Adatbányászat a Gyakorlatban

1. Előadás: Verziókezelés

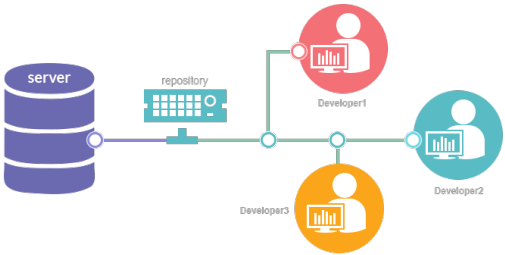
Kuknyó Dániel
Budapesti Gazdasági Egyetem

2024/25
1.félév

Verziókezelés alapjai

Miért van szükség verziókezelésre?

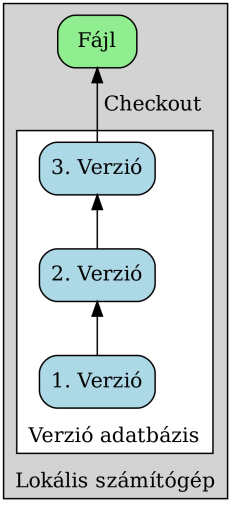
- A program változásainak követése
- A munka biztonságos elmentése
- Kollaboráció több fejlesztő között
- Programkód párhuzamos szerkesztése
- Feladatok szétosztása és követése



Lokális verziókezelők

A legegyszerűbb verziókezelés, ha a fejlesztő kézzel átmásol egy mappába fájlokat. Ezek lehetnek idő bélyegzett mappák is, ha okos a fejlesztő.

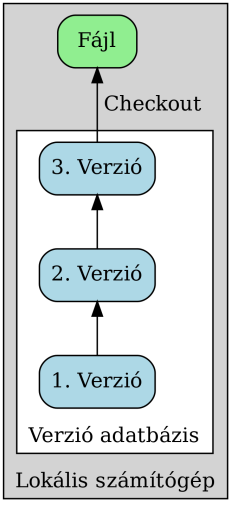
Ez a megoldás nagyon egyszerű, viszont fogékony a hibákra, mert sok a manuális munka. Ezenkívül sok benne a redundáns adat is, mert a nem változtatott adatot is el kell tárolni. Ezért hozták létre a lokális verziókezelőket.



Lokális verziókezelők

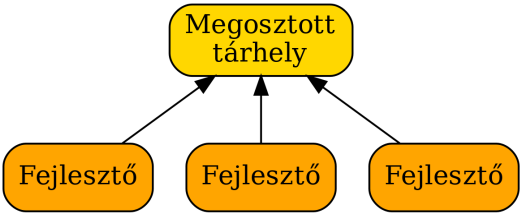
Lokális verziókezelő

A lokális verziókezelők a teljes fájlok helyett csak a változtatásokat tárolják el egy külön erre kifejlesztett adatbázisban lokálisan, a számítógépen. Egy gyakori ilyen szoftver volt az RCS.



Centralizált verziókezelők

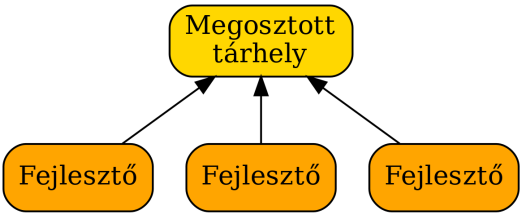
A következő jelentős probléma, amivel az emberek találkoznak, az az, hogy együtt kell működniük fejlesztőkkel más rendszereken. Ennek a problémának a kezelésére Központosított Verziókezelő Rendszerek (CVCS-ek) lettek kifejlesztve.



Centralizált verziókezelők

Centralizált verziókezelő

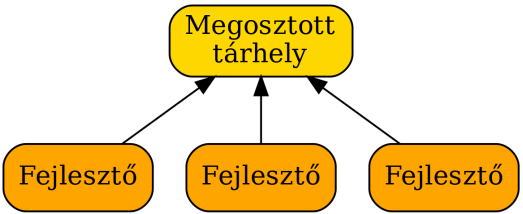
Ezek a rendszerek (például CVS, Subversion és Perforce) egyetlen szerverrel rendelkeznek, amely tartalmazza az összes verziózott fájlt, és számos klienst, akik a fájlokat ebből a központi helyről töltik be.



Centralizált verziókezelők

Előnyei:

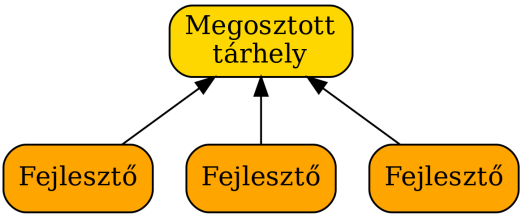
- Mindenki bizonyos mértékben tudja, hogy a projekt többi résztvevője mit csinál.
- Az adminisztrátorok részletes ellenőrzést gyakorolhatnak arról, hogy ki mit tehet meg, és sokkal könnyebb egy CVCS-t adminisztrálni, mint helyi adatbázisokkal foglalkozni minden kliens esetében.



