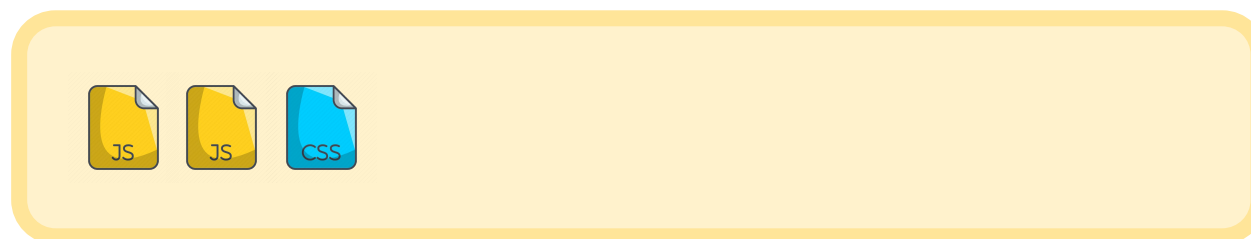
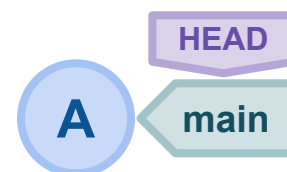






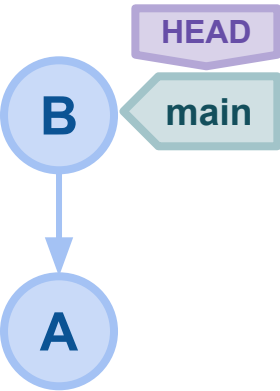
Git Simulator v.1.3.73 🧐

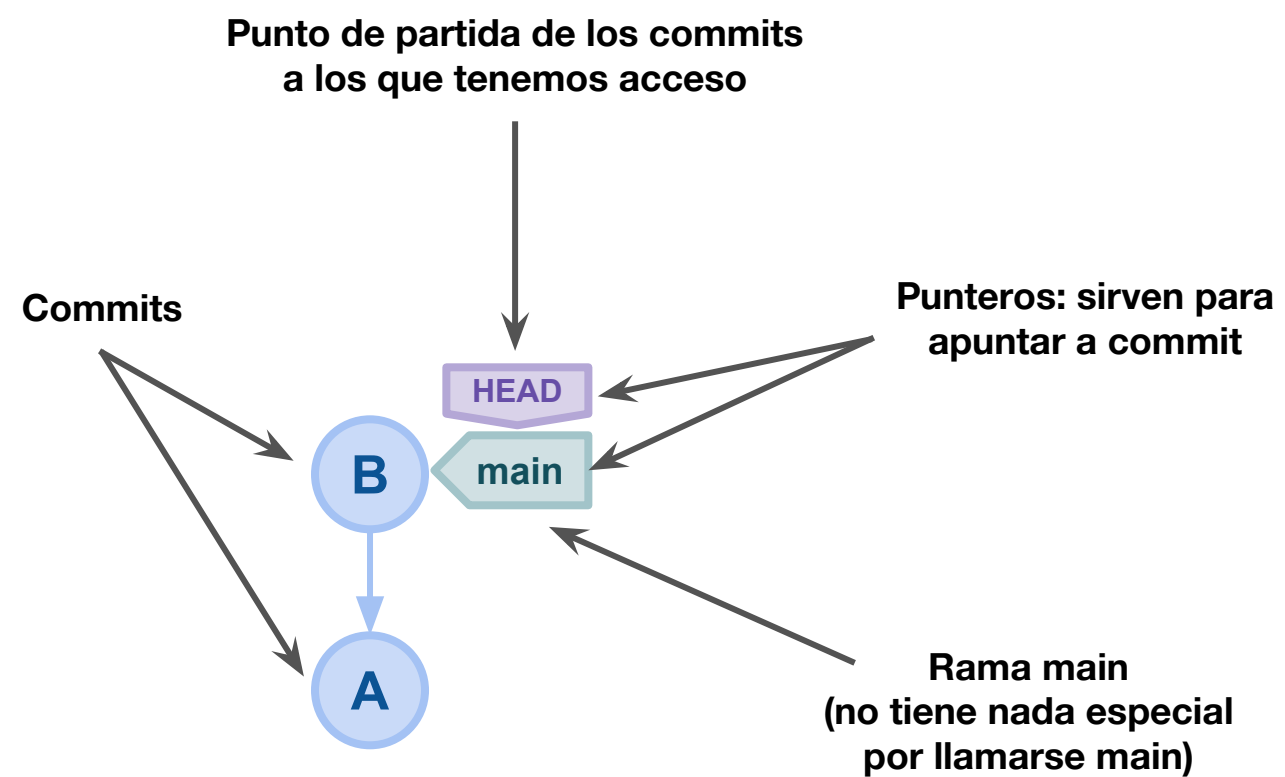
```
git commit -m "Mi segundo commit"
```





Git Simulator v.1.3.73





■ ¿Cómo podemos ver eso desde la consola?

`git log`

```
# Muestra la info los commits a los que tenemos acceso desde HEAD  
git log
```

```
# Muestra la info de los commits pintando un grafo  
git log --graph
```



■ ¿Cómo podemos el contenido de un commit?

git show

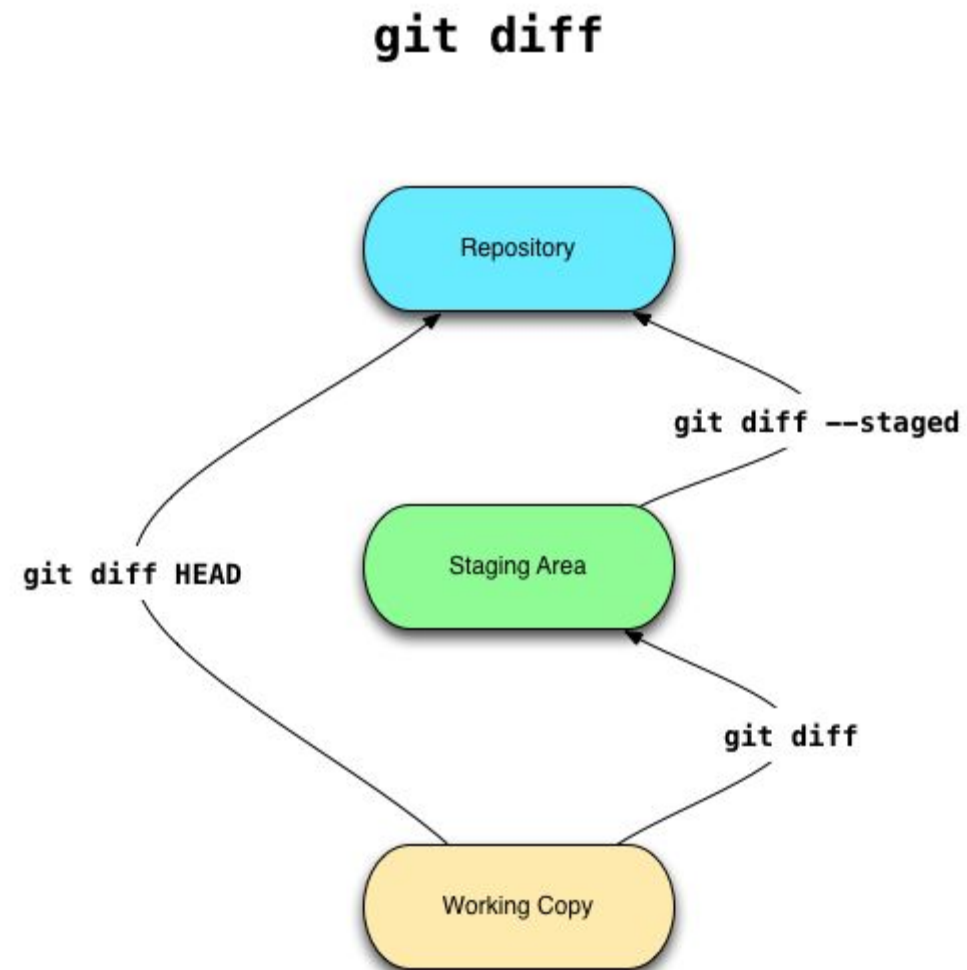
```
# Muestra el contenido del commit que indicamos
git show <commit id>

# Muestra el contenido del commit al que apunta HEAD
git show HEAD

# Muestra el contenido del commit al que apunta la rama main
git show main
```



■ ¿Cómo podemos comparar las tres zonas?



Donde dije digo, digo Diego

Deshaciendo lo que hemos hecho



■ Deshaciendo cosas

Antes de deshacer nada...hay que pensar



¿Qué es lo que quieres deshacer?



■ ¿Qué queremos deshacer?

- “He picado código durante 2 horas y he tocado tantas cosas que quiero descartar todos los cambios”
- “Tras picar código durante 5 horas, he hecho el commit pero me he equivocado en el mensaje (insultaba al jefe).”
- “He picado código durante 3 horas, he hecho el commit pero sigue sin funcionar nada. Quiero volver a la versión del código anterior”.



■ ¿Qué queremos deshacer?

¿Qué queremos deshacer?	Comando	Zonas afectadas
Los cambios realizados	<code>git restore <file></code>	Working Copy
El último commit	<code>git reset HEAD~1</code>	Graph (mentira)
El último commit + y los cambios realizados		Working Copy Graph (mentira)



■ ¿Qué queremos deshacer?

¿Qué queremos deshacer?	Comando	Zonas afectadas
Los cambios realizados	<code>git restore <file></code>	Working Copy
El último commit	<code>git reset HEAD~1</code>	Graph (mentira)
El último commit + y los cambios realizados	<code>git reset HEAD~1</code> + <code>git restore <file></code>	Working Copy Graph (mentira)



■ git restore

```
# Descarta los cambios realizados en el archivo <file>
git restore <file>

# Descarta los cambios realizados en archivos de la carpeta <file>
git restore <file>

# Descarta los cambios realizados en el archivos con extensión .md
git restore *.md
```

- Los cambios que hemos realizado en los archivos que le digamos.
- Sólo afecta a los archivos en **Working Copy**
- Realmente, lo que hace es volver a cargar la versión anterior del archivo
 - Por eso no podemos descartar archivos nuevos (que nunca han sido versionados)



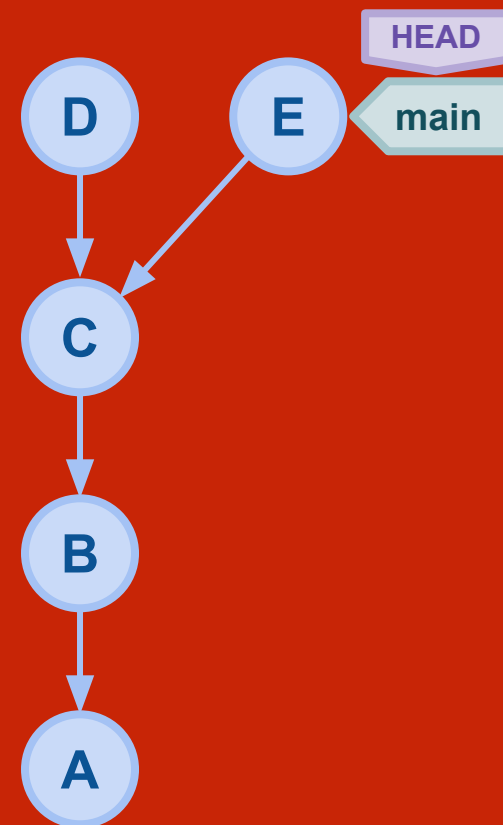
■ git reset

¿Qué pasa en el **Graph** cuando
ejecutamos `git reset HEAD~1`?



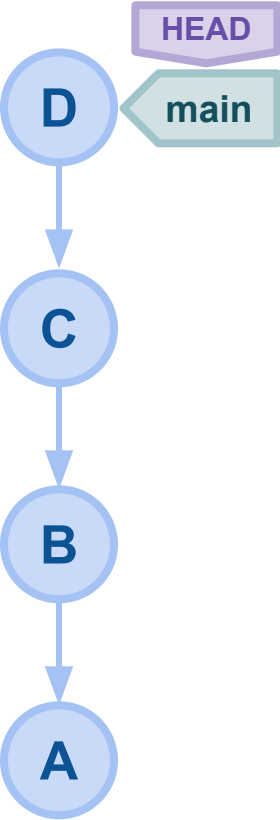
Git Simulator v.1.3.73 🤩

```
git commit -m "Lorem ipsum"
```





Git Simulator v.1.3.73 🤩

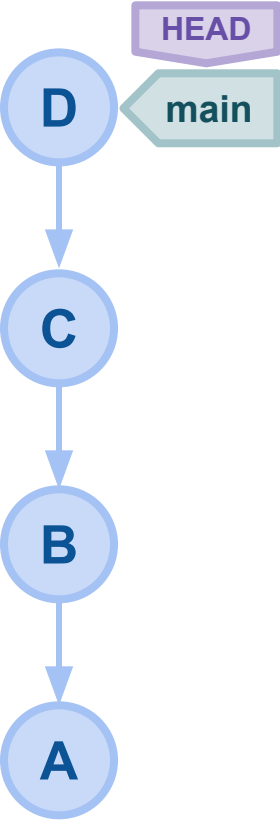




Git Simulator v.1.3.73 🤩

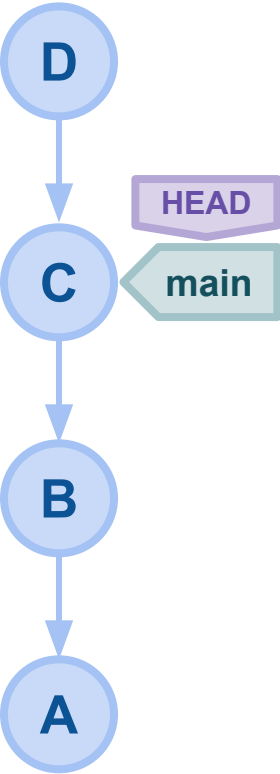


```
git reset HEAD~1
```

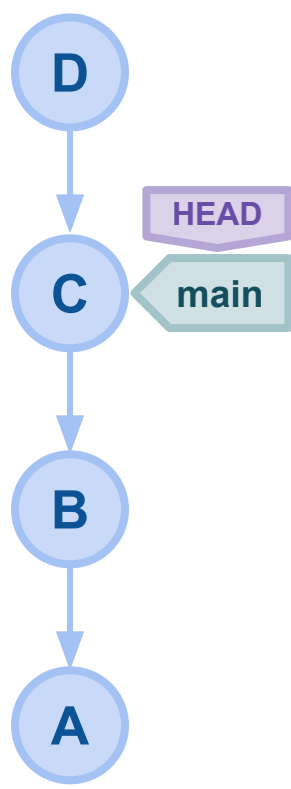




Git Simulator v.1.3.73 🤩

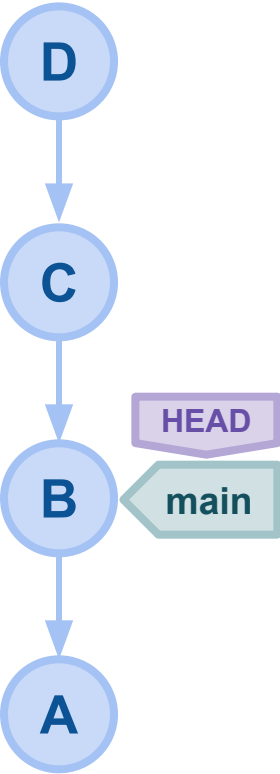


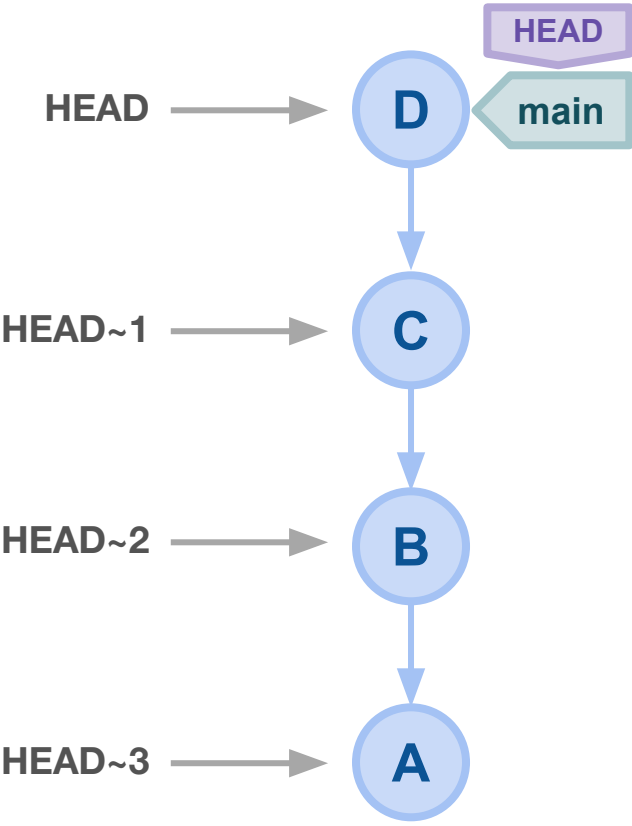
```
git reset HEAD~1
```





Git Simulator v.1.3.73 🤩





JS

JS

CSS



■ git reset

Ok, entonces
¿para qué sirve el comando `git reset`?

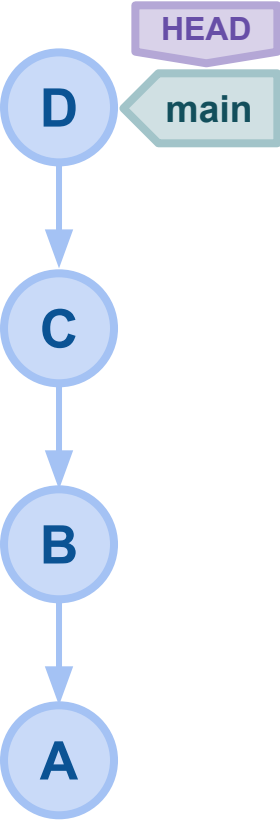


■ git reset

¡MENTIRA!

~~Para deshacer un commit~~





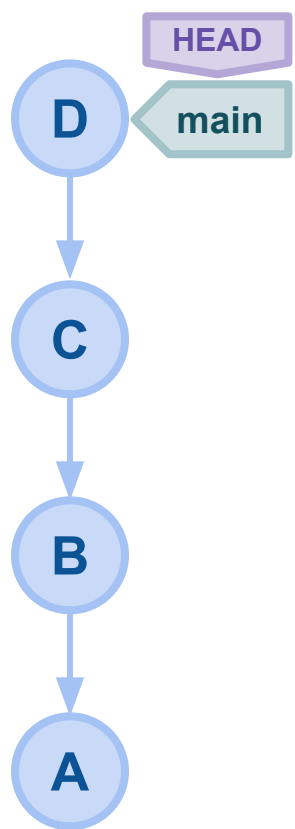
JS

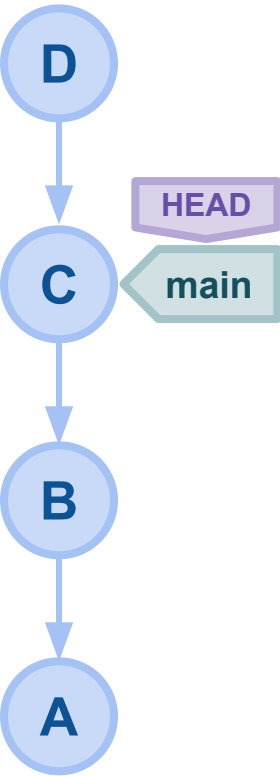
JS

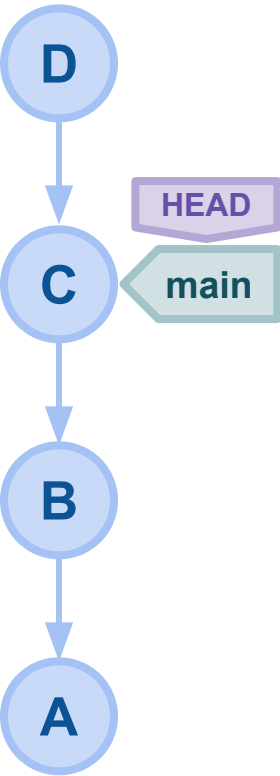
CSS



```
git reset HEAD~1
```





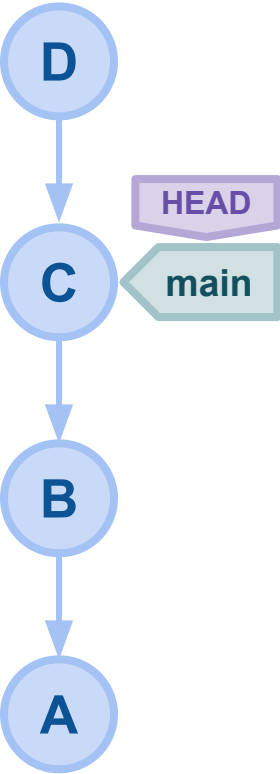


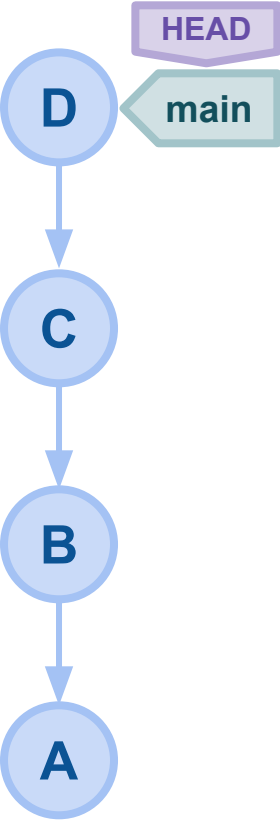


Git Simulator v.1.3.73 🧐



```
git reset D
```



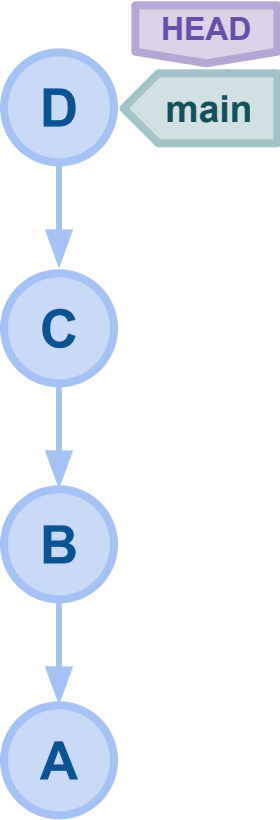




Git Simulator v.1.3.73 🧐

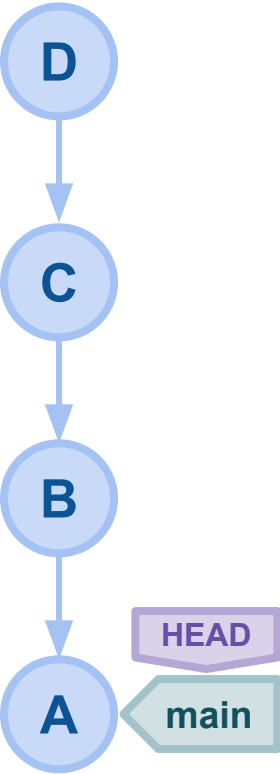


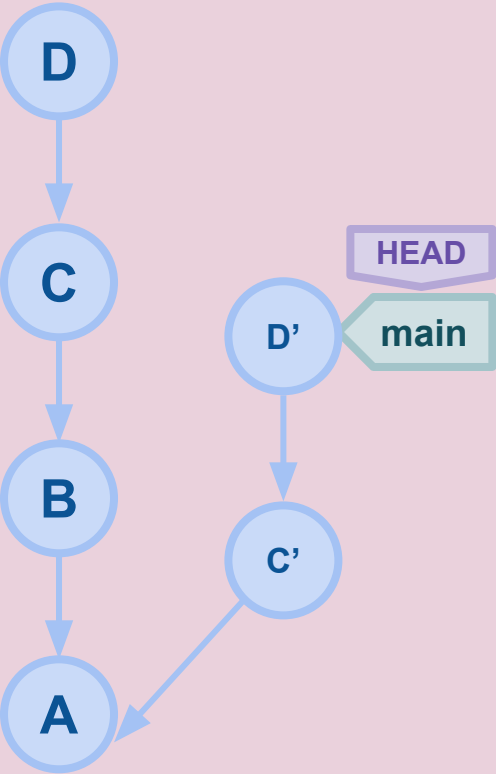
```
git reset A
```





Git Simulator v.1.3.73 🧐





■ git reset

Sirve para mover una rama
allá donde nosotros queramos.

git reset mueve ramas



■ ¿Qué queremos deshacer?

¿Qué queremos deshacer?	Comando	Zonas afectadas
Los cambios realizados	<code>git restore <file></code>	Working Copy
El último commit	<code>git reset HEAD~1</code>	Graph (mentira)
El último commit + y los cambios realizados	<code>git reset HEAD~1</code> + <code>git restore <file></code>	Working Copy Graph (mentira)



■ ¿Qué queremos deshacer?

¿Qué queremos deshacer?	Comando	Zonas afectadas
Los cambios realizados	<code>git restore <file></code>	Working Copy
El último commit	<code>git reset HEAD~1</code>	Graph (mentira)
El último commit + y los cambios realizados	<code>git reset --hard HEAD~1</code>	Working Copy Graph (mentira)



■ ¿Y cómo deshago lo que pongo en **Staging Area**?



```
• > git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   readme.md
```

`git restore --staged <file>`



■ ¿Qué pasa si borramos un archivo?

- Para Git, crear un archivo de 10 líneas es: un cambio en un archivo que pasa de tener 0 líneas a tener 10
- Borrar un archivo de 500 líneas es: ??????????????



■ ¿Qué pasa si borramos un archivo?

- Para Git, crear un archivo de 10 líneas es: un cambio en un archivo que pasa de tener 0 líneas a tener 10
- Borrar un archivo de 500 líneas es: un cambio en un archivo que pasa de tener 500 líneas a tener 0



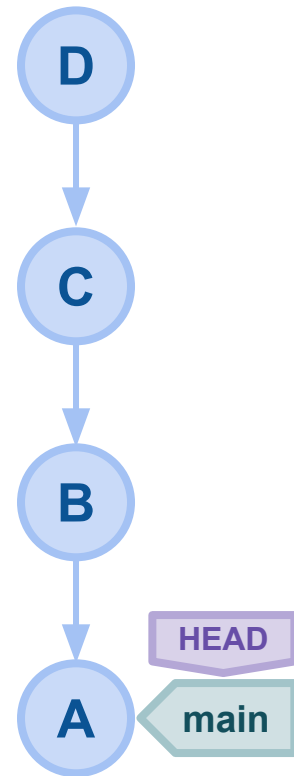
■ ¿Qué pasa si borramos un archivo?

- Para Git, crear un archivo de 10 líneas es: un cambio en un archivo que pasa de tener 0 líneas a tener 10
- Borrar un archivo de 500 líneas es: un cambio en un archivo que pasa de tener 500 líneas a tener 0
- A Git sólo le importan los cambios dentro de los archivos, no los archivos en sí



Git Simulator v.1.3.73 🧐

¿Y ahora cómo volvemos a D?







git reflog al rescate



■ Cuando las cosas se ponen feas...

