## Plumbing commands

$ git cat-file

$ git ls-files

the plumbing command, *cat-file*. You use two options here:

-t = type—shows the type of the object

-p = pretty—prints information about the object

$ git <git-options> <command> <command-options> <operands>

$ git

$ git help glossary

$ git help –a //list of over 150 commands

$ git help –g //list of common guides

$ git help config

$ git config –h

$ git config --help

## [Git from the inside out](https://maryrosecook.com/blog/post/git-from-the-inside-out)

The .git directory and its contents are Git’s. All the other files are collectively known as the working copy. They are the user’s.

### Add some files

$ git add data/letter.txt

This command has two effects.

* First, it creates a new blob file in the .git/objects/ directory.

For example, Git hashes the content to 2e65efe2a145dda7ee51d1741299f848e5bf752e. The first two characters are used as the name of a directory inside the objects database: .git/objects/2e/. The rest of the hash is used as the name of the blob file that holds the content of the added file: .git/objects/2e/65efe2a145dda7ee51d1741299f848e5bf752e.

* Second, git add adds the file to the index. The index is a list that contains every file that Git has been told to keep track of. It is stored as a file at .git/index.

### Make a commit

git commit -m 'a1'

The commit command has three steps. It creates a tree graph to represent the content of the version of the project being committed. It creates a commit object. It points the current branch at the new commit object

* Git records the current state of the project by creating a tree graph from the index. This tree graph records the location and content of every file in the project.

The graph is composed of two types of object: blobs and trees.

Blobs are stored by git add. They represent the content of files.

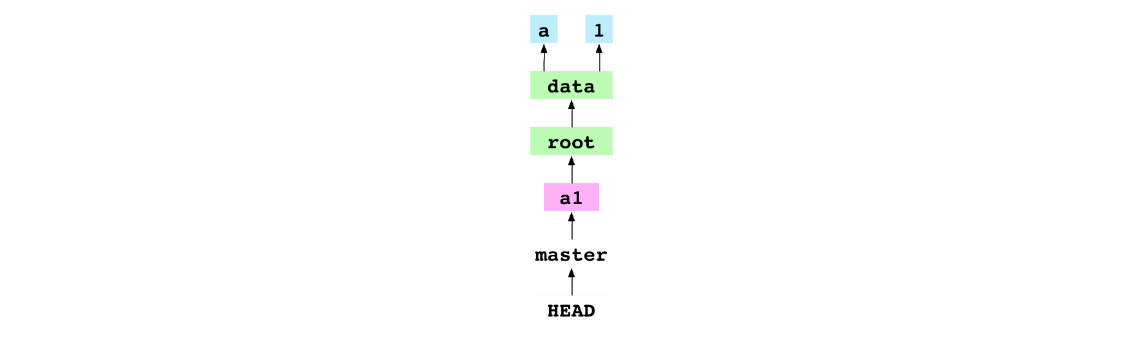
Trees are stored when a commit is made. A tree represents a directory in the working copy.

* git commit creates a commit object after creating the tree graph. The commit object is just another text file in .git/objects/:
* Finally, the commit command points the current branch at the new commit object. Which is the current branch? Git goes to the HEAD file at .git/HEAD and finds:

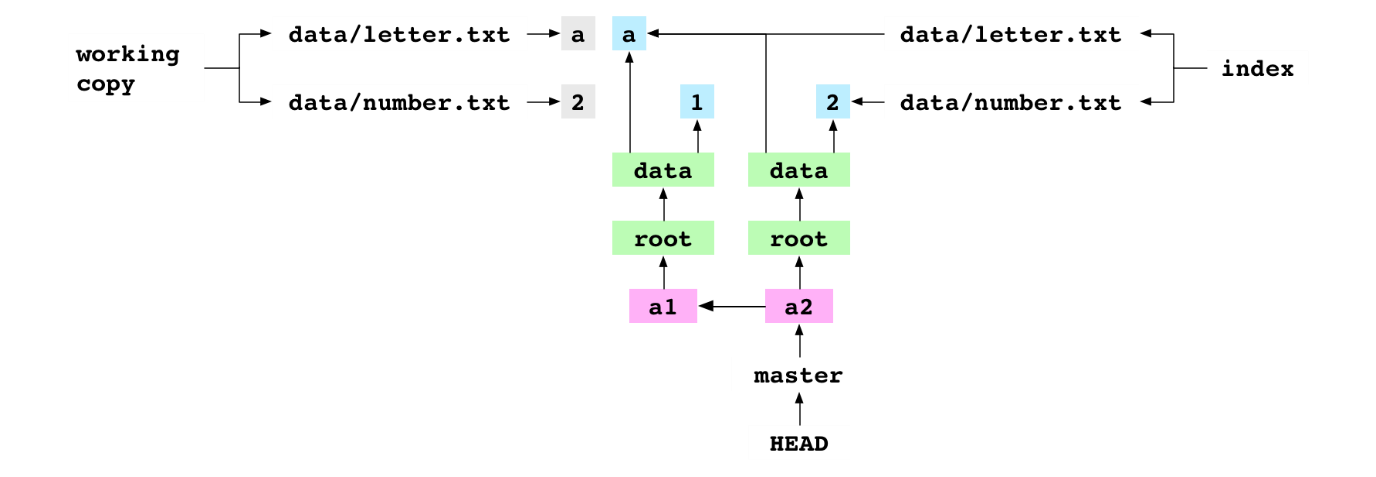
ref: refs/heads/master

This says that HEAD is pointing at master. master is the current branch.

HEAD and master are both refs. A ref is a label used by Git or the user to identify a specific commit.



HEAD pointing at master and master pointing at the a1 commit



### Check out a commit

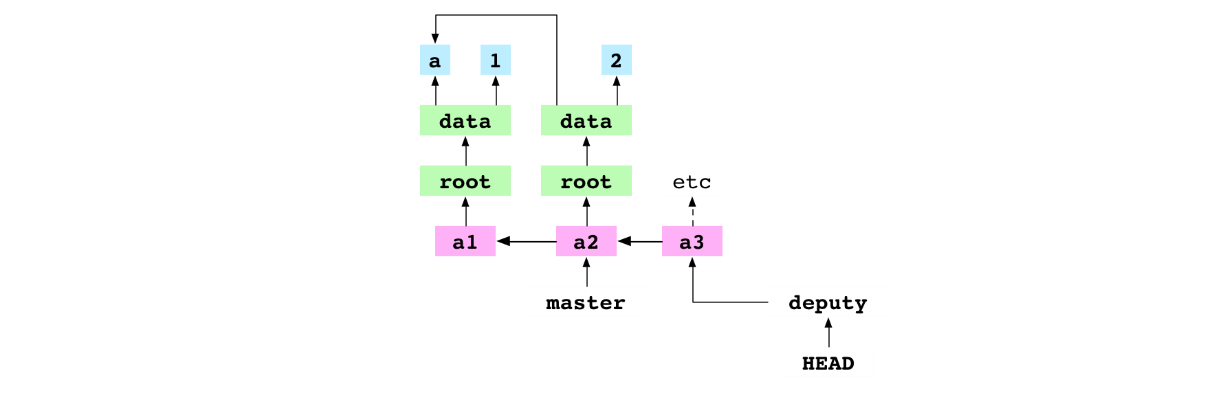
git checkout 37888c2

### Create a branch

git branch deputy

The user creates a new branch called deputy. This just creates a new file at .git/refs/heads/deputy that contains the hash that HEAD is pointing at: the hash of the a3 commit.