Centralizált verziókezelők

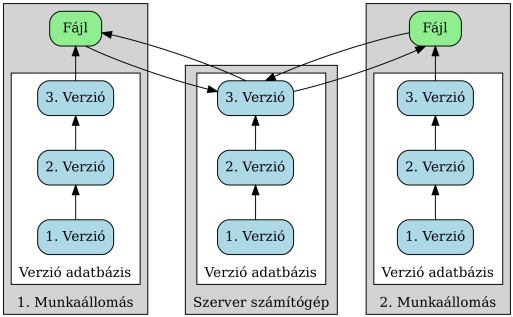
Hátrányai:

- Az egyetlen központi szerver hibapontot jelent, ahol akár egy órás leállás is lehetetlenné teszi a közös munkát és verziózási változtatások mentését.
- Adatvesztés veszélye, ha a központi adatbázis merevlemeze meghibásodik és nem rendelkezünk megfelelő biztonsági mentésekkel.



Elosztott verziókezelők

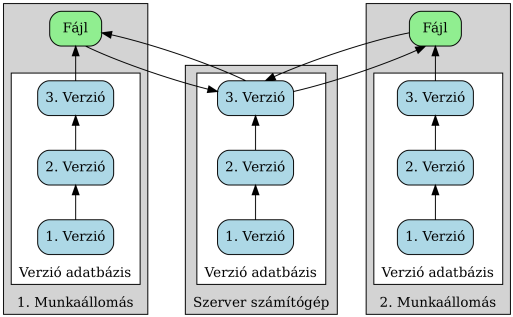
Ebben a helyzetben lépnek képbe az elosztott verziókezelő rendszerek (DVCS-ek). Sok ilyen rendszer nagyon jól kezeli a több távoli tárolóval való együttműködést, így lehetőség van különböző emberekkel egyidejűleg egyazon projekt keretein belül együttműködni.



Elosztott verziókezelők

Elosztott verziókezelő

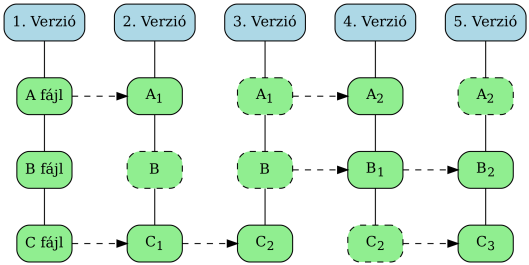
A kliensek nem csak a fájlok legfrissebb pillanatképét töltik le, hanem teljes egészében tükrözik a tárhelyet, beleértve annak teljes előzményeit is. Ezért minden klón valójában egy teljes adatbiztonsági másolat.



A Git verziókezelő

A Git úgy gondolkodik az adatokról, mint egy fájlrendszer pillanatképei. Segítségével minden alkalommal, amikor commit történik, (azaz elmentődik a projekt) készít egy képet arról, hogy az összes fájl hogyan néz ki abban a pillanatban, és eltárol egy hivatkozást erre a pillanatfelvételre.

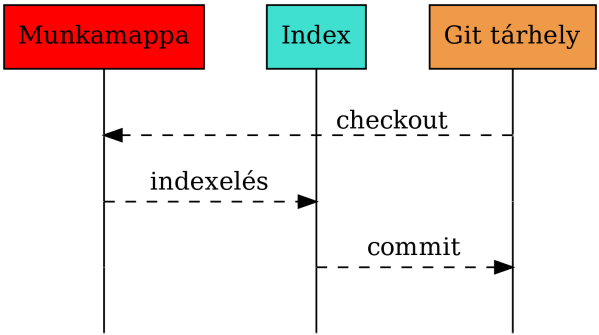
A hatékonyság érdekében, ha a fájlok nem változtak, a Git nem tárolja újra a fájlt, csak egy hivatkozást az előző, azonos fájlra, amit már tárolt.



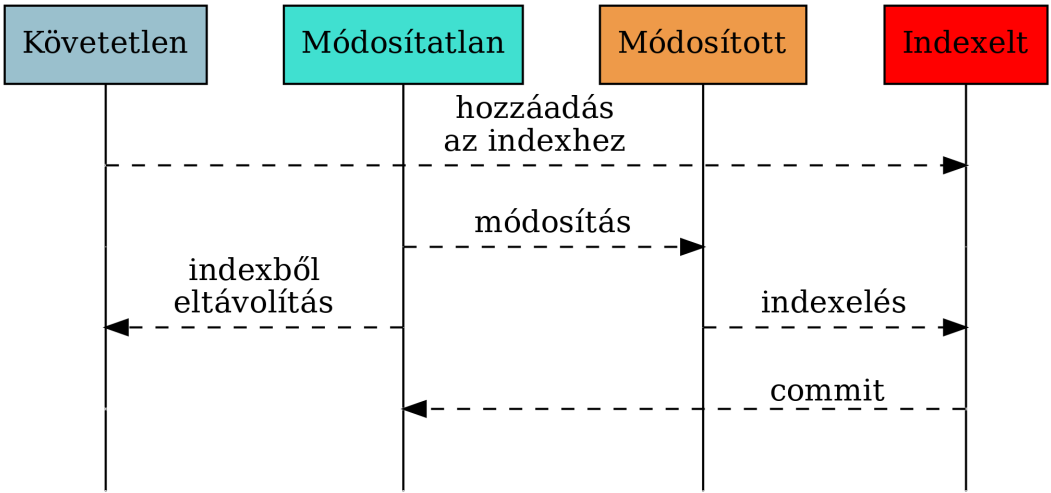
A három fájlállapot

A Git rendszerében három fő állapota van a fájloknak: módosított (modified), megjelölt (staged) és tárolt (committed):