`git reflog`

- Es un log que registra todos los commits por los que pasa HEAD
- Además, nos dice con qué comando se llegó a ese commit
- Es como un rastro de migas

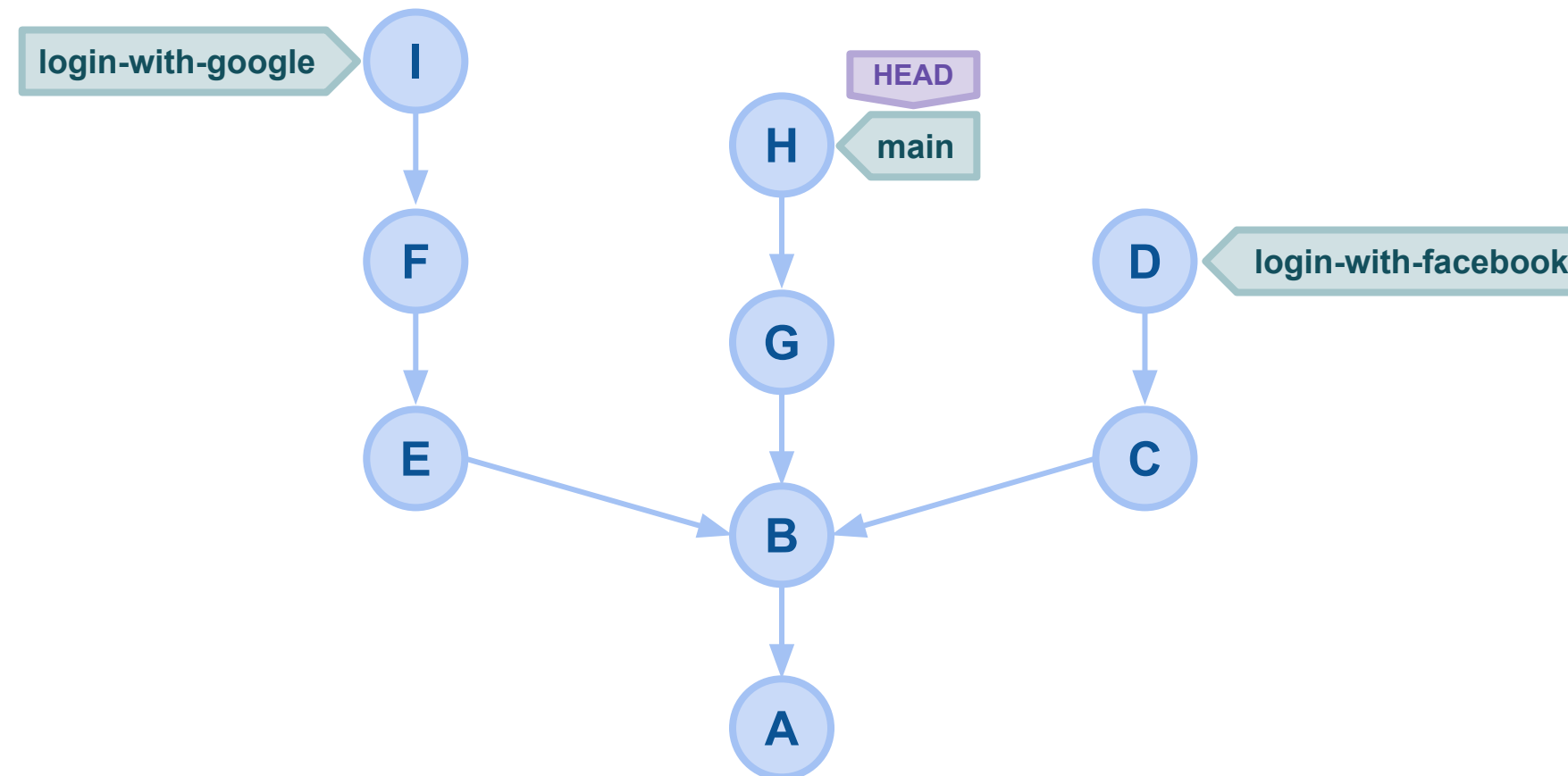


Ramas

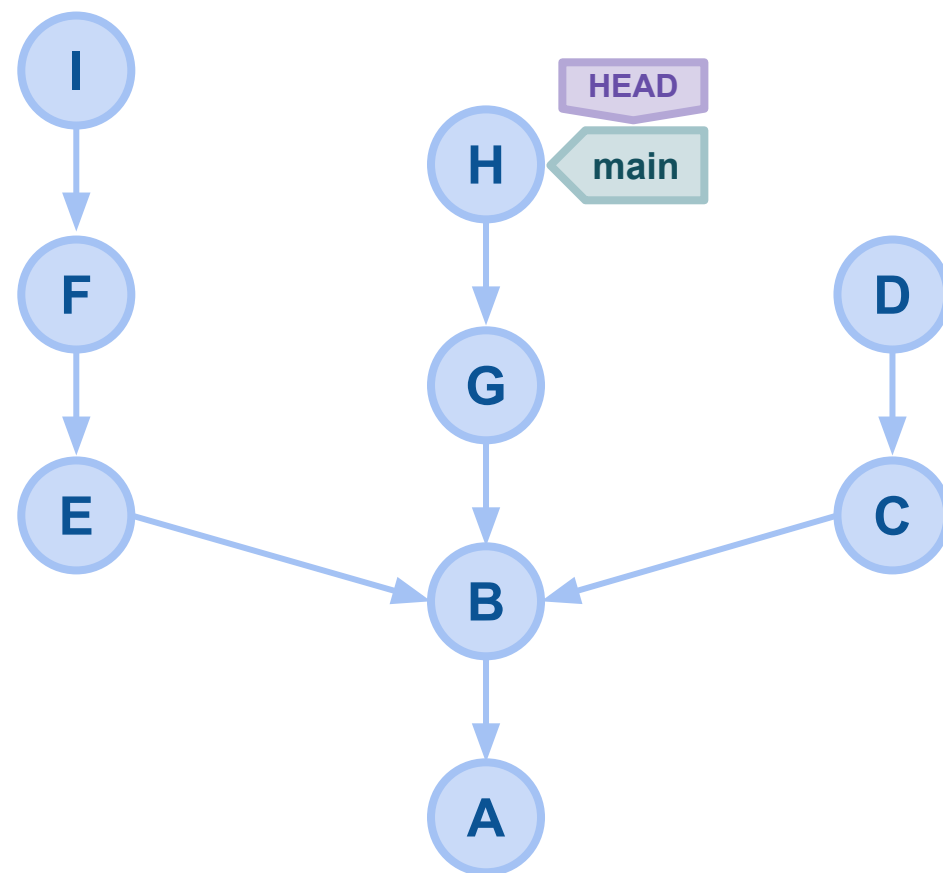
Branches para los amigos



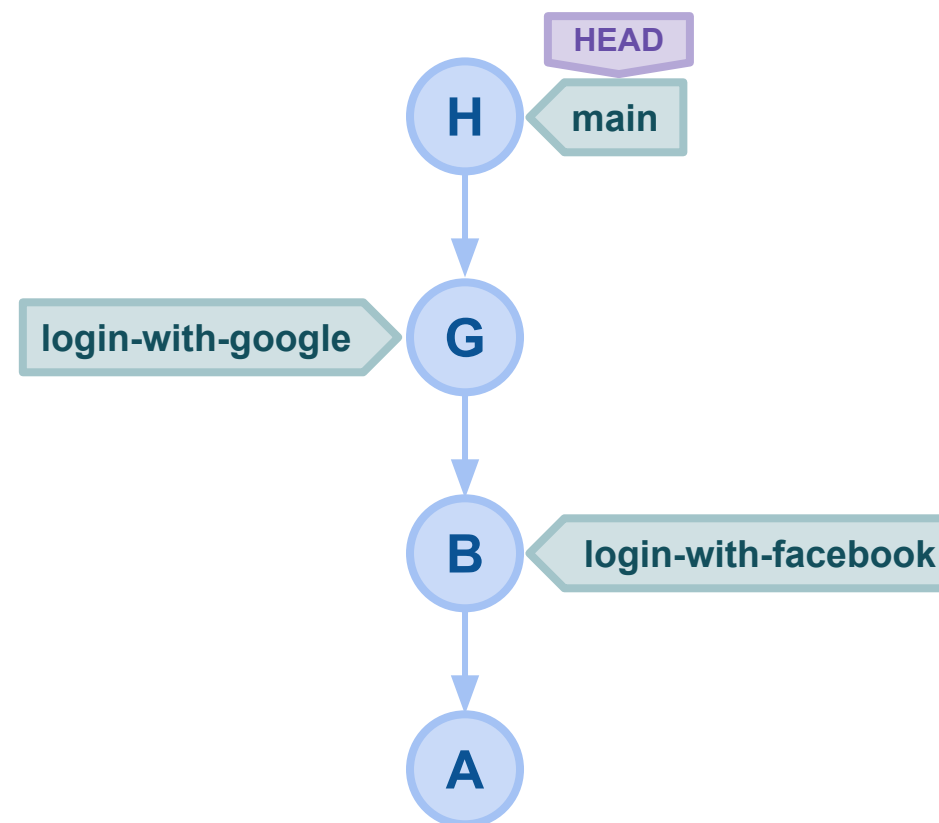
¿Cuántas ramas tiene este repo?



¿Y éste?



¿Y éste?



■ Ver las ramas existentes

`git branch`

- Muestra las ramas existentes en el repo
- Marca con * la rama en está HEAD



■ Crear una rama

git branch

```
# Crea la rama <branch name> en el commit al que apunta HEAD
git branch <branch name>

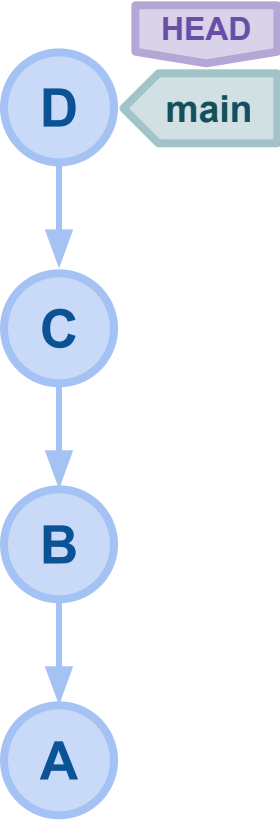
# Crea la rama <branch name> en el commit <commit id>
git branch <branch name> <commit id>

# Crea la rama <branch name> en el commit al que apunta la rama main
git branch <branch name> main
```

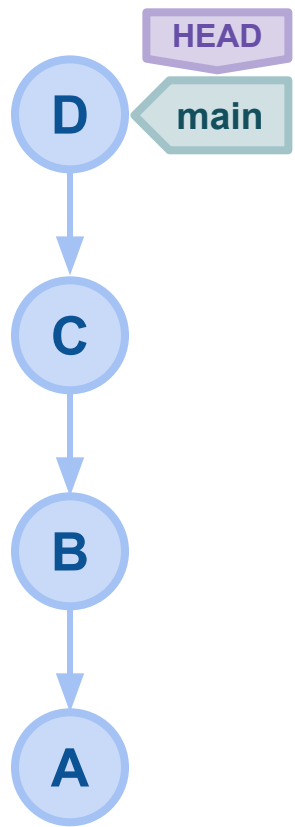




Git Simulator v.1.3.73 🥰

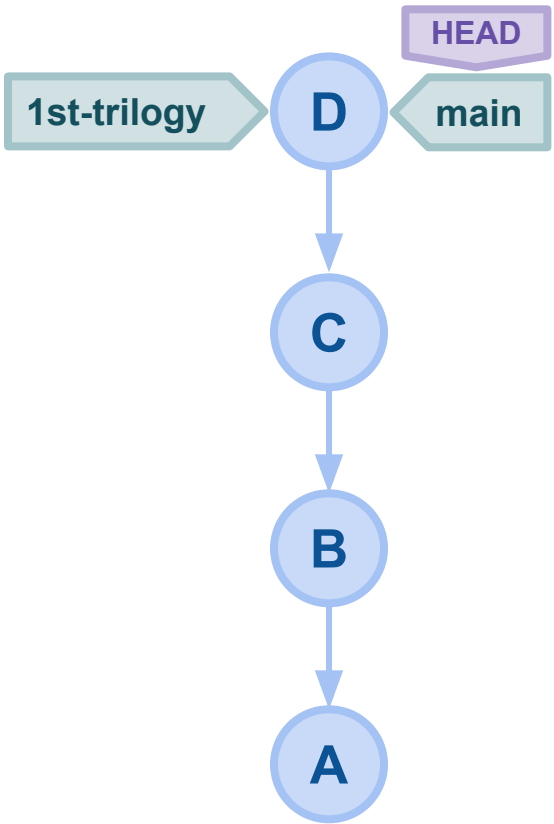


```
git branch 1st-trilogy
```

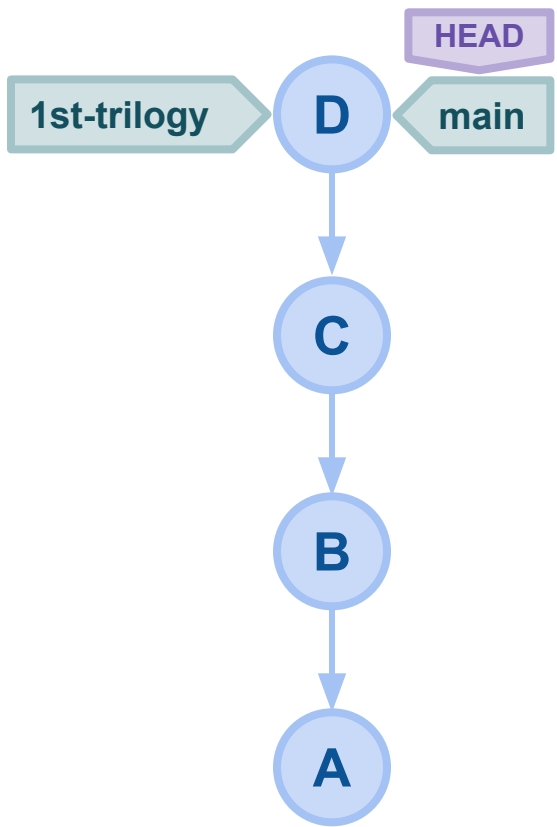


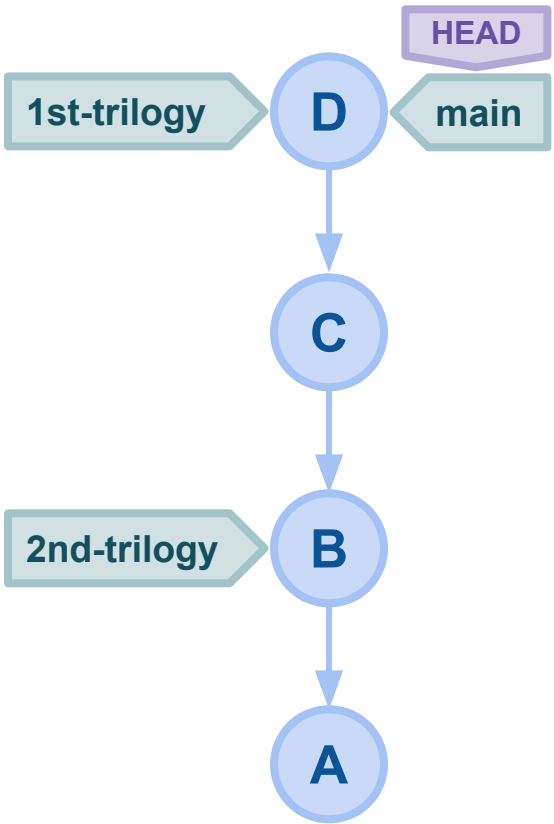


Git Simulator v.1.3.73 🥰



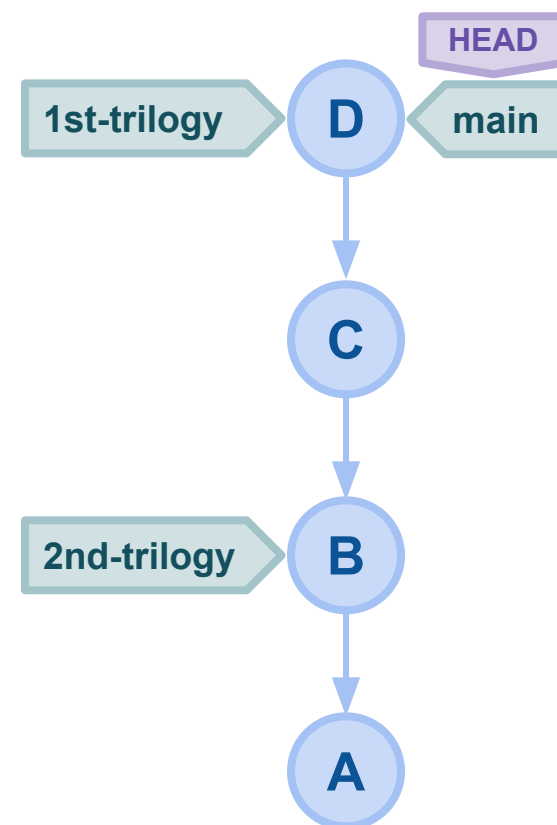
```
git branch 2nd-trilogy B
```

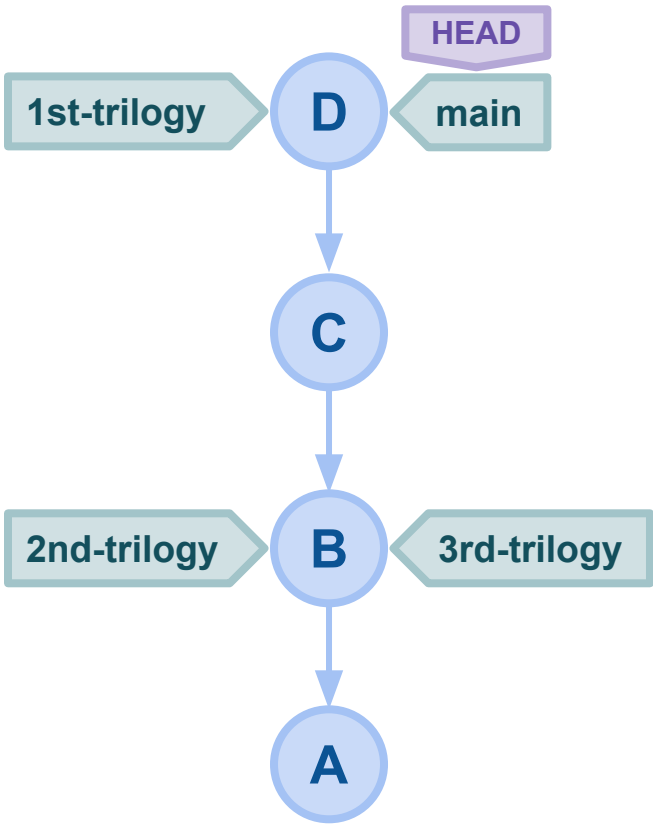




Git Simulator v.1.3.73 🥰

```
git branch 3rd-trilogy 2nd-trilogy
```



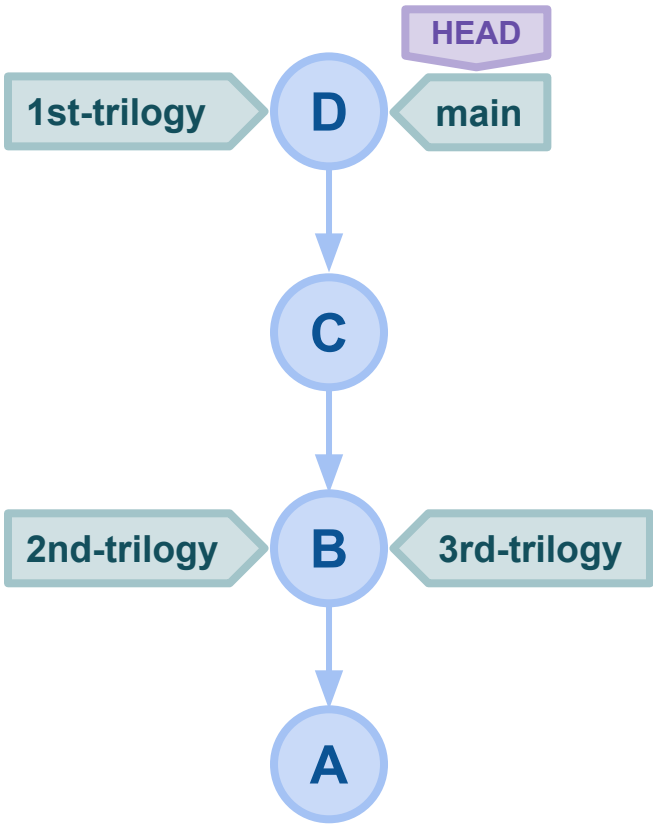


■ Cambiarse de una rama a otra

`git checkout`

- Permite cambiar de una rama a otra
- ⚠ Al cambiar de rama, cambia el contenido del **Working Copy** ⚠





JS

JS

CSS

CSS

CSS

CSS

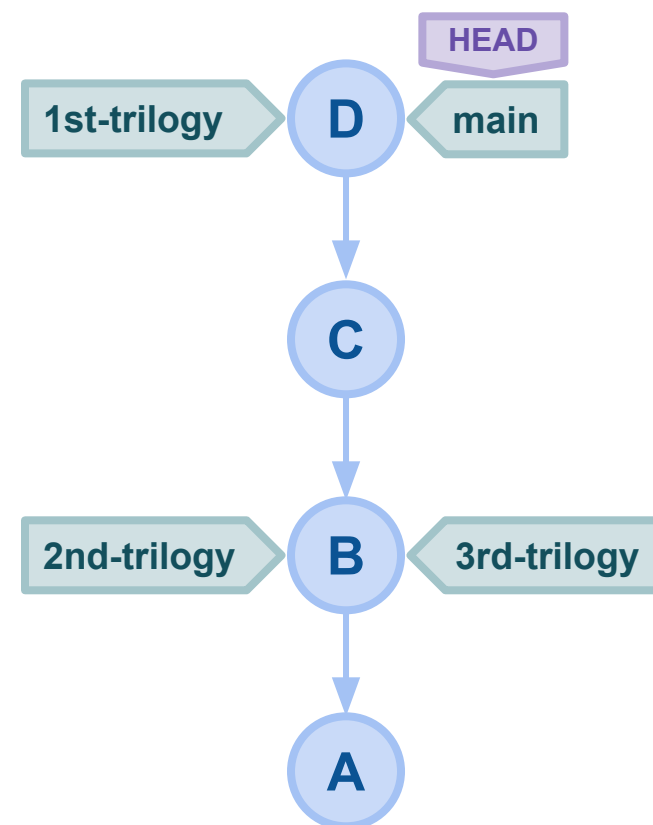
CSS

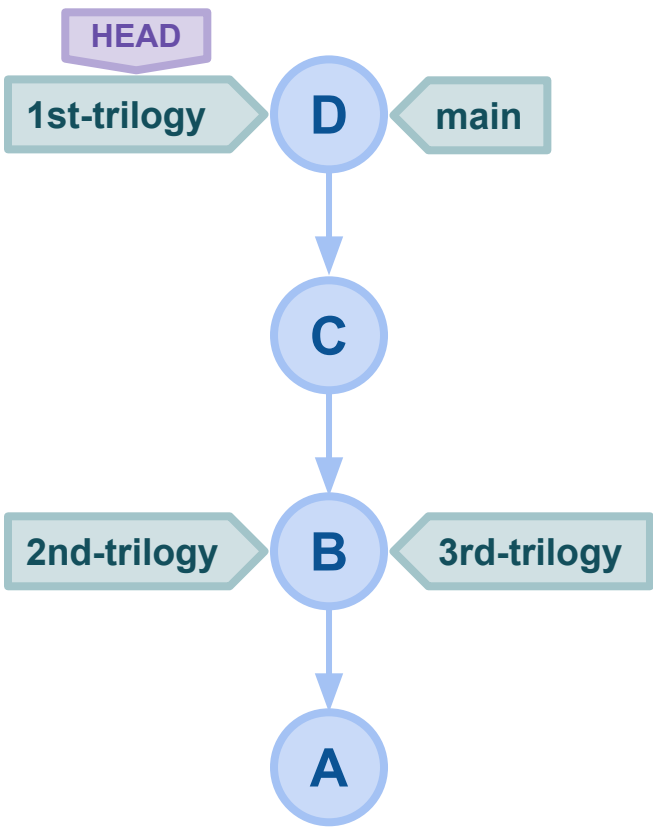
CSS



Git Simulator v.1.3.73 🤯

```
git checkout 1st-trilogy
```





JS

JS

CSS

CSS

CSS

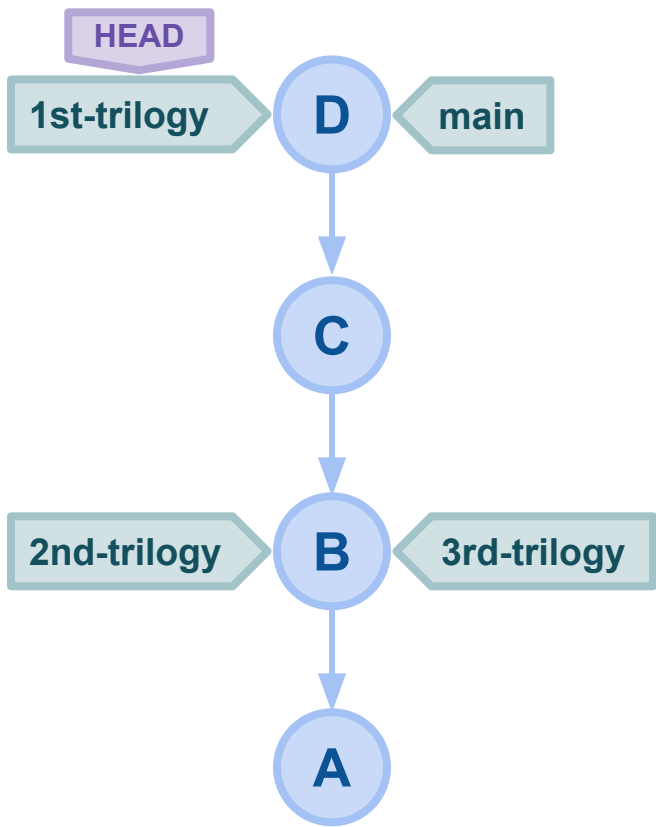
CSS

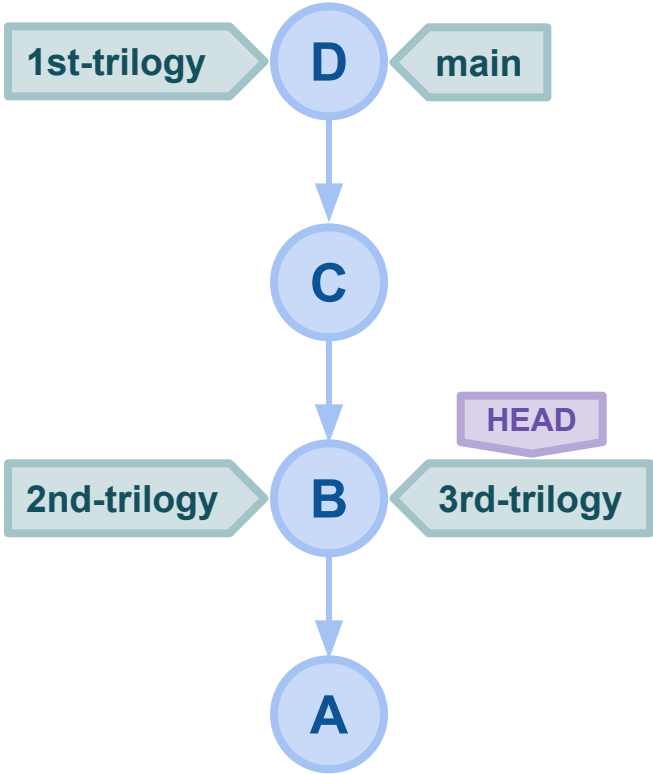
CSS

CSS

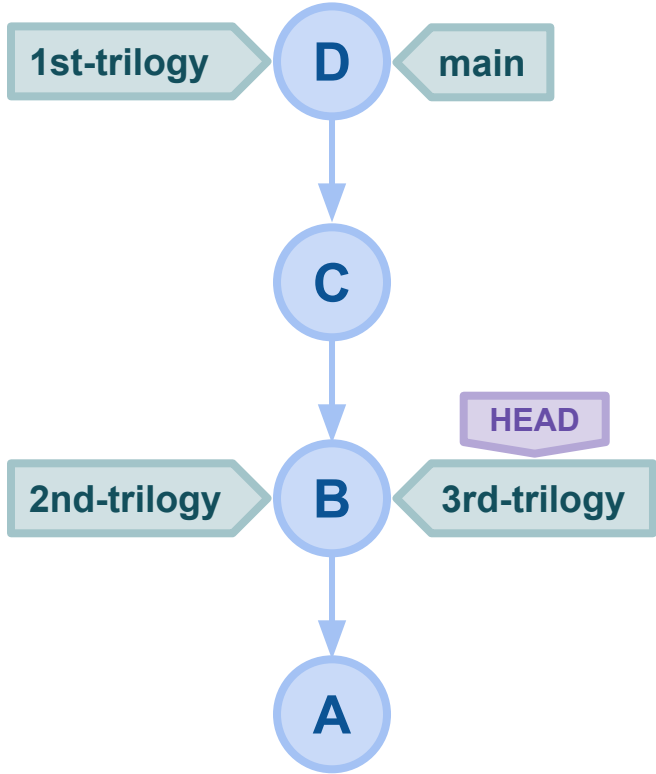


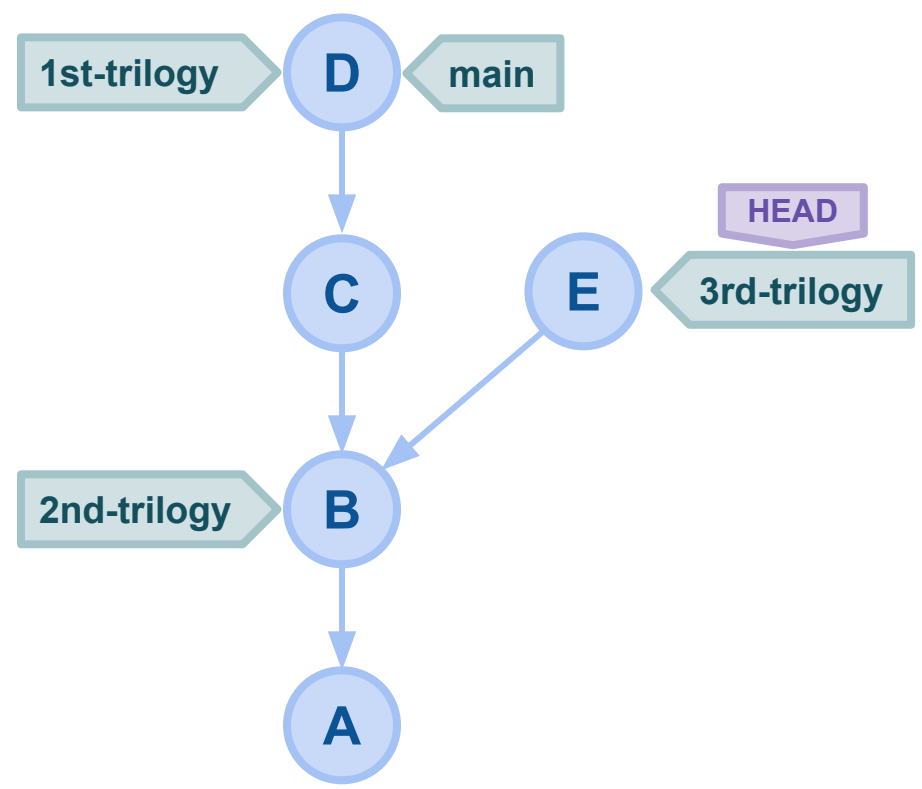
```
git checkout 3rd-trilogy
```





```
git commit -m "E"
```





JS

JS

CSS



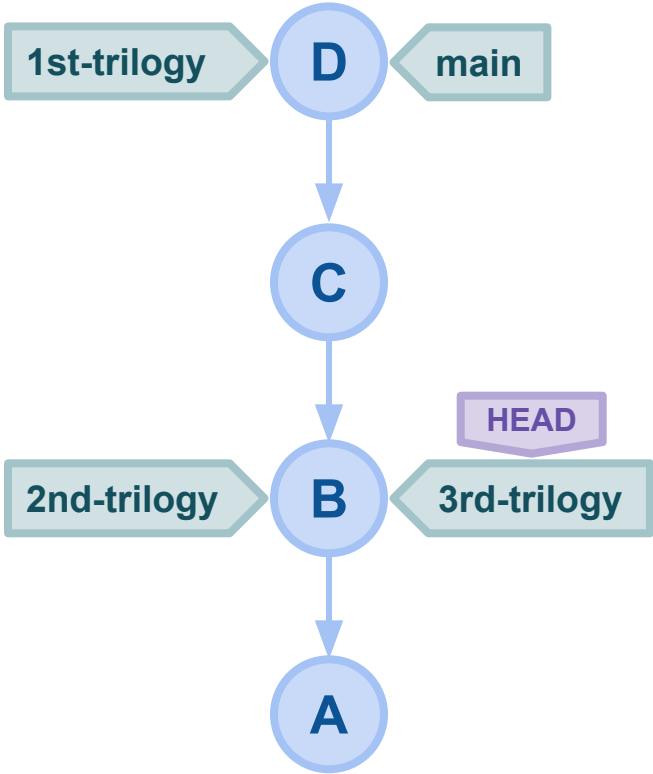
■ Eliminar una rama

git branch

```
# Elimina la rama <branch name> siempre y cuando no deje commits inaccesibles  
git branch -d <branch name>  
  
# Elimina la rama <branch name> aunque deje commits inaccesibles  
git branch -D <branch name>
```

