Understanding Git

Nelson Elhage Anders Kaseorg

Student Information Processing Board

September 29, 2009

Outline

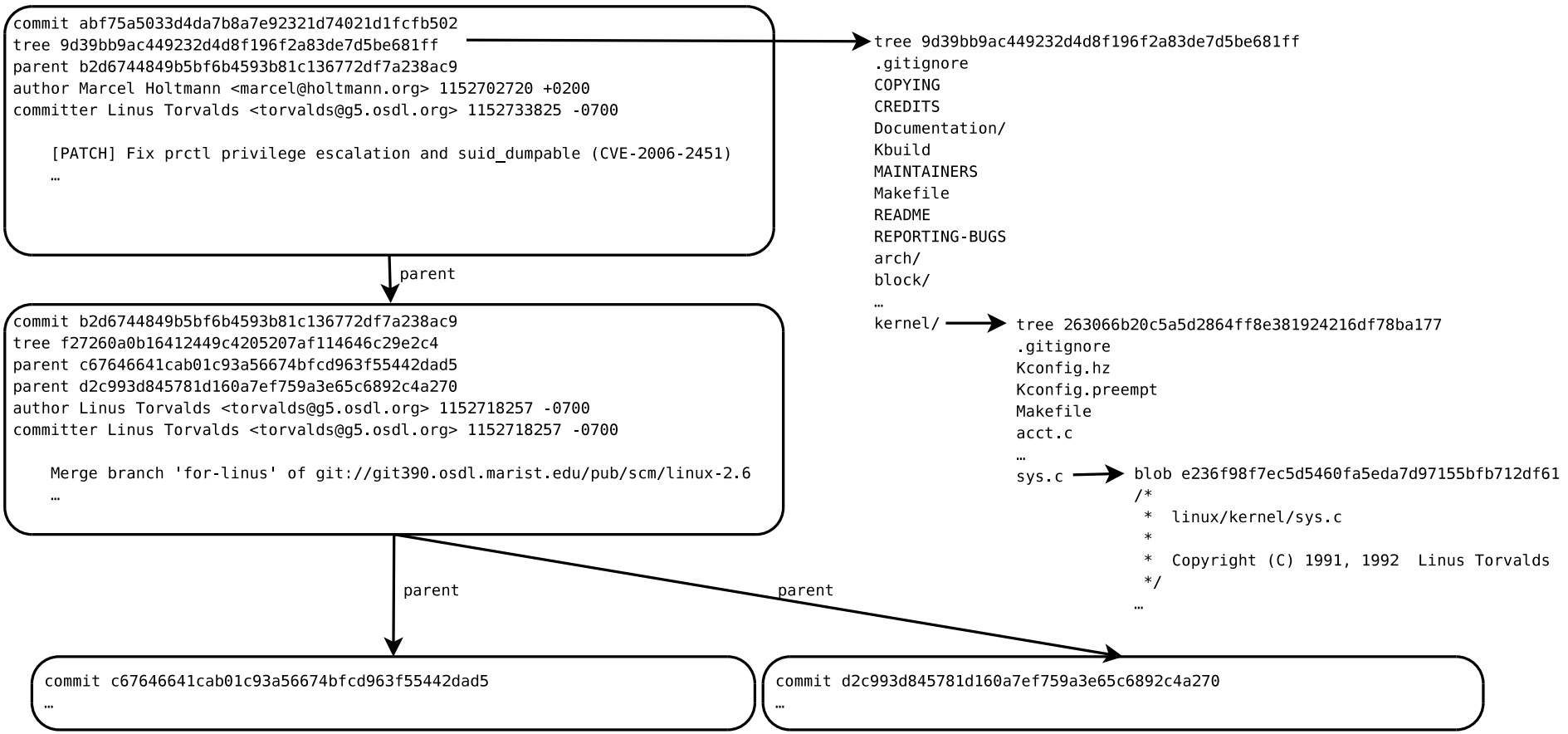
The Git model

- A Git repository contains four kinds of *objects*.
- An object is either a *blob* (file), a *tree* (directory), a *commit* (revision), or a *tag*.
- Every object is uniquely identified by a 40 hex digit number, which is the SHA-1 hash of its contents.
  - Don't worry—identifiers can be abbreviated by truncation, or referenced with human-readable names.
- Some objects refer to other objects using their identifiers.

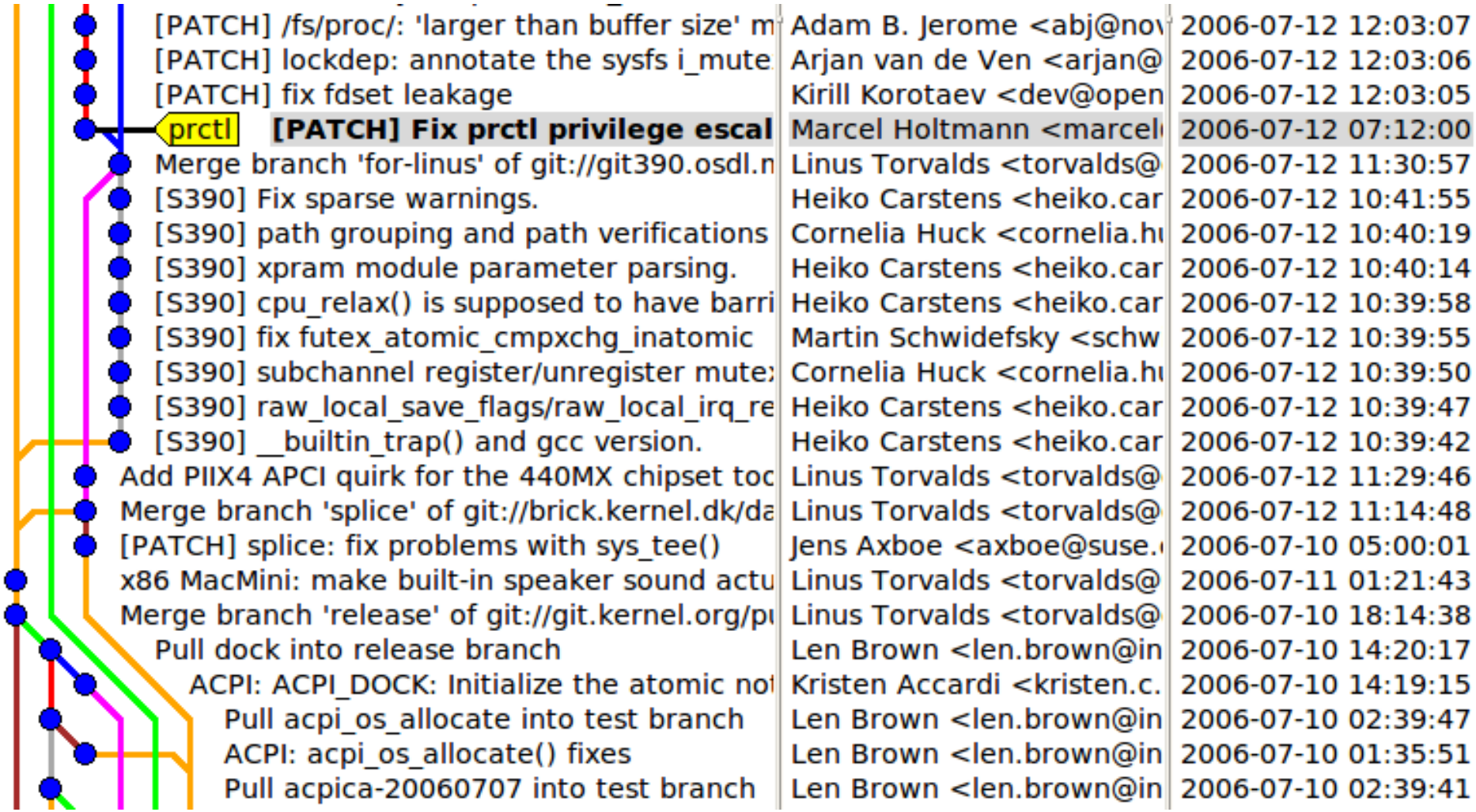
Objects

- Blobs and trees represent files and directories.
- Tags are named references to another object, along with some additional metadata.
- A commit object contains
  - a tree id
  - zero or more *parents*, which are commit ids
  - an *author* (name, email, date)
  - a *committer* (name, email, date)
  - a *log message*

A commit


commit abf75a5033d4da7b8a7e92321d74021d1fcfb502
tree 9d39bb9ac449232d4d8f196f2a83de7d5be681ff
parent b2d6744849b5bf6b4593b81c136772df7a238ac9
author Marcel Holtmann <marcel@holtmann.org> 1152702720 +0200
committer Linus Torvalds <torvalds@g5.osdl.org> 1152733825 -0700
[PATCH] Fix prctl privilege escalation and suid_dumpable (CVE-2006-2451)
…
parent
commit b2d6744849b5bf6b4593b81c136772df7a238ac9
tree f27260a0b16412449c4205207af114646c29e2c4
parent c67646641cab01c93a56674bfcd963f55442dad5
parent d2c993d845781d160a7ef759a3e65c6892c4a270
author Linus Torvalds <torvalds@g5.osdl.org> 1152718257 -0700
committer Linus Torvalds <torvalds@g5.osdl.org> 1152718257 -0700
Merge branch 'for-linus' of git://git390.osdl.marist.edu/pub/scm/linux-2.6
…
parent
parent
commit c67646641cab01c93a56674bfcd963f55442dad5
…
commit d2c993d845781d160a7ef759a3e65c6892c4a270
…
tree 9d39bb9ac449232d4d8f196f2a83de7d5be681ff
.gitignore
COPYING
CREDITS
Documentation/
Kbuild
MAINTAINERS
Makefile
README
REPORTING-BUGS
arch/
block/
…
kernel/
tree 263066b20c5a5d2864ff8e381924216df78ba177
.gitignore
Kconfig.hz
Kconfig.preempt
Makefile
acct.c
…
sys.c
blob e236f98f7ec5d5460fa5eda7d97155bfb712df61
/*
* linux/kernel/sys.c
*
* Copyright (C) 1991, 1992 Linus Torvalds
*/
…

More commits



A Git repository

- A Git repository is a collection of *refs*—*branches* and *tags*. (Branches are also known as *heads*.)
- A ref is a named mutable pointer to an object (usually a commit).
  - `HEAD` → `refs/heads/master`
  - `refs/heads/master` → `commit fec6ed...`
  - `refs/heads/ftrace` → `commit ce5c1e...`
  - `refs/tags/v2.6.8` → `commit e8ce2f...`
  - `refs/tags/v2.6.27` → `tag 4b5127...`
- The repository automatically stores the directed acyclic graph of objects rooted at these refs.