- A módosított azt jelenti, hogy a fájl meg lett változtatva, de még nem lett tárolva, sem tárolásra megjelölve
- A megjelölt állapot azt jelenti, hogy a módosított fájl az aktuális verziójában meg lett jelölve, hogy a következő commit pillanatképbe kerüljön
- A tárolt azt jelenti, hogy az adat biztonságosan tárolva van a helyi adatbázisban

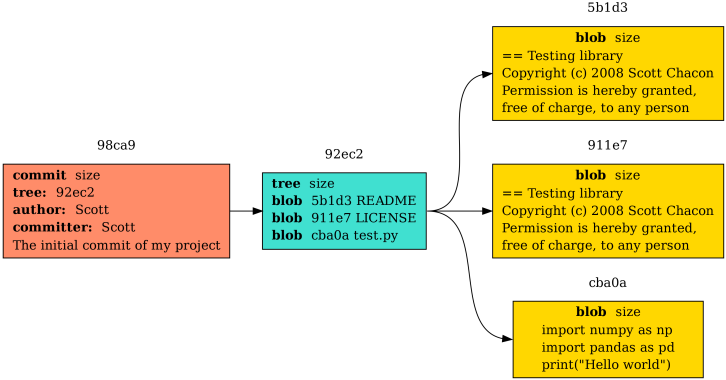


Státuszok változása



Commitok tárolása

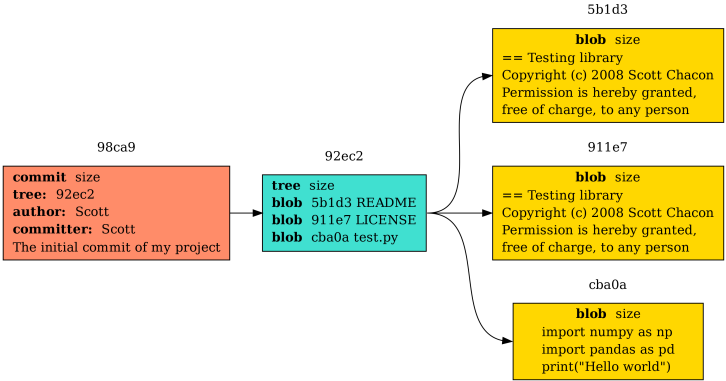
Amikor a commit létrejön a **git commit** futtatásával, a Git faobjektumként tárolja az adattárházban.
Ezután a létrehoz egy **commit objektumot**, amely a metaadatokat és egy mutatót tartalmaz a gyökérprojekt fához, így azt újra létre tudja hozni szükség esetén.



Commitok tárolása

Commit

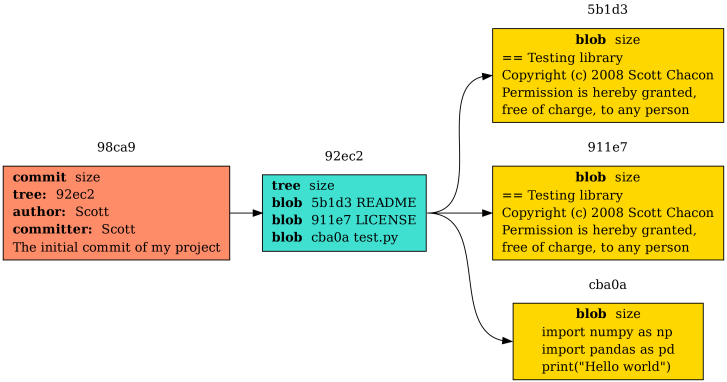
A **commit** a Git verziókezelő rendszerben egy olyan művelet, amely során a felhasználó rögzíti a változtatásokat a projektben, ezzel létrehozva egy új verziót az adattárházban. A commithoz commit üzenet társul.



Commitok tárolása

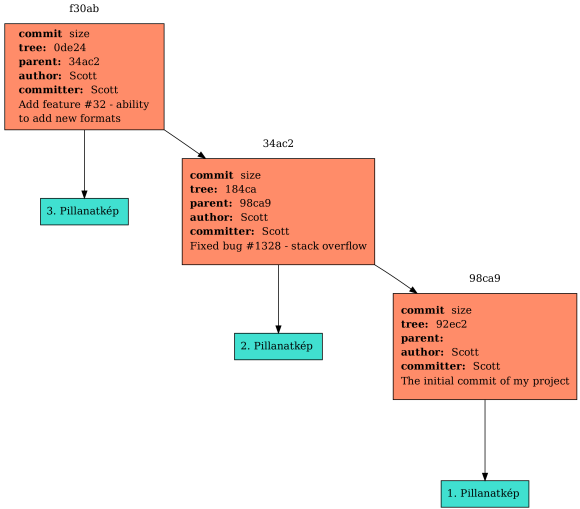
A Git adattárház most öt objektumot tartalmaz:

- Három **blobot** (amelyek mindegyike egy-egy fájl tartalmát képviseli)
- Egy faobjektumot, amely felsorolja a könyvtár tartalmát és megadja, hogy mely fájlnevek tárolódnak mely blobokként
- Egy commitot a gyökérfa mutatójával és a metaadatokkal



Több commit tárolása

Ha néhány változtatás után ismét egy commit következik, a következő commit egy mutatót tárol arra a commitra, amely közvetlenül megelőzte.