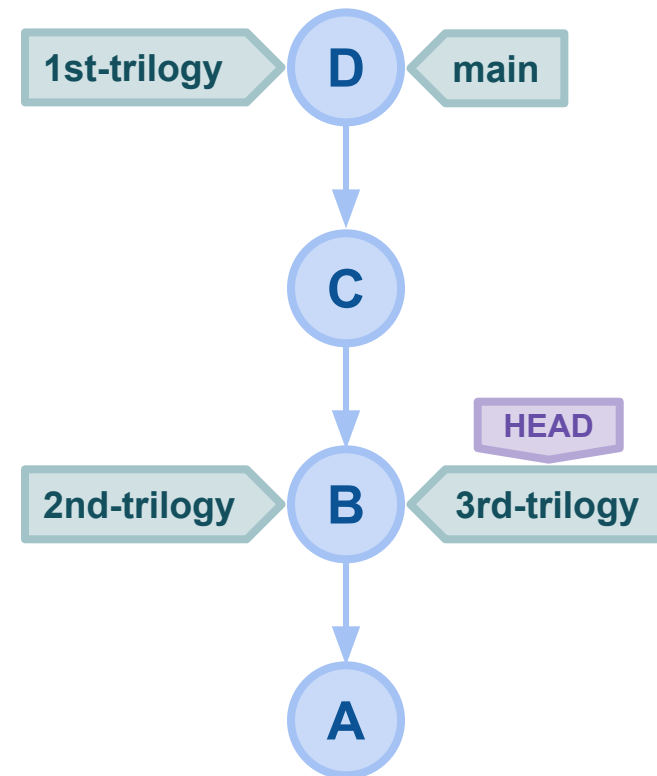
⚠ No podemos borrar la rama en la que estamos ⚠

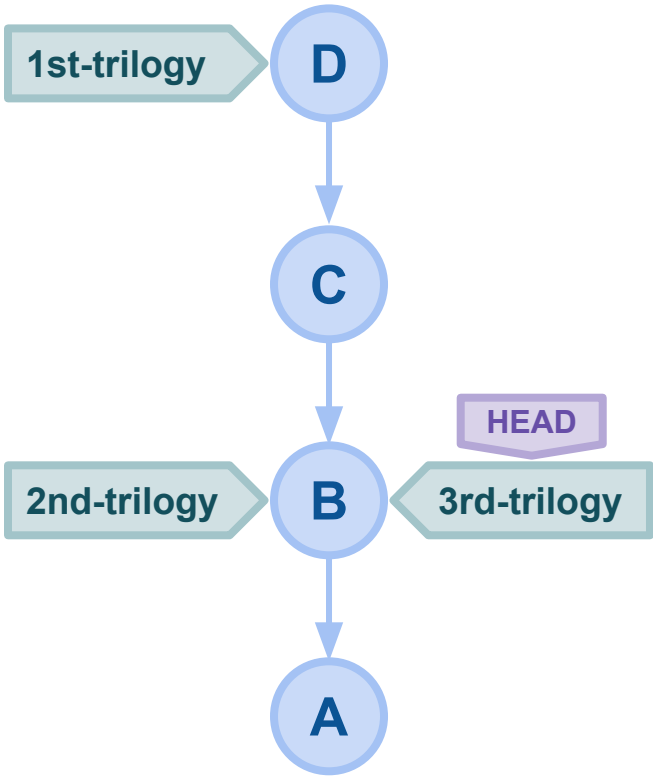




Git Simulator v.1.3.73 🤯

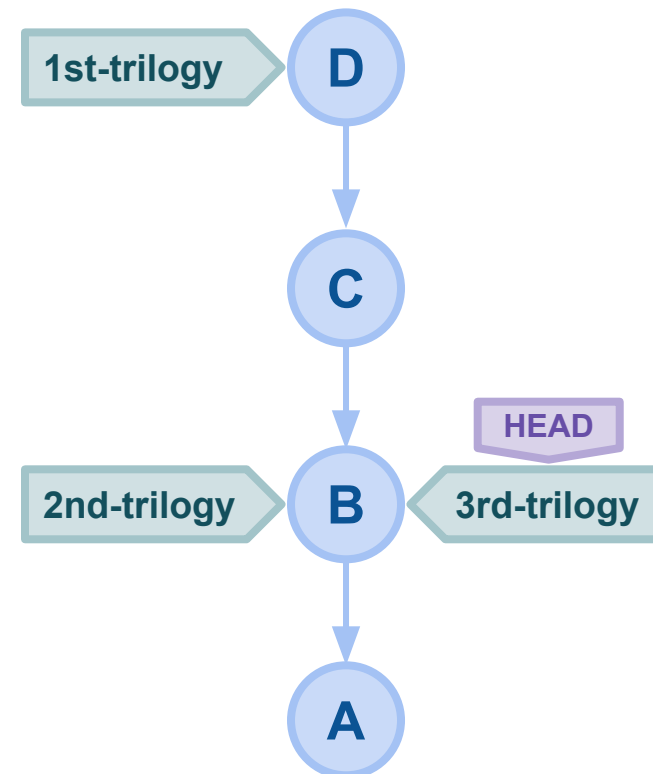
```
git branch -d main
```





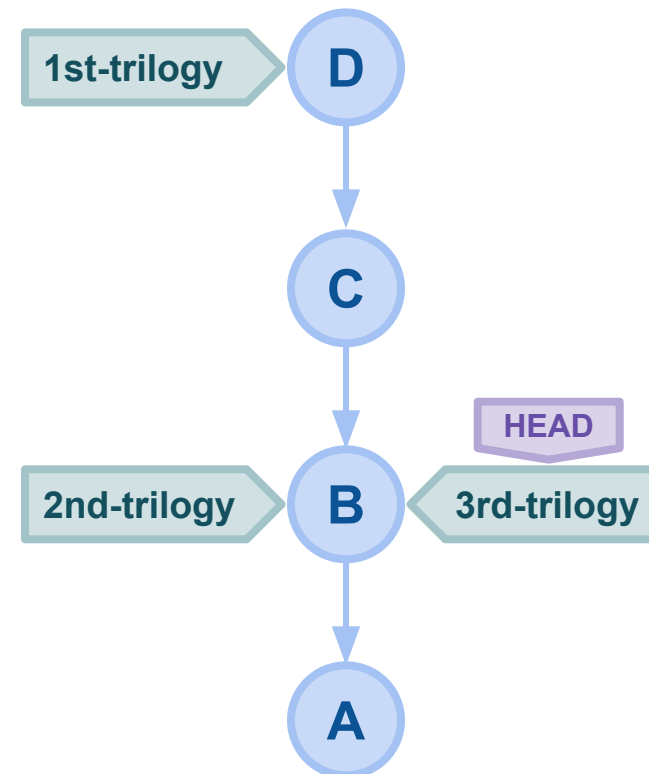
Git Simulator v.1.3.73 🤯

```
git branch -d 1st-trilogy
```



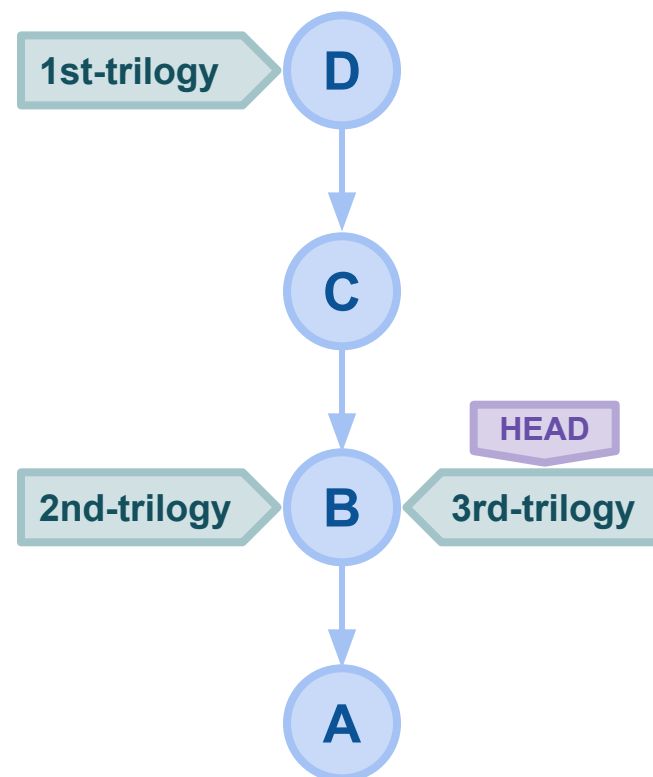
Git Simulator v.1.3.73 🤯

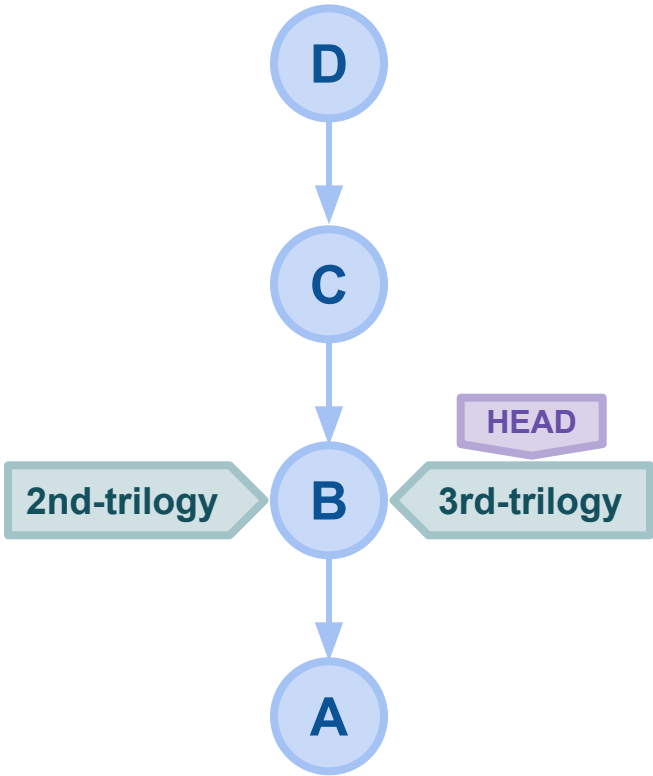
⚠️ ¡Error! ¿Estás seguro? ¡Que dejas commits inaccesibles! ⚠️



Git Simulator v.1.3.73 🤖

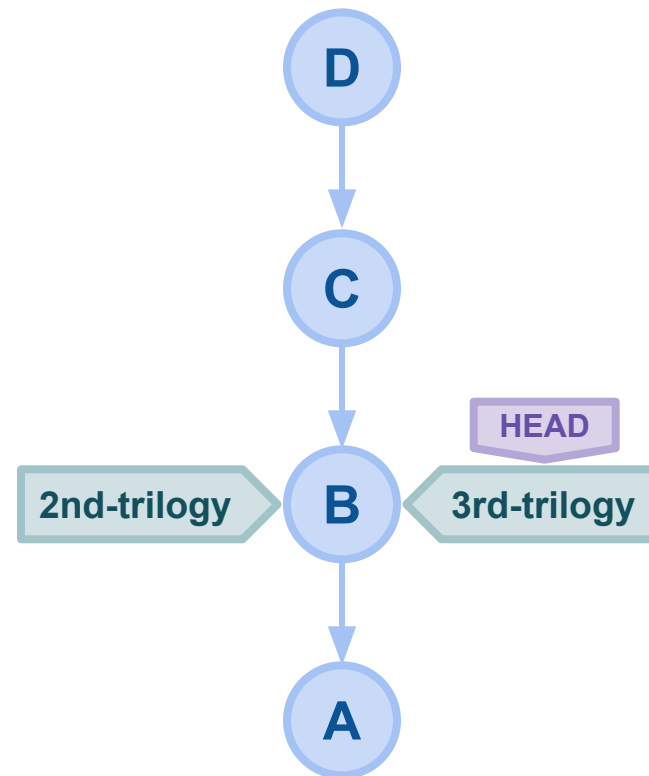
```
git branch -D 1st-trilogy
```

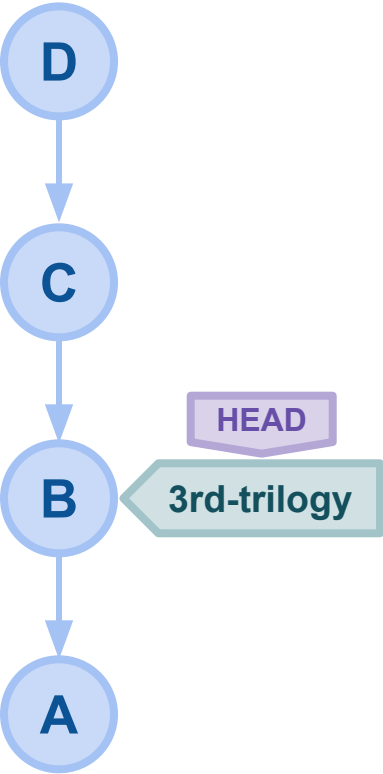




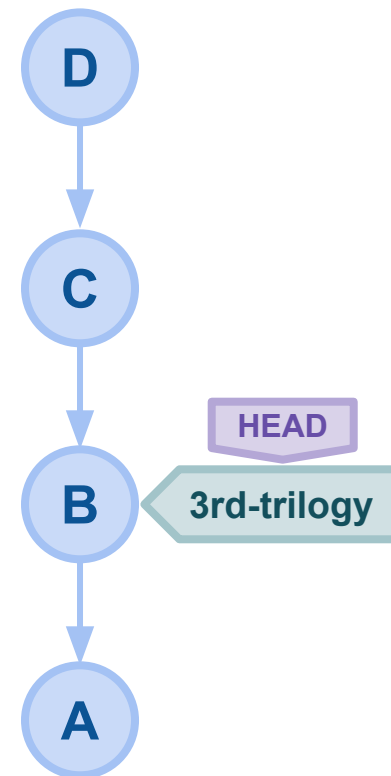
Git Simulator v.1.3.73 🤯

```
git branch -d 2nd-trilogy
```

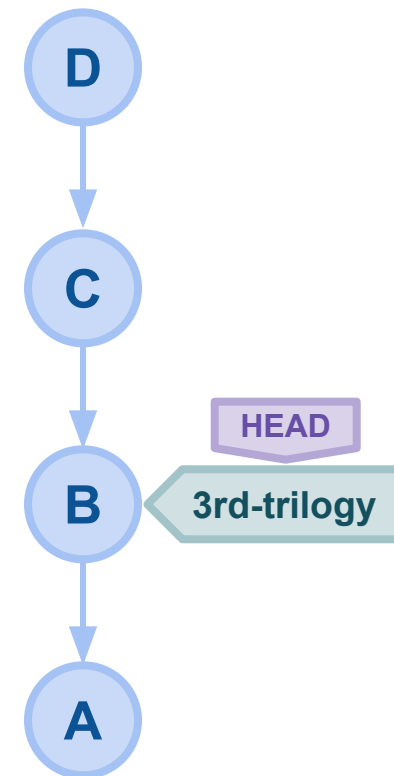





```
git branch -D 3rd-trilogy
```

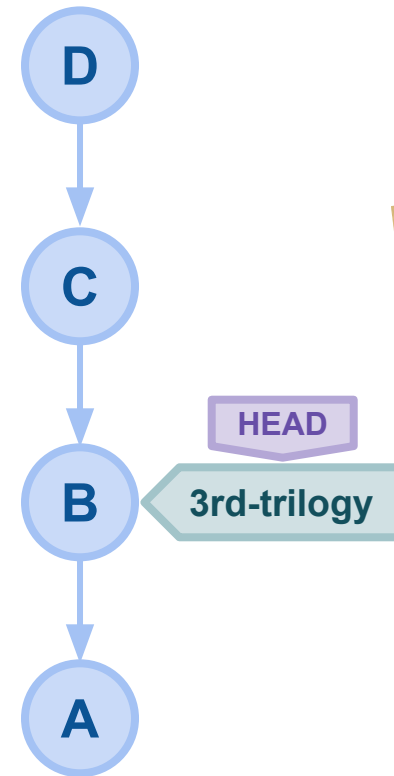


⚠️ ¡Error! No puedes eliminar la rama en la que estás ⚠️



Git Simulator v.1.3.73 🤯

¿Y ahora cómo recuperamos las ramas?







git reflog al rescate



Tags

Etiquetas para los amigos...y primas hermanas de las ramas



■ Tags

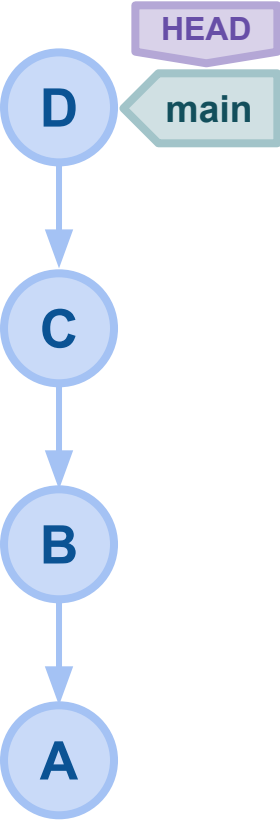
`git tag`

- Son como las ramas...pero que no se pueden mover
- Sirven para marcar hitos importantes en nuestro desarrollo:
 - Números de versión
 - Actualizaciones de framework

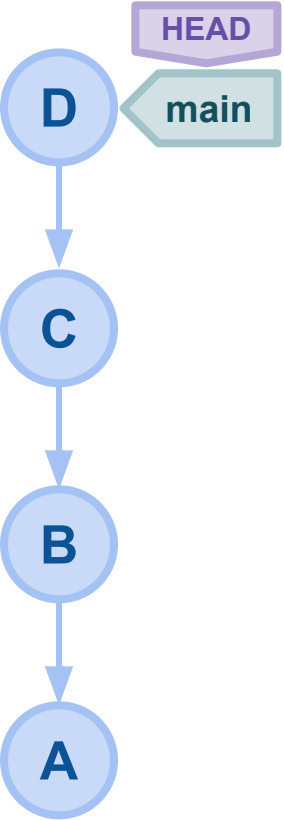




Git Simulator v.1.3.73

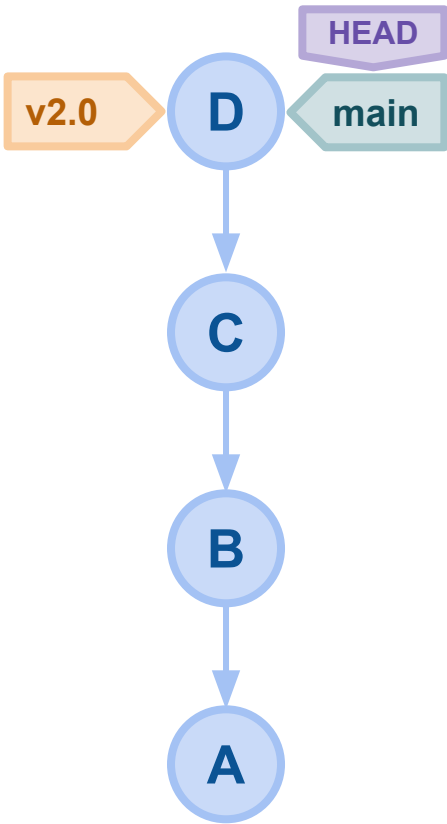



```
git tag v2.0
```

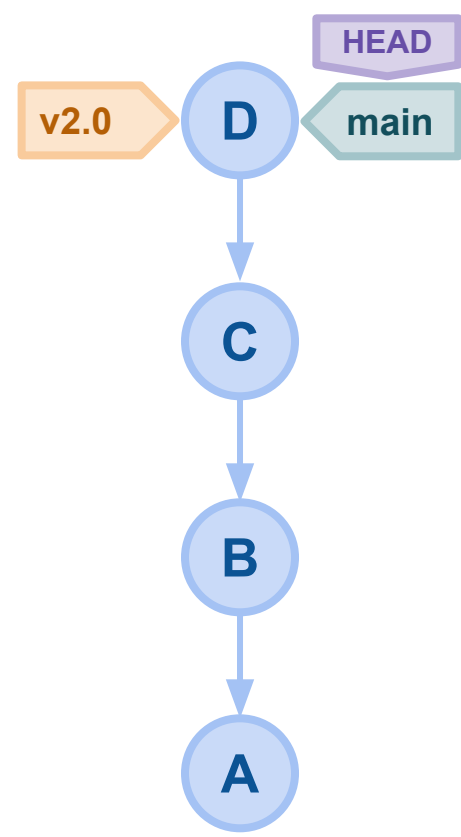




Git Simulator v.1.3.73 🥰💧

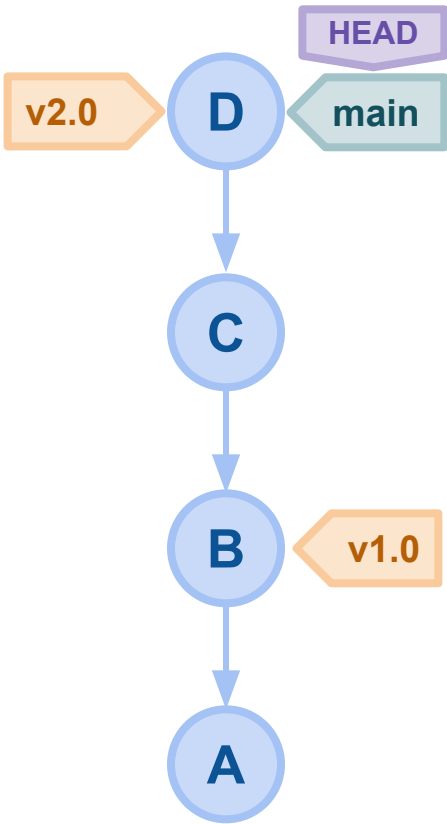


```
git tag v1.0 B
```

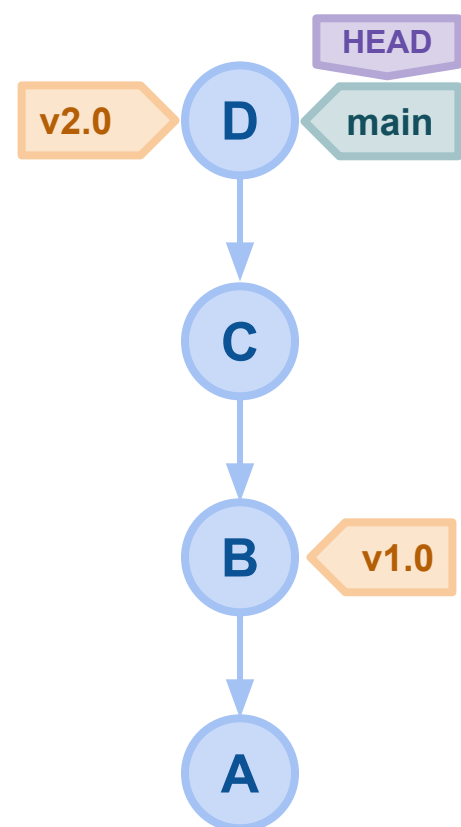


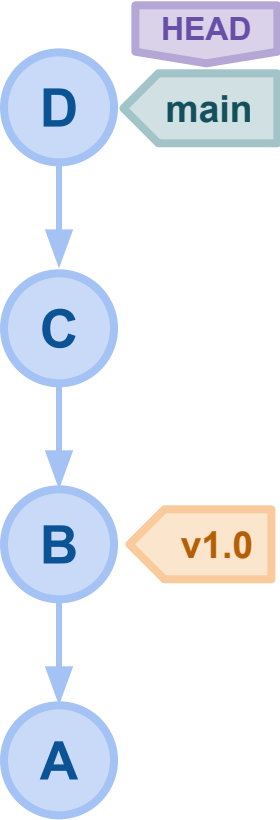


Git Simulator v.1.3.73



```
git tag -d v2.0
```



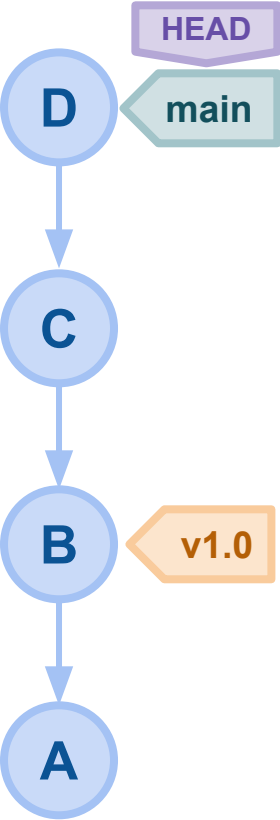


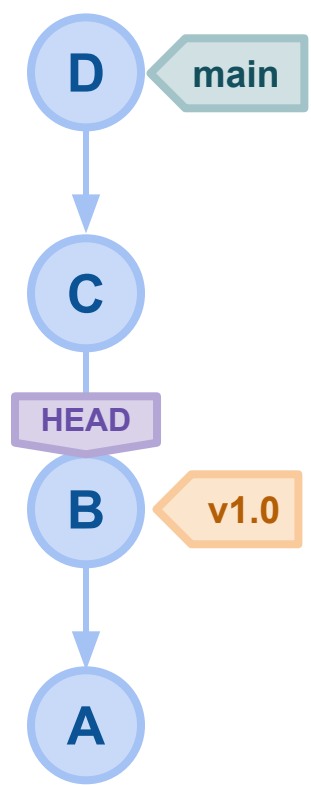


Git Simulator v.1.3.73 🥰💧



```
git checkout v1.0
```

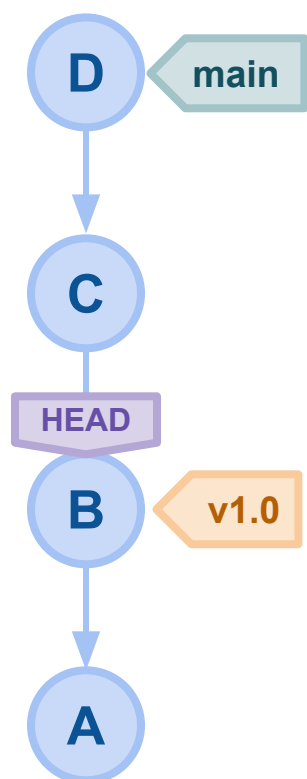


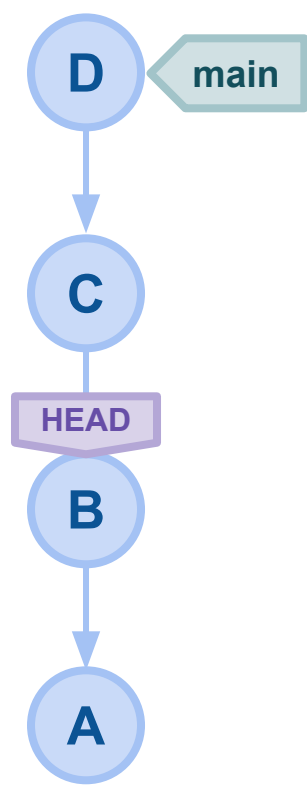







```
git tag -d v1.0
```



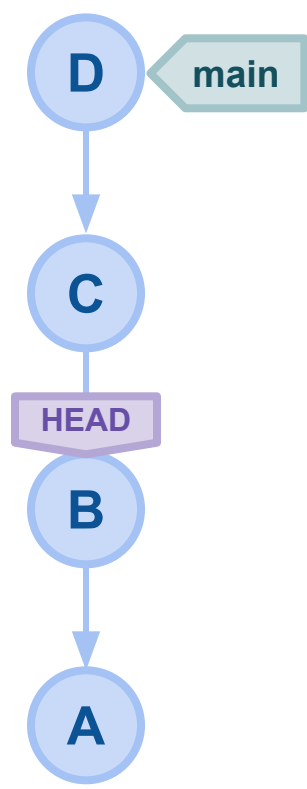






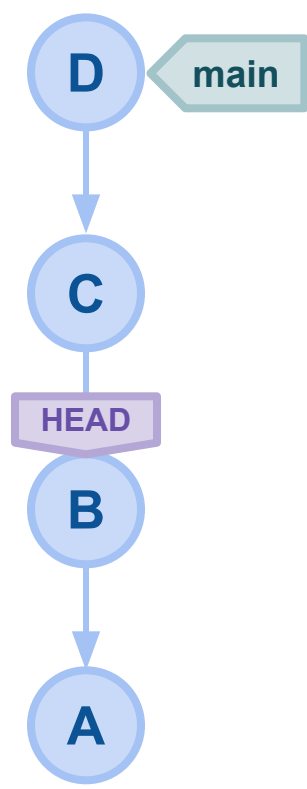


Git Simulator v.1.3.73 🥰💧







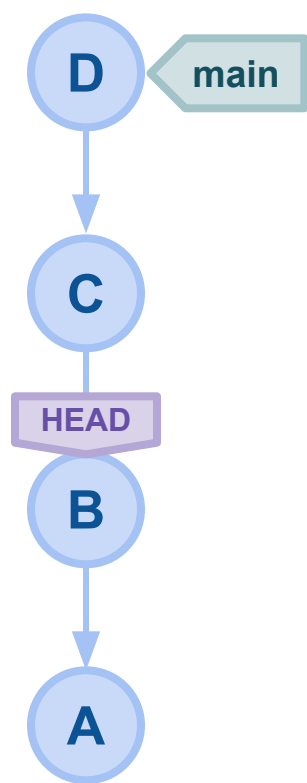








Git Simulator v.1.3.73



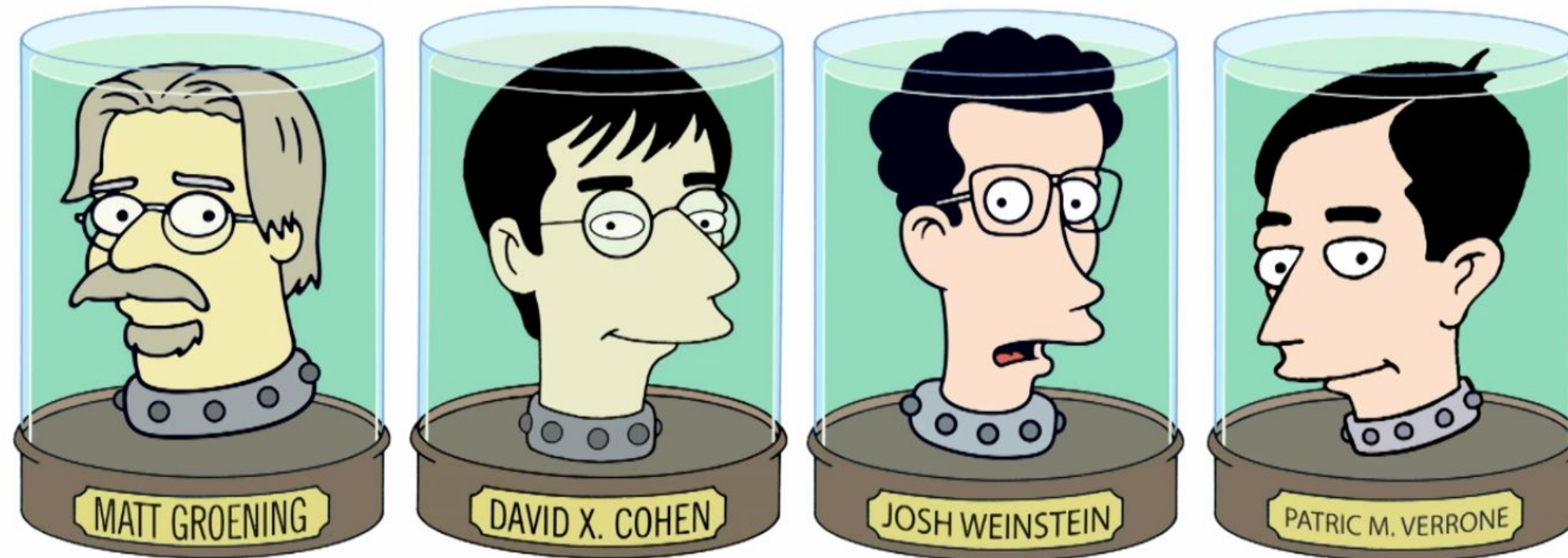
■ git checkout

¿Y qué pasa cuando no estamos en una rama?