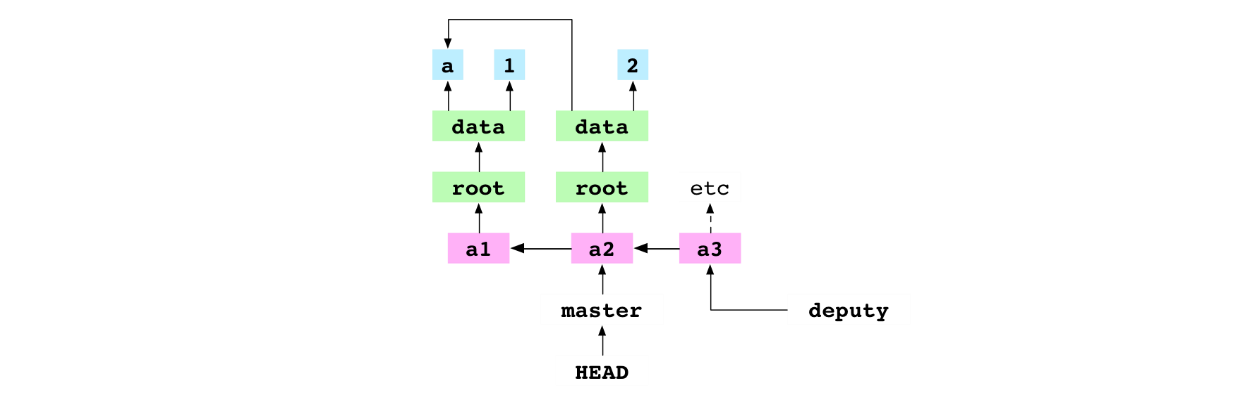
### Merge an ancestor



git merge master

Merging two branches means merging two commits. The first commit is the one that deputy points at: the receiver. The second commit is the one that master points at: the giver. For this merge, Git does nothing. It reports it is Already up-to-date.

### Merge a descendent



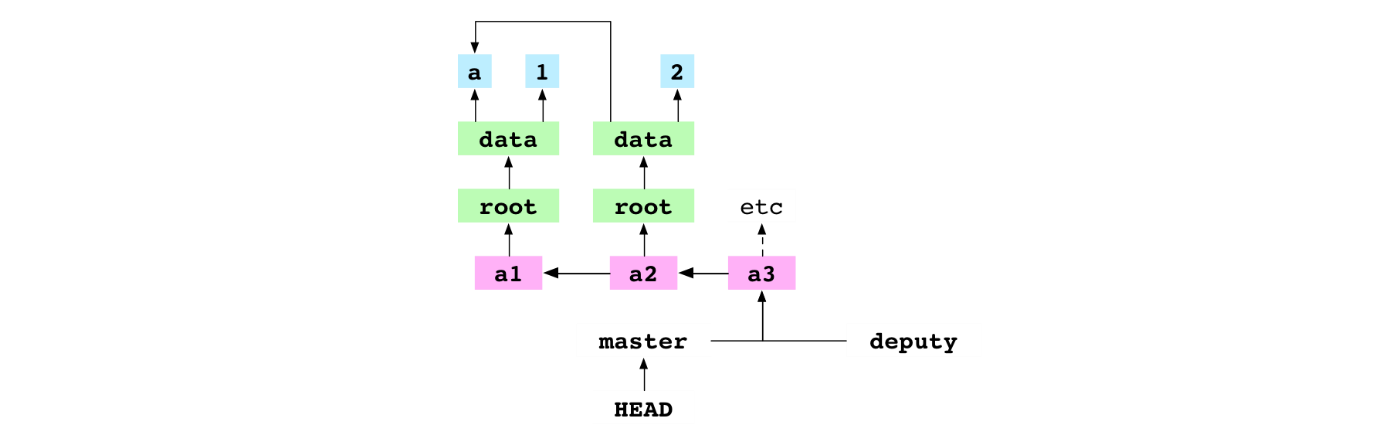
master checked out and pointing at the a2 commit

git checkout master

git merge deputy

They merge deputy into master. Git discovers that the receiver commit, a2, is an ancestor of the giver commit, a3. It can do a fast-forward merge.

It gets the giver commit and gets the tree graph that it points at. It writes the file entries in the tree graph to the working copy and the index. It “fast-forwards” master to point at a3.



a3 commit from deputy fast-forward merged into master

### Merge two commits from different lineages

$ printf '4' > data/number.txt

$ git add data/number.txt

$ git commit -m 'a4'

The user sets the content of number.txt to 4 and commits the change to master.

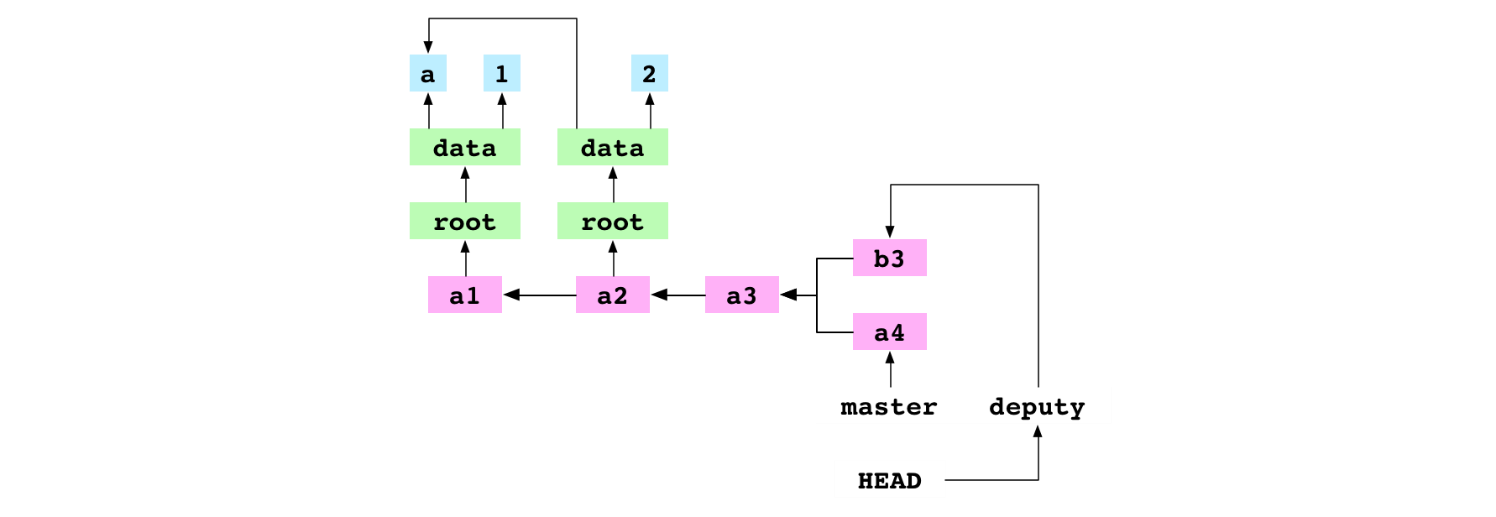
$ git checkout deputy

$ printf 'b' > data/letter.txt

$ git add data/letter.txt

$ git commit -m 'b3'

The user checks out deputy. They set the content of data/letter.txt to b and commit the change to deputy.



a4 committed to master, b3 committed to deputy and deputy checked out

The user merges master into deputy.

git merge master -m 'b4'

Git discovers that the receiver, b3, and the given, a4, are in different lineages. It makes a merge commit. This process has eight steps.

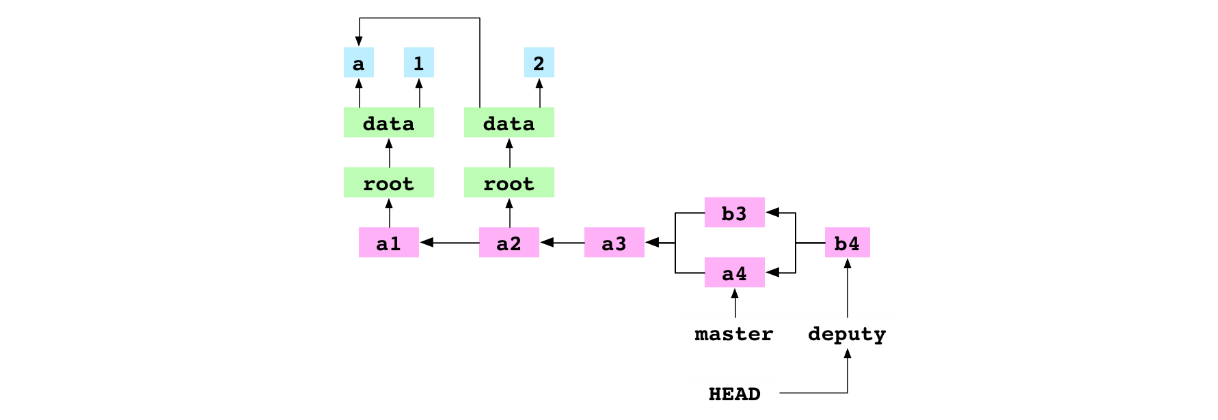
* First, Git writes the hash of the giver commit to a file at alpha/.git/MERGE\_HEAD. The presence of this file tells Git it is in the middle of merging.
* Second, Git finds the base commit: the most recent ancestor that the receiver and giver commits have in common.
* Third, Git generates the indices for the base, receiver and giver commits from their tree graphs.
* Fourth, Git generates a diff that combines the changes made to the base by the receiver commit and the giver commit. This diff is a list of file paths that point to a change: add, remove, modify or conflict.