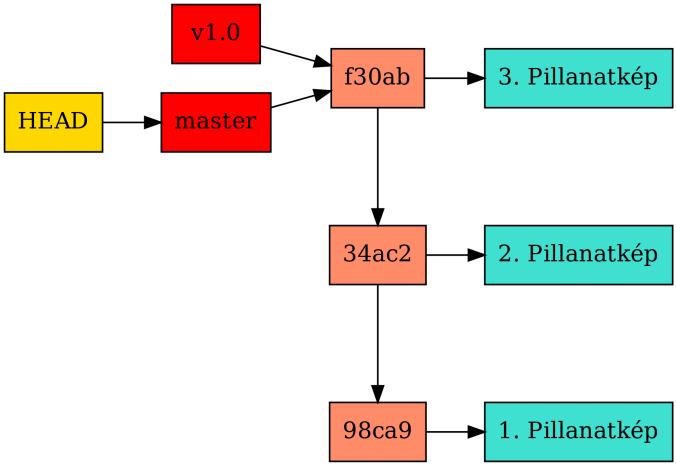


Egy Git ág és az előzményei

Fejlesztési ág

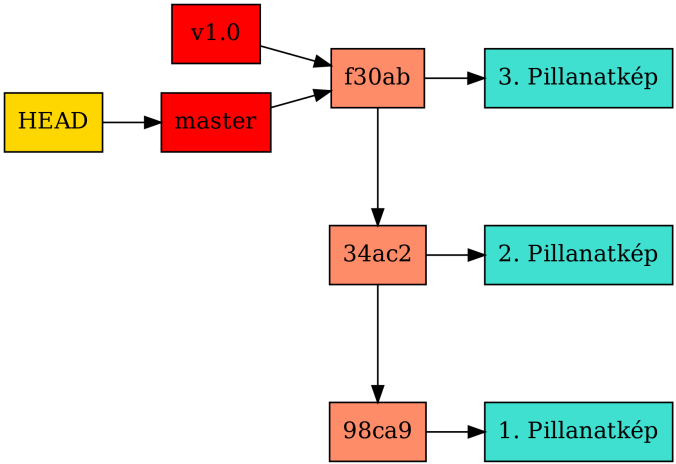
A Gitben egy **fejlesztési ág** egy egyszerű mozgatható mutató, amely valamely commitra mutat.

A példában fejlesztési ágak a `v1.0` és `master`.



Egy Git ág és az előzményei

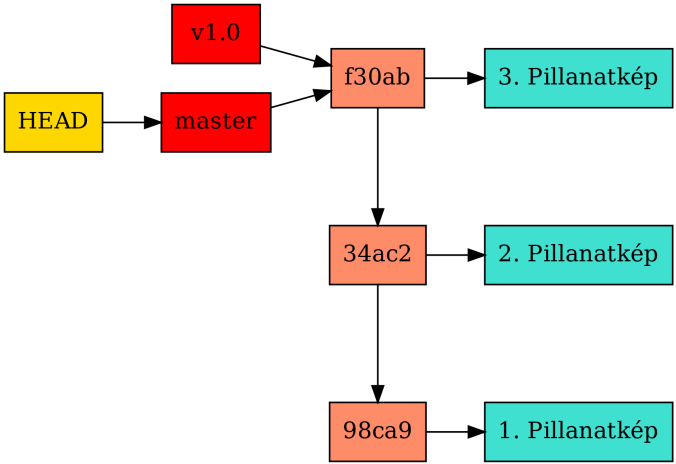
Az alapértelmezett ágnév a Gitben a **master**. Ahogy a commitok készülnek, a projekt kap egy master ágat, amely az utolsó, a felhasználó által készített commitra mutat. Minden egyes commitkor a master ág mutatója automatikusan előre mozog.



Egy Git ág és az előzményei

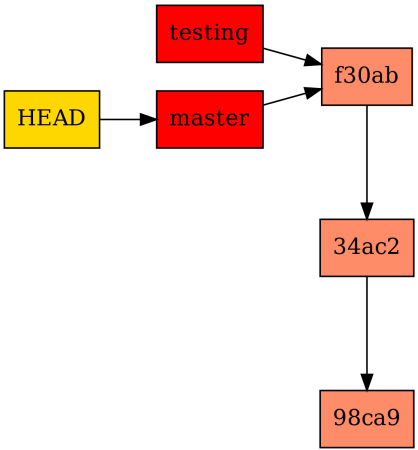
HEAD

A Git egy speciális mutatót tart nyilván, ami a **HEAD** néven ismert: ez egy mutató a jelenlegi helyi ágra.
Új fejlesztési ág létrehozásakor a HEAD nem az új ágra mutat, ezt kézzel kell megváltoztatni. Ez a **checkout**.