Branches

- Git was designed to enable lightweight branching and merging.
- Each repository can have any number of branches.
- Branches are just refs—pointers into the DAG of commits—and these pointers themselves are not versioned.
  - So you don't need to be afraid of making throwaway branches for experiments.

Consequences of the Git model

- Git tracks the history of your whole project, not the history of individual files.
  - Best practice is to keep projects that are logically separate in separate Git repositories.
- Git does not track renames as metadata in the repository.
  - Instead, renames are automatically detected based on content when this information is needed.
- A commit ID cryptographically certifies the integrity of the *entire history* of the repository up to that commit.
  - Git has powerful tools for rewriting history—but this requires communication with everyone that has pulled any affected commits from your repository.

Outline

Getting a Git repository

`git init`	Create an empty Git repository in the current directory. By default it will have one branch named `master`.
`git clone` *`url`*	Clone the Git repository from *`url`*. This may be over HTTP, SSH, or the Git protocol, or it may be a path to another local repository.

Both of these operations will create a *working copy*.

Working copy

- Every working copy has its own Git repository in the `.git` subdirectory (with arbitrarily many branches and tags).
  - The most important ref is `HEAD`, which refers to the current branch.
- The `.git` subdirectory also stores the *index*: a staging area for changes on top of `HEAD` that will become part of the next commit.
- Finally, the files outside of `.git` are the *working tree*.

Git workflow

- Changes made to the working tree can be *added* to the index.
- The index can be *committed* to the current branch (where it will then become the new `HEAD`).

Constructing commits

`git add` *file*	Add or update *file* from the working tree into the index.
`git reset` *file*	Unstage changes to *file* in the index, without touching the working tree.
`git checkout` *file*	Undo modifications to *file* in the working tree by reading it back from the index.
`git rm` *file*	Delete *file* from the index and the working tree.
`git mv` *oldfile newfile*	Shortcut for `mv` *oldfile newfile* plus the appropriate additions and removals in the index.
`git status`	Display the files changed in the index and in the working tree.
`git commit`	Make a commit out of the current index.
`git commit -a`	Shortcut for adding all modified files to the index and committing.

Referring to objects

`fc8da7a06bb66b707e7f5406657d5a3b7ee42c66` You can always refer to an object by its full SHA-1 ID, but this gets unwieldy very quickly.

`fc8da7` You can use a truncated SHA-1 as long as it is unambiguous.

refname You can refer to a branch or tag by name.

commit`^` Append a `^` to get the (first) parent of a commit.

commit`^2` The second parent of a commit, etc.

commit`~4` Short for *commit*`^^^^`—the great-great-grandparent of a commit.

commit`:`*filename* The given file or directory inside *commit*'s tree.

. . . and more (see `git help rev-parse` for a full description of the syntax).

Displaying changes

`git log`	List the commits on the current branch.
`git show` *`object`*	Show an object (e.g. the log information and patch for a commit, or the contents of a file).
`git diff`	Show the differences between the index and the working tree.
`git diff --cached`	Show the differences between `HEAD` and the index.
`git diff` *`commit`*	Show the differences between *commit* and the working tree.

Manipulating branches and tags

`git branch`	List the branches in your repository, with the current branch highlighted.
`git checkout` *branch*	Switch to the branch named *branch*. This updates `HEAD`, the index, and the working tree.
`git checkout -b` *branch* `[`*commit*`]`	Create a new branch named *branch* starting at *commit* (defaulting to current `HEAD`), and switch to it.
`git branch -d` *branch*	Delete the branch *branch*.
`git branch -m` *oldbranch newbranch*	Rename *oldbranch* to *newbranch*.
`git tag` *tag* `[`*commit*`]`	Attach a new tag named *tag* to *commit* (defaulting to current `HEAD`).
`git tag -d` *tag*	Delete the tag named *tag*.

Configuration hints

- You should tell Git who you are:

```
$ git config --global user.name "Your Name"
$ git config --global user.email "your@email.edu"
```

- And, if you're feeling colorful,

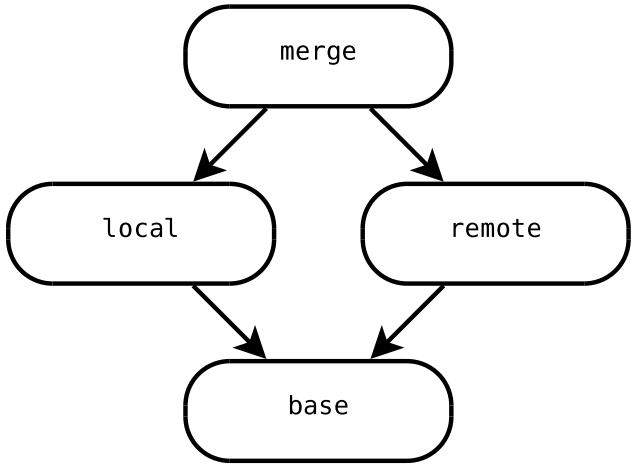
```
$ git config --global color.ui auto
```

(This configuration is stored in `~/.gitconfig`.)

Merging

`git merge` *commit* Merge *commit* into HEAD. The index must not contain any staged changes.

- In the general case, this will result in a *merge commit*—a commit with more than one parent.



- If *commit* is an ancestor of HEAD, then the merge is a no-op.
- If *commit* is a descendent of HEAD, then the merge degenerates into a *fast-forward*.

Resolving merge conflicts

- `git merge` works roughly by creating a diff against the common ancestor commit, and applying it against the current `HEAD`. (The general case is much more complicated.)
- Sometimes this patch will not apply to the current `HEAD`. This situation is called a *merge conflict*.
  - Git will respond by inserting *conflict markers* into the conflicted files, and asking you resolve the conflict.
  - Don't panic!
  - To resolve the conflict, edit the conflicted files appropriately and then `git add` them.
  - Alternatively, you can run `git mergetool` to resolve the conflicts interactively in a graphical diff program.

Merging example

```
$ seq 5 > numbers
```

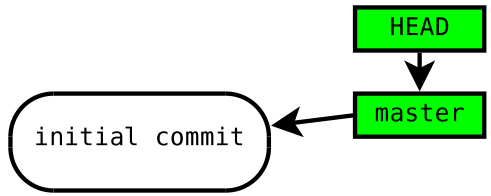
Merging example

```
$ seq 5 > numbers
$ git init
Initialized empty Git repository in /tmp/foo/.git/
```

Merging example

```
$ seq 5 > numbers
$ git init
Initialized empty Git repository in /tmp/foo/.git/
$ git add numbers
```

Merging example



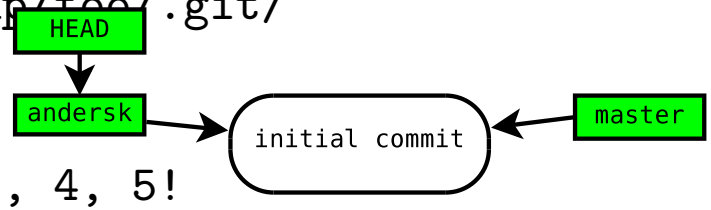

```
$ seq 5 > numbers
$ git init
Initialized empty Git repository in /tmp/foo/.git/
$ git add numbers
$ git commit -m '1, 2, 3, 4, 5!'
Created initial commit 4172330: 1, 2, 3, 4, 5!
 1 files changed, 5 insertions(+), 0 deletions(-)
 create mode 100644 numbers
```

Merging example


```
$ seq 5 > numbers
$ git init
Initialized empty Git repository in /tmp/foo/.git/
$ git add numbers
$ git commit -m '1, 2, 3, 4, 5!'
Created initial commit 4172330: 1, 2, 3, 4, 5!
 1 files changed, 5 insertions(+), 0 deletions(-)
 create mode 100644 numbers
$ git checkout -b andersk
Switched to a new branch "andersk"
```