Git gets the list of all the files that appear in the base, receiver or giver indices. For each one, it compares the index entries to decide the change to make to the file. It writes a corresponding entry to the diff. In this case, the diff has two entries.

The first entry is for data/letter.txt. The content of this file is a in the base, b in the receiver and a in the giver. The content is different in the base and receiver. But it is the same in the base and giver. Git sees that the content was modified by the receiver, but not the giver. The diff entry for data/letter.txt is a modification, not a conflict.

The second entry in the diff is for data/number.txt. In this case, the content is the same in the base and receiver, and different in the giver. The diff entry for data/letter.txt is also a modification.

* Fifth, the changes indicated by the entries in the diff are applied to the working copy. The content of data/letter.txt is set to b and the content of data/number.txt is set to 4.
* Sixth, the changes indicated by the entries in the diff are applied to the index. The entry for data/letter.txt is pointed at the b blob and the entry for data/number.txt is pointed at the 4 blob
* Seventh, the updated index is committed
* Eighth, Git points the current branch, deputy, at the new commit.



Notice that the commit has two parents.

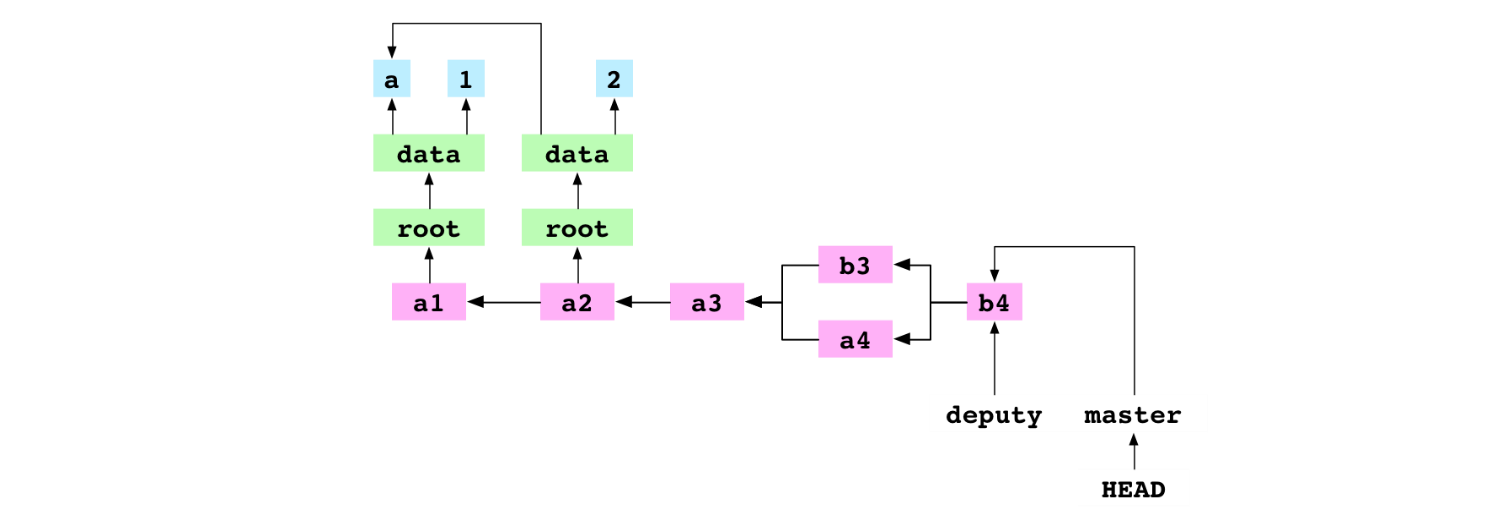
b4, the merge commit resulting from the recursive merge of a4 into b3

### Merge two commits from different lineages that both modify the same file

$ git checkout master

$ git merge deputy

The user checks out master. They merge deputy into master. This fast-forwards master to the b4 commit. master and deputy now point at the same commit.



deputy merged into master to bring master up to the latest commit, b4

The user checks out deputy. They set the content of data/number.txt to 5 and commit the change to deputy.

$ git checkout deputy

$ printf '5' > data/number.txt

$ git add data/number.txt

$ git commit -m 'b5'

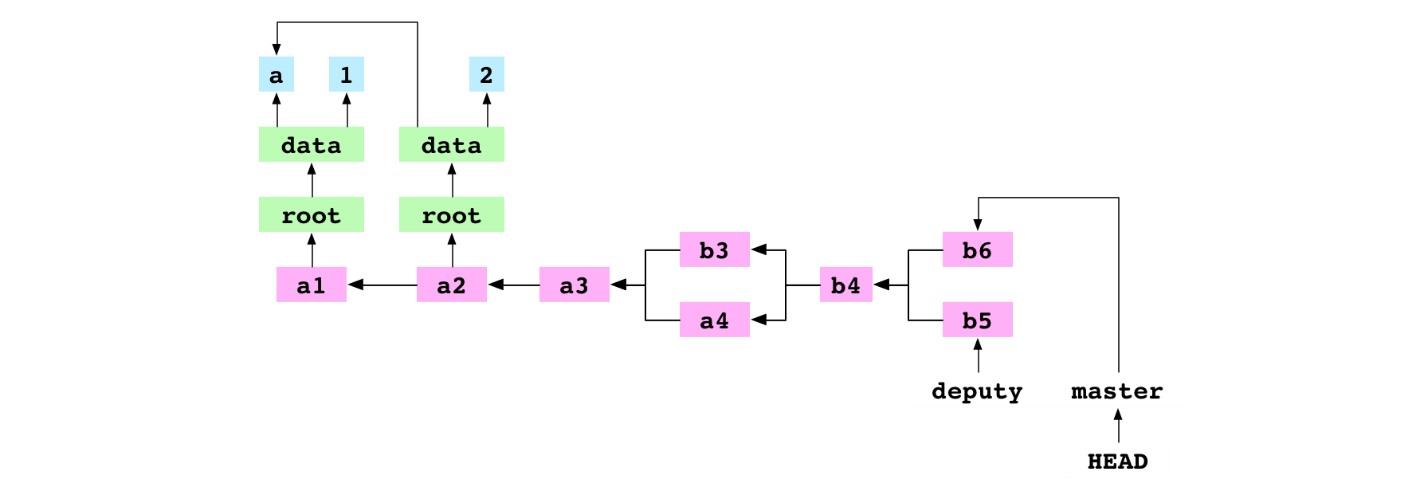
The user checks out master. They set the content of data/number.txt to 6 and commit the change to master.

$ git checkout master

$ printf '6' > data/number.txt

$ git add data/number.txt

$ git commit -m 'b6'



b5 commit on deputy and b6 commit on master

$ git merge deputy

The user merges deputy into master. There is a conflict and the merge is paused.

The user merges deputy into master. There is a conflict and the merge is paused. The process for a conflicted merge follows the same first six steps as the process for an unconflicted merge:

set .git/MERGE\_HEAD,

find the base commit,

generate the indices of the base, receiver and giver commits,

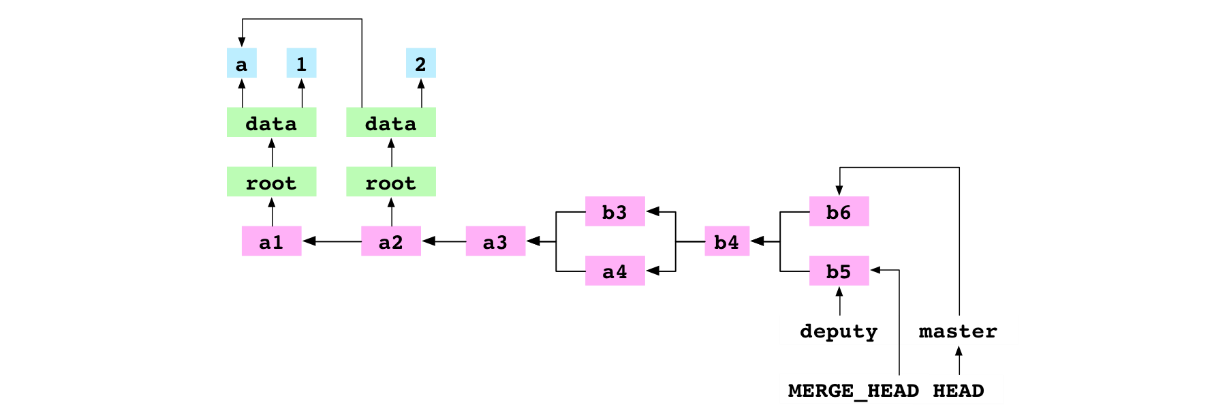
create a diff,

update the working copy

update the index.