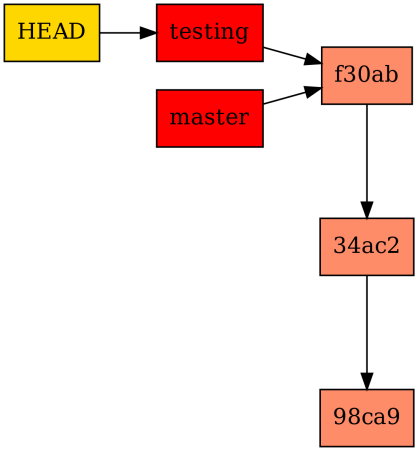
Checkout

Amikor új ágat hozunk létre, egy új mutató jön létre a verziókezelőben. Ebben az esetben a HEAD még az eredeti ágra mutat.



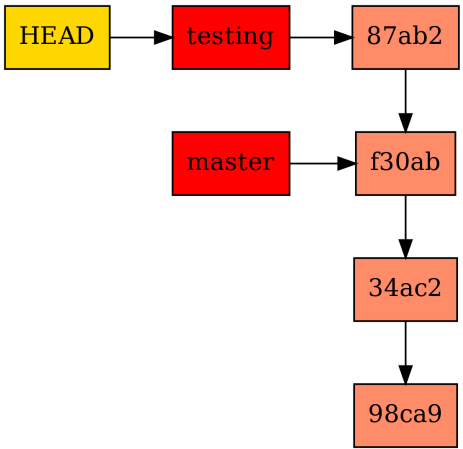
Checkout

A checkout művelete átírja a HEAD mutató pozícióját a megjelölt fejlesztési ágra.



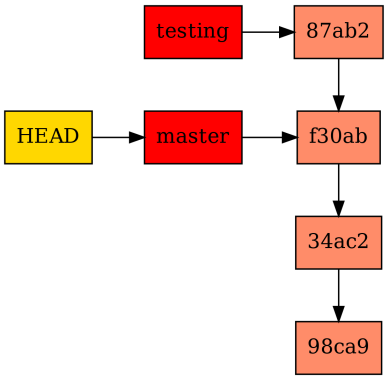
Commit az új ágra

Ha a jelenleg aktív (checkout által megjelölt) fejlesztési ágra történik egy commit, az ág lehagyja a main-t 1 commit-tal.



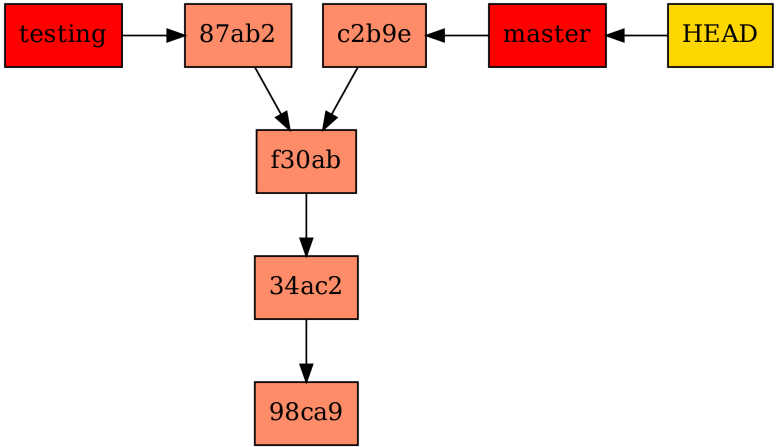
Másik ág aktiválása

A fejlesztési ágak között szabadon lehet váltani. Ha a checkout művelettel aktiválódik a master ág, a HEAD mutató onnantól kezdve rá mutat. Ebben az esetben a mappában lévő fájlok is megváltoznak a master-ben elmentett állapotukra.



Commit a master ágra

Ha ebben az állapotban a master ágra érkezik egy commit, egy új főág nyílik a projekten belül, és a két ág (testing, master) külön válnak.

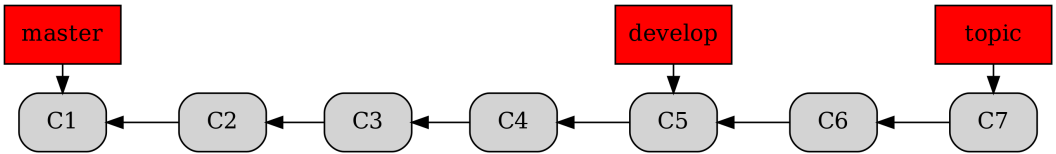


Hosszan futó fejlesztési ágak

Sok Git fejlesztő alkalmaz egy olyan munkafolyamatot, amely azt a megközelítést követi, hogy csak teljesen stabil kódot tartanak a `master` ágon.

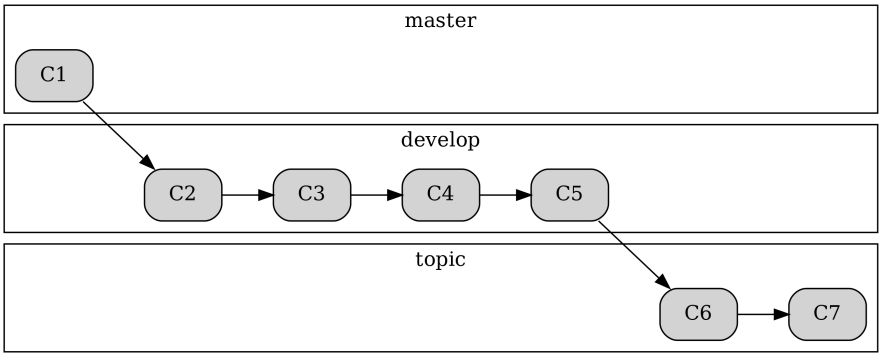
Van egy másik párhuzamos águk, amit `develop` vagy `next` néven neveznek, amelyen dolgoznak vagy a stabilitást tesztelik. Ezt használják a témakör ágak beolvasztására, amikor készen állnak, hogy megbizonyosodjanak arról, hogy minden teszt átment és nem vezetnek be hibákat.