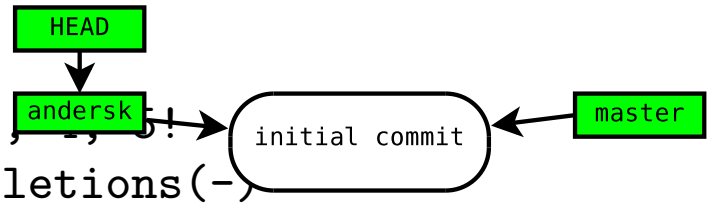
Merging example

```
$ seq 5 > numbers
$ git init
Initialized empty Git repository in /tmp/foo/.git/
$ git add numbers
$ git commit -m '1, 2, 3, 4, 5!'
Created initial commit 4172330: 1, 2, 3, 4, 5!
 1 files changed, 5 insertions(+), 0 deletions(-)
 create mode 100644 numbers
$ git checkout -b andersk
Switched to a new branch "andersk"
$ git branch
* andersk
  master
```



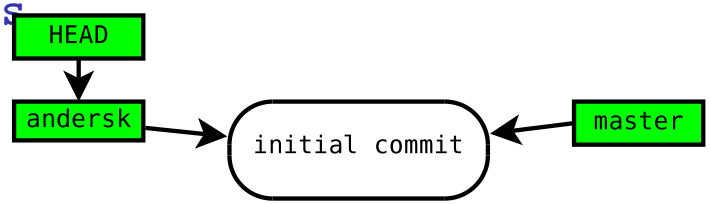
Merging example

```
$ seq 5 > numbers
$ git init
Initialized empty Git repository in /tmp/foo/.git/
$ git add numbers
$ git commit -m '1, 2, 3, 4, 5!'
Created initial commit 4172330: 1, 2, 3, 4, 5!
 1 files changed, 5 insertions(+), 0 deletions(-)
 create mode 100644 numbers
$ git checkout -b andersk
Switched to a new branch "andersk"
$ git branch
* andersk
  master
$ (echo 0; cat numbers) | sponge numbers
```



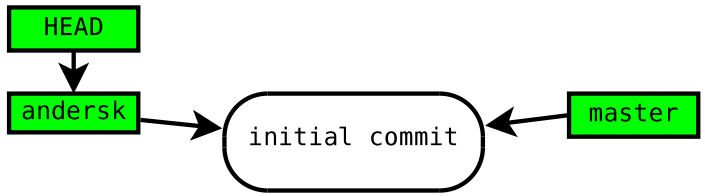
Merging example

```
 1 files changed, 5 insertions(+), 0 deletions(-)
 create mode 100644 numbers
$ git checkout -b andersk
Switched to a new branch "andersk"
$ git branch
* andersk
  master
$ (echo 0; cat numbers) | sponge numbers
$ git diff
diff --git a/numbers b/numbers
index 8a1218a..e8371f0 100644
--- a/numbers
+++ b/numbers
@@ -1,3 +1,4 @@
+0
 1
 2
 3
```



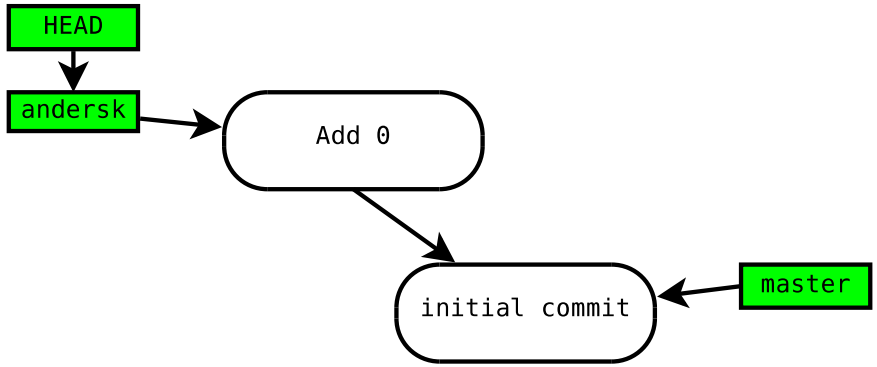
Merging example

```
 create mode 100644 numbers
$ git checkout -b andersk
Switched to a new branch "andersk"
$ git branch
* andersk
  master
$ (echo 0; cat numbers) | sponge numbers
$ git diff
diff --git a/numbers b/numbers
index 8a1218a..e8371f0 100644
--- a/numbers
+++ b/numbers
@@ -1,3 +1,4 @@
+0
 1
 2
 3
$ git add numbers
```



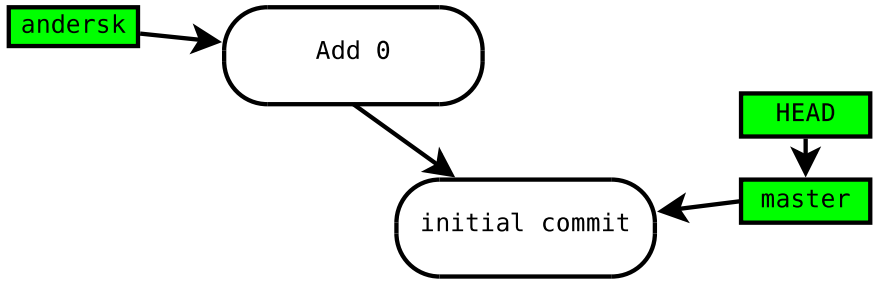
Merging example

```
$ git branch
* andersk
  master
$ (echo 0; cat numbers) | sponge numbers
$ git diff
diff --git a/numbers b/numbers
index 8a1218a..e8371f0 100644
--- a/numbers
+++ b/numbers
@@ -1,3 +1,4 @@
+0
 1
 2
 3
$ git add numbers
$ git commit -m 'Numbers start at 0.'
Created commit 7aeb494: Numbers start at 0.
 1 files changed, 1 insertions(+), 0 deletions(-)
```



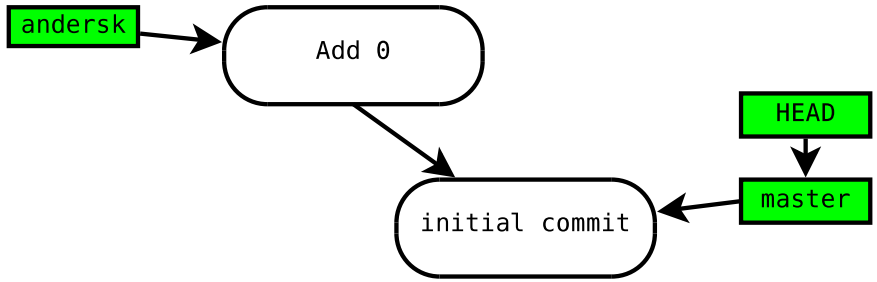
Merging example

```
  master
$ (echo 0; cat numbers) | sponge numbers
$ git diff
diff --git a/numbers b/numbers
index 8a1218a..e8371f0 100644
--- a/numbers
+++ b/numbers
@@ -1,3 +1,4 @@
+0
 1
 2
 3
$ git add numbers
$ git commit -m 'Numbers start at 0.'
Created commit 7aeb494: Numbers start at 0.
 1 files changed, 1 insertions(+), 0 deletions(-)
$ git checkout master
Switched to branch "master"
```



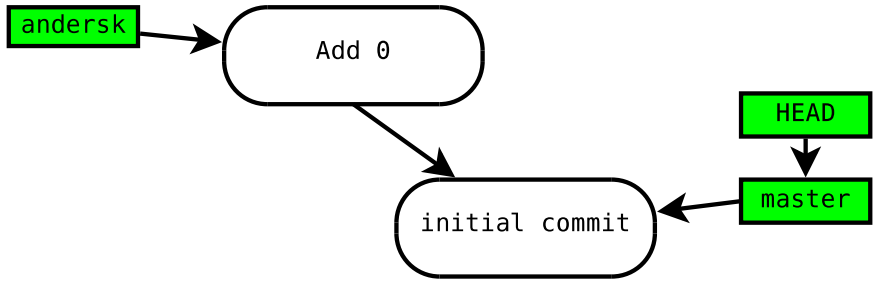
Merging example

```
$ (echo 0; cat numbers) | sponge numbers
$ git diff
diff --git a/numbers b/numbers
index 8a1218a..e8371f0 100644
--- a/numbers
+++ b/numbers
@@ -1,3 +1,4 @@
+0
 1
 2
 3
$ git add numbers
$ git commit -m 'Numbers start at 0.'
Created commit 7aeb494: Numbers start at 0.
 1 files changed, 1 insertions(+), 0 deletions(-)
$ git checkout master
Switched to branch "master"
$ echo 6 >> numbers
```



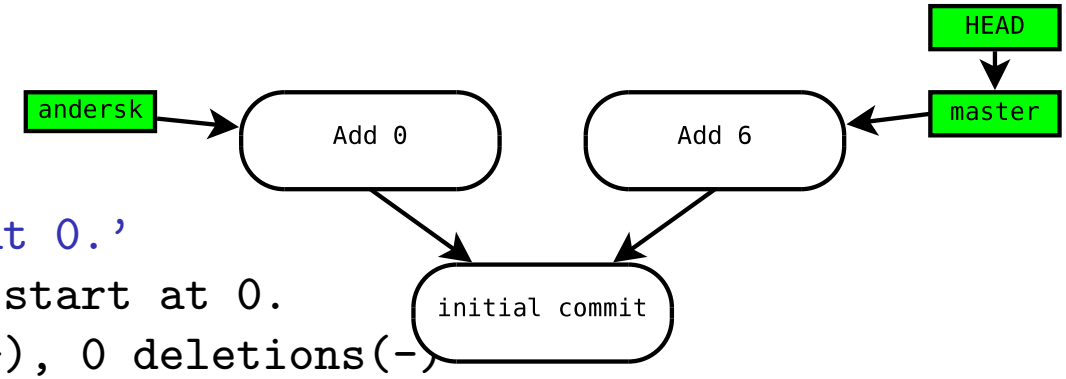
Merging example

```
$ git diff
diff --git a/numbers b/numbers
index 8a1218a..e8371f0 100644
--- a/numbers
+++ b/numbers
@@ -1,3 +1,4 @@
+0
 1
 2
 3
$ git add numbers
$ git commit -m 'Numbers start at 0.'
Created commit 7aeb494: Numbers start at 0.
 1 files changed, 1 insertions(+), 0 deletions(-)
$ git checkout master
Switched to branch "master"
$ echo 6 >> numbers
$ git add numbers
```



