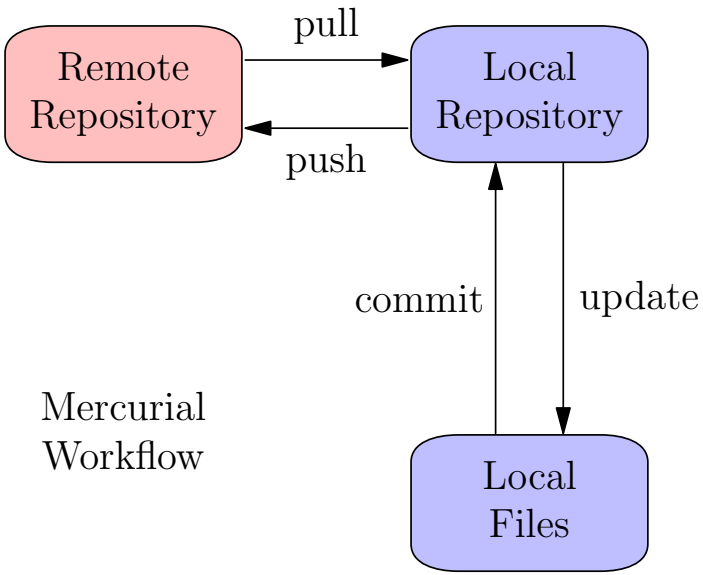
Introduction

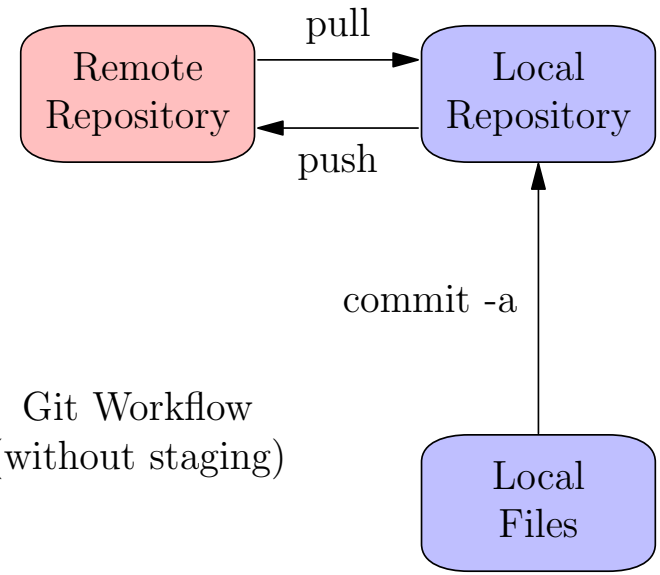
to Git

by Daniel Bump

Mercurial vs. Git Workflows



Mercurial Workflow

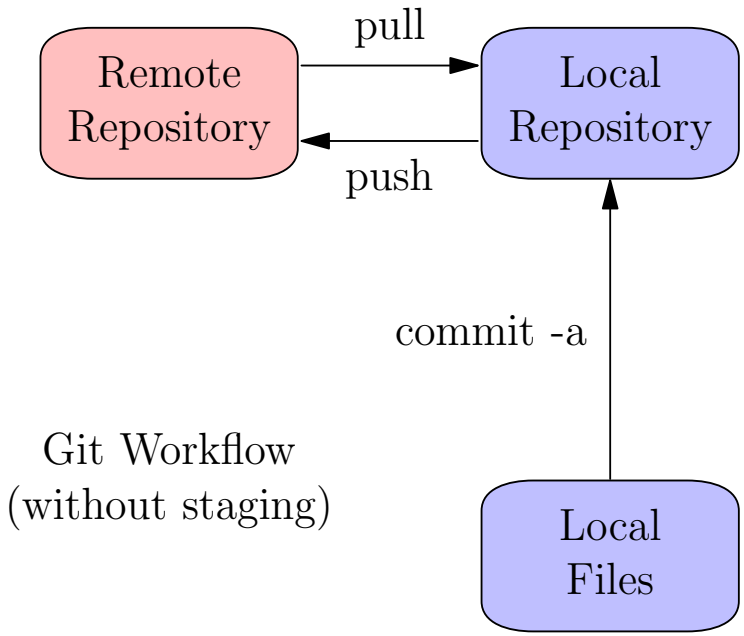


Git Workflow (without staging)

If we ignore Git's **staging** feature, a simple workflow is similar to **Mercurial**.