Valójában arról beszélünk, hogy a mutatók feljebb mozognak a készített commitok sorában. A stabil ágak lejjebb vannak a commit történetben, míg a legújabb fejlesztések ágai feljebb találhatók.



Hosszan futó fejlesztési ágak

Ezzel ekvivalens definíció, ha a fejlesztési ágak külön-külön silókban vannak ábrázolva. Ezt lehetséges tovább végezni több szinten keresztül, de az általános elv a fejlesztési ágak létrehozására, hogy különböző stabilitási szintekkel rendelkeznek. Amikor egy ág elér egy stabilabb állapotot, akkor beolvad a felette lévőbe.

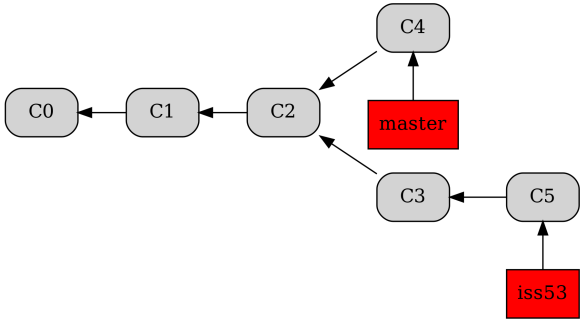


Több ág összeolvasztása

Összeolvasztás

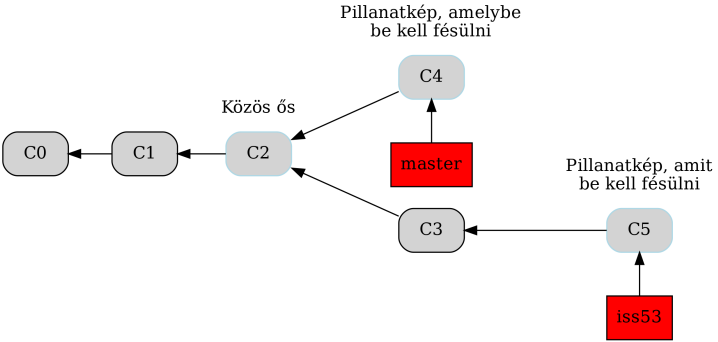
Az összeolvasztás (**merge**) a Git verziókezelő rendszerben egy olyan művelet, amelynek során egyik ágból a másikba importálódnak a változtatások. Az összeolvasztás folyamatában a Git megpróbálja összehangolni a különböző ágakban végzett módosításokat, hogy azokat egyetlen, integrált állapotban rögzítse.

Az 53-as probléma munkája befejeződött és készen áll arra, hogy a főprogrammal együtt legyen kezelve. Ahhoz, hogy ezt meg lehessen tenni, össze kell olvasztani az **iss53** ágat a **master** ággal.



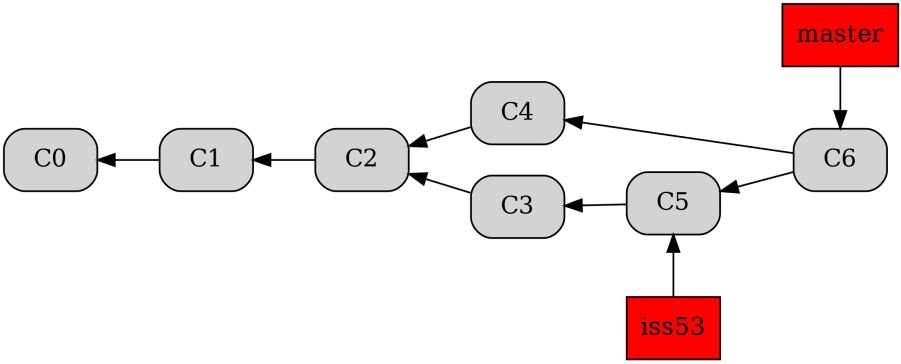
Megjelölés összeolvasztásra

Mivel a master ágon lévő commit nem közvetlen őse az összeolvasztásra kerülő ágnak, a Gitnek el kell végeznie némi munkát. Ebben az esetben a Git egyszerű háromutas összeolvasztást végez, a két ág pillanatképeire mutató referenciák és a két ág közös őse alapján.



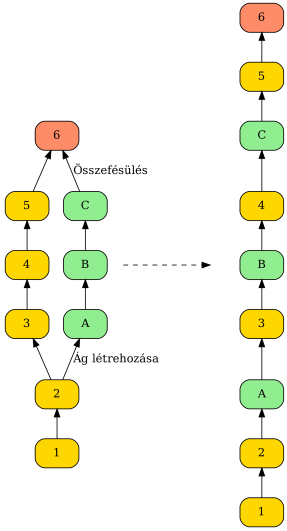
Összeolvasztás befejezése

Ahelyett, hogy egyszerűen előre mozgatná az ág mutatóját, a Git létrehoz egy új pillanatképet, amely az ebből a háromutas összeolvasztásból ered, és automatikusan létrehoz egy új commitot, amely erre mutat. Ezt nevezzük **összeolvasztási commitnak**, és különlegessége, hogy több mint egy szülője van.