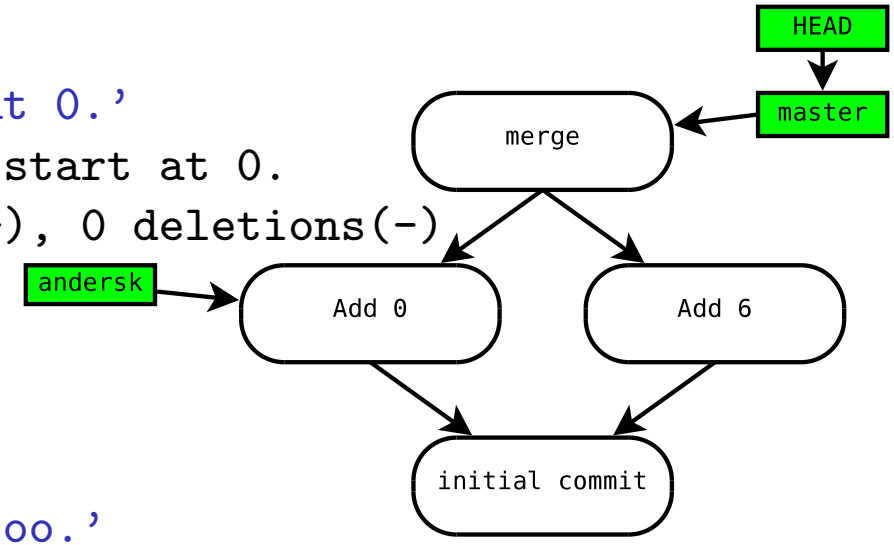
Merging example

```
--- a/numbers
+++ b/numbers
@@ -1,3 +1,4 @@
+0
 1
 2
 3
$ git add numbers
$ git commit -m 'Numbers start at 0.'
Created commit 7aeb494: Numbers start at 0.
 1 files changed, 1 insertions(+), 0 deletions(-)
$ git checkout master
Switched to branch "master"
$ echo 6 >> numbers
$ git add numbers
$ git commit -m '6 is a number too.'
Created commit 383c158: 6 is a number too.
 1 files changed, 1 insertions(+), 0 deletions(-)
```



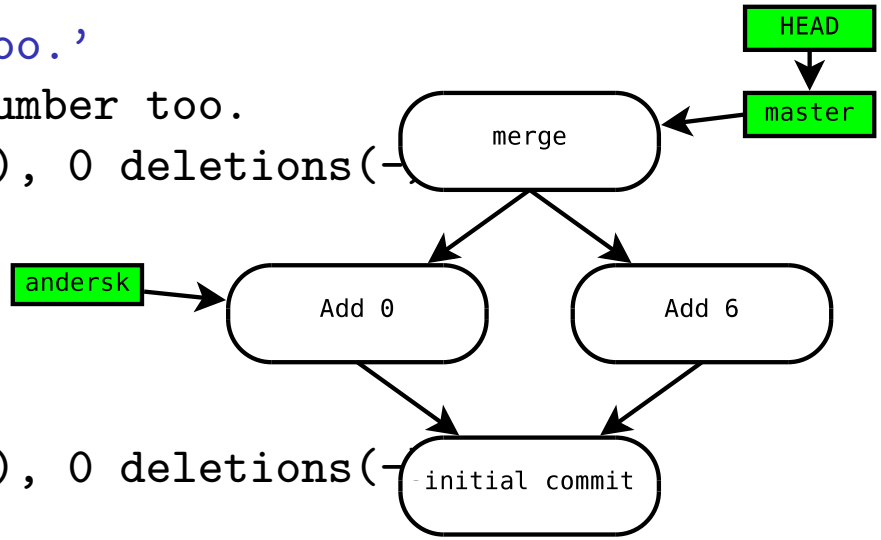
Merging example

```
 2
 3
$ git add numbers
$ git commit -m 'Numbers start at 0.'
Created commit 7aeb494: Numbers start at 0.
 1 files changed, 1 insertions(+), 0 deletions(-)
$ git checkout master
Switched to branch "master"
$ echo 6 >> numbers
$ git add numbers
$ git commit -m '6 is a number too.'
Created commit 383c158: 6 is a number too.
 1 files changed, 1 insertions(+), 0 deletions(-)
$ git merge andersk
Auto-merged numbers
Merge made by recursive.
 numbers |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```



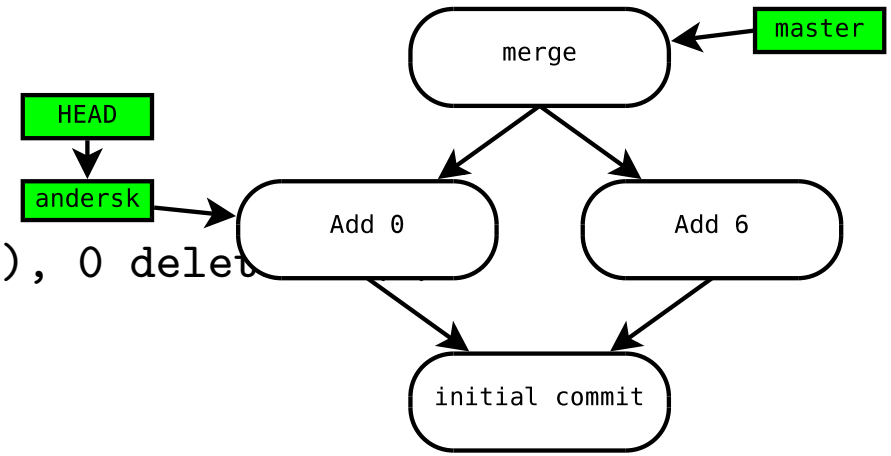
Merging example

```
$ echo 6 >> numbers
$ git add numbers
$ git commit -m '6 is a number too.'
Created commit 383c158: 6 is a number too.
 1 files changed, 1 insertions(+), 0 deletions(-)
$ git merge andersk
Auto-merged numbers
Merge made by recursive.
 numbers |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
$ cat numbers
0
1
2
3
4
5
6
```



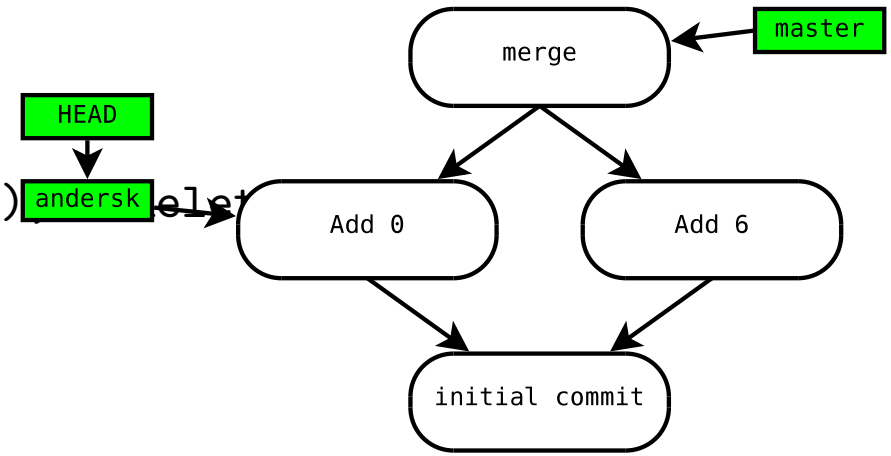
Merging example

```
$ git commit -m '6 is a number too.'
Created commit 383c158: 6 is a number too.
 1 files changed, 1 insertions(+), 0 deletions(-)
$ git merge andersk
Auto-merged numbers
Merge made by recursive.
 numbers |    1 +
 1 files changed, 1 insertions(+), 0 delet
$ cat numbers
0
1
2
3
4
5
6
$ git checkout andersk
Switched to branch "andersk"
```



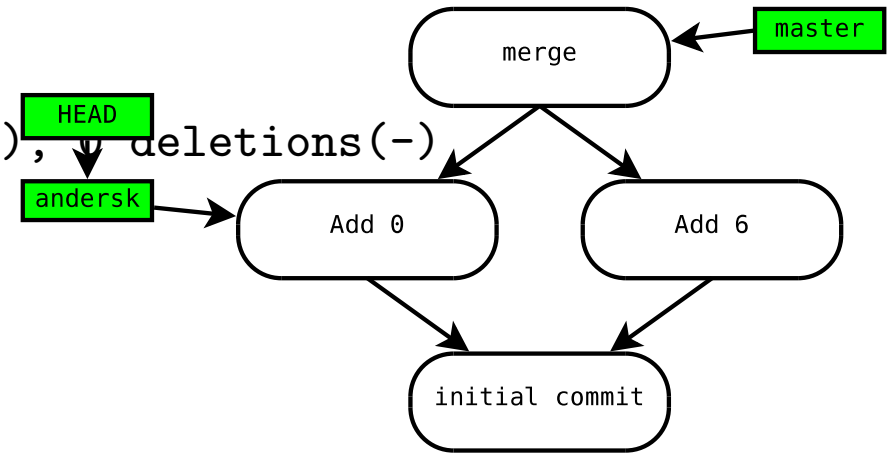
Merging example