Because of the conflict, the seventh commit step and eighth ref update step are never taken. Let’s go through the steps again and see what happens.

First, Git writes the hash of the giver commit to a file at .git/MERGE\_HEAD



MERGE\_HEAD written during merge of b5 into b6

Second, Git finds the base commit, b4.

Third, Git generates the indices for the base, receiver and giver commits.

Fourth, Git generates a diff that combines the changes made to the base by the receiver commit and the giver commit. This diff is a list of file paths that point to a change: add, remove, modify or conflict.

In this case, the diff contains only one entry: data/number.txt. The entry is marked as a conflict because the content for data/number.txt is different in the receiver, giver and base.

Fifth, the changes indicated by the entries in the diff are applied to the working copy. For a conflicted area, Git writes both versions to the file in the working copy. The content of data/number.txt is set to:

<<<<<<< HEAD

6

=======

5

>>>>>>> deputy

Sixth, the changes indicated by the entries in the diff are applied to the index. Entries in the index are uniquely identified by a combination of their file path and stage. The entry for an unconflicted file has a stage of 0. Before this merge, the index looked like this, where the 0s are stage values:

0 data/letter.txt 63d8dbd40c23542e740659a7168a0ce3138ea748

0 data/number.txt 62f9457511f879886bb7728c986fe10b0ece6bcb

After the merge diff is written to the index, the index looks like this:

0 data/letter.txt 63d8dbd40c23542e740659a7168a0ce3138ea748

1 data/number.txt bf0d87ab1b2b0ec1a11a3973d2845b42413d9767

2 data/number.txt 62f9457511f879886bb7728c986fe10b0ece6bcb

3 data/number.txt 7813681f5b41c028345ca62a2be376bae70b7f61

he entry for data/letter.txt at stage 0 is the same as it was before the merge. The entry for data/number.txt at stage 0 is gone. There are three new entries in its place. The entry for stage 1 has the hash of the base data/number.txt content. The entry for stage 2 has the hash of the receiver data/number.txt content. The entry for stage 3 has the hash of the giver data/number.txt content. The presence of these three entries tells Git that data/number.txt is in conflict.

The merge pauses

~/alpha $ printf '11' > data/number.txt

~/alpha $ git add data/number.txt

The user integrates the content of the two conflicting versions by setting the content of data/number.txt to 11. They add the file to the index. Git adds a blob containing 11. Adding a conflicted file tells Git that the conflict is resolved. Git removes the data/number.txt entries for stages 1, 2 and 3 from the index. It adds an entry for data/number.txt at stage 0 with the hash of the new blob. The index now reads:

0 data/letter.txt 63d8dbd40c23542e740659a7168a0ce3138ea748

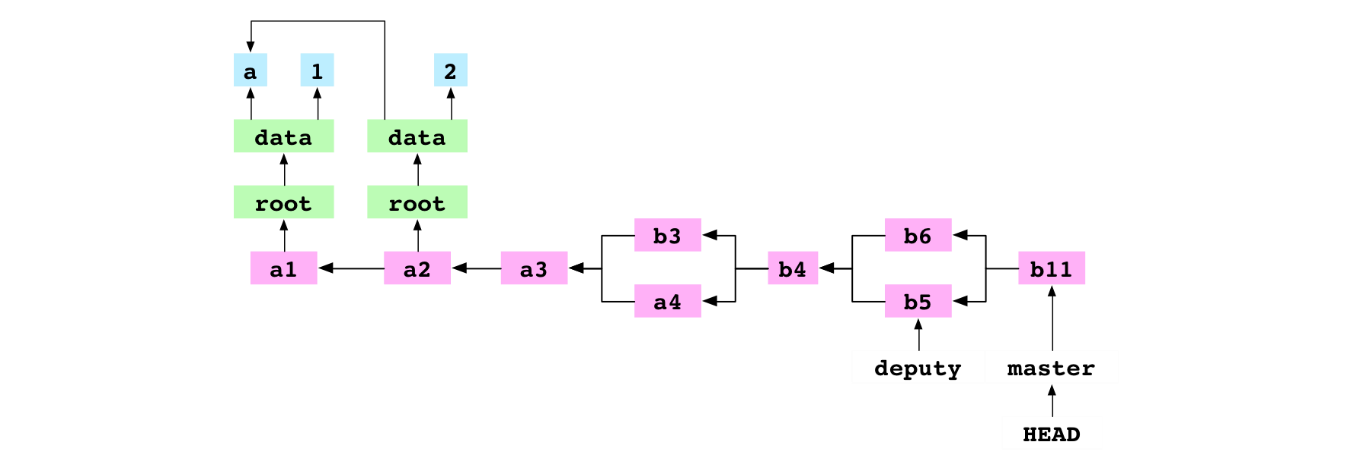
0 data/number.txt 9d607966b721abde8931ddd052181fae905db503

Seventh, the user commits.

git commit -m 'b11'

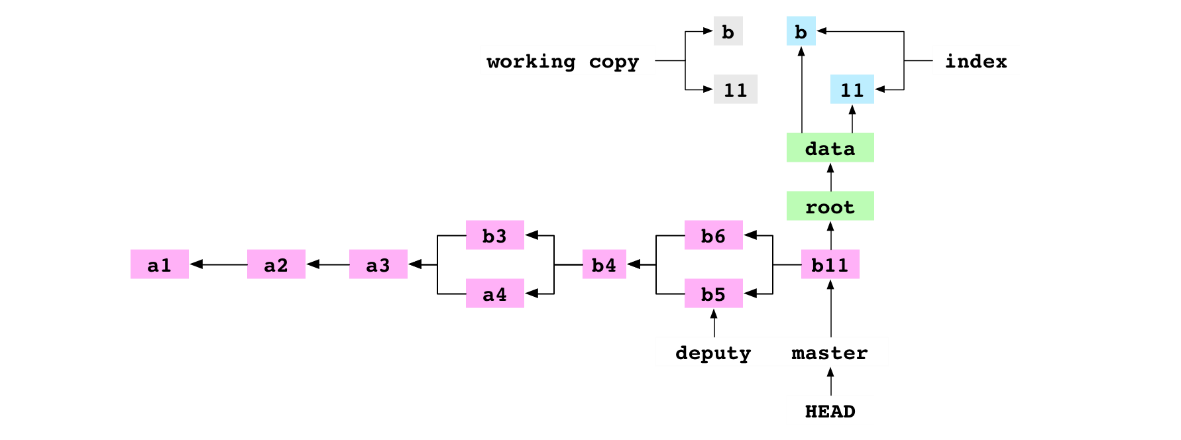
Git sees .git/MERGE\_HEAD in the repository, which tells it that a merge is in progress. It checks the index and finds there are no conflicts. It creates a new commit, b11, to record the content of the resolved merge. It deletes the file at .git/MERGE\_HEAD. This completes the merge.