■ git checkout

Que estaremos en un estado *DETACHED HEAD*



■ git checkout

Ok, entonces

¿para qué sirve el comando `git checkout`?



■ git checkout

¡MENTIRA!

~~Para cambiar de una rama a otra~~



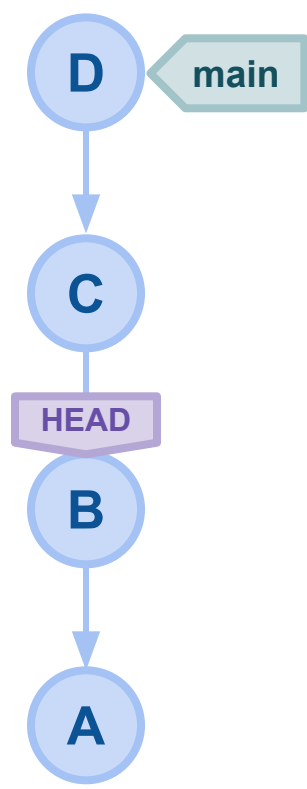
■ git checkout

Sirve para mover HEAD
allá donde nosotros queramos.





Git Simulator v.1.3.73 🤪

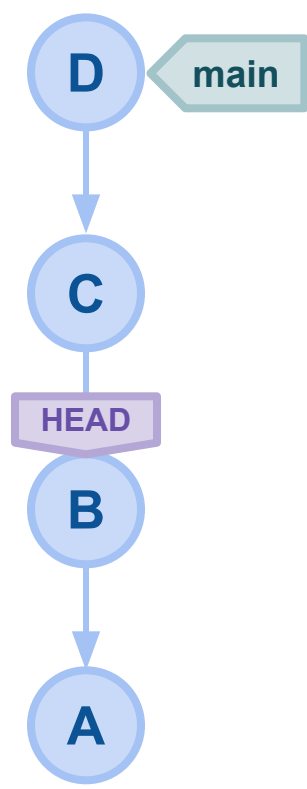










Git Simulator v.1.3.73 🤪



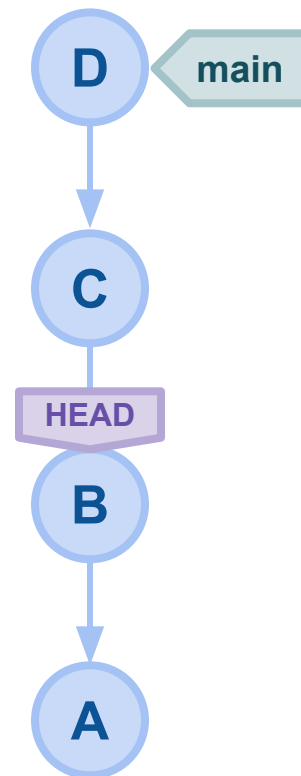


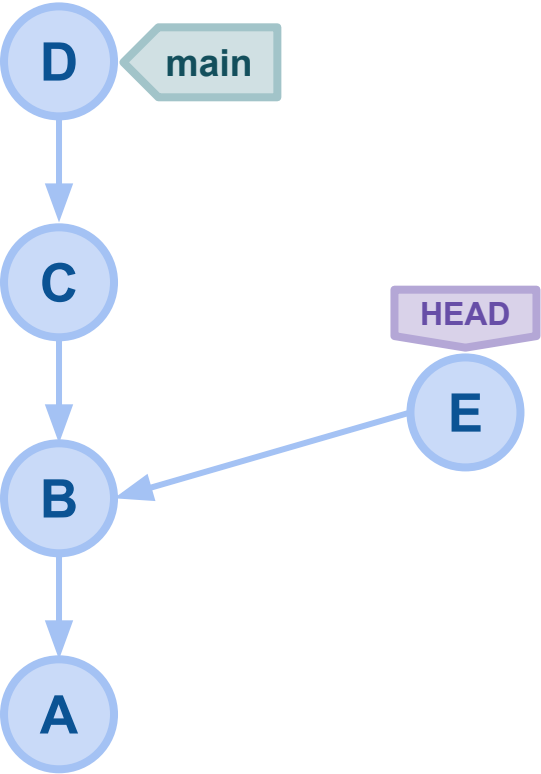




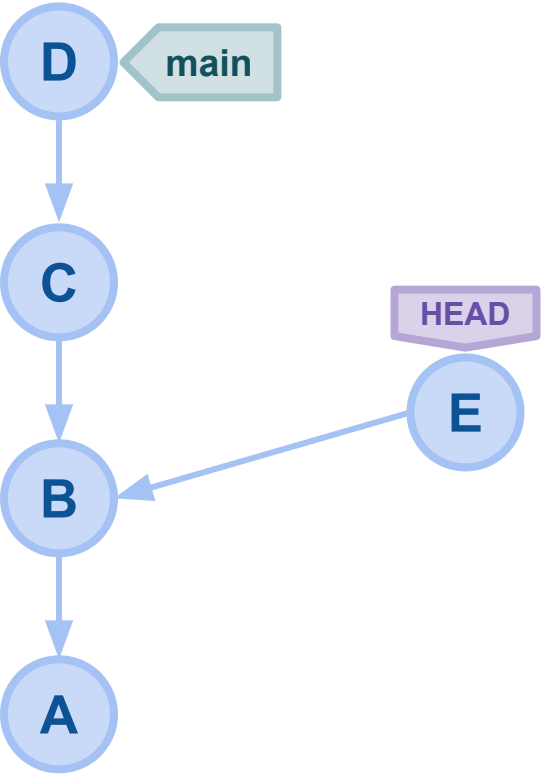
Git Simulator v.1.3.73 🤪

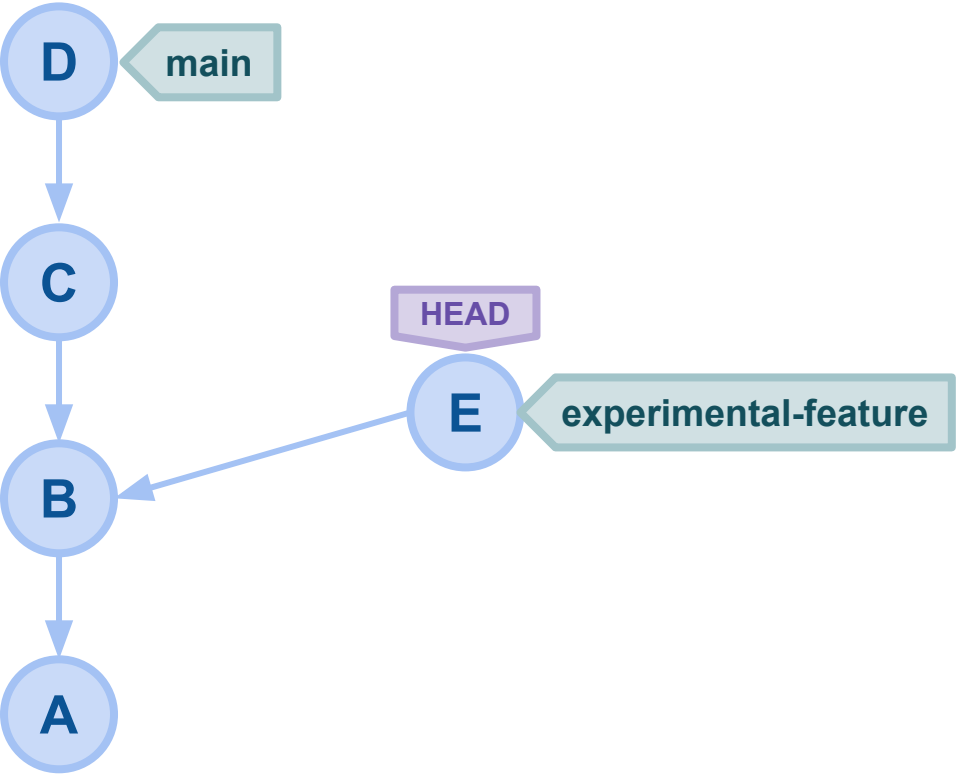
```
git commit -m "Commit experimental"
```

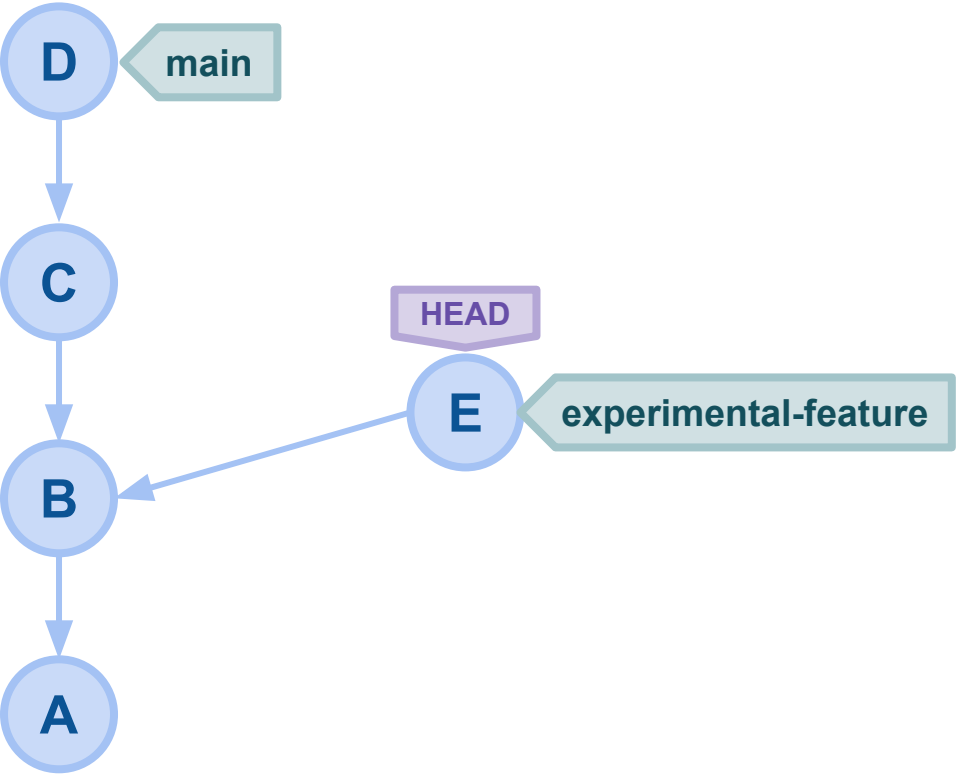




```
git branch experimental-feature
```







JS

JS

CSS

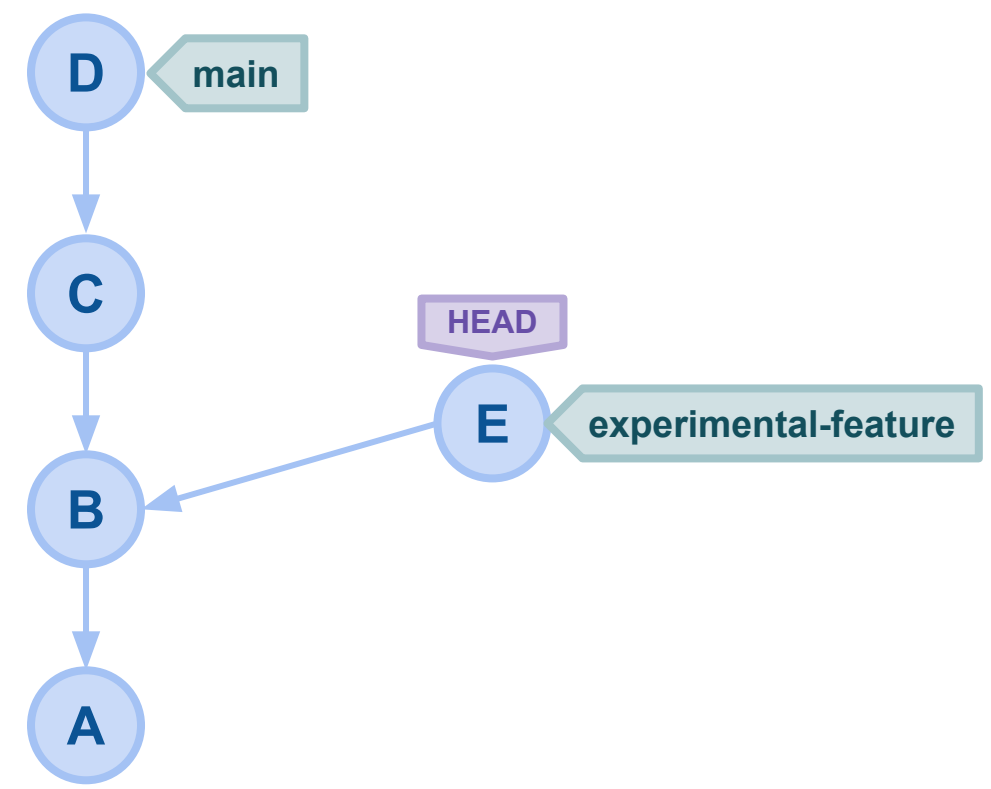
?

CSS




Git Simulator v.1.3.73 🤪

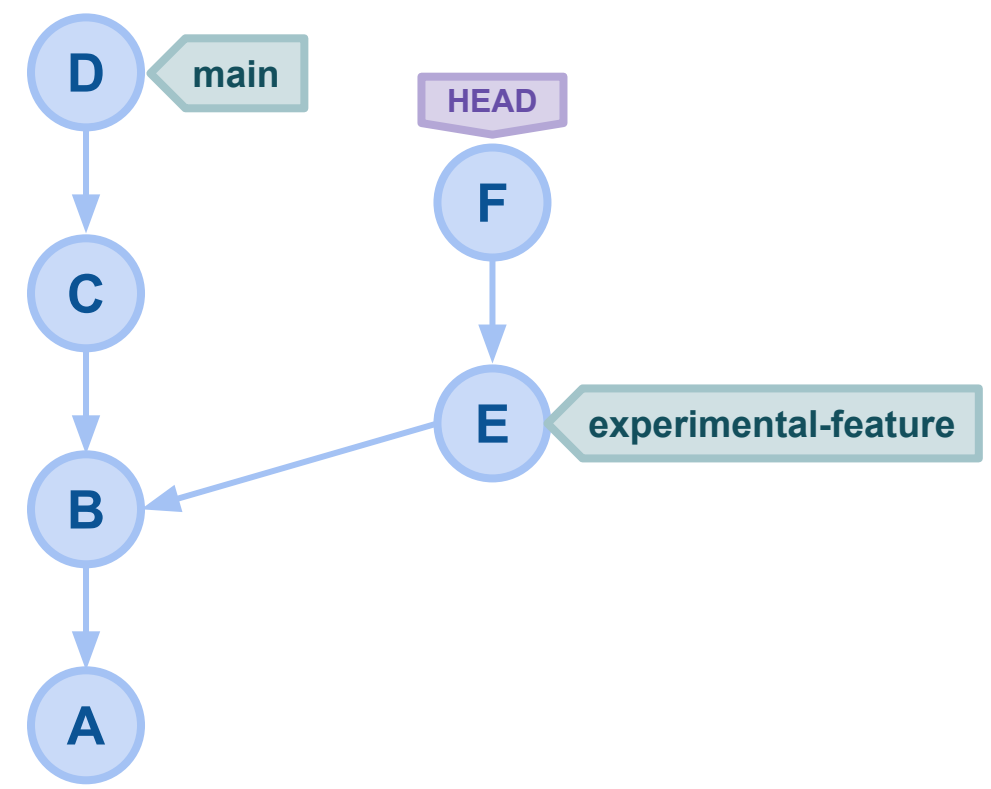
```
git commit -m "Otra prueba experimental"
```