```
Created commit 383c158: 6 is a number too.
 1 files changed, 1 insertions(+), 0 deletions(-)
$ git merge andersk
Auto-merged numbers
Merge made by recursive.
 numbers |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
$ cat numbers
0
1
2
3
4
5
6
$ git checkout andersk
Switched to branch "andersk"
$ echo 5½ >> numbers
```



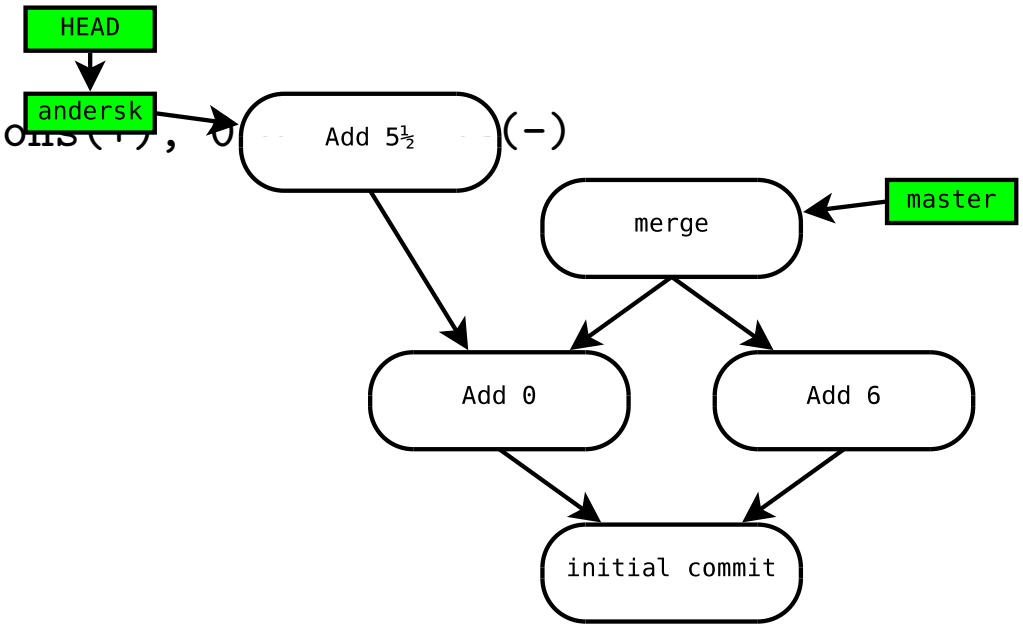
Merging example

```
 1 files changed, 1 insertions(+), 0 deletions(-)
$ git merge andersk
Auto-merged numbers
Merge made by recursive.
 numbers |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
$ cat numbers
0
1
2
3
4
5
6
$ git checkout andersk
Switched to branch "andersk"
$ echo 5½ >> numbers
$ git add numbers
```



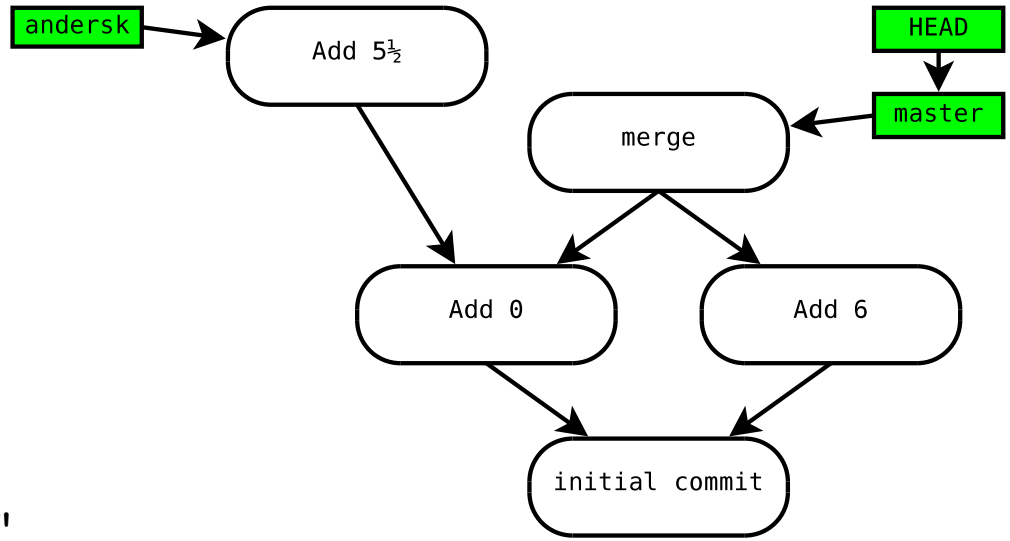
Merging example

```
Merge made by recursive.
 numbers |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
$ cat numbers
0
1
2
3
4
5
6
$ git checkout andersk
Switched to branch "andersk"
$ echo 5½ >> numbers
$ git add numbers
$ git commit -m '5½ is a better number.'
Created commit 5360c2d: 5½ is a better number.
 1 files changed, 1 insertions(+), 0 deletions(-)
```



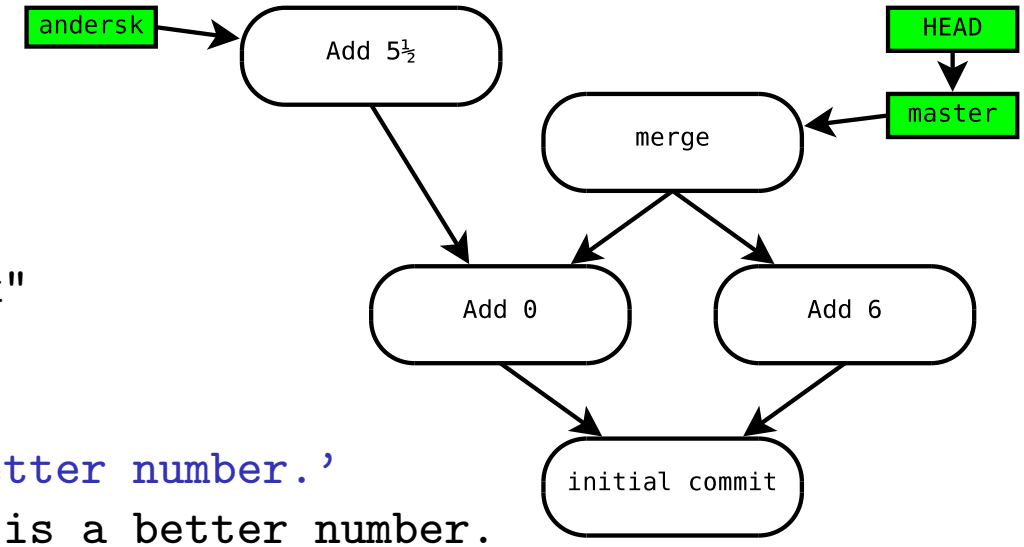
Merging example

```
 1 files changed, 1 insertions(+), 0 deletions(-)
$ cat numbers
0
1
2
3
4
5
6
$ git checkout andersk
Switched to branch "andersk"
$ echo 5½ >> numbers
$ git add numbers
$ git commit -m '5½ is a better number.'
Created commit 5360c2d: 5½ is a better number.
 1 files changed, 1 insertions(+), 0 deletions(-)
$ git checkout master
Switched to branch "master"
```



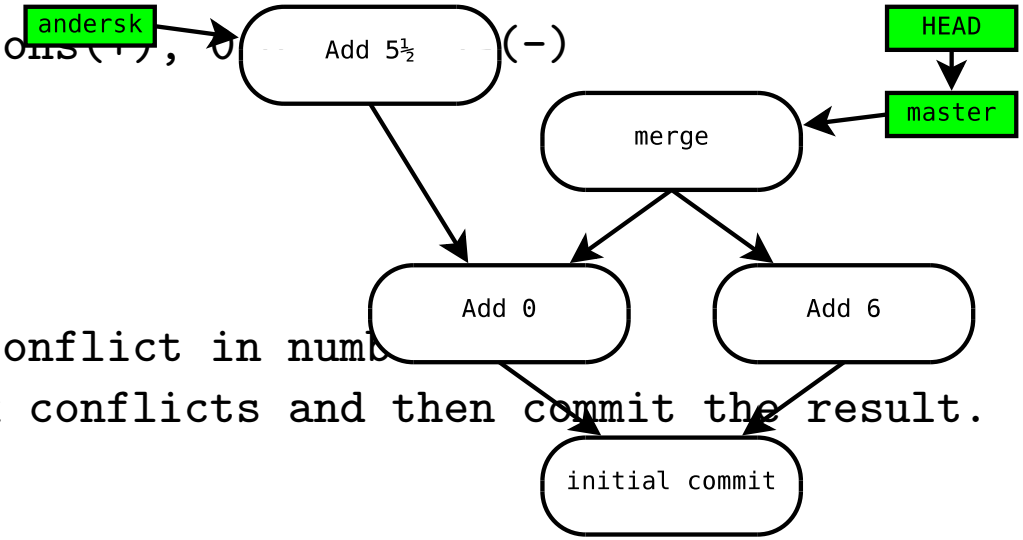
Merging example