Eighth, Git points the current branch, master, at the new commit



b11, the merge commit resulting from the conflicted, recursive merge of b5 into b6

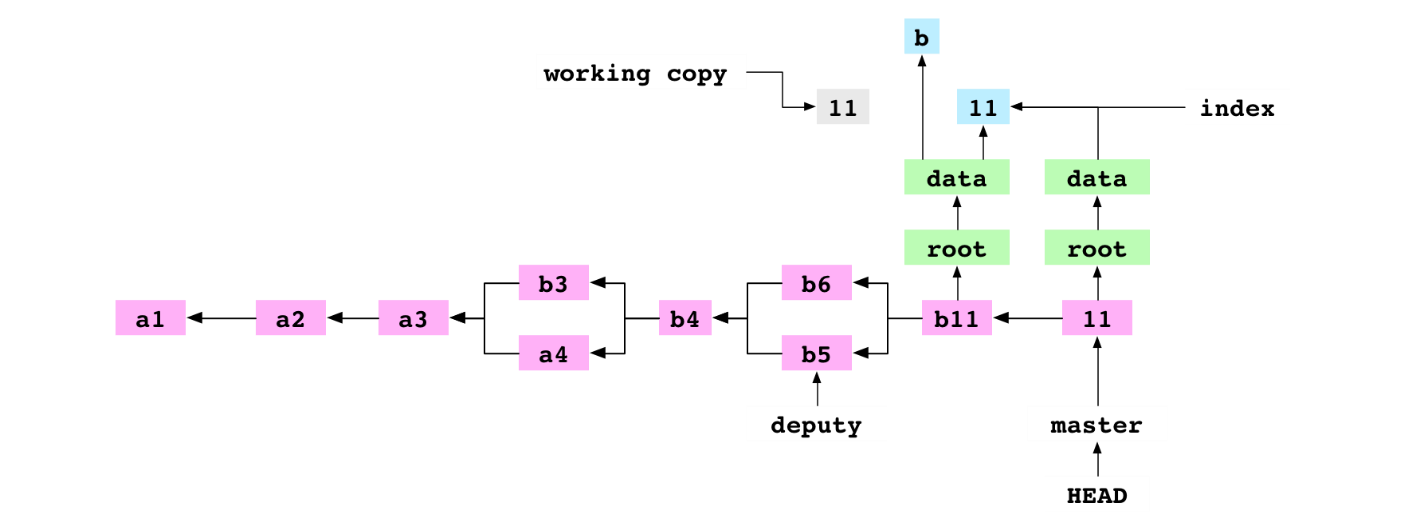
### Remove a file



git rm data/letter.txt

The user tells Git to remove data/letter.txt. The file is deleted from the working copy. The entry is deleted from the index

git commit -m '11'



11 commit made after data/letter.txt rm ed

### Copy a repository

~/alpha $ cd ..

~ $ cp -R alpha bravo

The user copies the contents of the alpha/ repository to the bravo/ directory. This produces the following directory structure:

├── alpha

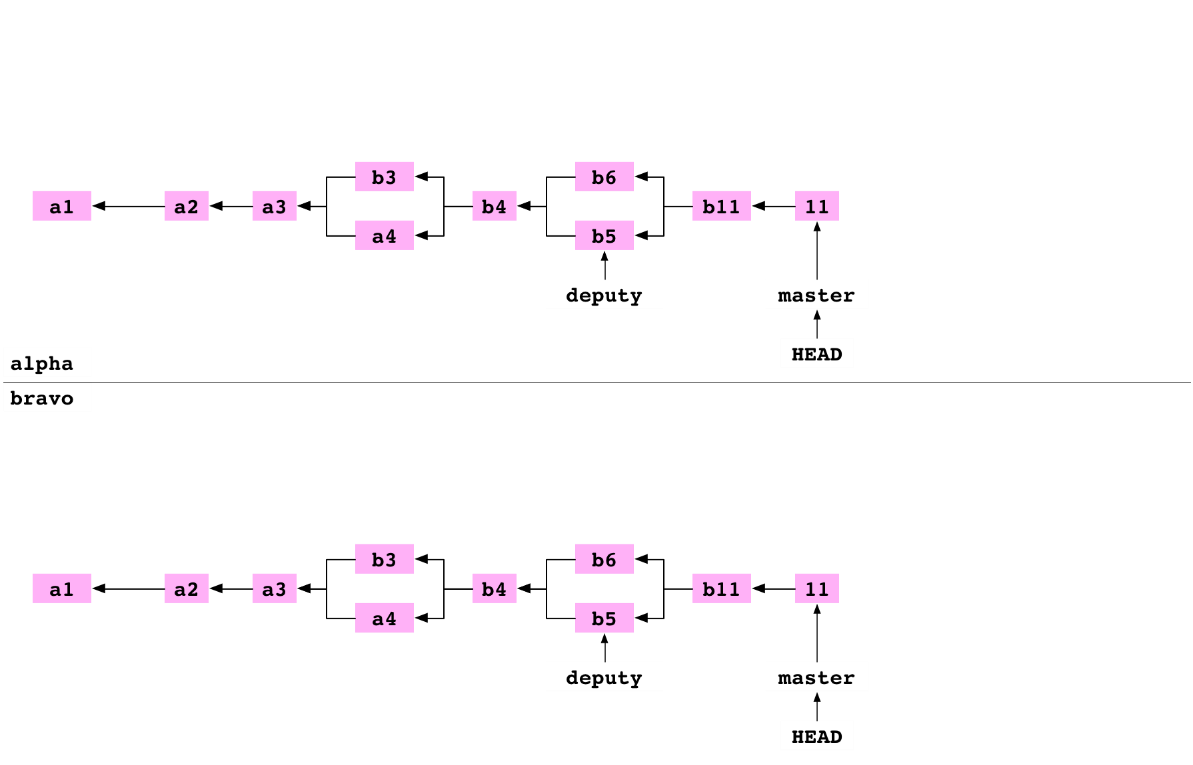
| └── data

| └── number.txt

└── bravo

└── data

└── number.txt



New graph created when alpha cp ed to bravo

### Link a repository to another repository

~ $ cd alpha

~/alpha $ git remote add bravo ../bravo

The user moves back into the alpha repository. They set up bravo as a remote repository on alpha. This adds some lines to the file at alpha/.git/config:

[remote "bravo"]

url = ../bravo/

These lines specify that there is a remote repository called bravo in the directory at ../bravo.

### Fetch a branch from a remote

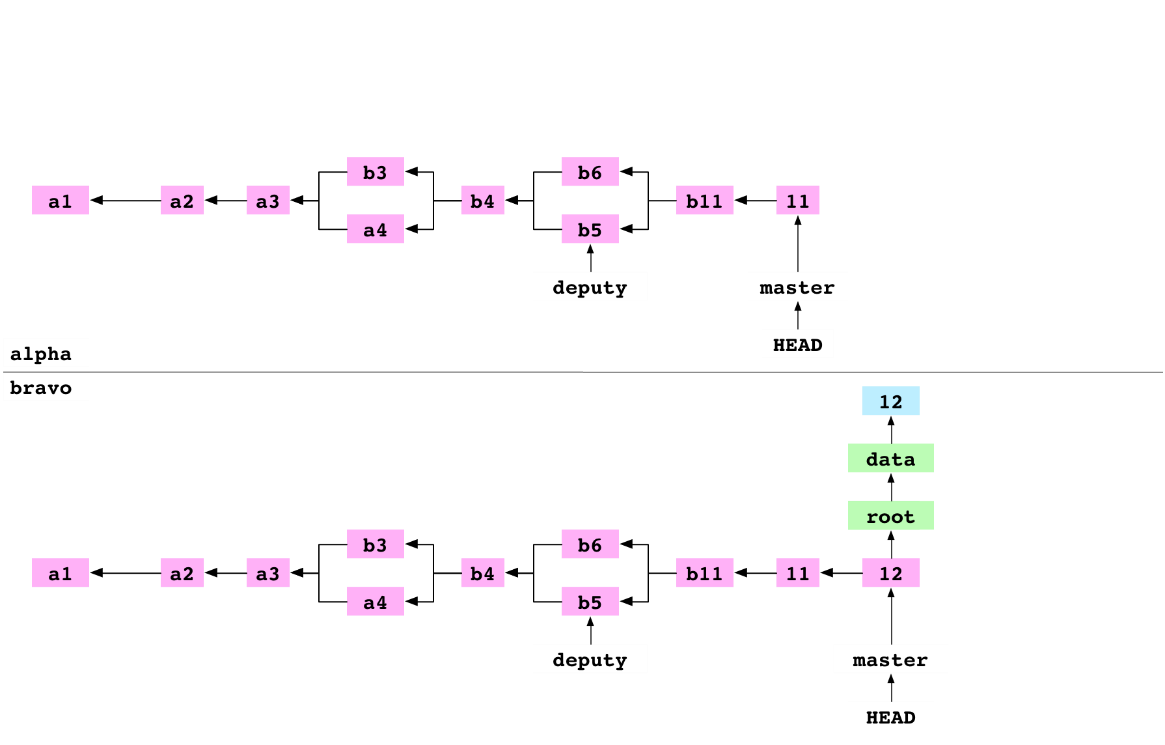
~/alpha $ cd ../bravo

~/bravo $ printf '12' > data/number.txt

~/bravo $ git add data/number.txt

~/bravo $ git commit -m '12'

The user goes into the bravo repository. They set the content of data/number.txt to 12 and commit the change to master on bravo.