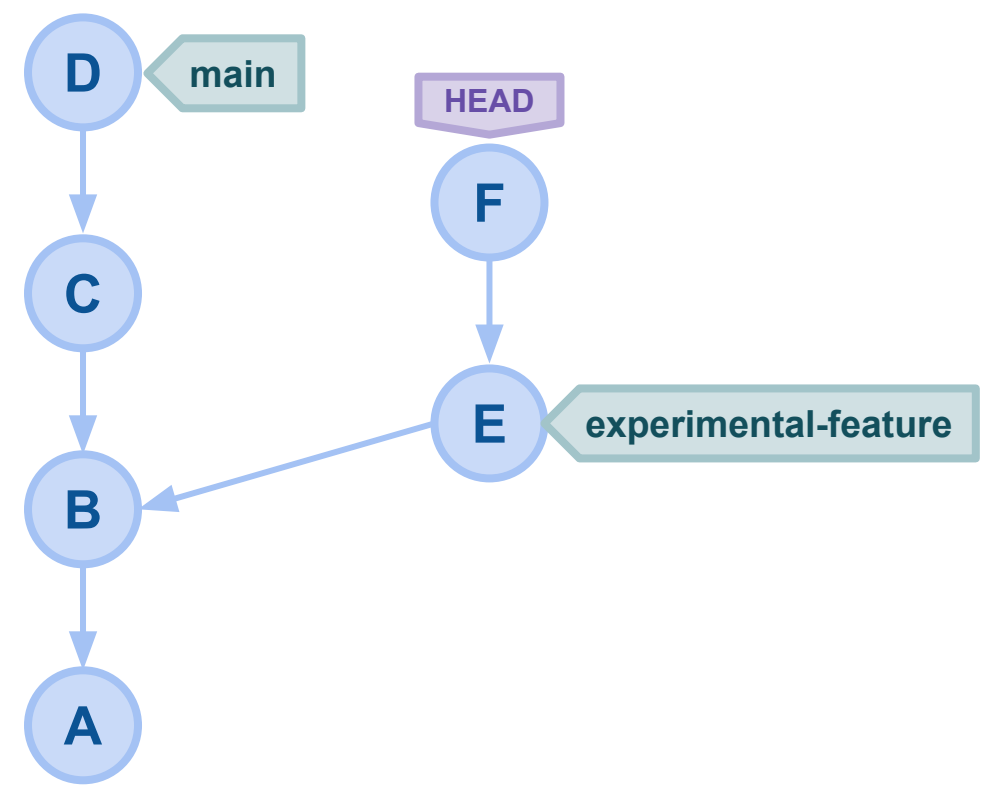


JS

JS

CSS

```
git checkout C
```



JS

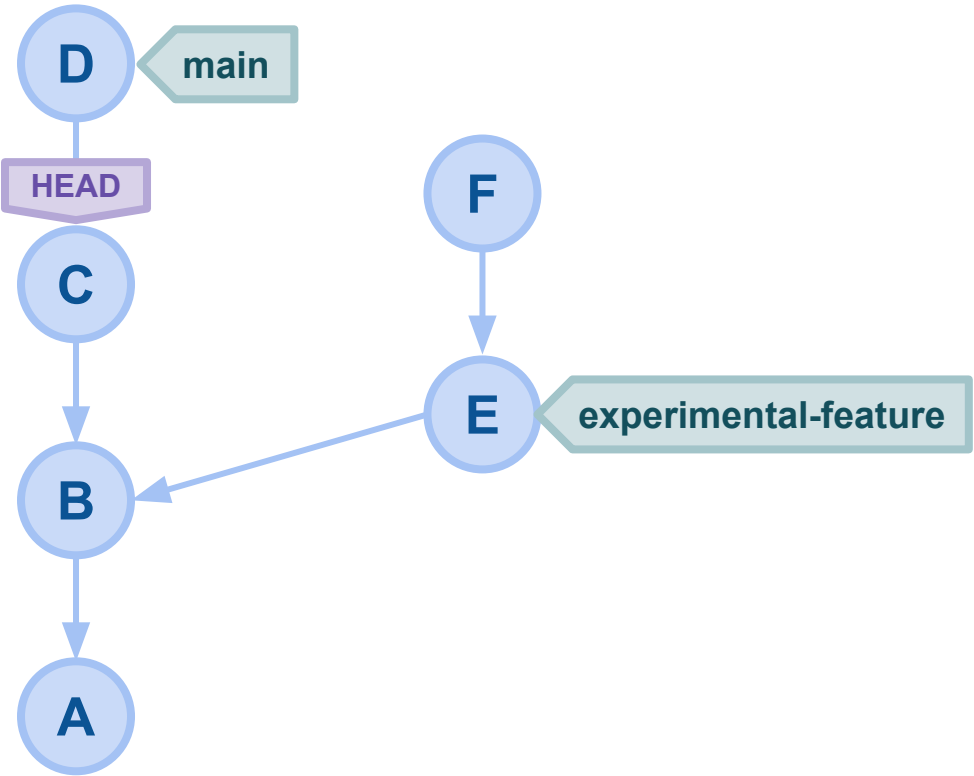
JS

CSS





Git Simulator v.1.3.73 🤪

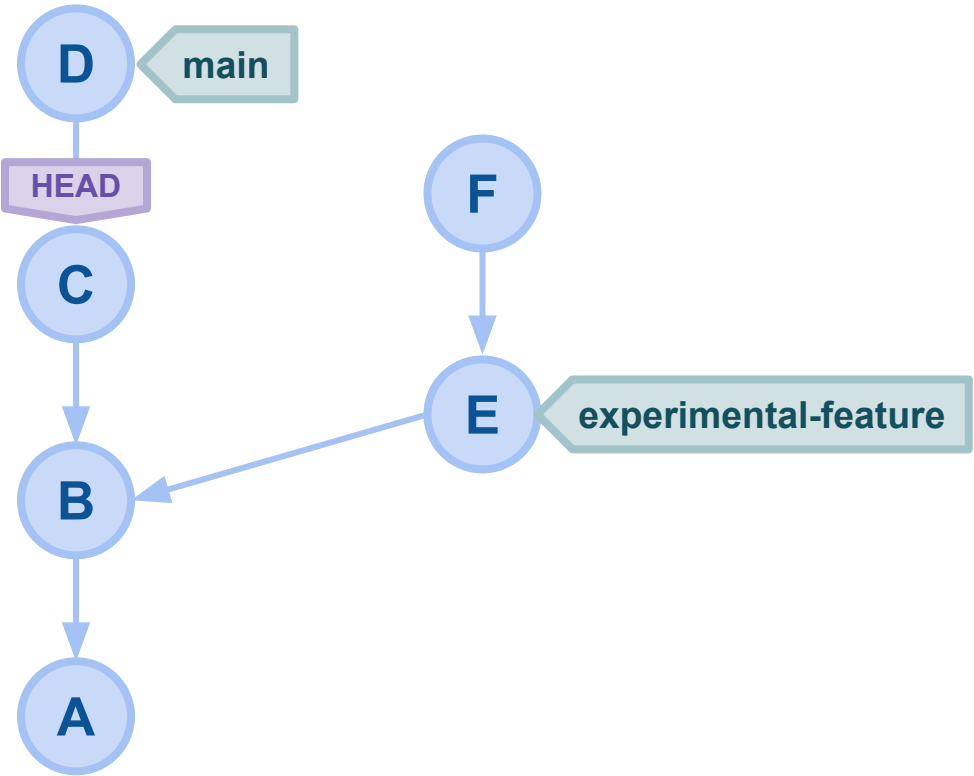




Git Simulator v.1.3.73 🤪

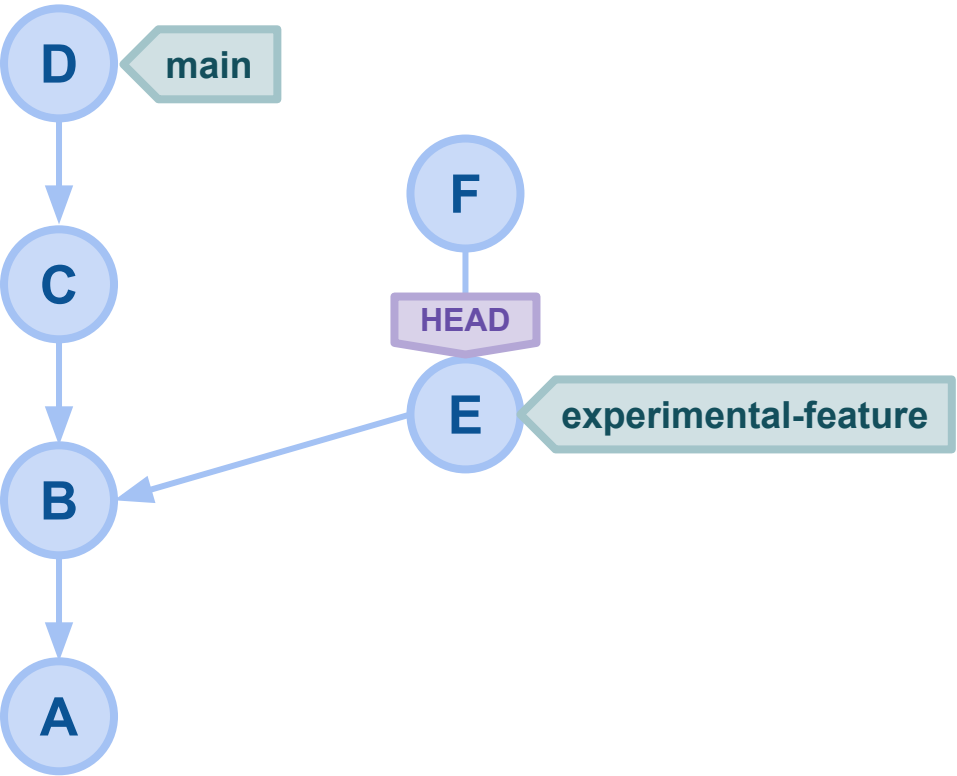


```
git checkout E
```

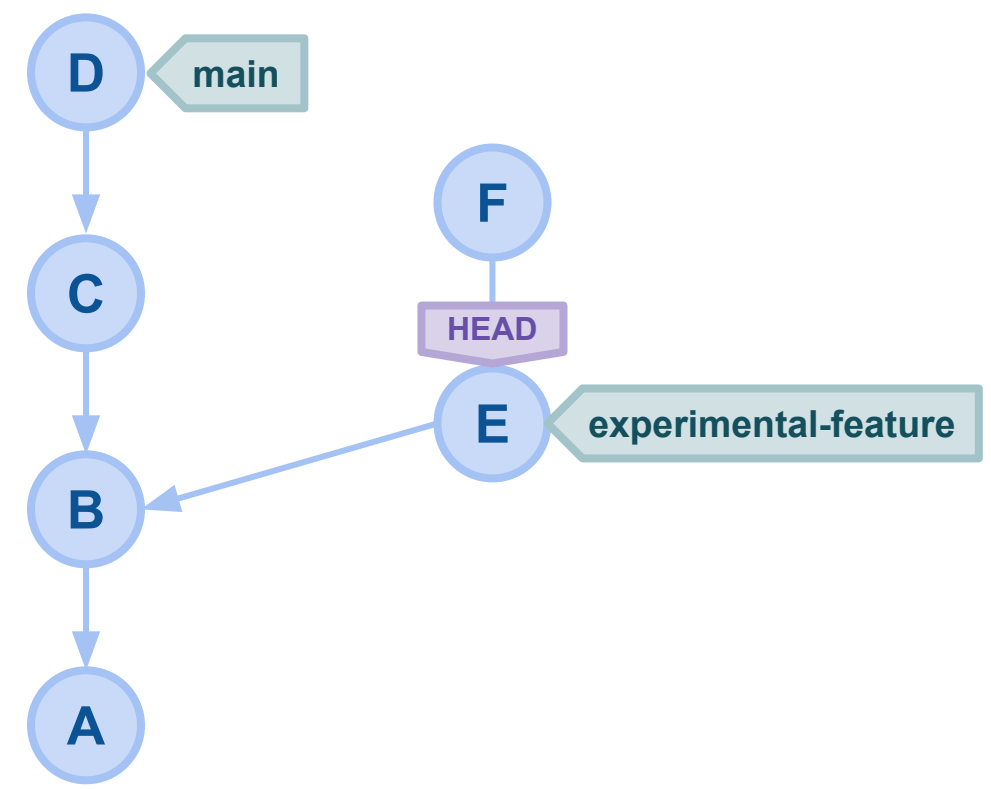




Git Simulator v.1.3.73 🤪



```
git checkout experimental-feature
```



JS

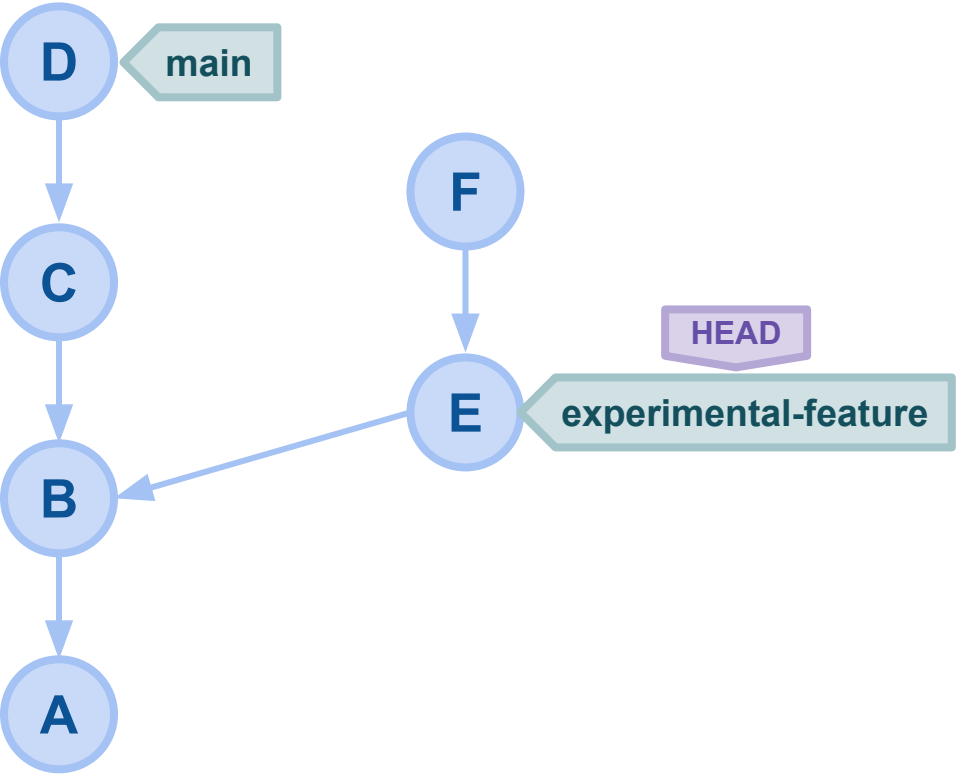
JS

CSS



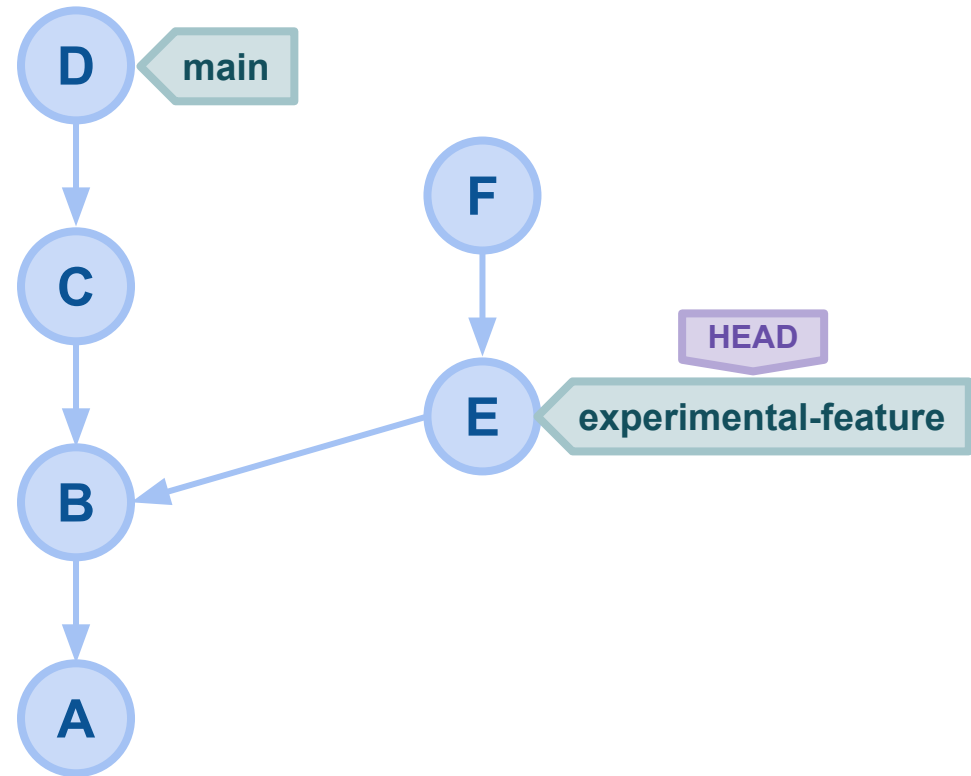


Git Simulator v.1.3.73 🤪



Git Simulator v.1.3.73 🤪

```
git reset --hard F
```

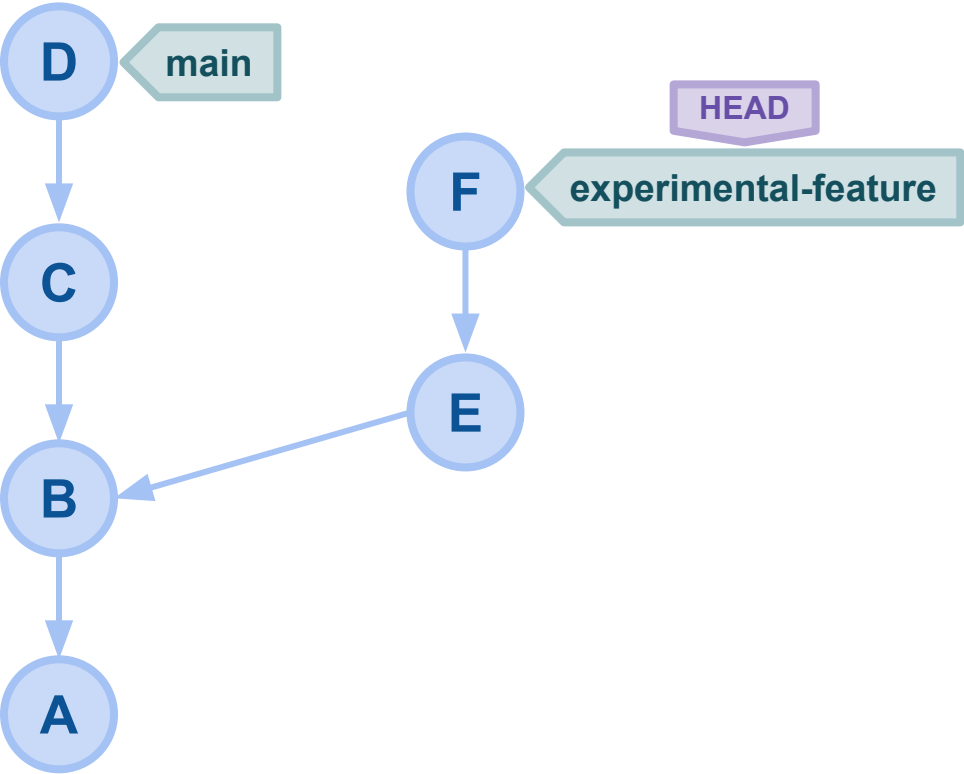








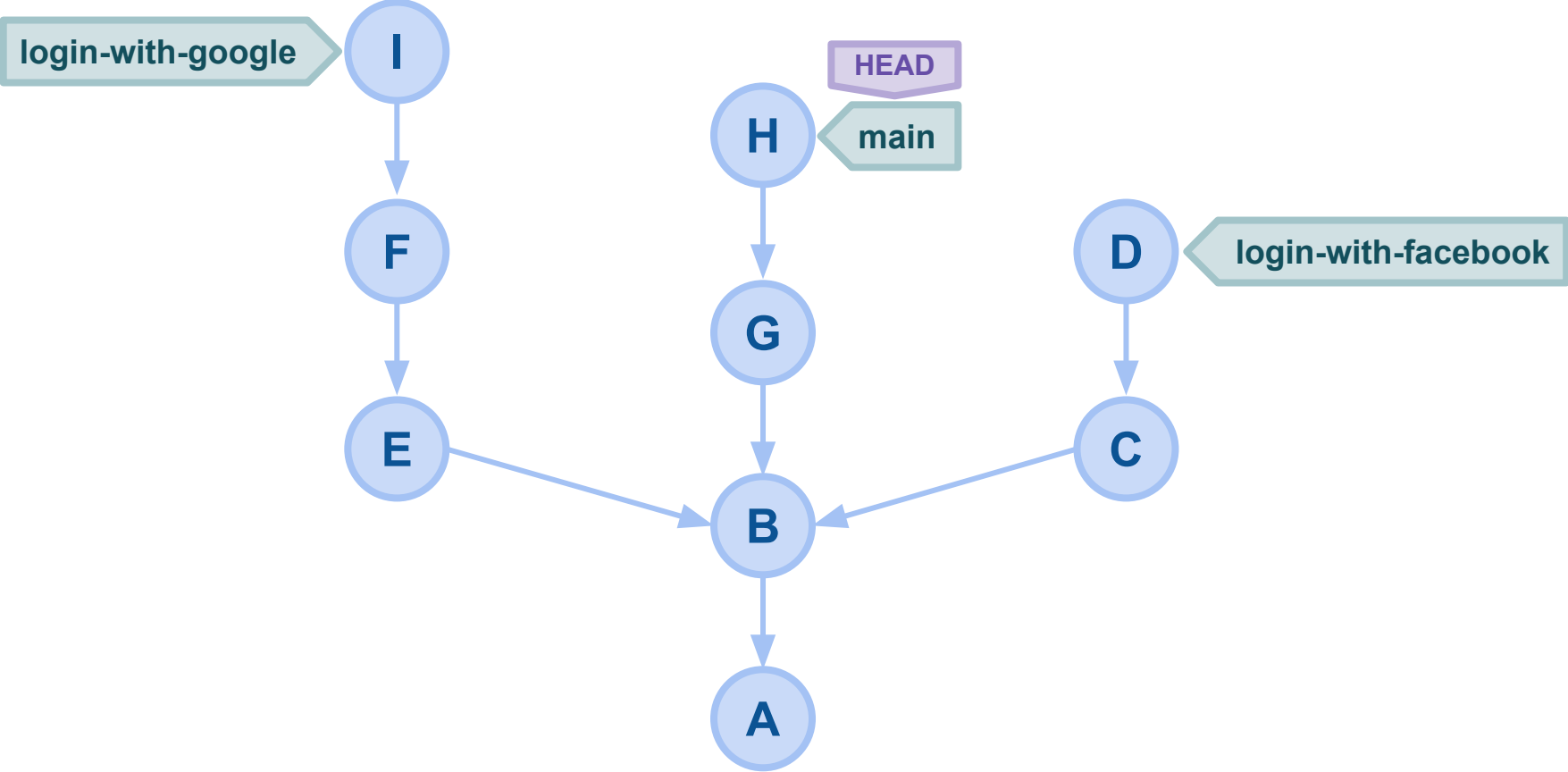
Git Simulator v.1.3.73 🤪

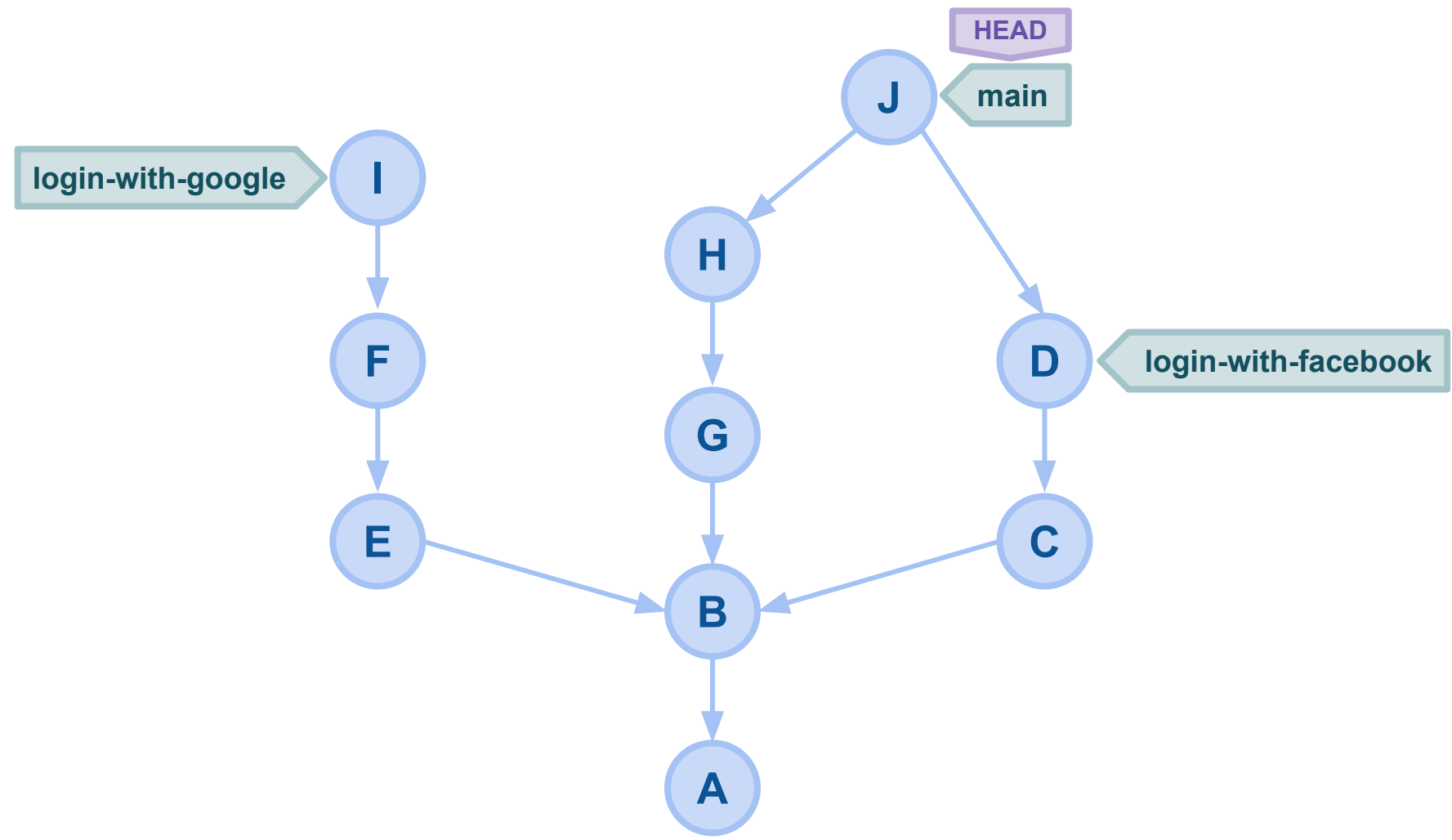





Merging

Fusionando branches

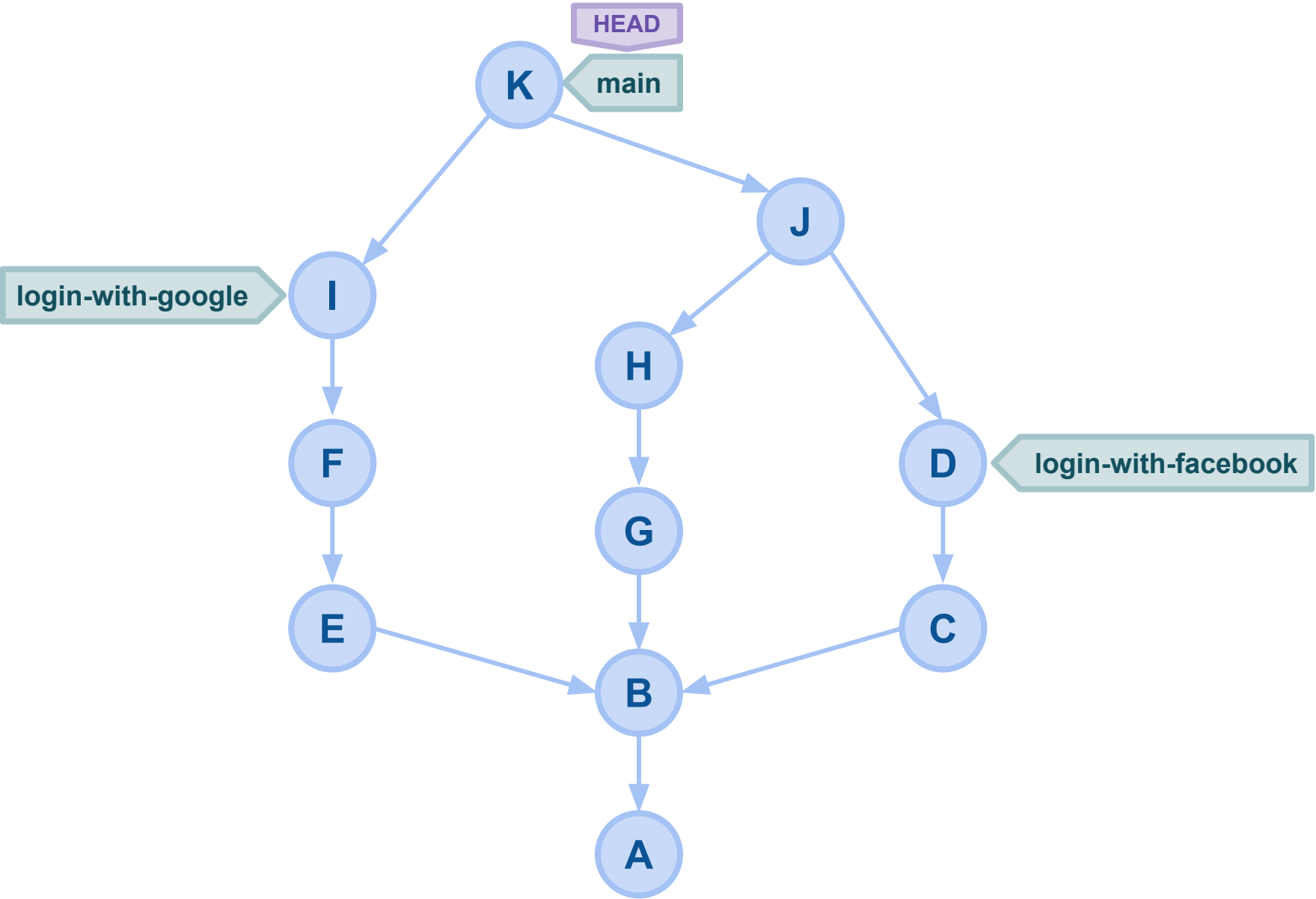


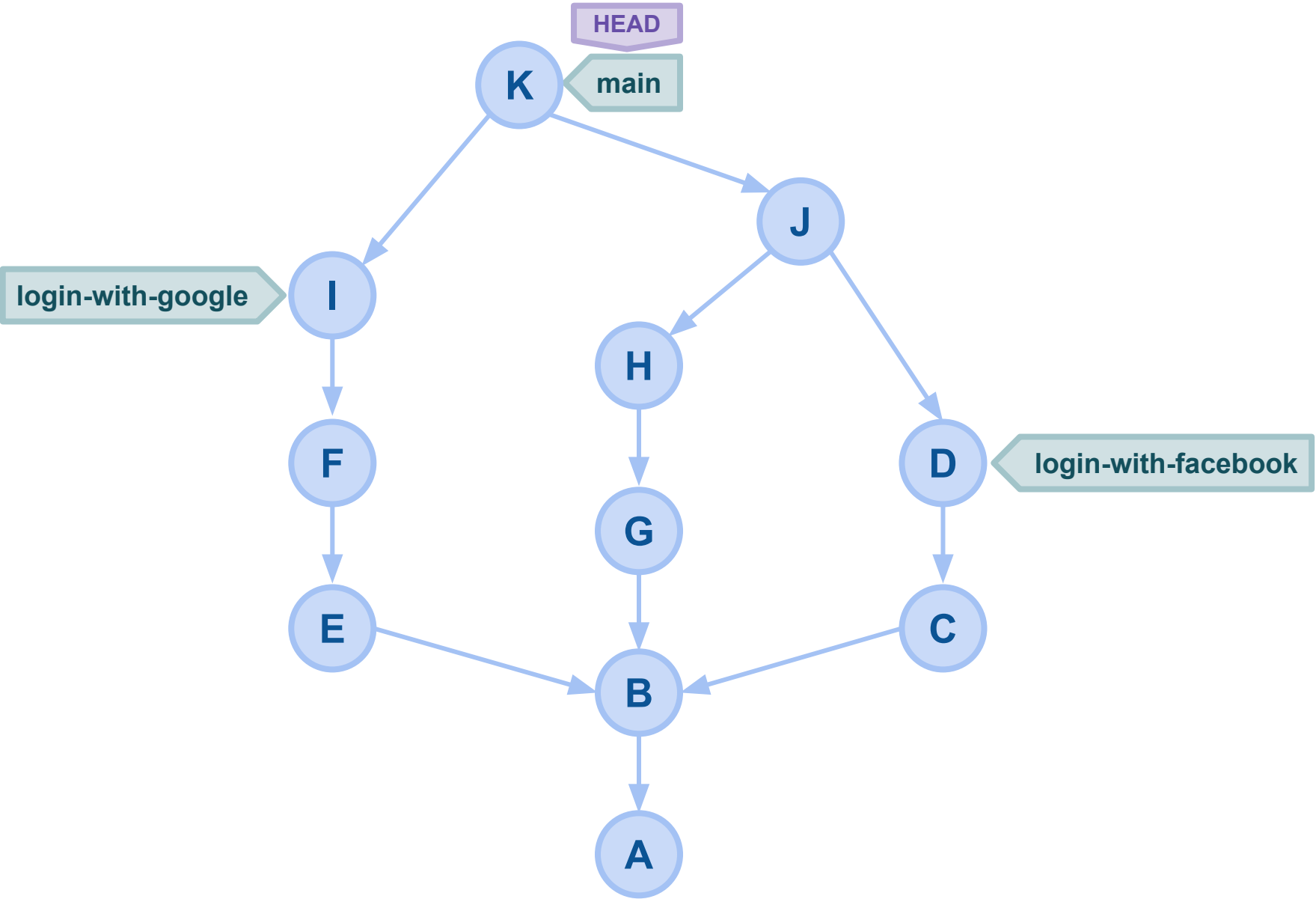










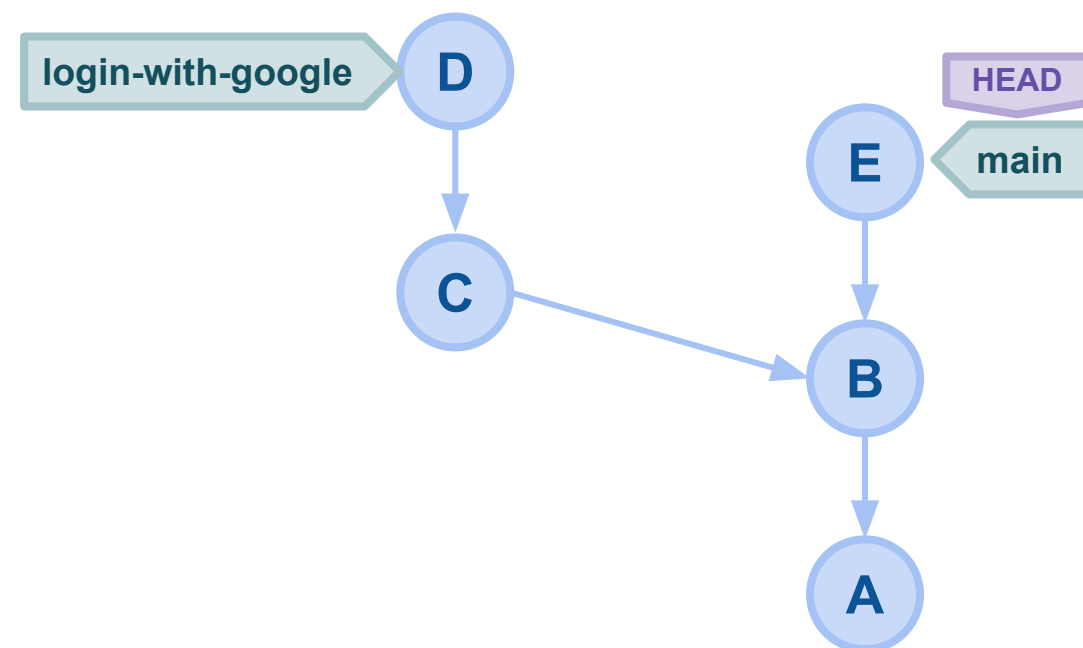


git merge

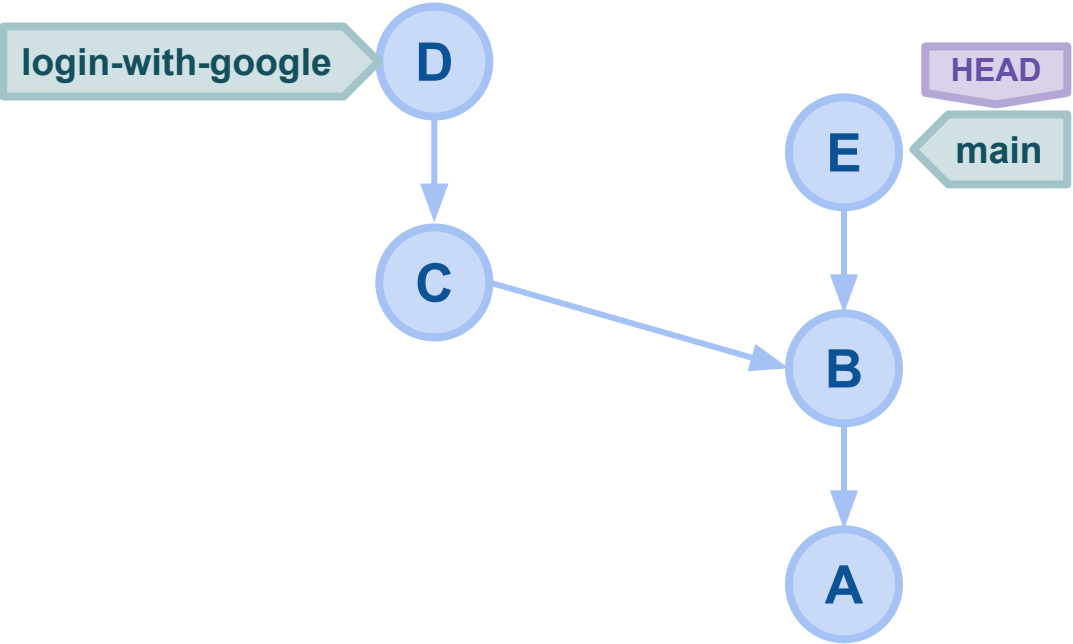
```
# Da acceso a los commits de la rama <branch name> en la rama en la que apunta HEAD  
git branch <branch name>
```



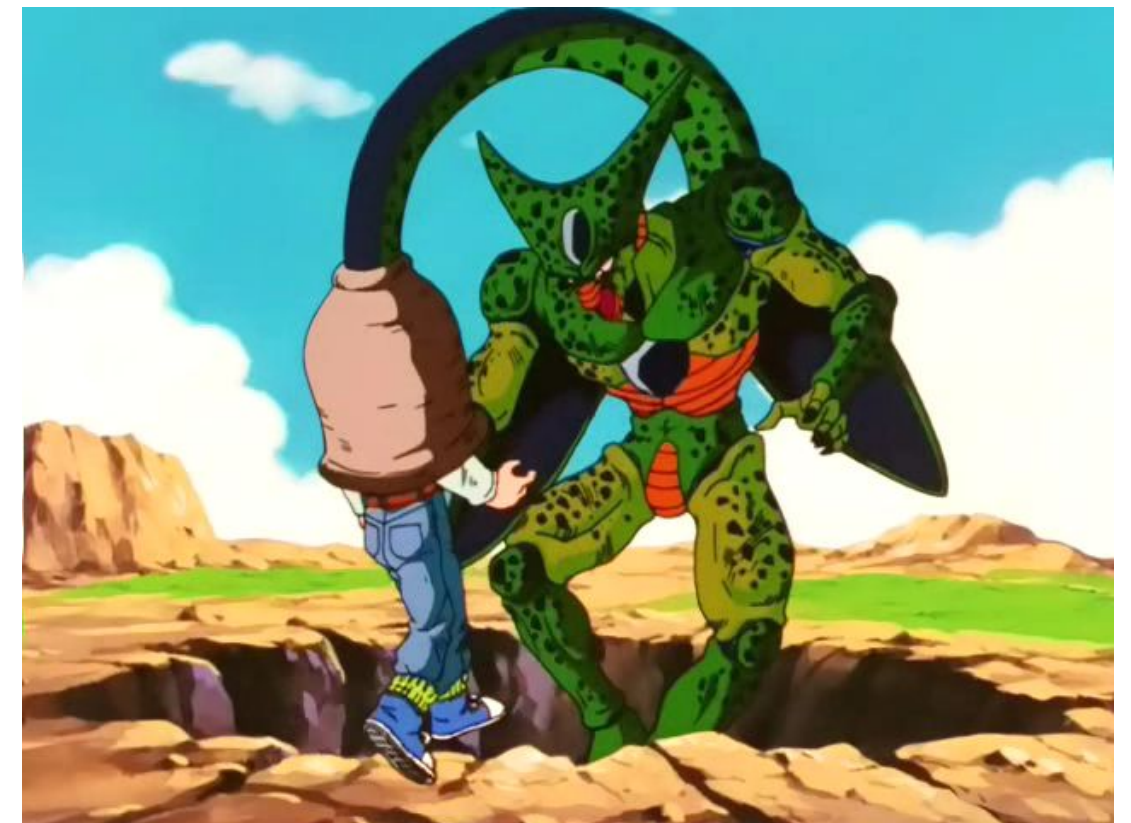
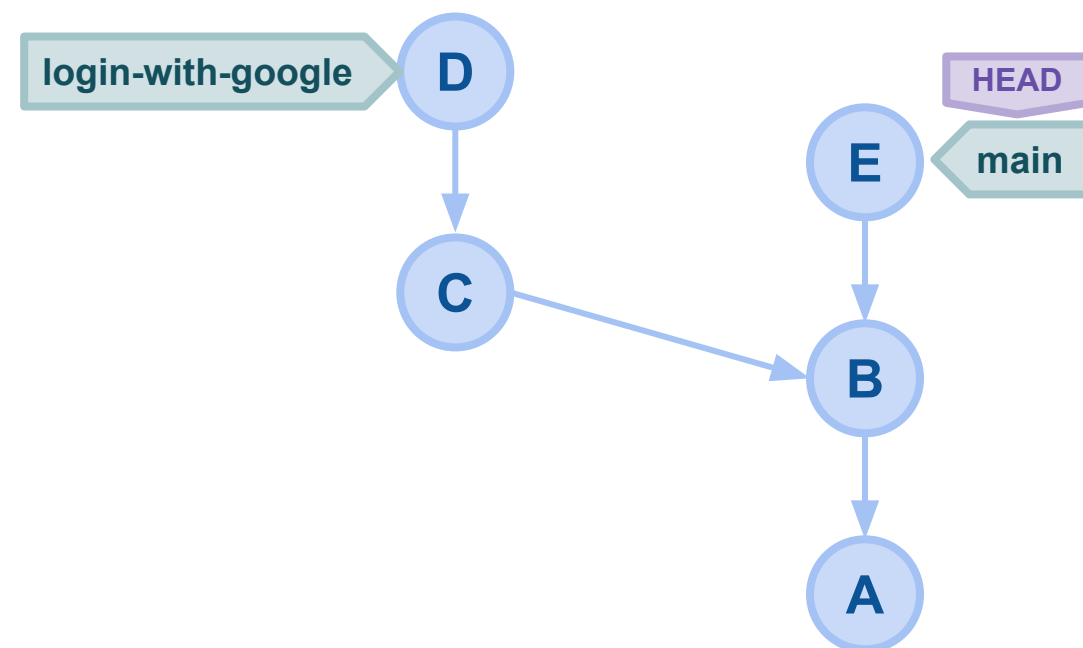
Incorporar el login con Google a la rama main



```
git merge login-with-google
```

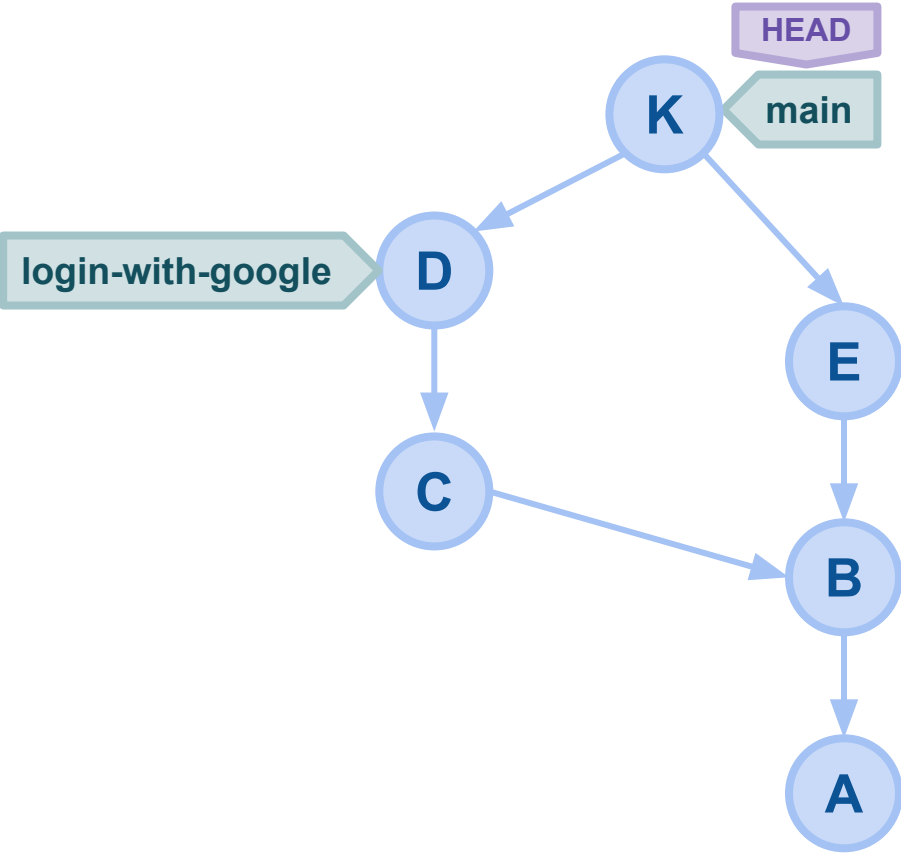


```
git merge login-with-google
```



Regla nemotécnica: HEAD es Célula y absorbe la rama que indicas.