```
2
3
4
5
6
$ git checkout andersk
Switched to branch "andersk"
$ echo 5½ >> numbers
$ git add numbers
$ git commit -m '5½ is a better number.'
Created commit 5360c2d: 5½ is a better number.
 1 files changed, 1 insertions(+), 0 deletions(-)
$ git checkout master
Switched to branch "master"
$ git merge andersk
Auto-merged numbers
CONFLICT (content): Merge conflict in numbers
Automatic merge failed; fix conflicts and then commit the result.
```



Merging example

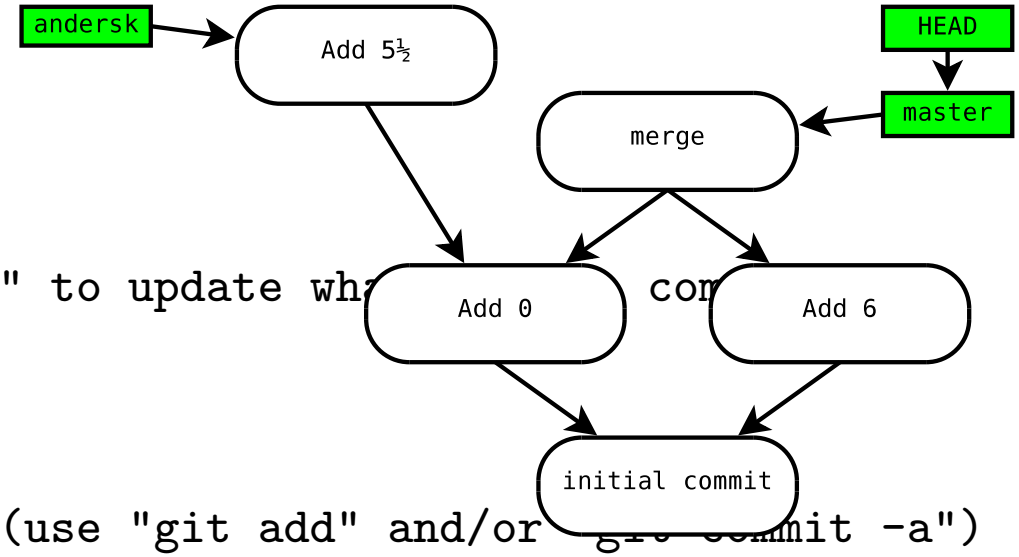
```
$ git commit -m '5½ is a better number.'
Created commit 5360c2d: 5½ is a better number.
 1 files changed, 1 insertions(+), 0 deletions(-)
$ git checkout master
Switched to branch "master"
$ git merge andersk
Auto-merged numbers
CONFLICT (content): Merge conflict in numb
Automatic merge failed; fix conflicts and then commit the result.
$ git status
numbers: needs merge
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
# unmerged:   numbers
#
no changes added to commit (use "git add" and/or "git commit -a")
```



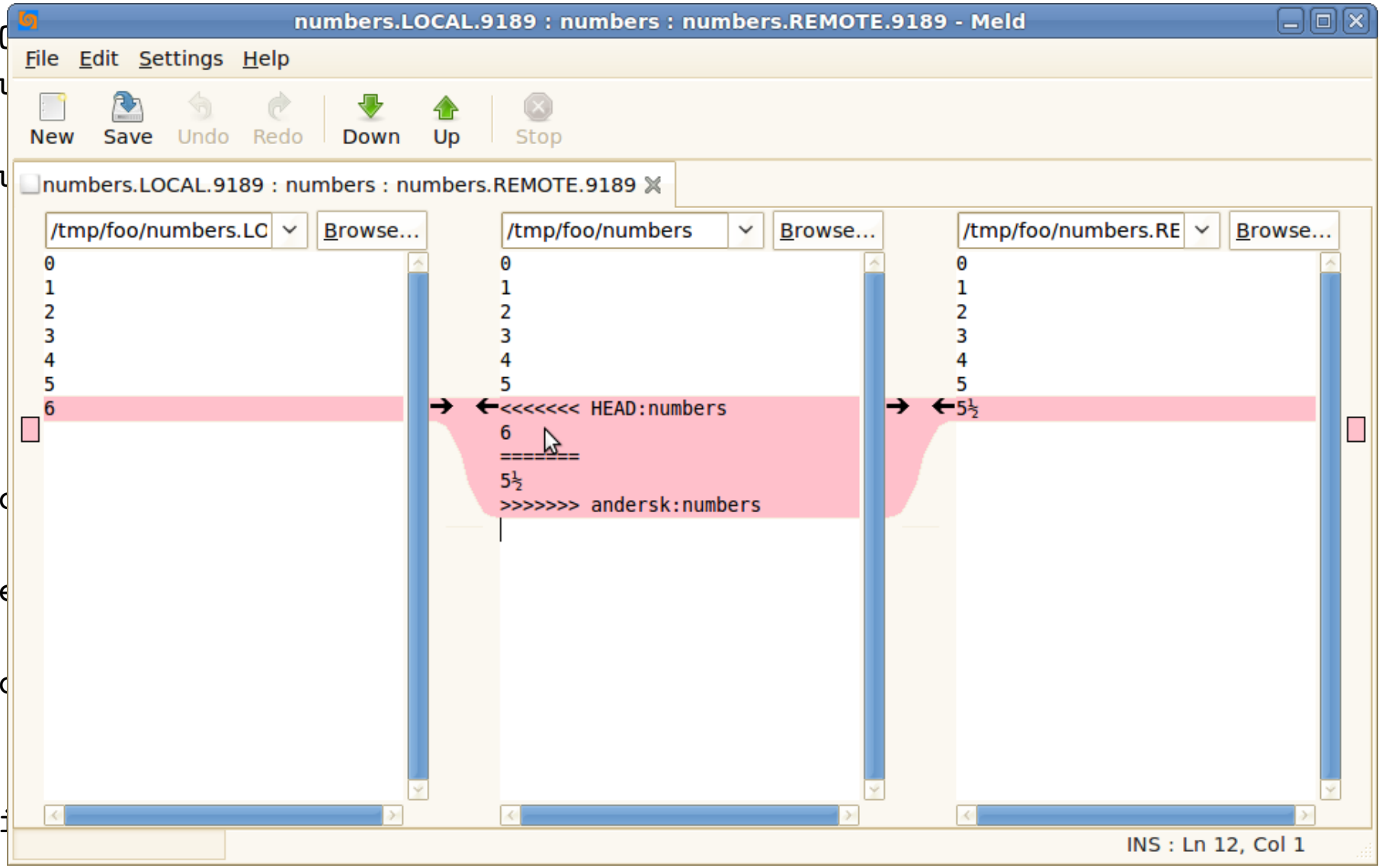
Merging example

```
CONFLICT (content): Merge conflict in numbers
Automatic merge failed; fix conflicts and then commit the result.
$ git status
numbers: needs merge
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update wh      com
#
# unmerged:   numbers
#
no changes added to commit (use "git add" and/or      commit -a")
$ git mergetool
Merging the files: numbers

Normal merge conflict for 'numbers':
  local: modified
  remote: modified
Hit return to start merge resolution tool (meld):
```



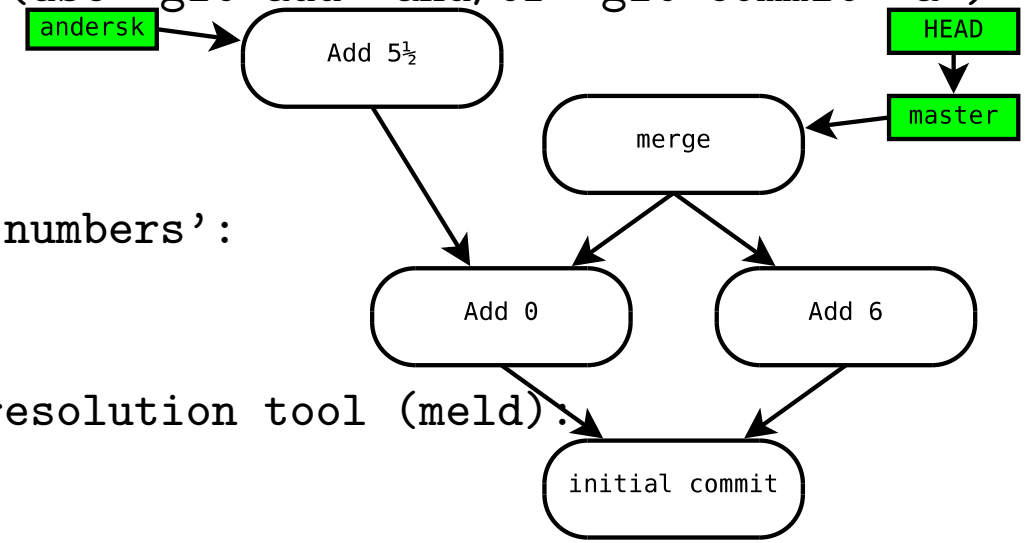
Merging example



Merging example

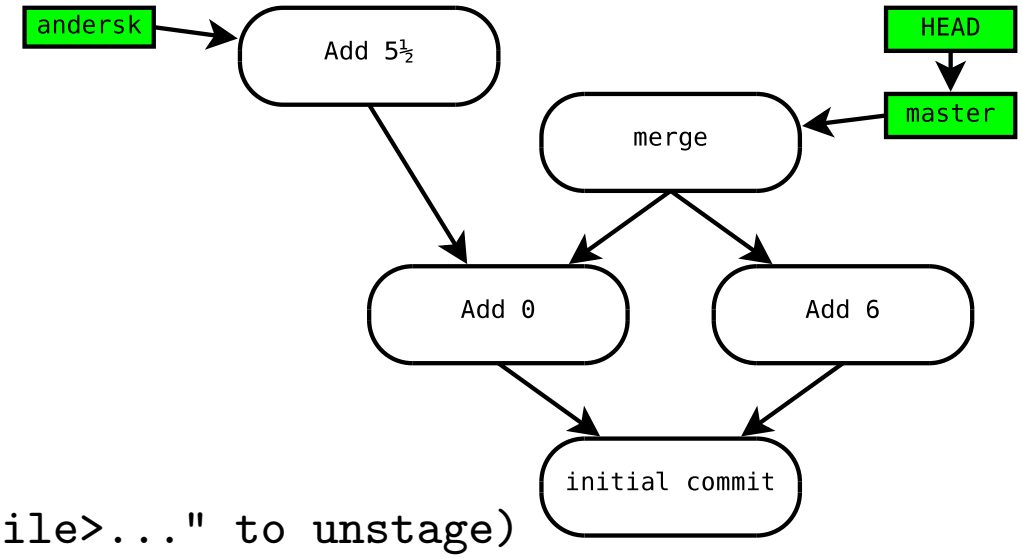
```
#
no changes added to commit (use "git add" and/or "git commit -a")
$ git mergetool
Merging the files: numbers