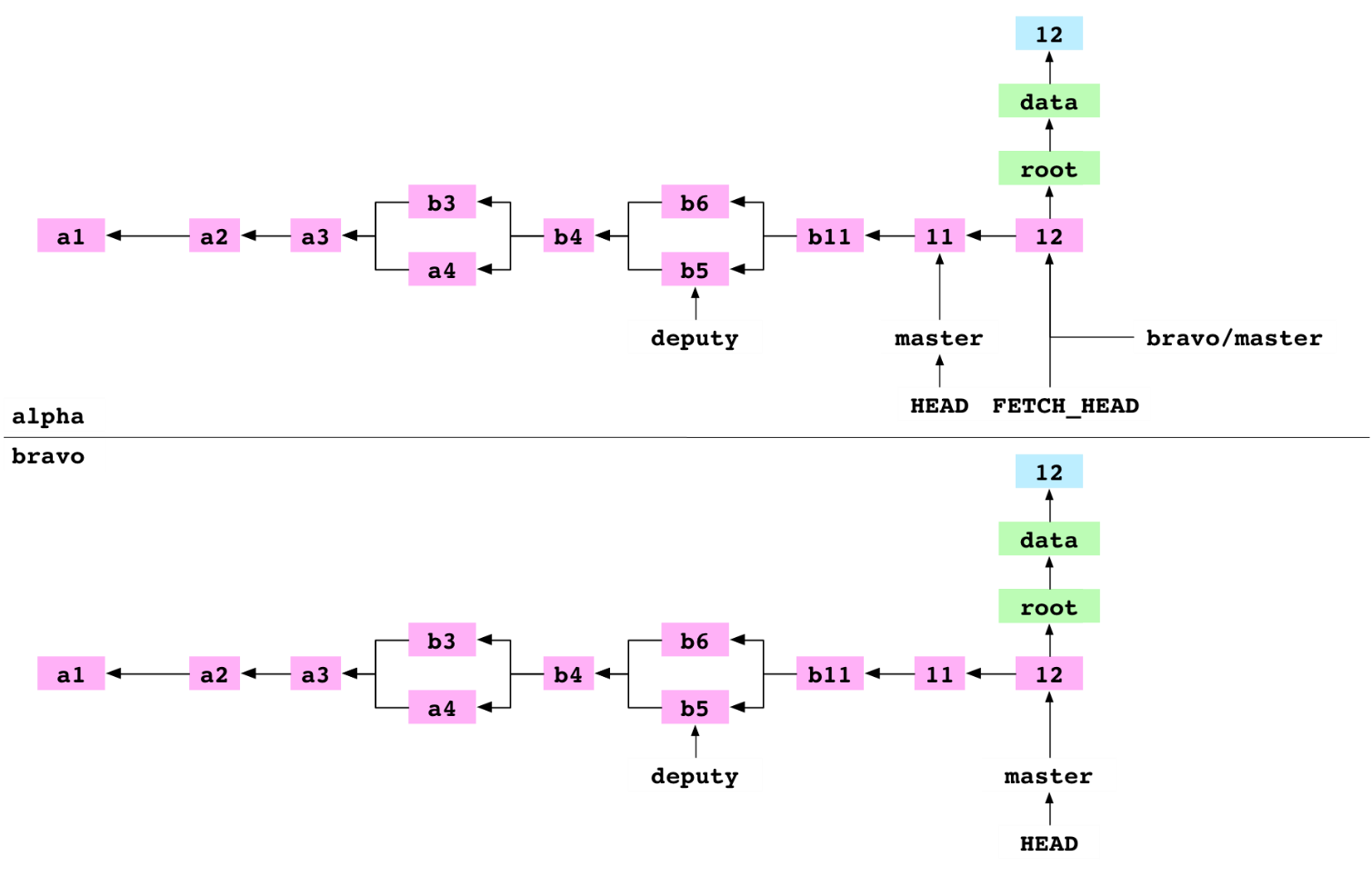
~/bravo $ cd ../alpha

~/alpha $ git fetch bravo master

The user goes into the alpha repository. They fetch master from bravo into alpha. This process has four steps.

* First, Git gets the hash of the commit that master is pointing at on bravo. This is the hash of the 12 commit.
* Second, Git makes a list of all the objects that the 12 commit depends on: the commit object itself, the objects in its tree graph, the ancestor commits of the 12 commit and the objects in their tree graphs. It removes from this list any objects that the alpha object database already has. It copies the rest to alpha/.git/objects/.
* Third, the content of the concrete ref file at alpha/.git/refs/remotes/bravo/master is set to the hash of the 12 commit.
* Fourth, the content of alpha/.git/FETCH\_HEAD is set to:

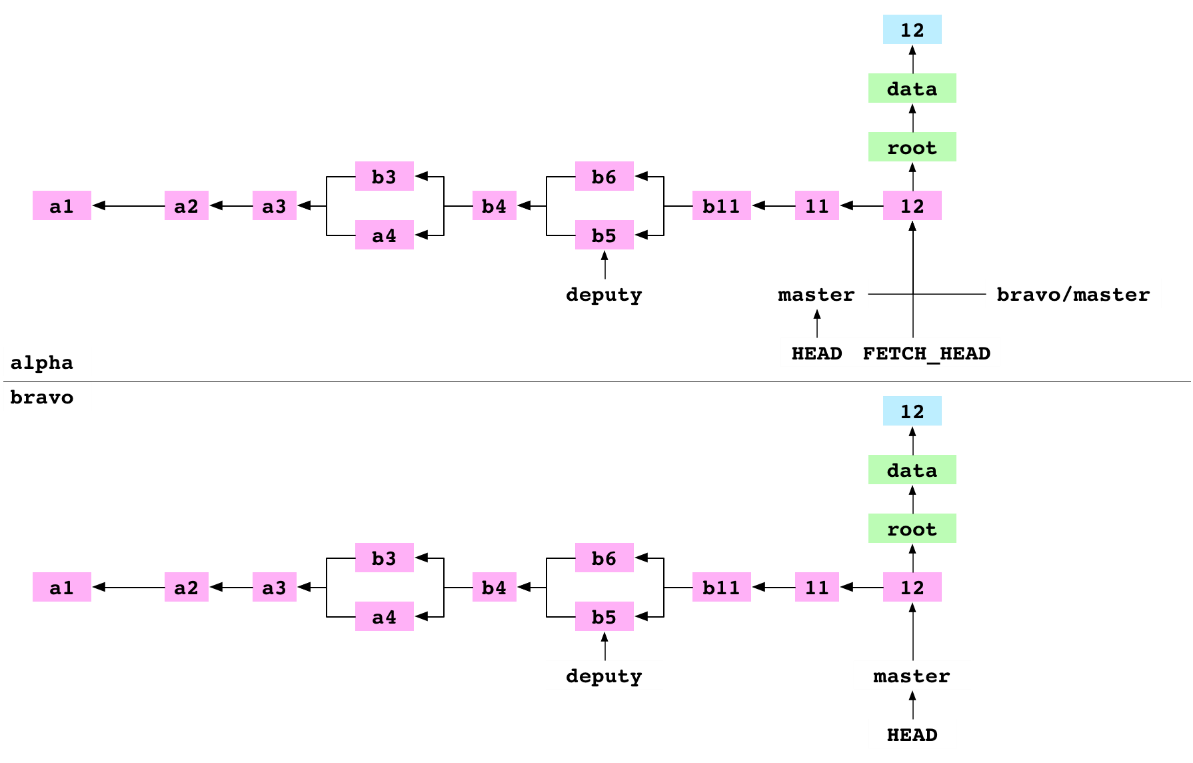
This indicates that the most recent fetch command fetched the 12 commit of master from bravo.