■ Los dos tipos de merge

No fast forward



Fast forward



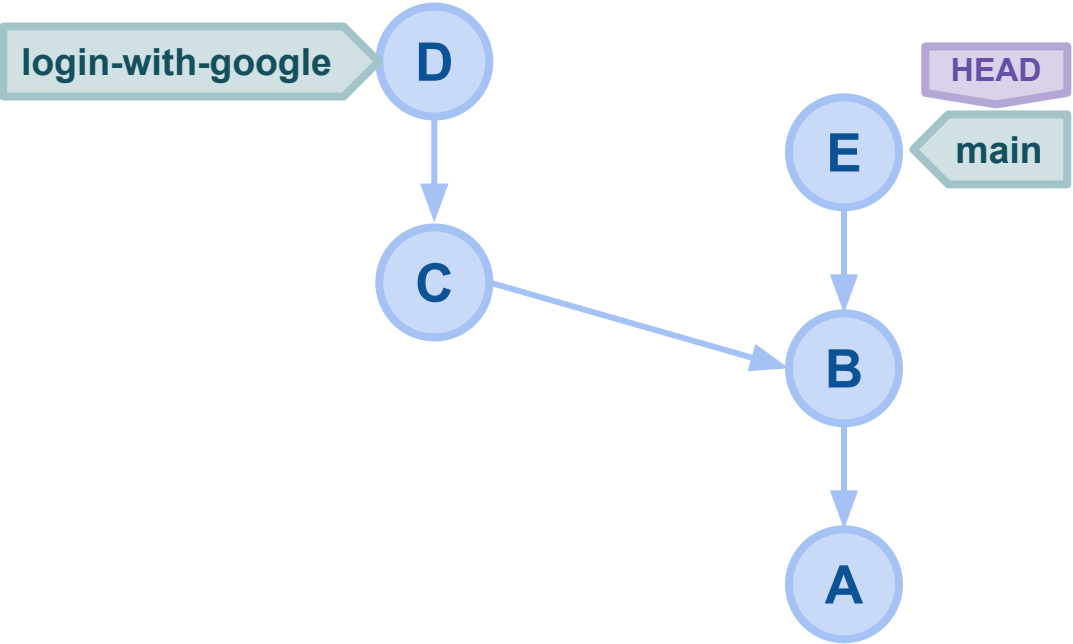
■ Los dos tipos de merge: no fast forward

No fast forward



Git Simulator v.1.3.73

No fast forward



JS

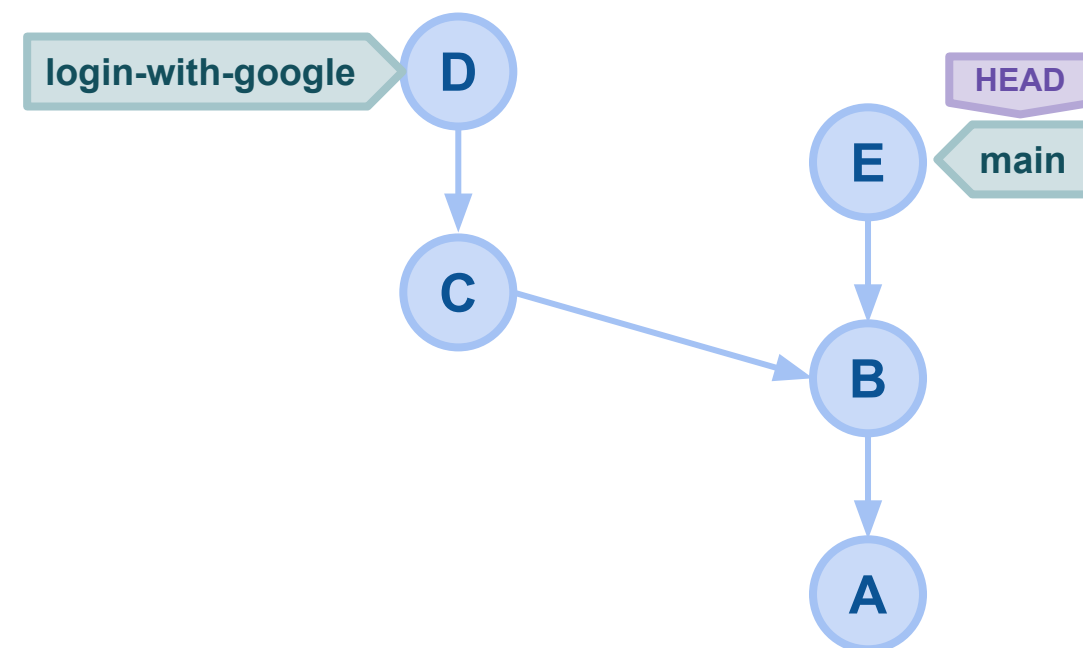
JS

CSS

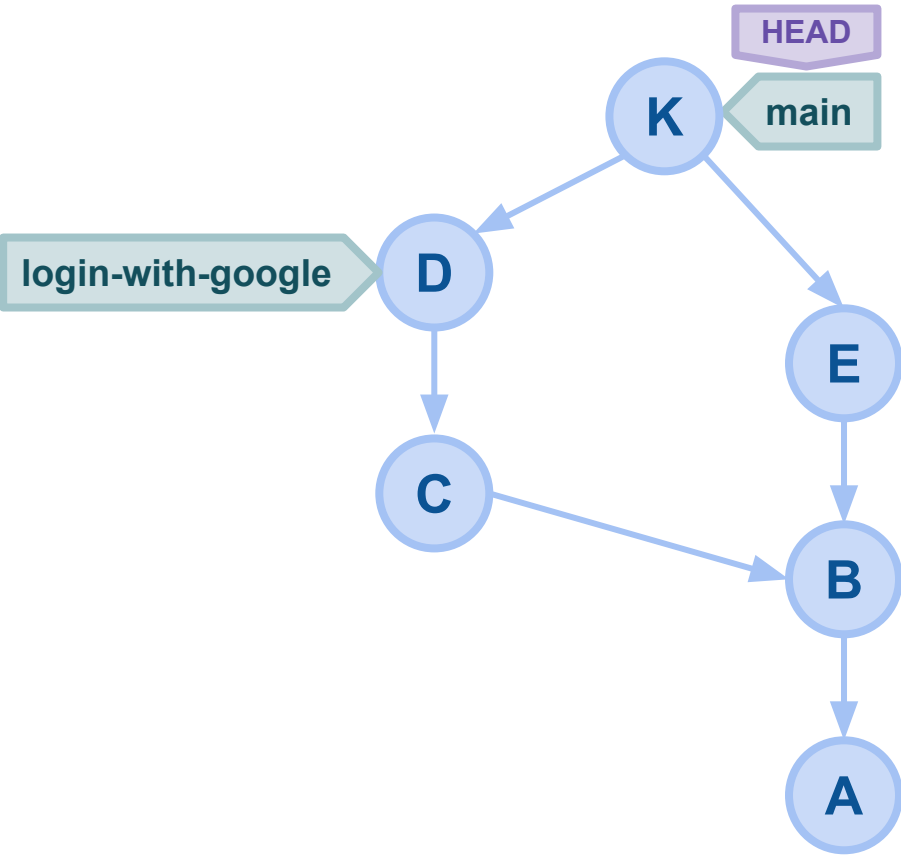
Git Simulator v.1.3.73 🤗

No fast forward

```
git merge login-with-google
```



No fast forward



■ Los dos tipos de merge: fast forward

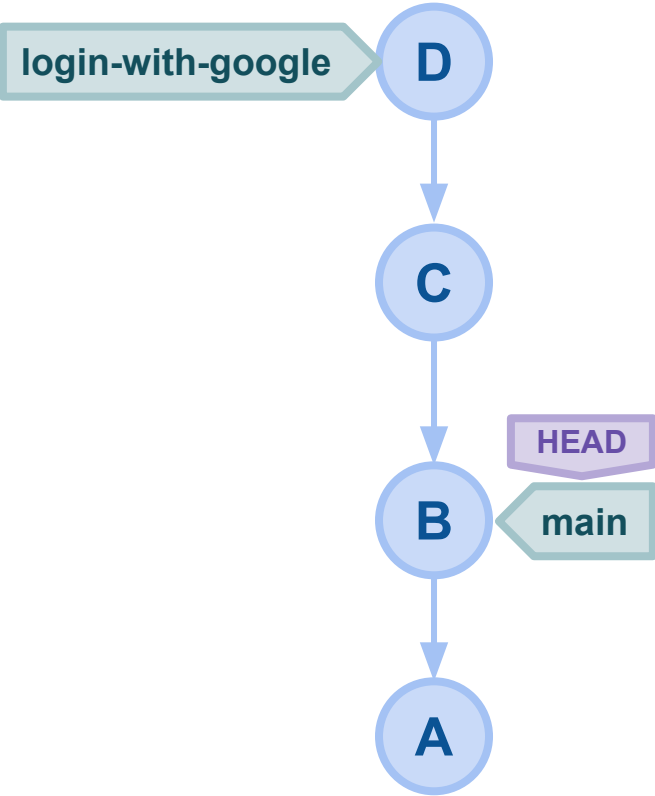
Fast forward



Git Simulator v.1.3.73

🤔

Fast forward

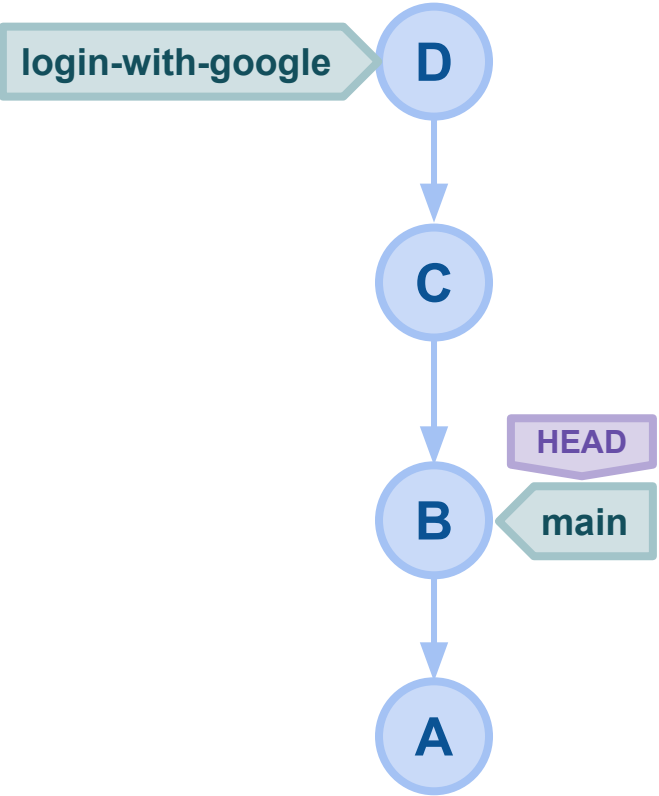


Git Simulator v.1.3.73

🤔

Fast forward

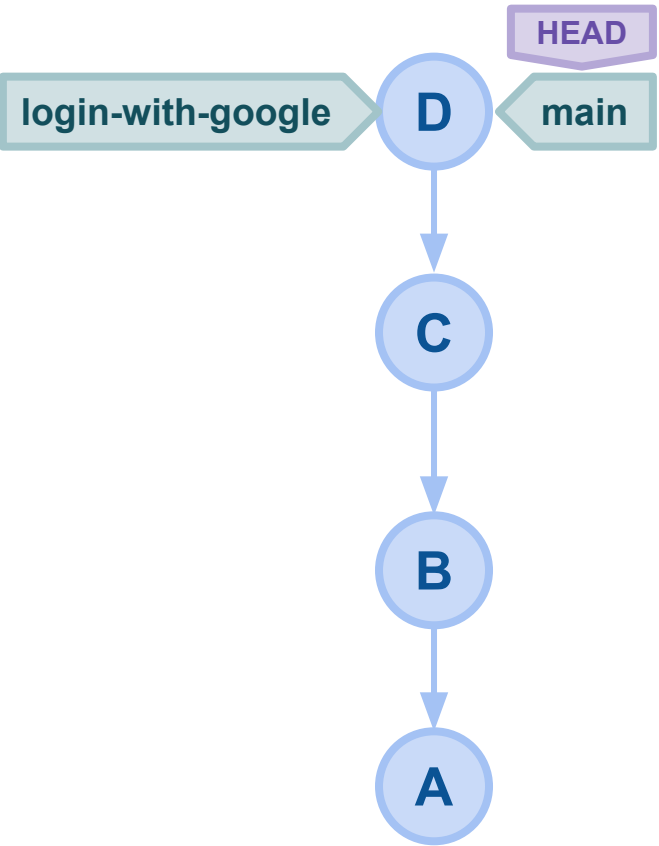
```
git merge login-with-google
```



Git Simulator v.1.3.73

🤔

Fast forward



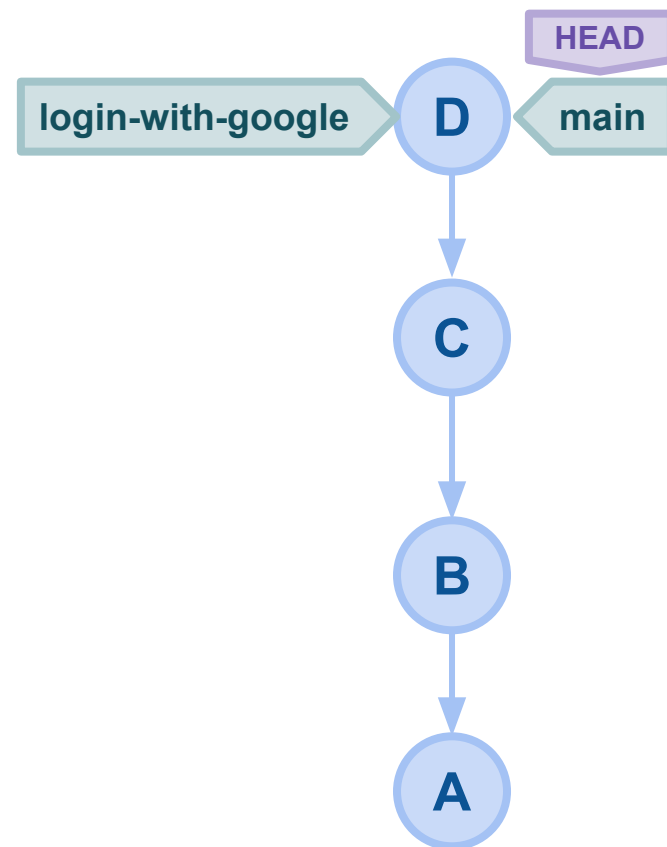
JS

JS

CSS

Fast forward

Fast forward se basa en mover la rama en la que está HEAD hasta el commit de la otra rama



JS

JS

CSS

■ Los dos tipos de merge: fast forward

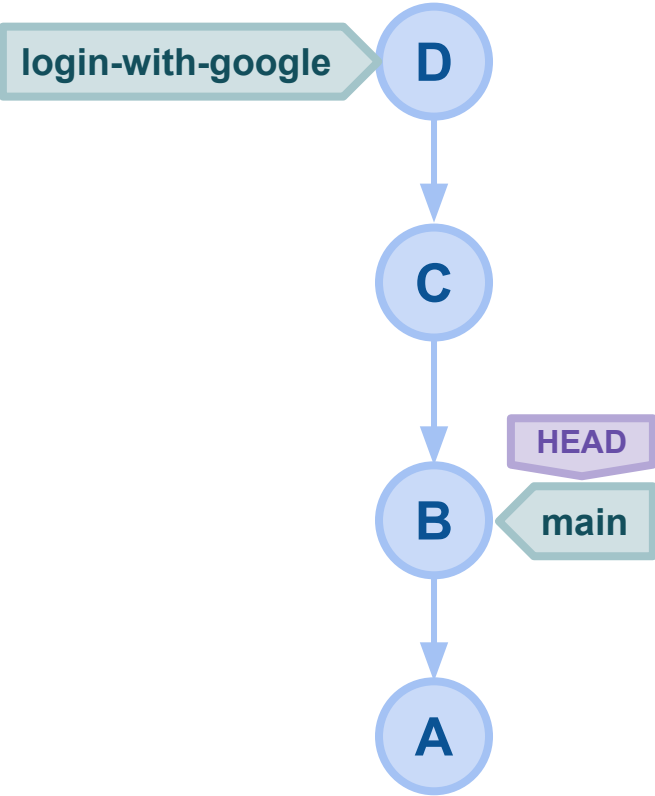
¿Puedo obligar a git a hacer
no fast forward?



Git Simulator v.1.3.73

🤔

Fast forward



JS

JS

CSS

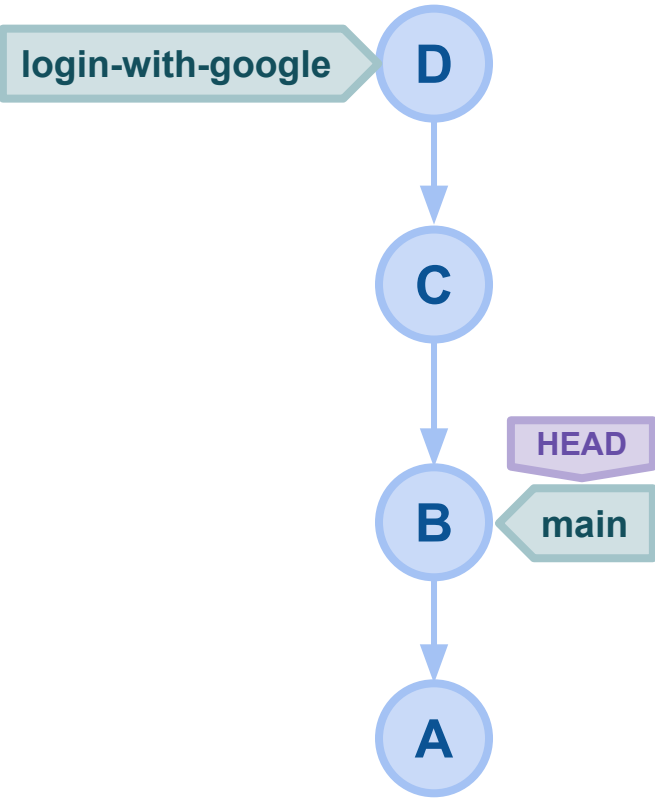


Git Simulator v.1.3.73

🤔

Fast forward

```
git merge --no-ff login-with-google
```



JS

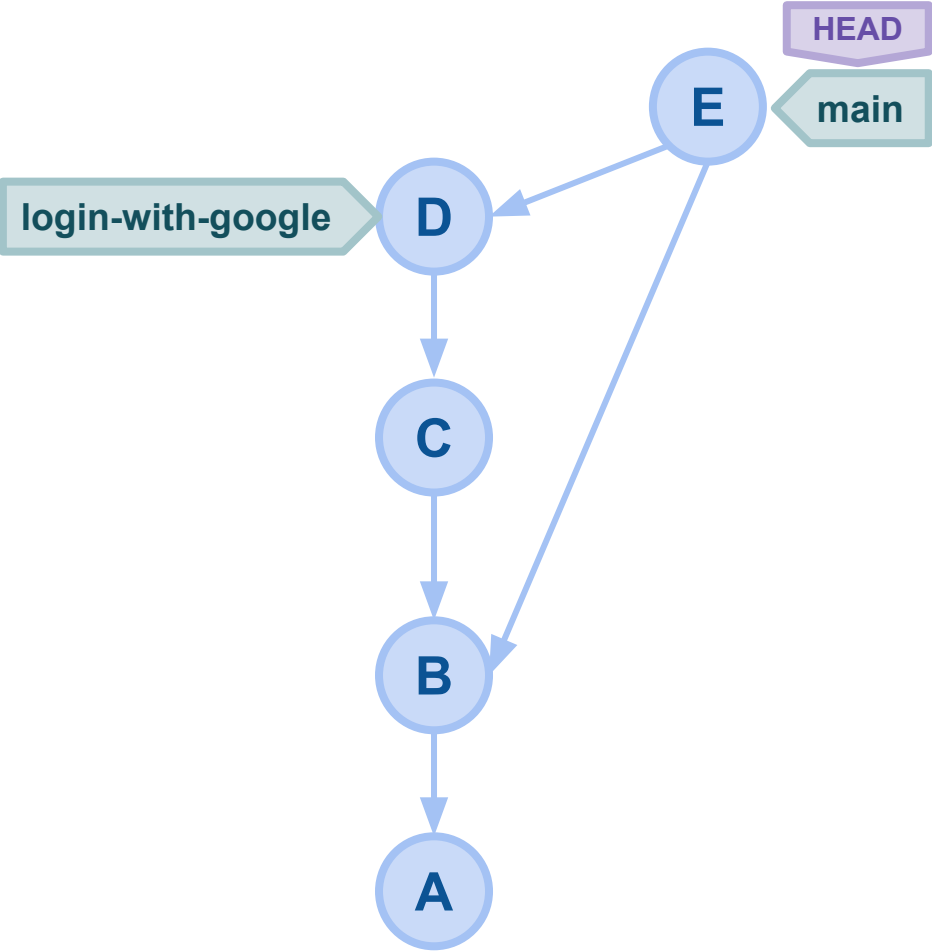
JS

CSS



Git Simulator v.1.3.73 🤔

Fast forward



JS

JS

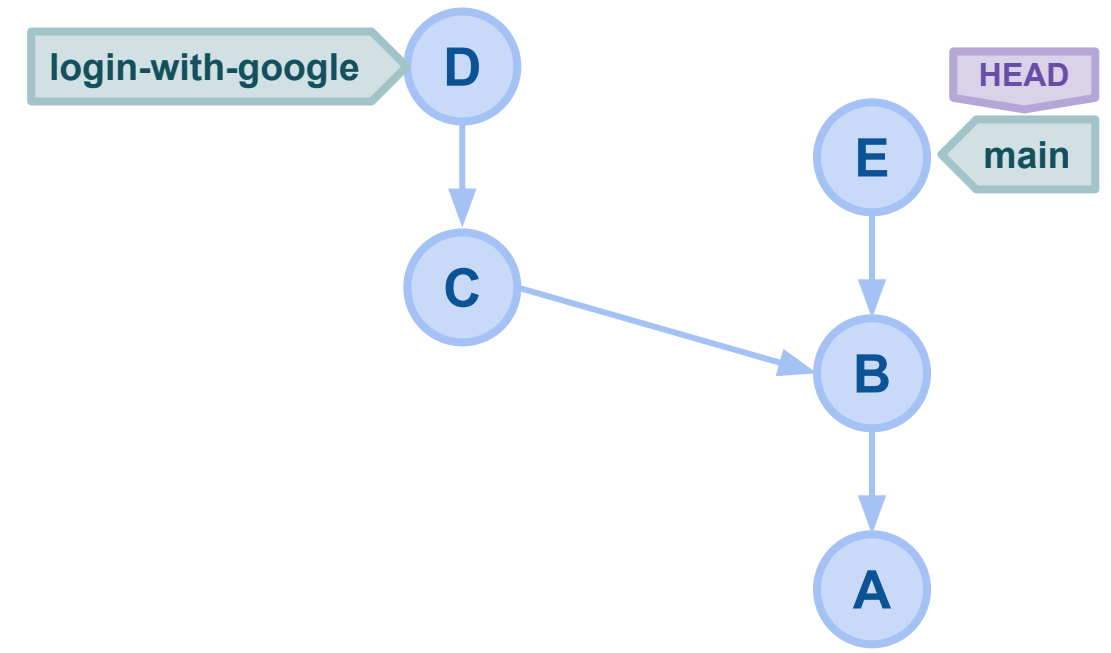
CSS

■ Los dos tipos de merge: fast forward

¿Puedo obligar a git a hacer siempre
fast forward?



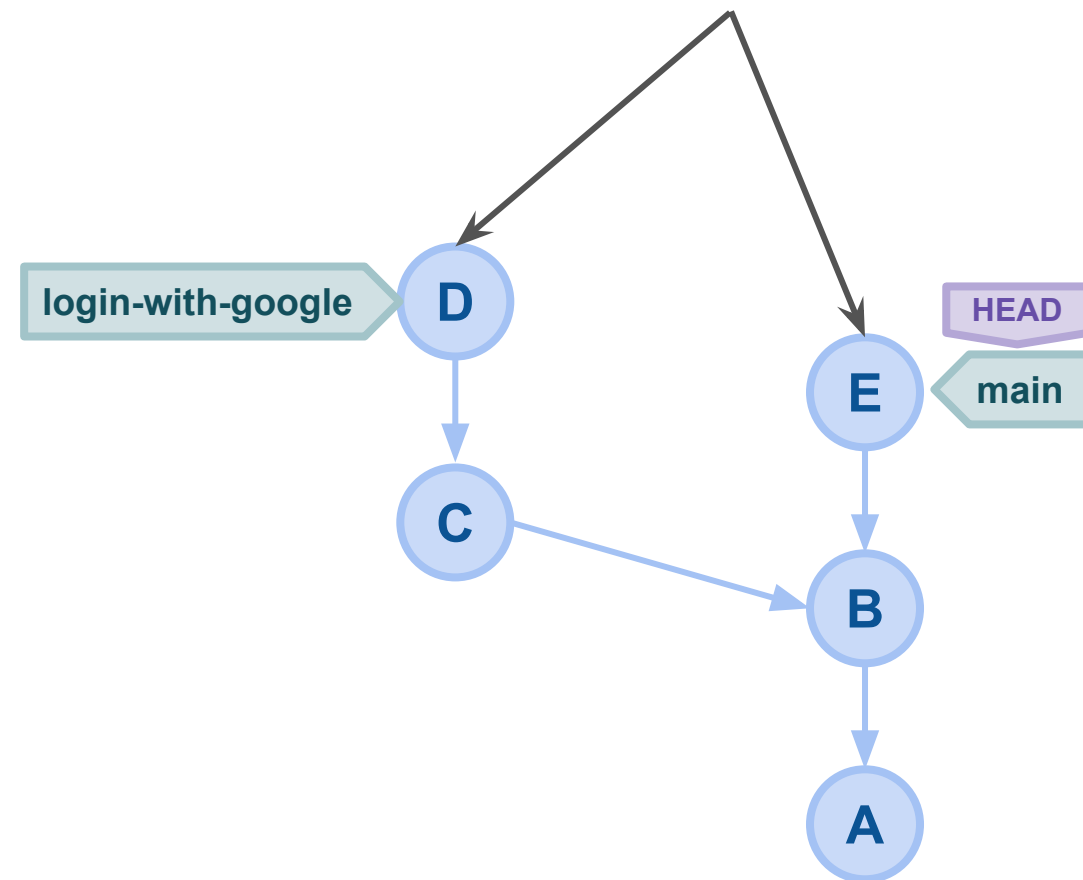
¿Podemos hacer aquí un **fast forward**? ¿Por que?