Normal merge conflict for 'numbers':
  local: modified
  remote: modified
Hit return to start merge resolution tool (meld):
$ cat numbers
0
1
2
3
4
5
5½
6
```

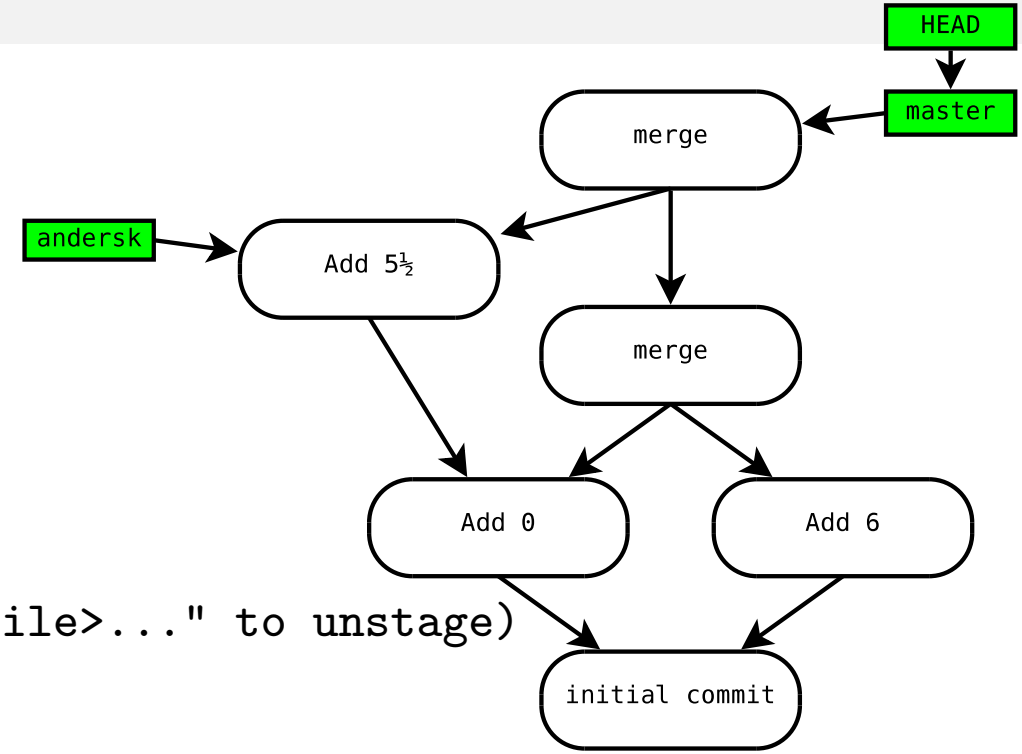


Merging example

```
1
2
3
4
5
5½
6
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# modified:   numbers
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
# numbers.orig
```



Merging example




```
3
4
5
5½
6
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# modified:   numbers
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
# numbers.orig
$ git commit
Created commit fc8da7a: Merge branch 'andersk'
```

Getting out of trouble

`gitk --all`	The graphical repository browser is immensely useful for visualizing what's going on in your repository.
`git reflog`	Show the *reflog* entries for `HEAD`.
`git reflog show` *ref*	Show the *reflog* entries for *ref*.
`git reset --hard` *commit*	Resets the *ref* pointed to by `HEAD`, as well as the index and working tree, to *commit*.

- The reflog tracks all local changes to refs. Whenever a ref is updated to point at a new commit, it gets an entry in the reflog.
- If you find yourself somewhere you don't expect, you can examine the log or the reflog, and then use `reset` to get back to a known point.
- This works even in a conflicted merge or rebase, if you just want to bail out and try something different.

A peek at the reflog

```
$ git reflog
fc8da7a... HEAD@0: commit (merge): Merge branch 'andersk'
994be80... HEAD@1: checkout: moving from andersk to master
5360c2d... HEAD@2: commit: 5½ is a better number.
7aeb494... HEAD@3: checkout: moving from master to andersk
994be80... HEAD@4: merge andersk: Merge made by recursive.
383c158... HEAD@5: commit: 6 is a number too.
4172330... HEAD@6: checkout: moving from andersk to master
7aeb494... HEAD@7: commit: Numbers start at 0.
4172330... HEAD@8: checkout: moving from master to andersk
$ git reflog show master
fc8da7a... master@0: commit (merge): Merge branch 'andersk'
994be80... master@1: merge andersk: Merge made by recursive.
383c158... master@2: commit: 6 is a number too.
4172330... master@3: commit (initial): 1, 2, 3, 4, 5!
$ git reflog show andersk
5360c2d... andersk@0: commit: 5½ is a better number.
7aeb494... andersk@1: commit: Numbers start at 0.
4172330... andersk@2: branch: Created from HEAD
```

Cherry-picking and reverting

`git cherry-pick` *commit*	Constructs a new commit on `HEAD` that performs the same changes as *commit*.
`git revert` *commit*	Constructs a new commit on `HEAD` that performs the reverse of the changes in *commit*.

- These commands construct a new commit that does not preserve any parent information pointing back to the old one. Use with care.
- Instead of cherry-picking from your development branch into your stable branch, for example, it is usually better to make the commit on stable and merge the entire stable branch into development.

Outline

Collaboration with Git

- Git allows bidirectional communication between any pair of repositories.
- Git speaks many protocols.
  - SSH
  - HTTP/HTTPS
  - DAV
  - Git protocol
  - rsync
  - direct filesystem access
- This flexibility lets you implement a wide range of centralized or distributed development models.

The simple case

- A freshly cloned repository has one remote called `origin`, which is the default source for pulls and destination for pushes.

`git fetch`	Download commits from `origin`. Each remote branch *`branch`* will be made available with the name `origin/`*`branch`*.
`git branch -r`	List the available remote branches.
`git branch -a`	List the available local and remote branches.

- Development is done on local branches. To work on a remote branch, you first create a local *tracking branch*, and then push any changes back to the remote branch as a separate operation.

Tracking branches

`git checkout -b` *branch* `origin/`*branch*	Create and switch to a new tracking branch named *branch*, set up to track the remote branch `origin/`*branch*.
`git pull`	Update the current tracking branch from `origin/`*branch*. Short for `git fetch; git merge origin/`*branch*.
`git push`	Push the current tracking branch back to `origin/`*branch*. This will only fast-forward the remote branch by default, so you may need to `git pull` first.
`git push origin :`*branch*	Delete the remote branch *branch*.
`git remote prune origin`	Clean up any refs to branches that have been deleted remotely.

Remotes

- A Git repository can be configured with references to any number of *remotes*.
- By default, a newly cloned repository has one remote named `origin` pointing to the source of the clone.