### Merge FETCH\_HEAD

git merge FETCH\_HEAD

The user merges FETCH\_HEAD. FETCH\_HEAD is just another ref. It resolves to the 12 commit, the giver. HEAD points at the 11 commit, the receiver. Git does a fast-forward merge and points master at the 12 commit.



alpha after FETCH\_HEAD merged

### Pull a branch from a remote

git pull bravo master

The user pulls master from bravo into alpha. Pull is shorthand for “fetch and merge FETCH\_HEAD”. Git does these two commands and reports that master is Already up-to-date

### Clone a repository

~/alpha $ cd ..

~ $ git clone alpha charlie

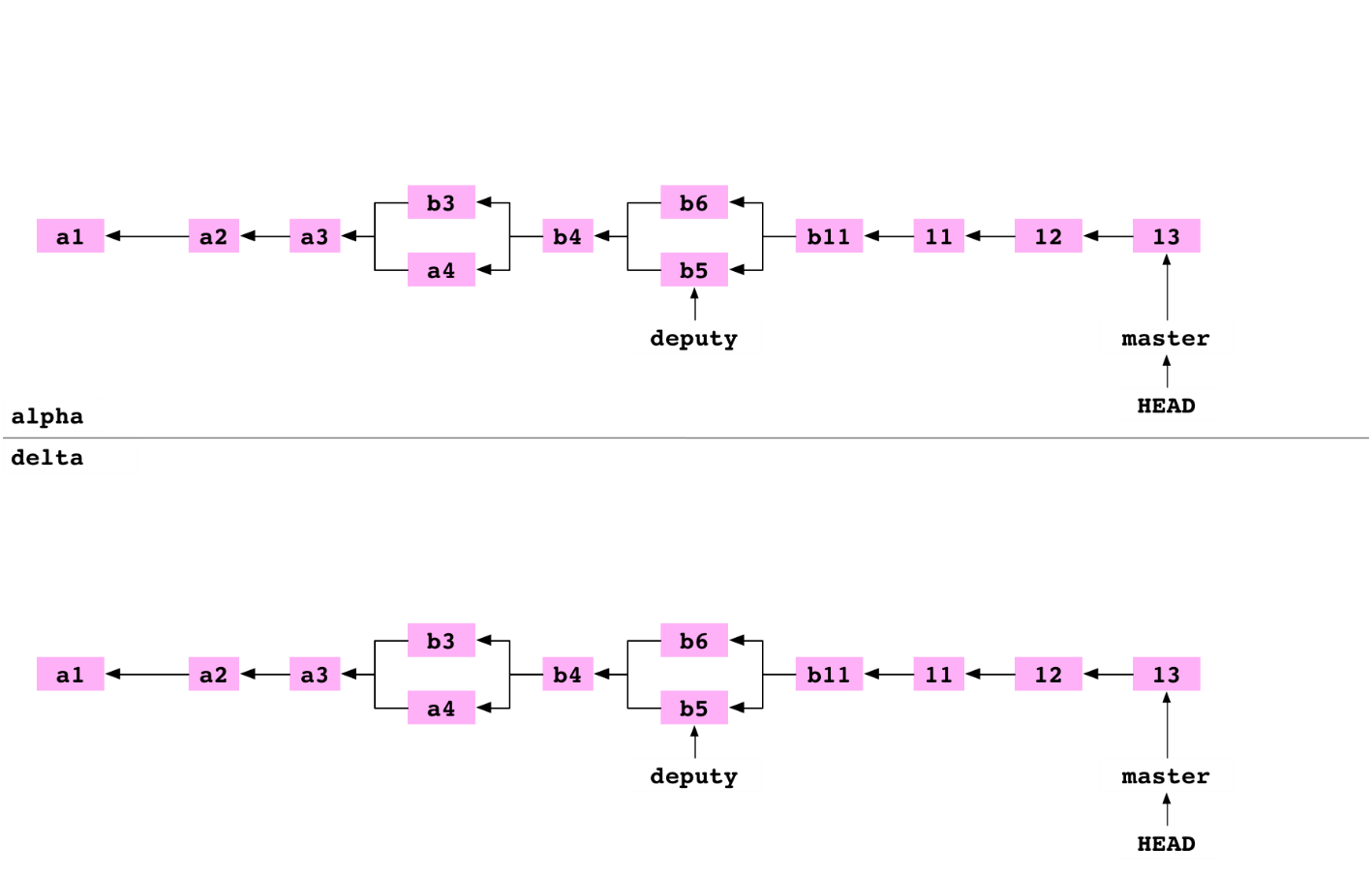
The user moves into the directory above. They clone alpha to charlie. Cloning to charlie has similar results to the cp the user did to produce the bravo repository. Git creates a new directory called charlie. It inits charlie as a Git repo, adds alpha as a remote called origin, fetches origin and merges FETCH\_HEAD.

### Clone a bare repository

/alpha $ cd ..

~ $ git clone alpha delta --bare

Cloning into bare repository 'delta'



The user moves into the directory above. They clone delta as a bare repository. This is an ordinary clone with two differences. The config file indicates that the repository is bare. And the files that are normally stored in the .git directory are stored in the root of the repository:

### Push a branch to a bare repository

The user goes back into the alpha repository. They set up delta as a remote repository on alpha.

~ $ cd alpha

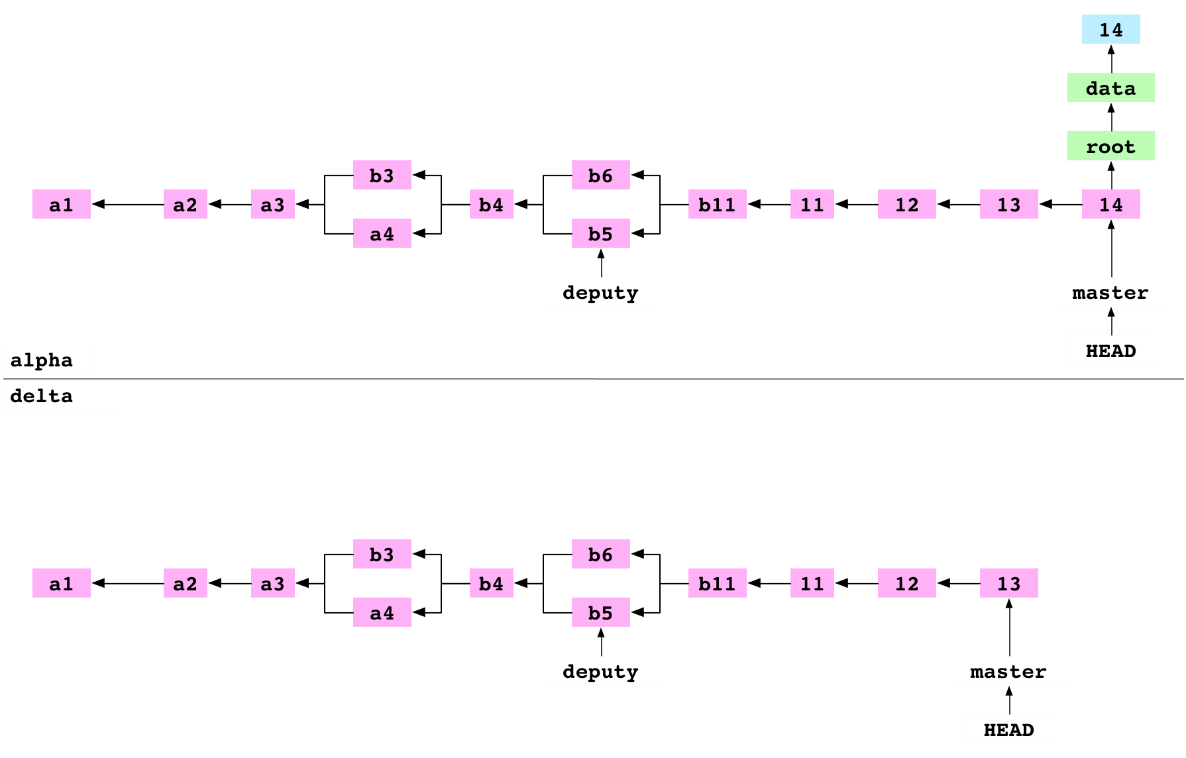
~/alpha $ git remote add delta ../delta

They set the content of data/number.txt to 14 and commit the change to master on alpha.