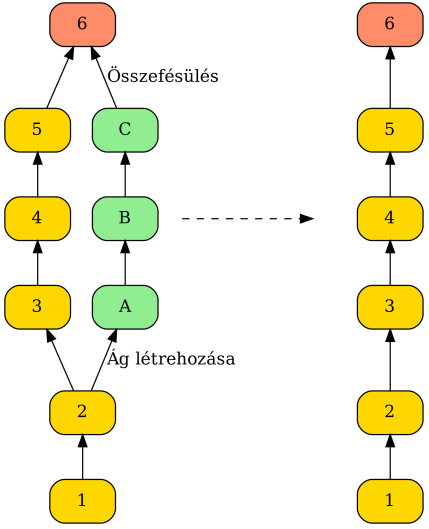
Összeolvasztás típusai

- **Egyesítés (merge)**: Minden commitot átvesz az összeolvasztandó ágból, és hozzáadja az alapág előzményeihez a létrehozásuk időbélyege alapján.



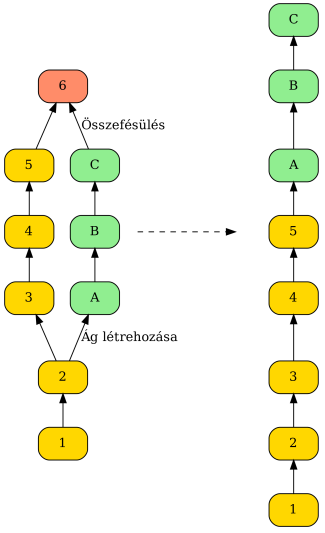
Összeolvasztás típusai

- **Egyesítés (merge)**: Minden commitot átvesz az összeolvasztandó ágból, és hozzáadja az alapág előzményeihez a létrehozásuk időbélyege alapján.
- **Összeolvasztás (squash)**: Minden commitot átvesz az ágból és egyetlen commitra olvasztja őket össze. Ez a commit hozzáadódik a történethez, de az ág commitjai közül egyik sem marad meg.



Összeolvasztás típusai

- **Egyesítés (merge)**: Minden commitot átvesz az összeolvasztandó ágból, és hozzáadja az alapág előzményeihez a létrehozásuk időbélyege alapján.
- **Összeolvasztás (squash)**: Minden commitot átvesz az ágból és egyetlen commitra olvasztja őket össze. Ez a commit hozzáadódik a történethez, de az ág commitjai közül egyik sem marad meg.
- **Újra alapozás (rebase)**: Az ág létrehozásának helyét veszi figyelembe, és azt a pontot helyezi át az alapág utolsó commitjára, majd újra alkalmazza a commitokat az ezekre a változásokra.



Konfliktusok előfordulása

```
 $ git merge iss53
Auto-merging index.html
CONFLICT (content):  Merge conflict in
index.html
Automatic merge failed; fix conflicts and
then commit the result.
```

Időnként az összefésülési folyamat nem megy zökkenőmentesen. Ha ugyanazon fájl ugyanazon részét különböző módon módosították a két egyesítendő ágon, a Git nem lesz képes tisztán egyesíteni őket.

A Git ebben az esetben nem hoz létre összefésülési commitot, hanem leállítja a folyamatot addig, amíg a konfliktust manuálisan fel nem oldja a fejlesztő

Konfliktusok előfordulása

```
 $ git status
On branch master
You have unmerged paths.
(fix conflicts and run "git commit")
Unmerged paths:
(use "git add <file>..." to mark
resolution)
both modified:  index.html
no changes added to commit (use "git add"
and/or "git commit -a"
```

A `git status` parancs füttatásával lehet látni, melyek azok a fájlok, amelyekben összefésületlen konfliktusok találhatóak.

Minden olyan fájl, amelyikben konfliktus található, összefésületlen státuszba kerül.

Konfliktusok jelölése fájlokban

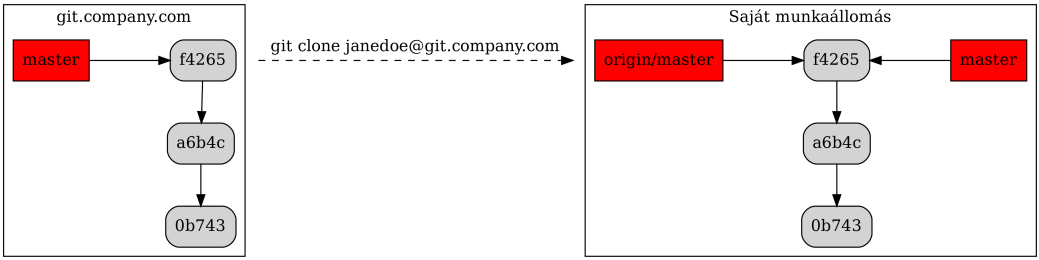
```
<<<<<<< HEAD:index.html
<div id="footer">contact :
email.support@github.com</div>
=======
<div id="footer">
please contact us at support@github.com
</div>
>>>>>>> iss53:index.html
```

A konfliktusos fájlok megfelelő részeiben ehhez hasonló szekciók lesznek találhatóak, amik jelzik a különbséget a két ág, ebben az esetben a `master` és a `iss53` között.