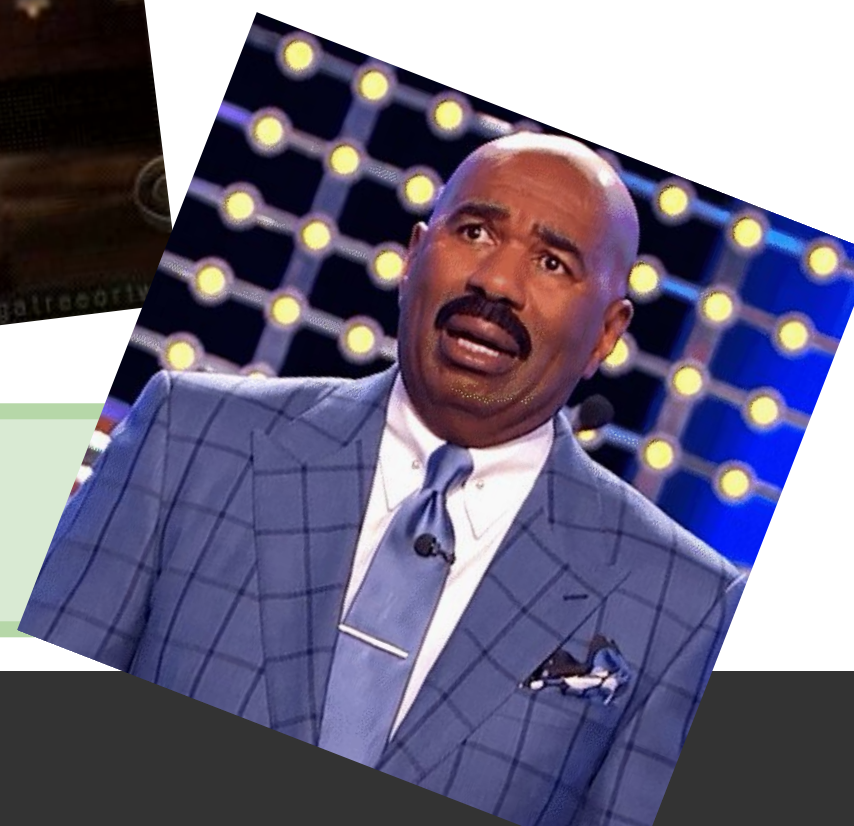
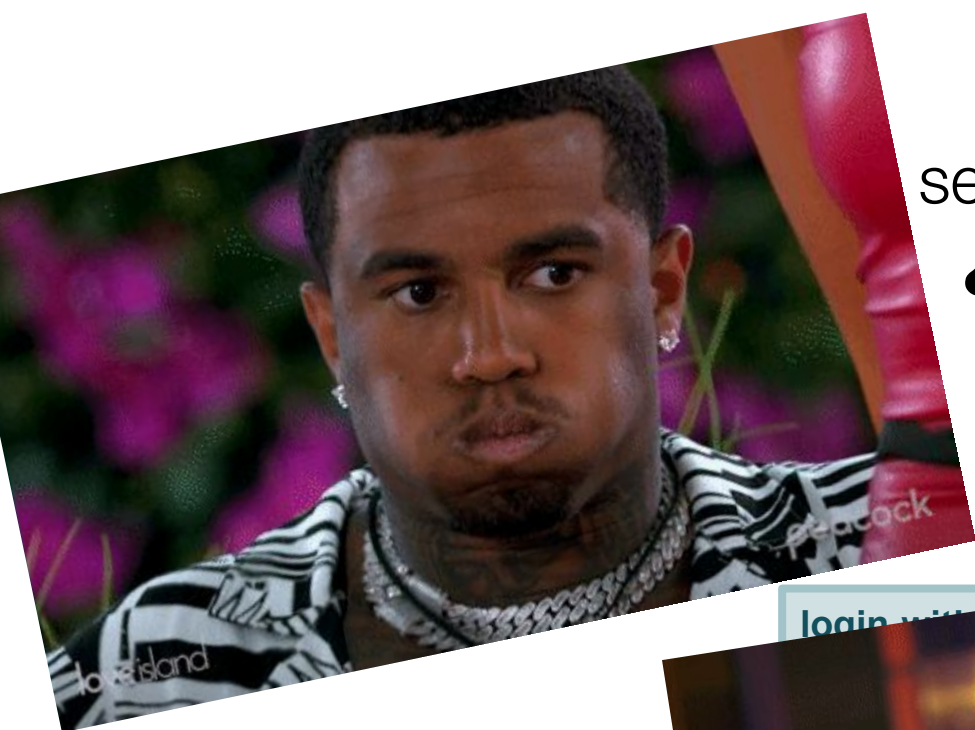
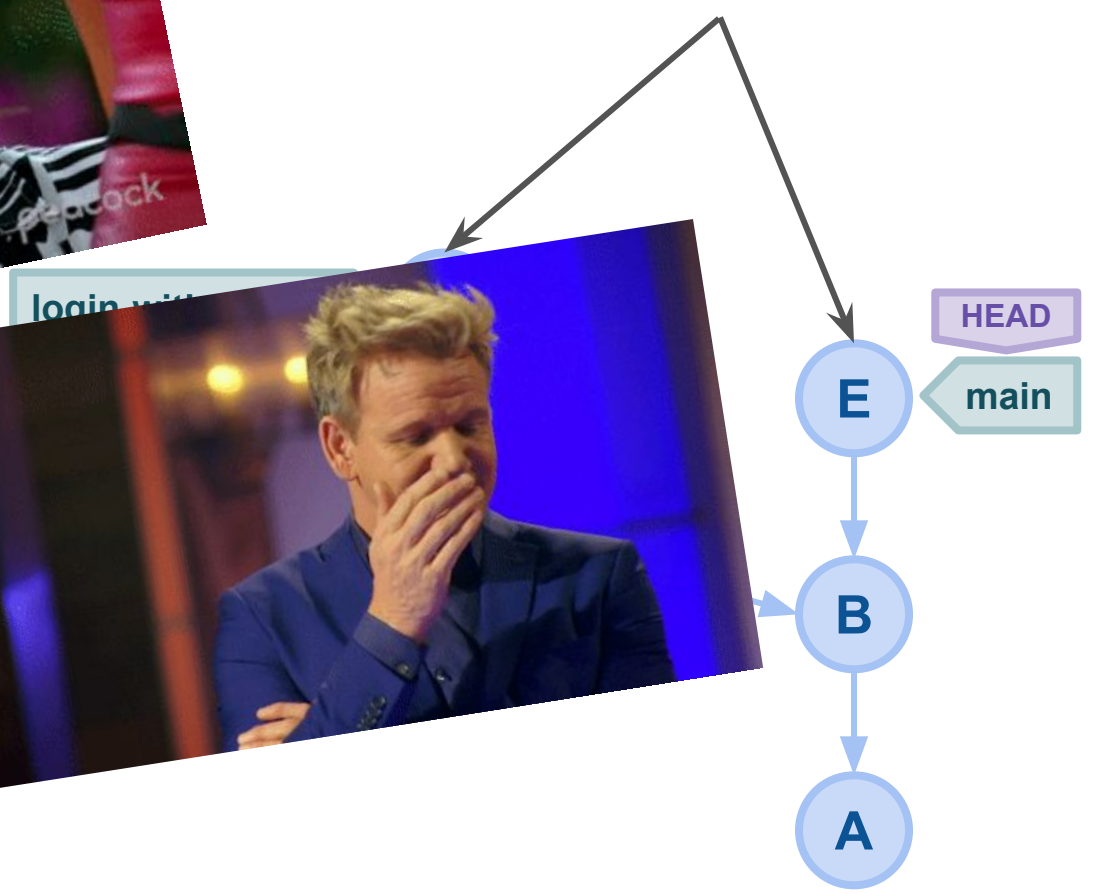
Git Simulator v.1.3.73 🙄

En ambos commits, se ha hecho un cambio en el archivo `index.js` en la línea 5.
¿Qué pasa si hacemos un merge?



Git Simulator v.1.3.73 🙄

se ha hecho un cambio en el archivo `index.js`
¿Qué pasa si hacemos un merge?









NO PROBLEMO



Conflictos

Cuando se monta el pollo



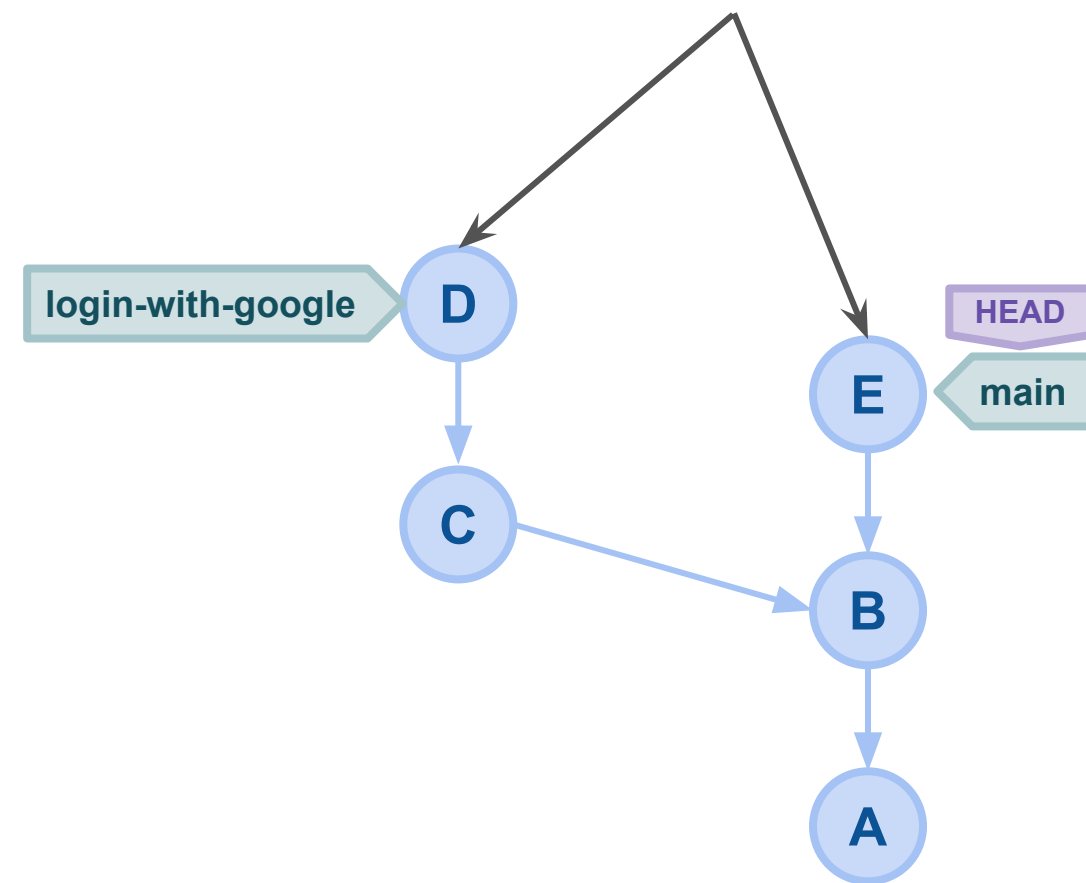
```
$ git merge master
```

```
Auto-merging war.txt  
CONFLICT (content): Merge conflict in war.txt  
Automatic merge failed; fix conflicts and then commit the result.
```



■ ¿Cómo y cuándo se crea un conflicto?

Al hacer `merge` entre dos ramas diferentes (`main` y `login-with-google`) que contienen commits que hacen cambios en uno o varios archivos en las mismas líneas.



■ ¿Cómo se solucionan?

- Un conflicto se produce al hacer un `merge` que no puede ser fast forward
- El conflicto sucede creando el commit que une las ramas. Se queda “a medias”
- Tenemos dos opciones:
 - Abortar el merge con `git merge --abort`
 - Solucionar los conflictos:
 - i. Editando los archivos en conflicto (con nuestro editor de texto favorito)
 - ii. Usar `git add <file>` en cada archivo que haya resuelto el conflicto
 - iii. Usar `git commit` para finalizar el commit que une las dos ramas

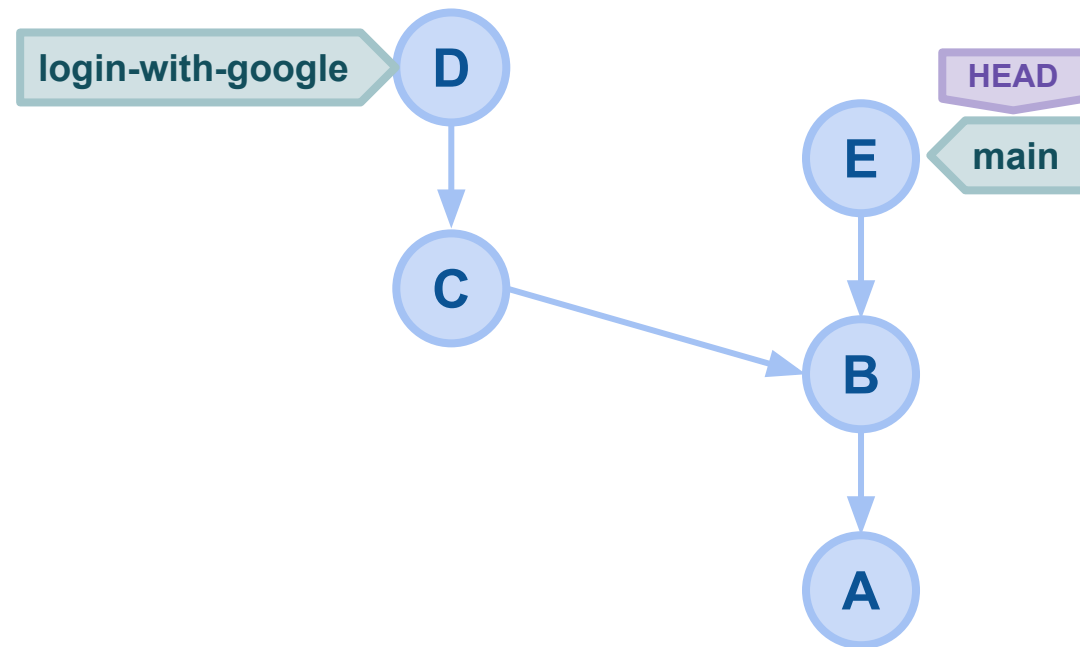


¿Qué pinta tienen los archivos en conflicto?





■ ¿Qué pinta tienen los archivos en conflicto?



index.js en main

```
Roses are red
violets are #0000ff
all my base
Are belong to you.
```

index.js en conflicto

```
Roses are red
<<<<<<<< HEAD
violets are #0000ff
all my base
=====
Violets are blue,
All of my base
>>>>>>> BranchToMerge
Are belong to you.
```

index.js en login-with-google

```
Roses are red
Violets are blue,
All of my base
Are belong to you.
```



Repos remotos

Júntate con los demás frikis en GitHub



■ ¿Qué es GitHub?

- Plataforma para alojar proyectos con Git y trabajo colaborativo
- Infinidad de [funcionalidades](#)
- Es **GRATUITA** (modelo freemium)
- Con páginas de [404 not found](#) y [500 server](#) error muy chulas
- El Open Source creció gracias a sus [forks](#) y [pull requests](#)
- Su logo es un "octogato" (Octocat)



■ git clone

```
# Clona el repositorio alojado en <repo-url> en una carpeta con su mismo nombre  
git clone <repo-url>
```

```
# Clona el repositorio alojado en <repo-url> en la carpeta <folder>  
git clone <repo-url> <folder>
```



¿Y si he creado el repo en mi ordenador?



■ git remote

```
# Ver los repositorios remoto definidos en el repo que estoy trabajando
```

```
git remote
```

```
# Añadir el repositorio remoto con nombre <repo-name> y url <repo-url> en el repo que estoy trabajando
```

```
git remote add <repo-url> <repo-name>
```

```
# Elimina el repositorio remoto con nombre <repo-name> del repo que estoy trabajando
```

```
git remote remove <repo-name>
```



■ ¿Cómo funcionan los repos remotos?



■ ¿Cómo funcionan los repos remotos?



■ ¿Cómo funcionan los repos remotos?

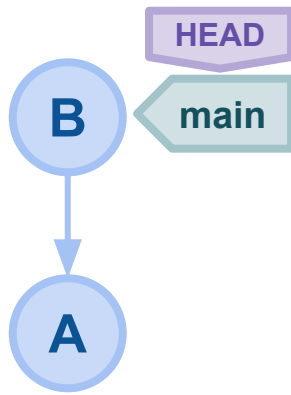
**¿Sabes trabajar con ramas?
Entonces también con repos remotos.**

Los repos remotos usan “ramas remotas”, que funcionan igual que las ramas normales.

Esas “ramas remotas”, son las ramas en las que trabajan otros miembros del equipo (que son los que te generan conflictos).







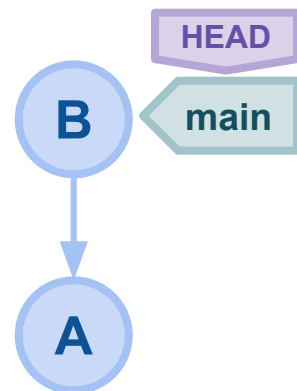
remote

local



Git Simulator v.1.4.0 🤪

```
git clone user@remote/repo.git
```

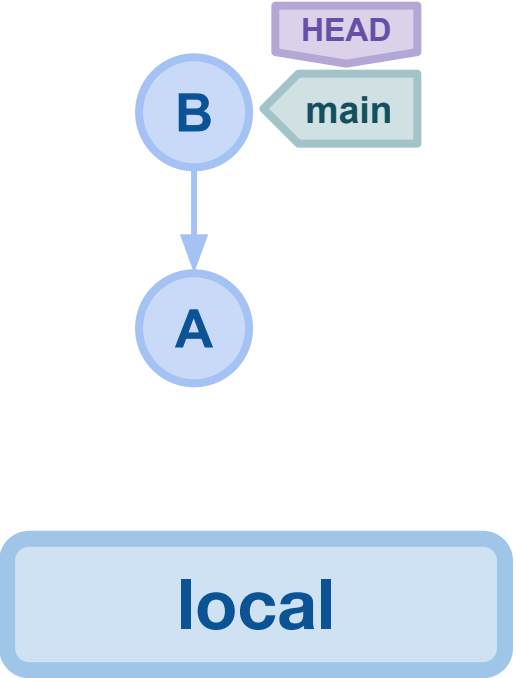
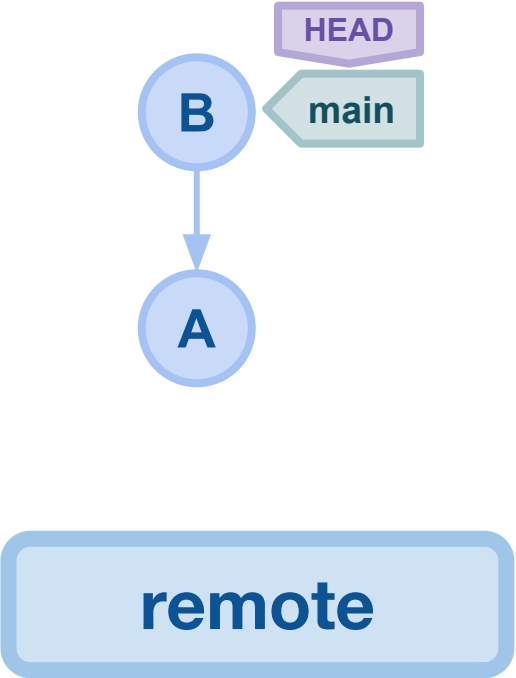


remote

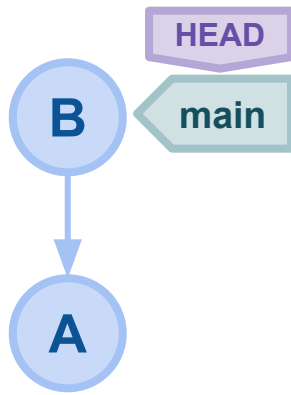
local



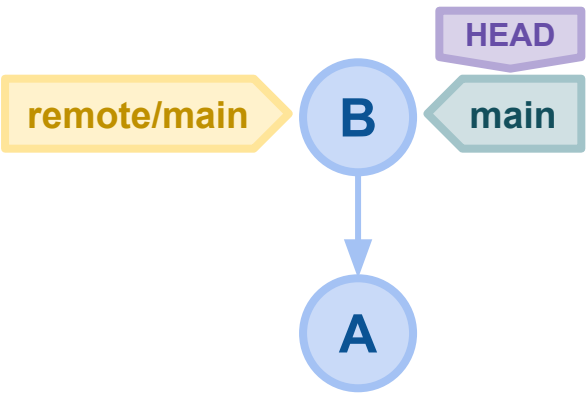
Git Simulator v.1.4.0 🤪



Git Simulator v.1.4.0 🤪



remote

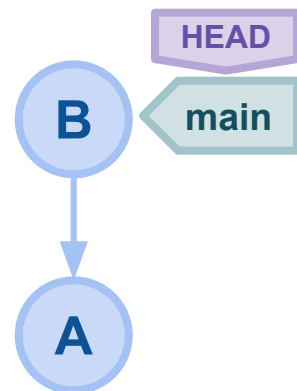


local

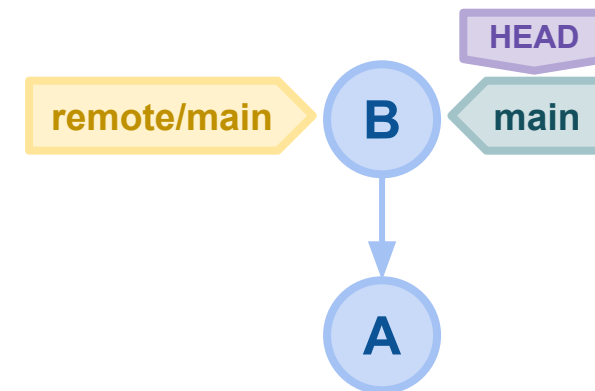


Git Simulator v.1.4.0 🤪

```
git commit -m "Añado commit C"
```



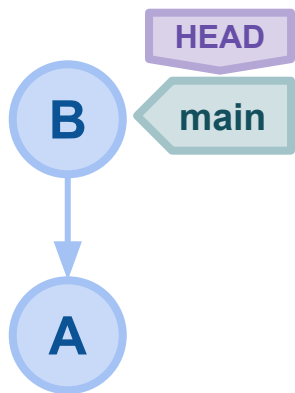
remote



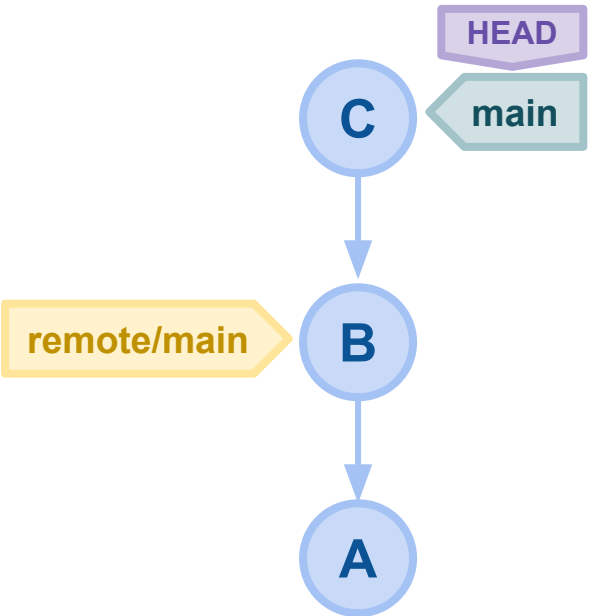
local



Git Simulator v.1.4.0 🤪



remote

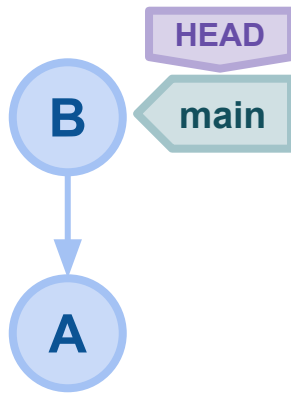


local

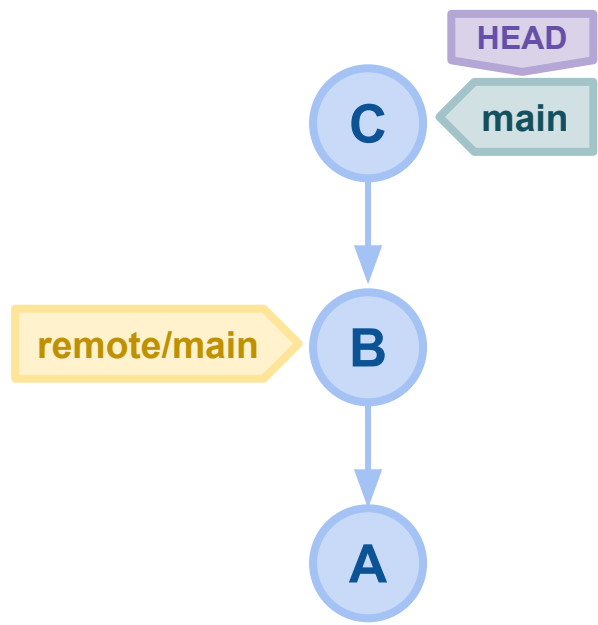


Git Simulator v.1.4.0 🤪

git push



remote

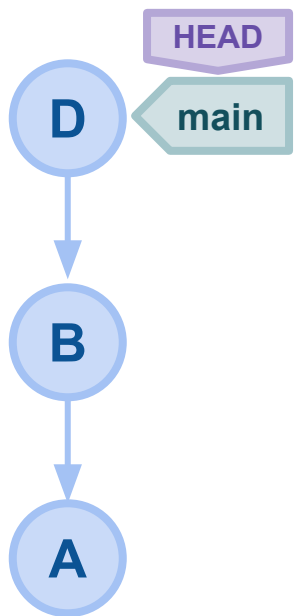


local

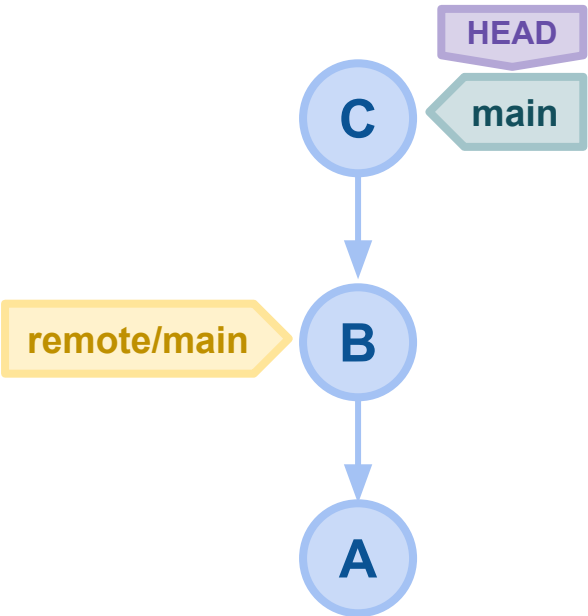




Git Simulator v.1.4.0 🤪



remote

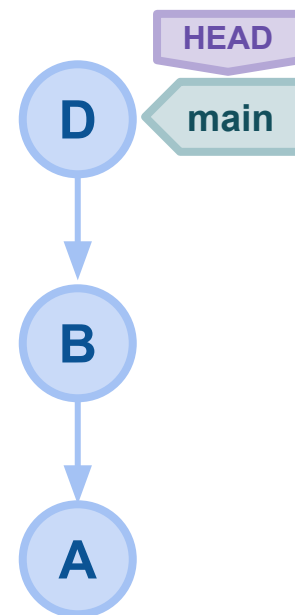


local

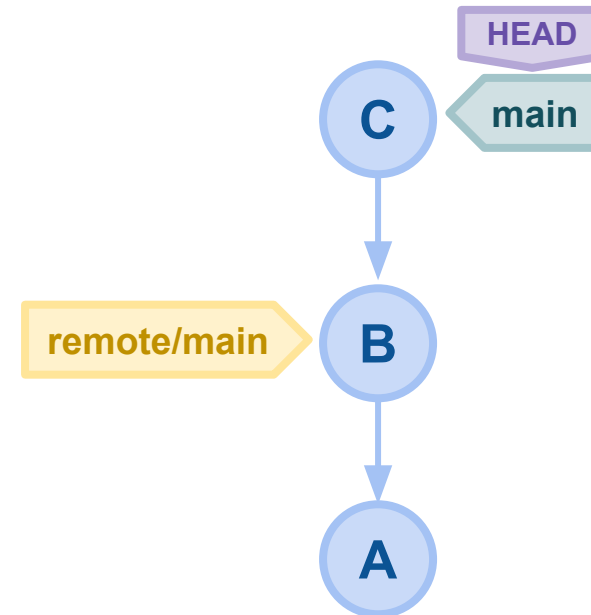


Git Simulator v.1.4.0 🤪

```
git push remote main
```



remote

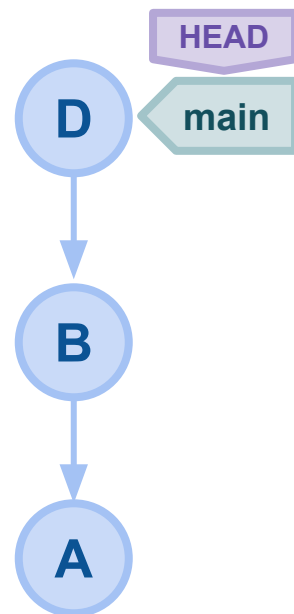


local

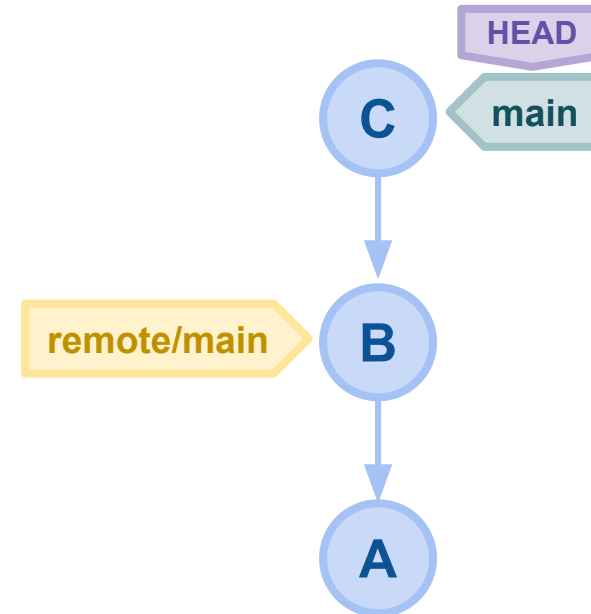


Git Simulator v.1.4.0 🤪

```
! [rejected] main -> main  
error: failed to push some refs to remote
```