~/alpha $ printf '14' > data/number.txt

~/alpha $ git add data/number.txt

~/alpha $ git commit -m '14'



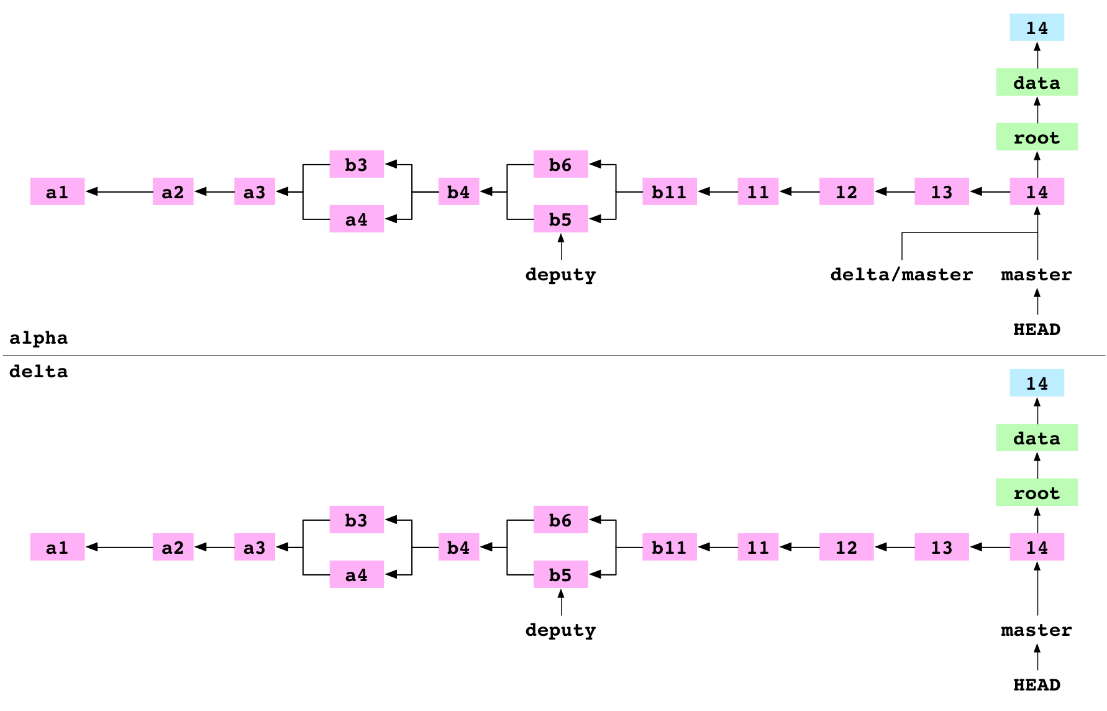
git push delta master

They push master to delta. Pushing has three steps.

First, all the objects required for the 14 commit on the master branch are copied from alpha/.git/objects/ to delta/objects/.

Second, delta/refs/heads/master is updated to point at the 14 commit.

Third, alpha/.git/refs/remotes/delta/master is set to point at the 14 commit. alpha has an up-to-date record of the state of delta.



***Git Object Types***

Git objects are the actual data of Git, the main thing that the repository is made up of.

All of these types of objects are stored in the Git Object Database, which is kept in the Git Directory. Each object is compressed (with Zlib) and referenced by the SHA-1 value of its contents plus a small header (SHA stands for Secure Hash Algorithm)

In Git, the contents of files are stored as blobs. It is important to note that it is the contents that are stored, not the files. The names and modes of the files are not stored with the blob, just the contents.

Directories in Git basically correspond to **trees**. A tree is a simple list of trees and blobs that the tree contains, along with the names and modes of those trees and blobs.

**The Commit**

The commit is very simple, much like the tree. It simply points to a tree and keeps an *author*, *committer*, *message* and any *parent* com­mits that directly preceded it.

The Tag

The final type of object you will find in a Git database is the **tag**. This is an object that provides a permanent shorthand name for a par­ticular commit.



The Git Data Model

the Git object data is a *directed acyclic graph.*

The cheap references I’ve represented as the grey boxes, the immutable objects are the colored round cornered boxes.



References

In addition to the Git objects, which are immutable – that is, they cannot ever be changed, there are references also stored in Git. Unlike the objects, references can constantly change. They are simple pointers to a particular commit, something like a tag, but eas­ily moveable.

A branch in Git is nothing more than a file in the .git/refs/heads/ directory that con­tains the SHA-1 of the most recent commit of that branch

In fact, in Git the act of creating a new branch is simply writing a file in the .git/refs/heads directory that has the SHA-1 of the last commit for that branch.

How does Git actually retrieve these objects in practice?

Well, it gets the initial SHA-1 of the starting commit object by looking in the .git/refs directory for the branch, tag or remote you specify. Then it tra­verses the objects by walking the trees one by one, checking out the blobs under the names listed.

In fact, in Git the act of creating a new branch is simply writing a file in the .git/refs/heads directory that has the SHA-1 of the last commit for that branch.

Switching to that branch simply means having Git make your work­ing directory look like the tree that SHA-1 points to and updating the HEAD file so each commit from that point on moves that branch pointer forward (

***The Treeish***

Besides branch heads, there are a number of shorthand ways to refer to particular objects in the Git data store. These are often referred to as a *treeish*. Any Git command that takes an object – be it a commit, tree or blob – as an argument can take one of these shorthand versions as well.

* Full SHA-1

dae86e1950b1277e545cee180551750029cfe735

* PARTIAL SHA-1

dae86e

the full SHA-1 can be referenced fine with the first 6 or 7 characters. Git is smart enough to figure out a partial SHA-1 as long as it’s unique.

* Branch or tag name

Anything in *.git/refs/heads* or *.git/refs/tags* can be used to refer to the commit it points to.

* date spec

master@{yesterday}

master@{1 month ago}

* ordinal spec

master@{5}

This indicates the 5th prior value of the master branch. Like the *Date Spec*, this depends on special files in the *.git/log* directory that are written during commits, and is specific to *your* repository

* Carrot parent

dae86e^N

this refers to the Nth parent of that commit. Only really helpful for commits that merged two or more commits

* Tilde spec

dae86e~N

refers to the Nth generation grandparent of that commit

dae86e~5 ⬄ dae86e^^^^^

* tree pointer

e65s46^{tree}

This points to the tree of that commit. Any time you add a ^{tree} to any commit-ish, it resolves to its tree.



* Tree pointer

dae86e^{tree}

This points to th tree of that commit

blob spec

master:/path/to/file

This is very helpful for referring to a blob under a particular commit or tree.

**Git repository**

When you initialize a Git repository, either by cloning an existing one or creating a new one, the first thing Git does is create a Git directory. This is the directory that stores all the object data, tags, branches, hooks and more. Everything that Git permanently stores goes in this single directory. When you clone someone else’s reposi­tory, it basically just copies the contents of this directory to your computer.

When you run git init to initialize your repository, the Git directory is by default installed in the directory you are currently in as .git. The Git directory for our little project looks something like this:

For now, let’s go over some of the more important contents of this directory.

* .git/config

This is the main Git configuration file. It keeps your project specific Git options, such as your remotes, push configurations, tracking branches and more.

* .git/index

This is the default location of the index file for your Git project.

* .git/objects/

This is the main directory that holds the data of your Git objects – that is, all the contents of the files you have ever checked in, plus your commit, tree and tag objects.

The files are stored by their SHA-1 values. The first two characters make up the subdirectory and the last 38 is the filename

* .git/refs/

This directory normally has three subdirectories in it – *heads*, *remotes* and *tags*. Each of these directories will hold files that correspond to your local branches, remote branches and tags, respectively

* ..git/HEAD

This file holds a reference to the branch you are currently on. This tells Git what to use as the parent of your next commit

* .git/hooks

Contains shell scripts that are invoked after the git command

**Working directory**

Your working directory is temporary – everything is stored permanently in your git repository. Your working directory is a just a copy of a tree so you can edit it and commit changes

The Index

The index was called the cache for a while, because that’s largely what it does. It is a staging area for changes that are made to files or trees that are not committed to your repository yet. It acts as sort of a middle ground between your working directory and your repository. When you run git commit, the resulting tree and commit object will be built based on the contents of the index.

Now that you *hopefully* understand what Git is designed to do at a fundamental level – how it tracks and stores content, how it stores branches and merges and tracks remote copies of the repository,