A programozónak olyan formára kell hoznia a fájlokat, hogy a konfliktus feloldódjon a megfelelő részek kitörtésével.

Távoli tárhely lemásolása (`clone`)

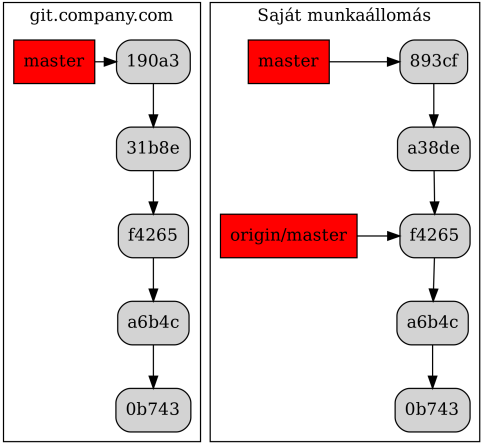
Ha van egy Git szerver a hálózaton a `git.company.com` címen, innen a Git képes automatikusan letölteni a az összes adatot. A távoli tárhelyet a Git `origin` néven nevezi el, és létrehoz egy mutatót a `master` ágra. Lokálisan `origin/master` néven inicializál mutatót egy helyi `master` ágra.



Fejlesztés a távoli ágon (commit)

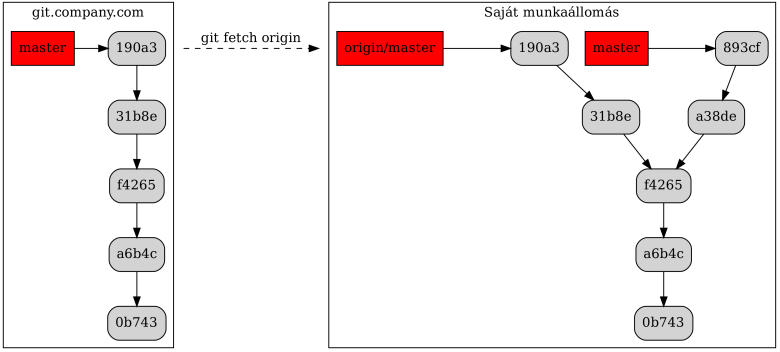
Ha a lokális `master` ágon történő munka közben egy másik fejlesztő feltölt változtatást a `git.company.com` szerverre és frissítani annak `master` ágát, a két fejlesztési történet divergál.

Amíg nem töltődnek le a frissítések a távoli szerverről, a helyi `master` mutató változatlan marad.



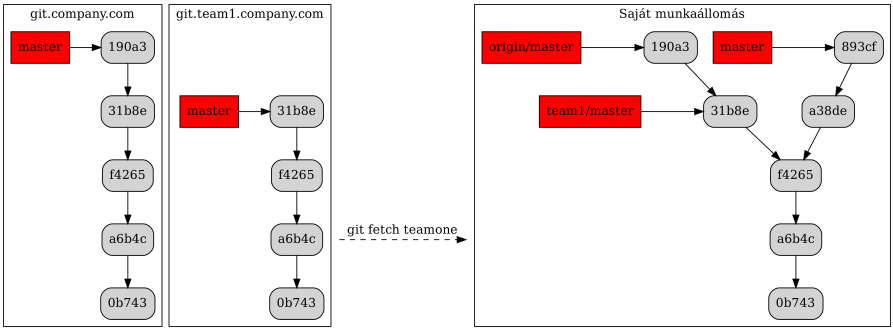
Két ág szinkronizálása (`fetch`)

Ágak szinkronizálását a `git fetch` parancs teszi lehetővé. Ez az utasítás megkeresi, hogy melyik szerver az `origin` (jelen esetben a `git.company.com`), letölti azokat az adatokat, amelyek még nincsenek jelen a lokális munkaállomáson, frissíti a helyi adatbázist és az `origin/master` mutatót naprakész pozícióba helyezi.



Harmadik ág szinkornizálása (`fetch`)

Ha a programozó a `git fetch teamone` parancsot futtatja, hogy letöltsön minden, a `teamone` ágon található adatot, a Git nem kér le új adatot, mert már rendelkezik az `origin` szerveren található adatok egy részhalmazával. Egy új mutató inicializálódik, `teamone/master` néven, ami arra a commitra mutat, amelyet a `teamone` szerver `master` néven tart nyilván.



Változtatások véglegesítése

push

A `git push` parancs a lokális fájlrendszeren végrehajtott változtatásokat feltölti a távoli tárhelyre. Ezáltal a commitok és ágak szinkronizálódnak a repozitóriummal.

pull

A `git pull` parancs a távoli repozitóriumban lévő változtatások letöltésére és a helyi repozitóriummal való egyesítésére szolgál.

Ez a parancs valójában két műveletet hajt végre: először a `git fetch` parancsot, amely letölti a változtatásokat, majd a `git merge` parancsot, amely egyesíti azokat a helyi ággal.