```
$ git clone /mit/andersk/Public/git/nss_nonlocal.git
Initialized empty Git repository in /tmp/nss_nonlocal/.git/
$ cat nss_nonlocal/.git/config
...
[remote "origin"]
url = /mit/andersk/Public/git/nss_nonlocal.git
fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"]
remote = origin
merge = refs/heads/master
```

Hosting a public Git repository

- A repository that's used for cloning, pulling, and pushing should usually be a *bare repository* (`git clone --bare`). A bare repository has no working tree, and lives in a directory named *`project`*`.git` instead of *`project`*`/.git`.
- The quickest solution at MIT is to drop your repository into AFS.
- To serve a repository on the web, you need to run `git update-server-info`, and enable the `hooks/post-update` hook.
- To serve a repository via the Git protocol, you need to create the `git-daemon-export-ok` file inside it.
  - `scripts.mit.edu` provides a Git hosting service. Drop your repository into `/mit/`*`locker`*`/Scripts/git/`*`project`*`.git` and access it at `git://`*`locker`*`.scripts.mit.edu/`*`project`*`.git`.

Outline

Rewriting history

- Git includes powerful tools for rewriting history.
- Of course, since modifying a commit changes its SHA-1 identifier, by "rewriting history" we actually mean "transforming a sequence of commits into a different sequence of commits".
- You need to be careful about rewriting commits that others may have already pulled.
  - By default, Git will prevent you from pushing changes that are not fast-forwards, unless you ask very hard.
- Rewriting is extremely useful for cleaning up a private branch before making it publicly available.

Why rewriting is useful

- A good history will include one commit for each self-contained logical change to the tree.
- Avoid cluttering the history with typos and trivial bugs that are fixed in the following commits.
  - This makes things more pleasant for anyone who wants to read or review your changes.
  - It also makes it easier to pinpoint bugs with `git bisect`.
- You don't need to worry about making your commits perfect as you write them, since you can rearrange them later.

Resetting branches

`git reset --hard` *commit*	Resets the current `HEAD`, as well as the index and working tree, to *commit*.
`git reset` *commit*	Resets the current `HEAD` and index to *commit*, without touching the working tree.
`git reset --soft` *commit*	Resets the current `HEAD` to *commit*, without touching the index or the working tree.
`git commit --amend`	Adds the modifications in the index to the current commit at `HEAD` "in place". Approximately equivalent to `git reset HEAD^; git commit`.

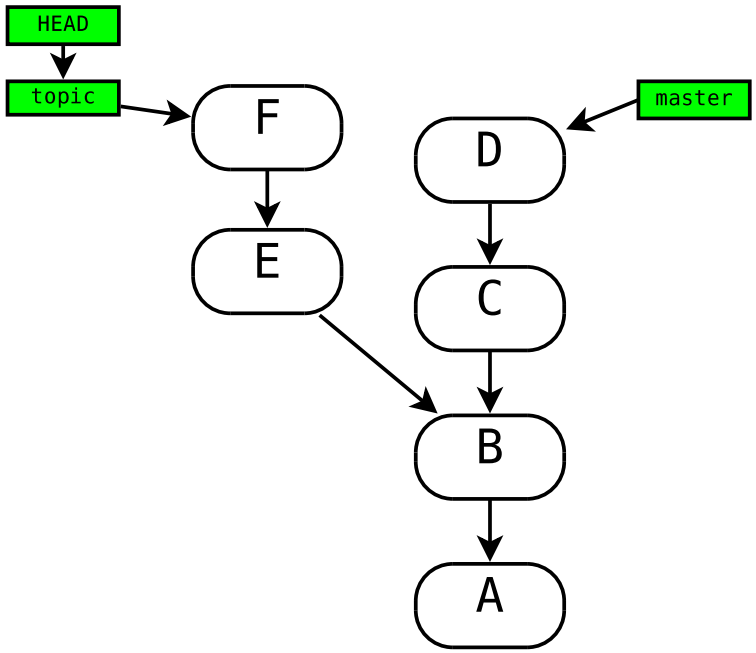
Rebasing

`git rebase` *commit*	Rebase `HEAD` onto *commit*.
`git pull --rebase`	Short for `git fetch; git rebase origin/`*branch*.

- `rebase` finds all commits that are in `HEAD` but not in *commit*, and re-applies them starting with *commit*. The current branch is reset to the result.
- This has a similar effect to a merge, but maintains a linear history, at the cost of losing some information.
- `rebase` *changes the object identifiers* of the re-applied commits.
- `rebase` is often preferred to keep history clean.

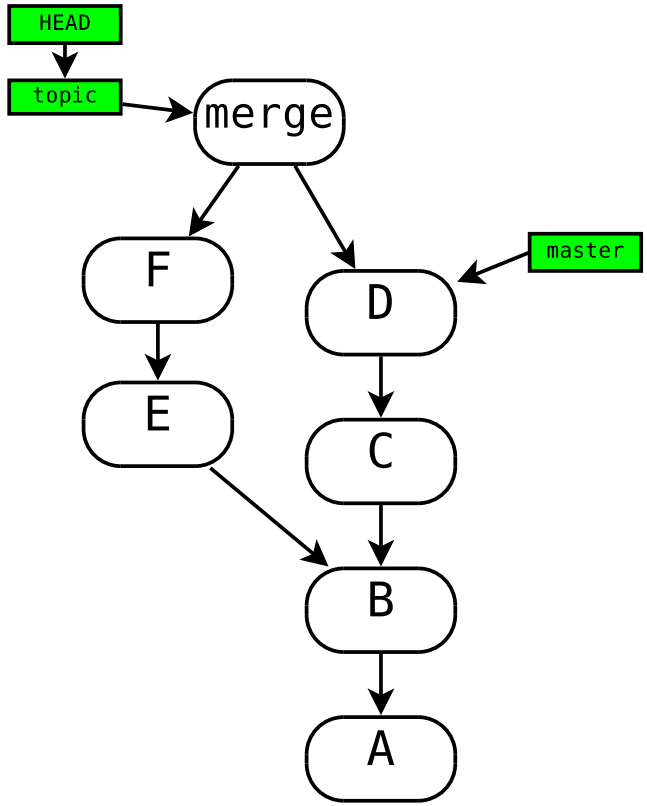
Rebase vs. merge

- We have development on both a *topic branch* and `master`.



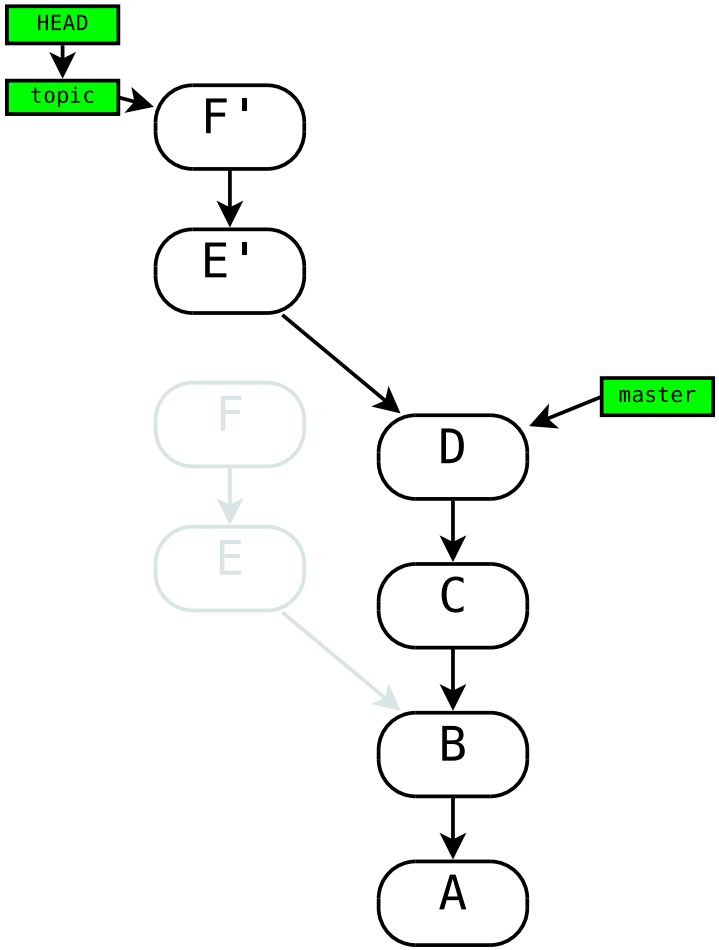
Rebase vs. merge

- `merge` results in a forked history:



Rebase vs. merge

- `rebase` rewrites commits and maintains a linear history:



Interactive rebasing

`git rebase -i commit` Rebase `HEAD` onto *commit*, letting you interactively edit the resulting history. (Typically *commit* will already be an ancestor of `HEAD`, to edit history "in place".)

- Git will start your editor on a list of the commits to be applied on top of *commit*.
- You can cut and paste to arbitrarily reorder the commits.
- You can delete a line to remove that commit completely.
- You can insert the `squash` directive to fuse a commit into the previous commit.
- You can insert the `edit` directive to have Git pause after applying a commit, so you can amend it in place or insert new commits, before further commits are applied.

Advanced rewriting

`git filter-branch`	Rewrite history by mapping each commit through an arbitrary script.
`.git/info/grafts`	Causes the local repository to pretend that certain commits have different parents than their real ones. (`git filter-branch` can then rewrite the fake parents into real ones.)
`git fast-export`	Dump history in a human-readable format, with SHA-1 IDs replaced by symbolic marks, so that it can be edited by hand.
`git fast-import`	Read back commits produced by `git fast-export`.

Outline

Other awesome Git commands

`git bisect`	Easily pinpoint a regression in your history using a repeated bisection search.
`git blame`	Annotate each line of a file with information about its last modification.
`git cvsimport`, `git svn`	Use Git to work with repositories in other formats. (I think Git makes a much better CVS or SVN client than the native ones!)
`git format-patch`, `git send-email`, `git am`	Send and receive Git patches by email.
`git grep`	Search for a regex in a Git tree.
`git stash`	Quickly stash away and reapply temporary changes while you do other work.
`git submodule`	Manage a group of related Git repositories.

Exploring Git yourself

- There are many commands we haven't talked about, and the ones we have take additional options that can help you work more efficiently.
- Anything you think you should be able to do within the Git model can probably be done.
- Git is designed to be conveniently scriptable.
- Git has extensive documentation—start with `man git`.
  - To get documentation on any `git` *`command`*, run `git help` *`command`* or (equivalently) `man git-`*`command`*.