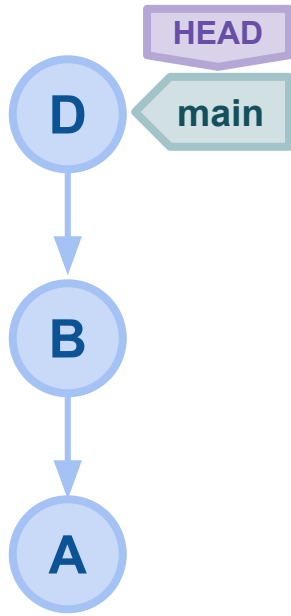


remote

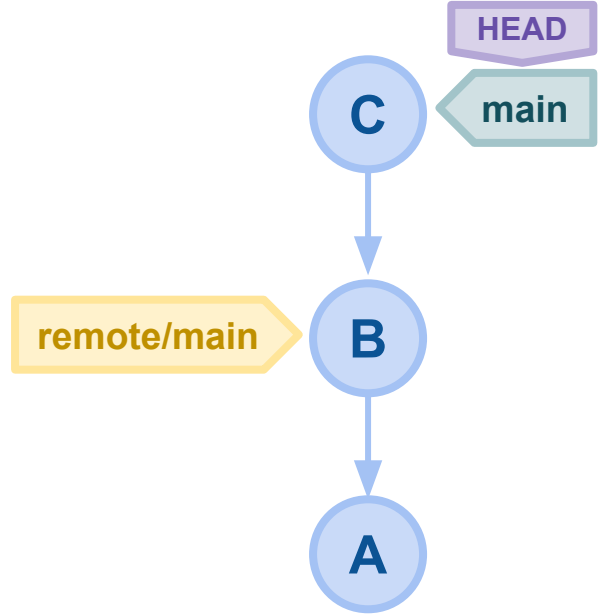


local





remote

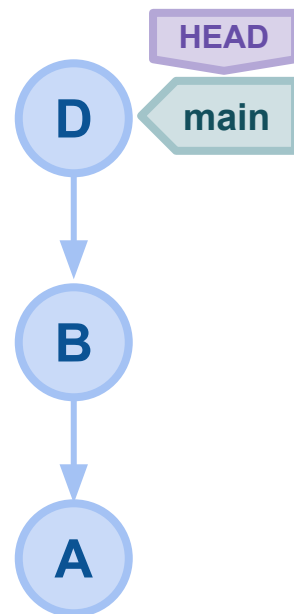


local

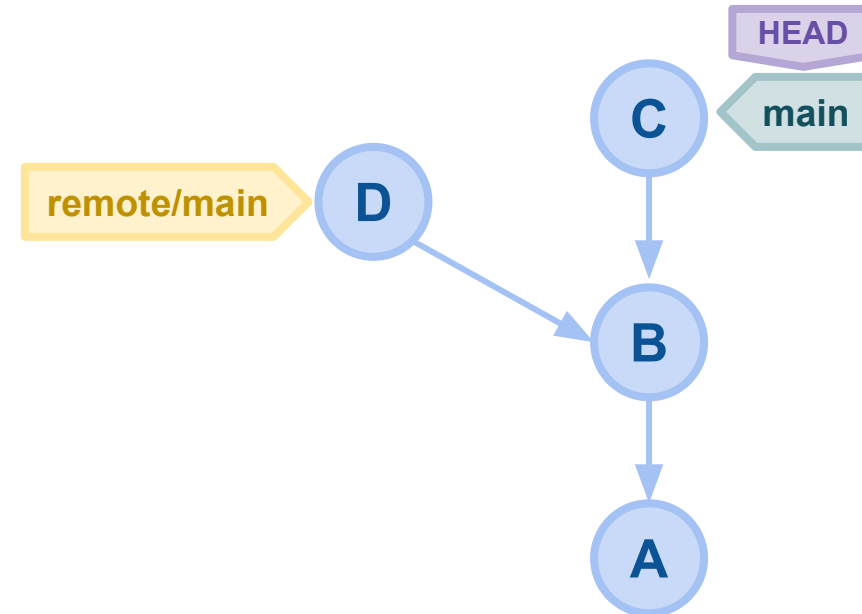


Git Simulator v.1.4.0 🤪

```
git fetch
```



remote

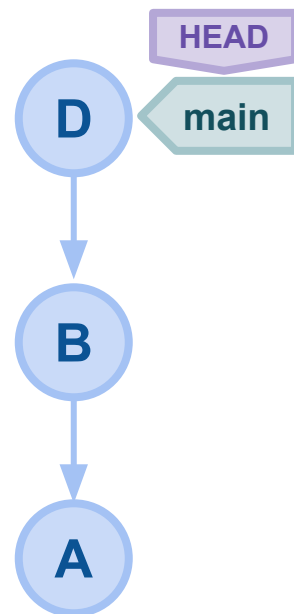


local

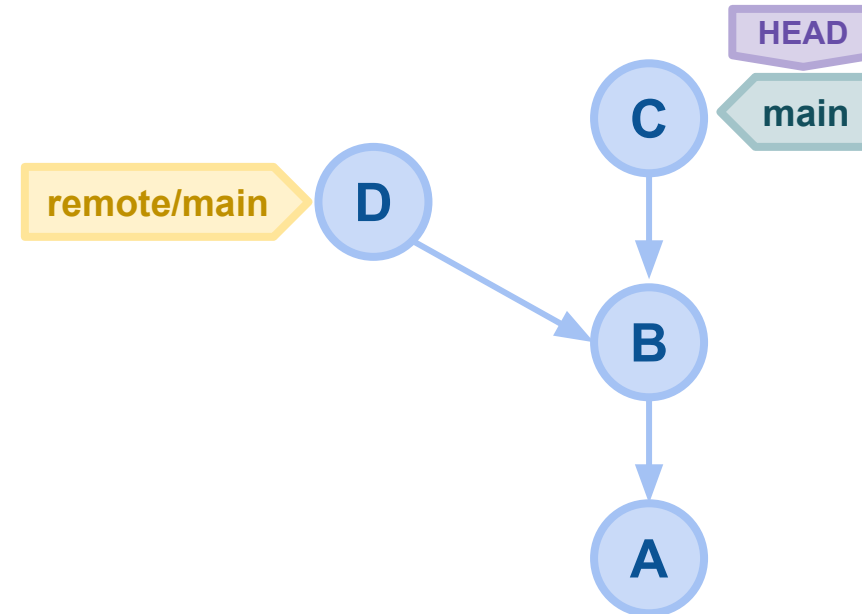


Git Simulator v.1.4.0 🤪

```
git merge remote/main
```



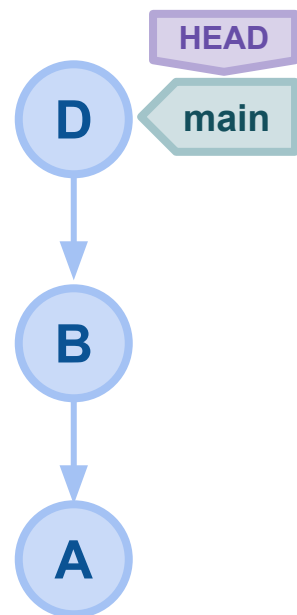
remote



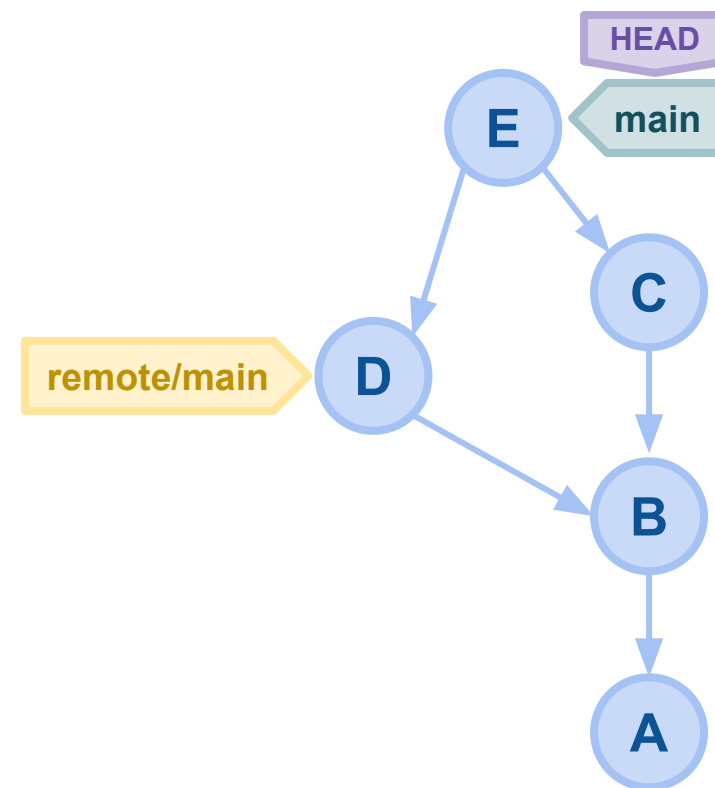
local



Git Simulator v.1.4.0 🤪



remote

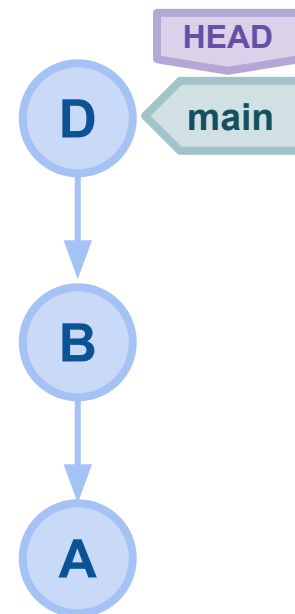


local

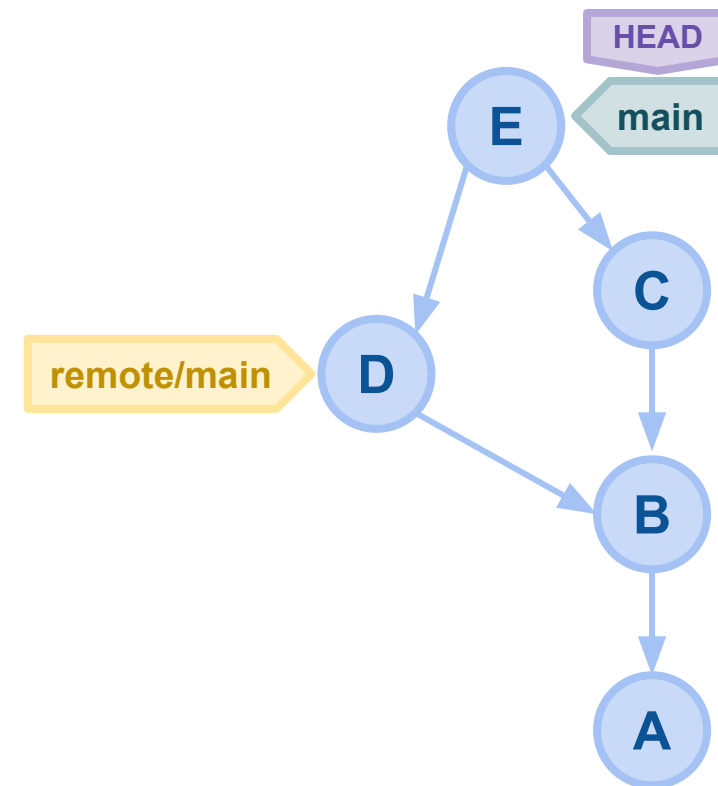


Git Simulator v.1.4.0 🤪

```
git push
```



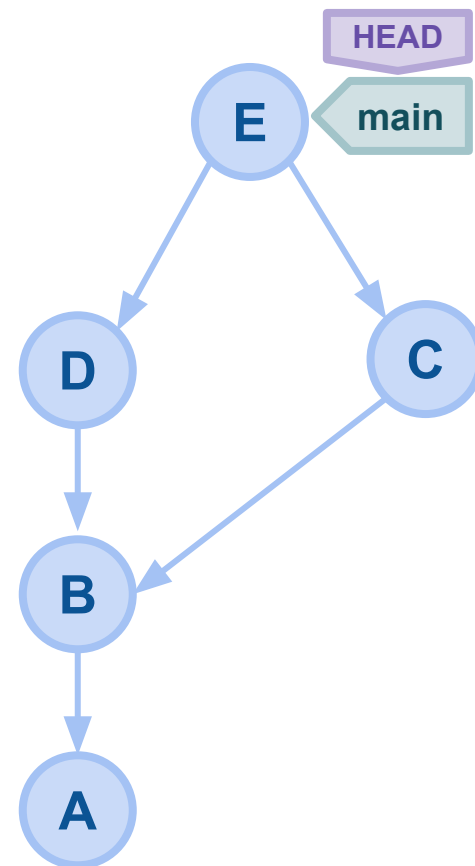
remote



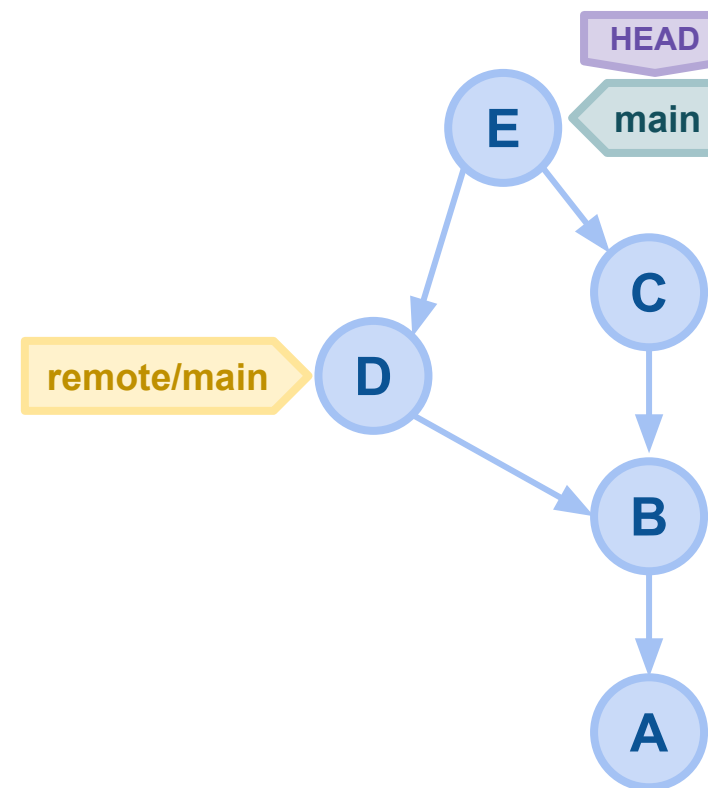
local



Git Simulator v.1.4.0 🤪



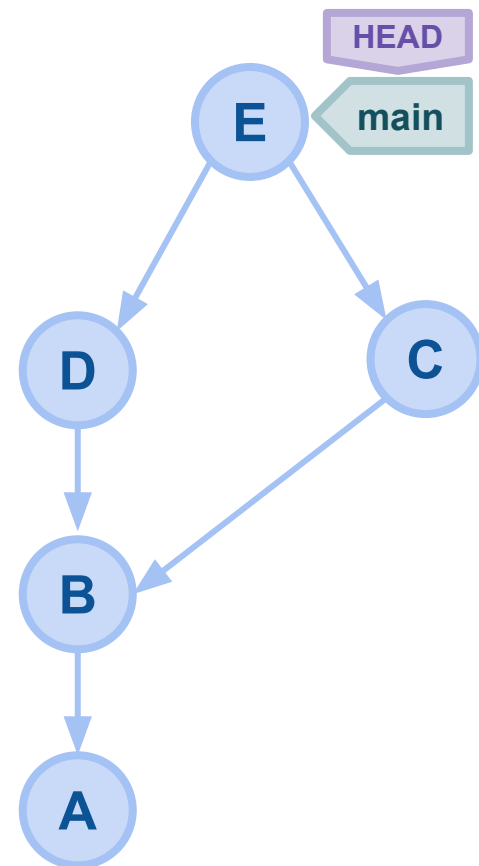
remote



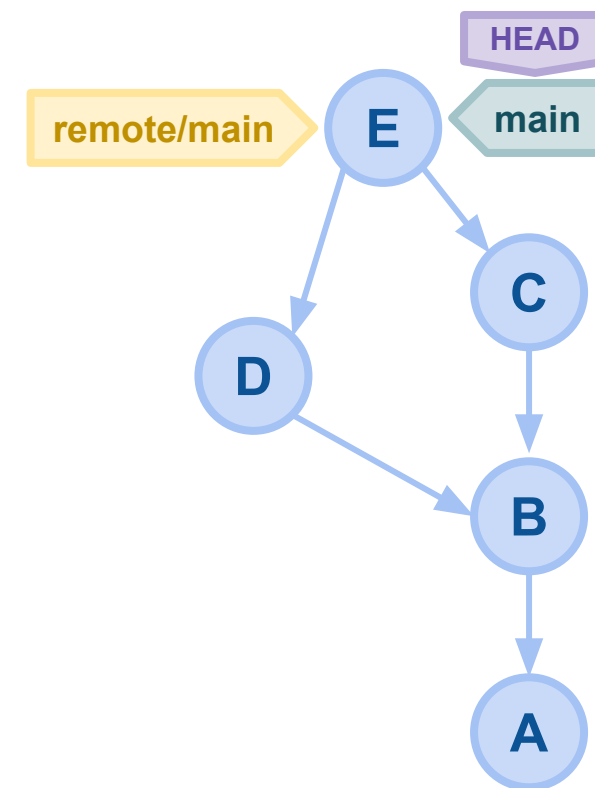
local



Git Simulator v.1.4.0 🤪



remote



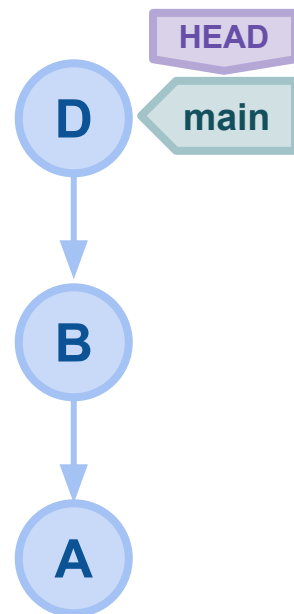
local



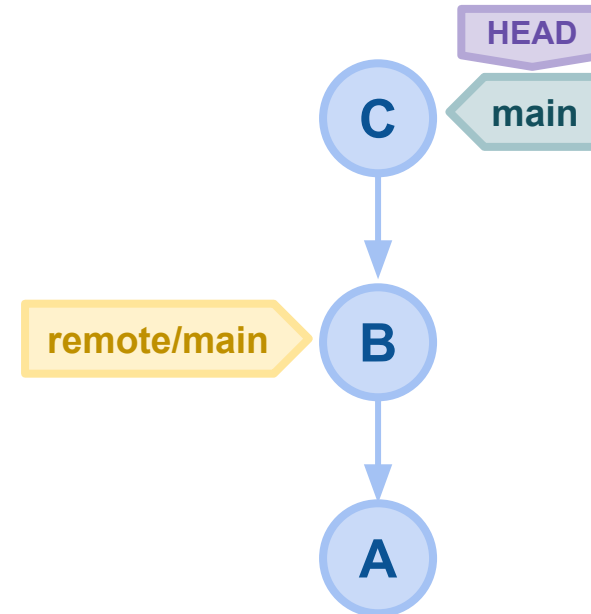


Git Simulator v.1.4.0 🤪

```
! [rejected] main -> main  
error: failed to push some refs to remote
```

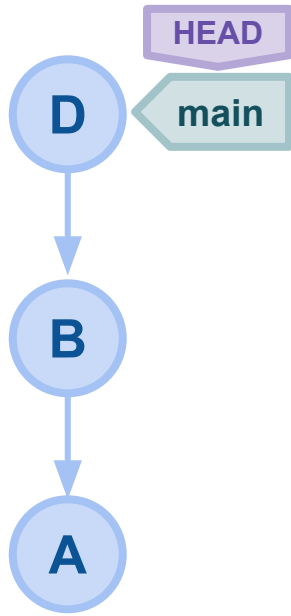


remote

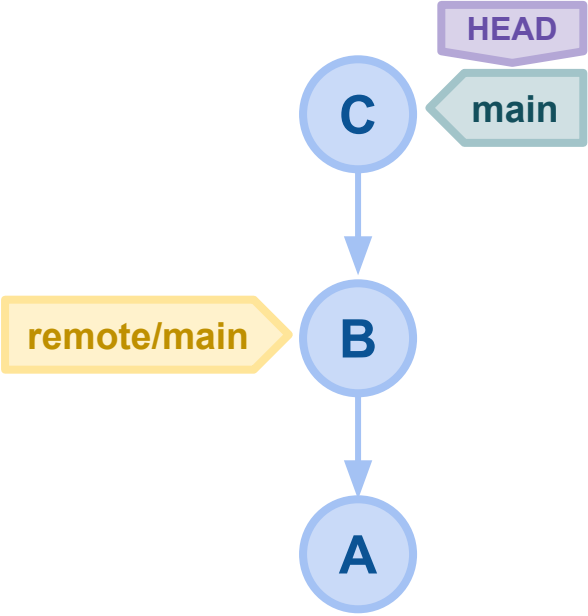


local





remote

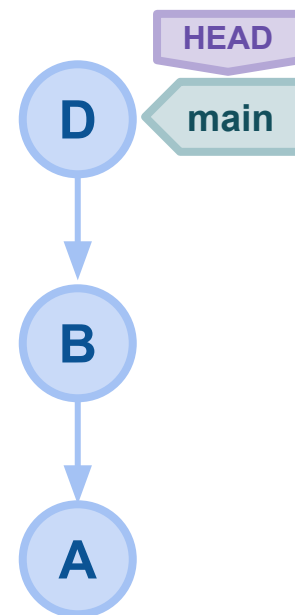


local

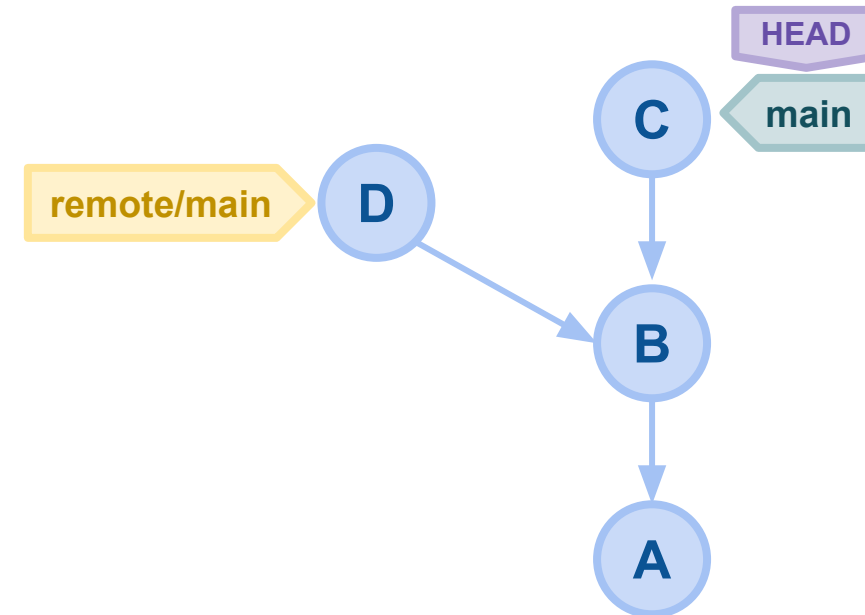


Git Simulator v.1.4.0 🤪

```
git pull
```



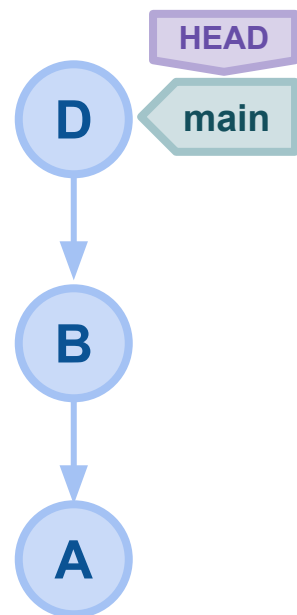
remote



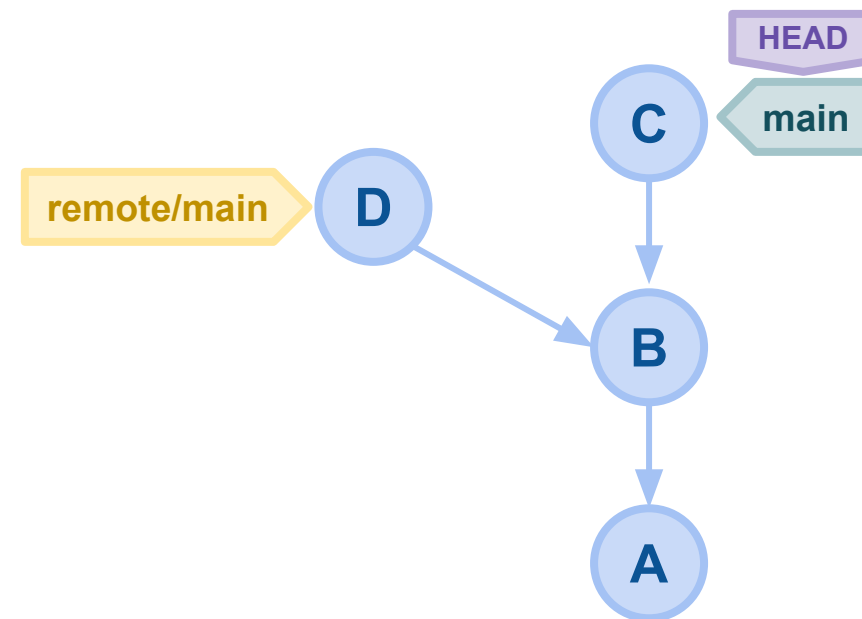
local



Git Simulator v.1.4.0 🤪



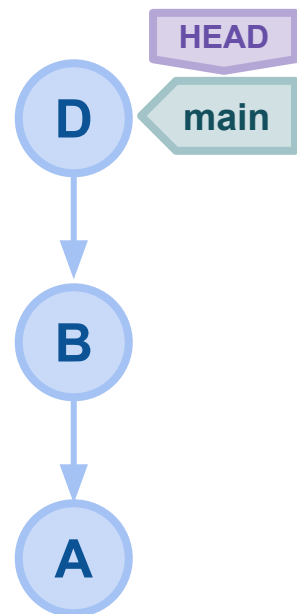
remote



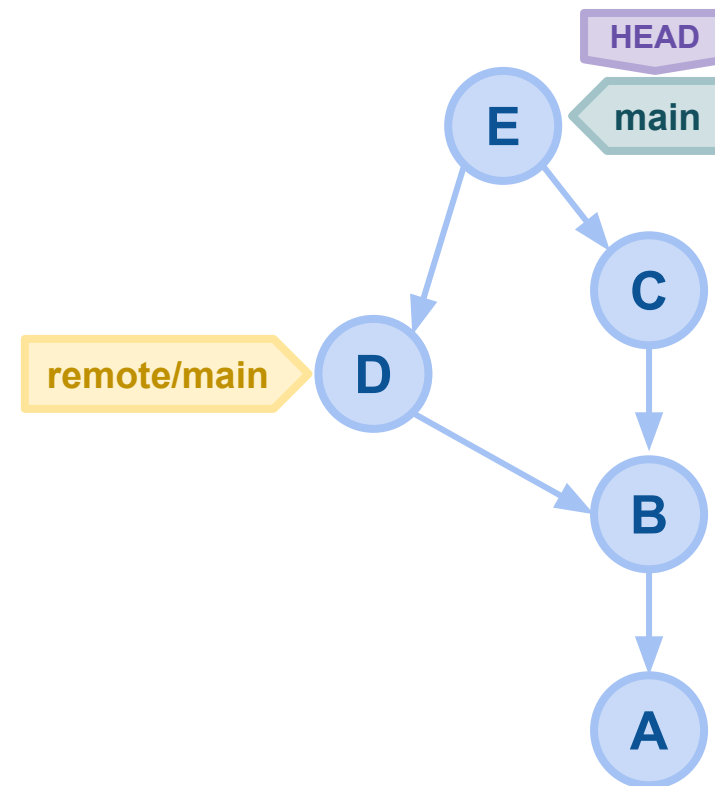
local



Git Simulator v.1.4.0 🤪



remote



local



¿Qué ha hecho `git pull`?



`git fetch + git merge = git pull`



■ Repos remotos

```
# Sube los cambios de la rama en la que está HEAD a su remote por defecto  
git push
```

```
# Sube los cambios del <branch/tag> al remote <remote>  
git push <remote> <branch/tag>
```

```
# Sube los cambios y establece el remote por defecto  
git push -u <remote> <branch/tag>
```

```
# Fuerza a reestablecer la rama remota al mismo commit que en la local  
git push -f
```

```
# Descarga cambios de todas las ramas y tags del repo remoto  
git fetch
```

```
# Hace un fetch y mergea la rama en la que estoy con su rama remota  
git pull
```





■ Forks, pull requests & code review

- Un **fork** es una copia de un repositorio en tu cuenta
 - Por tanto, como está en tu cuenta, puedes modificar
- Un **pull request** es una solicitud para *mergear* entre dos ramas
 - Las dos ramas pueden ser del mismo repo o de repos distintos
 - En el pull request, se pueden hacer code reviews



■ ¿Cómo colaborar con un proyecto Open Source?

1. Hacemos un fork del proyecto
2. Como el fork está en nuestra cuenta, podemos hacer cambios
3. Una vez hechos los cambios, podemos hacer un pull request para solicitar que incorporen el código del fork al del repo principal (si lo consideran oportuno)





■ ¿Por dónde sigo?

- Aprende más Git jugando: [Learn Git Branching](#), [Oh My Git!](#)
- Como el fork está en nuestra cuenta, podemos hacer cambios
- Aprende algunos flujos de trabajo, como [Gitflow](#) o [Gitlab Flow](#)
- Aprende [GitHub Actions](#) para automatizar cosas cuando hagas push
- En unos meses, aprende `git rebase`



GRACIAS
www.keepcoding.io