- A difference: local files are **updated automatically** on pulling with Git.
- If you don't want that, us `git fetch` instead of `git pull`.

Git without staging:

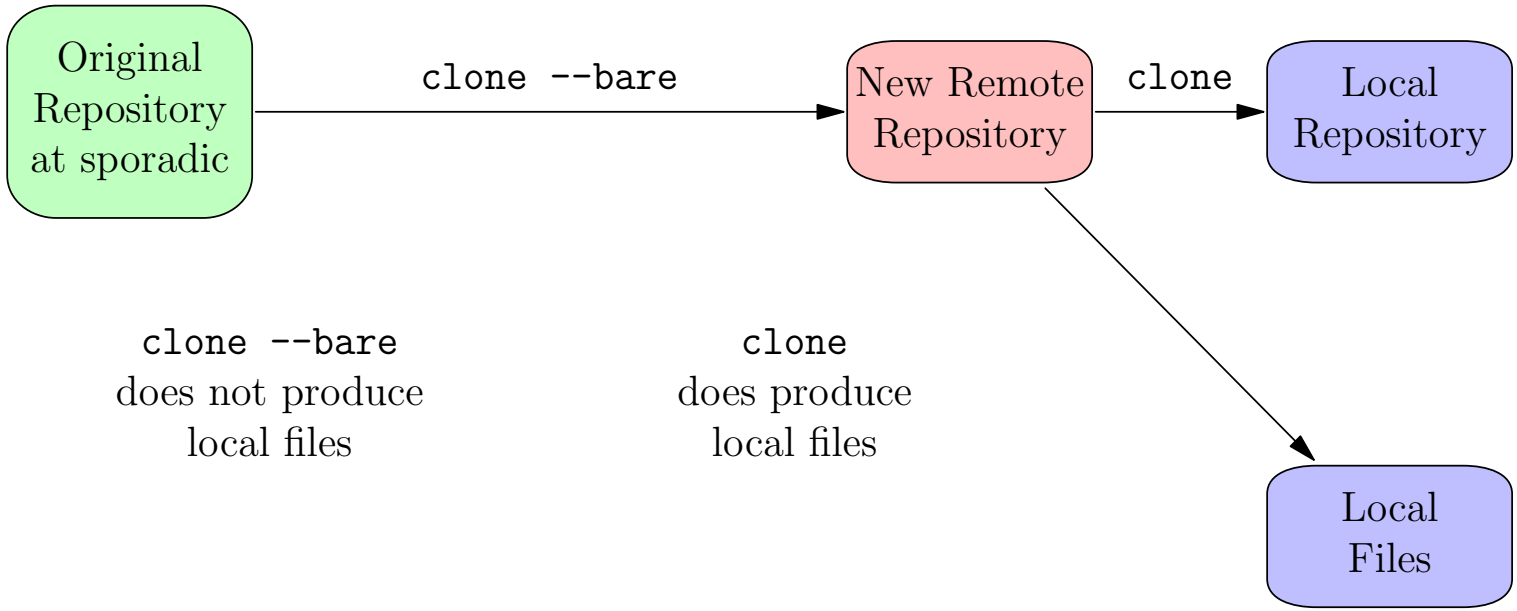


For this workflow, always use `commit` with the `-a` flag (**add**) in this workflow.

- Advantage: **simple** and similar to **mercurial**.
- Disadvantage: **staging** can help avoid half-finished commits. You lose this advantage.
- **You can follow along** on your own computer.

What we'll do.

- To get started quickly, we'll clone an **existing repository**.
- The cloned repository will be our new **remote repository**.
- We'll clone it again to get our new **local repository**.
- The **remote** and **local** repositories will be on the same computer (yours)
- This will simplify matters of permission.



The remote repository

- For simplicity, let's just clone an existing repository.
- Use `clone --bare`.
- A bare repository contains **git files** but not **working files.**

```
$ mkdir remote; cd remote; pwd
/home/bump/remote
$ git clone --bare http://sporadic.stanford.edu/pi.git
Cloning into bare repository 'pi.git'...
```

The Local Repository(s)

Now clone the "remote" repository to make the local repository.

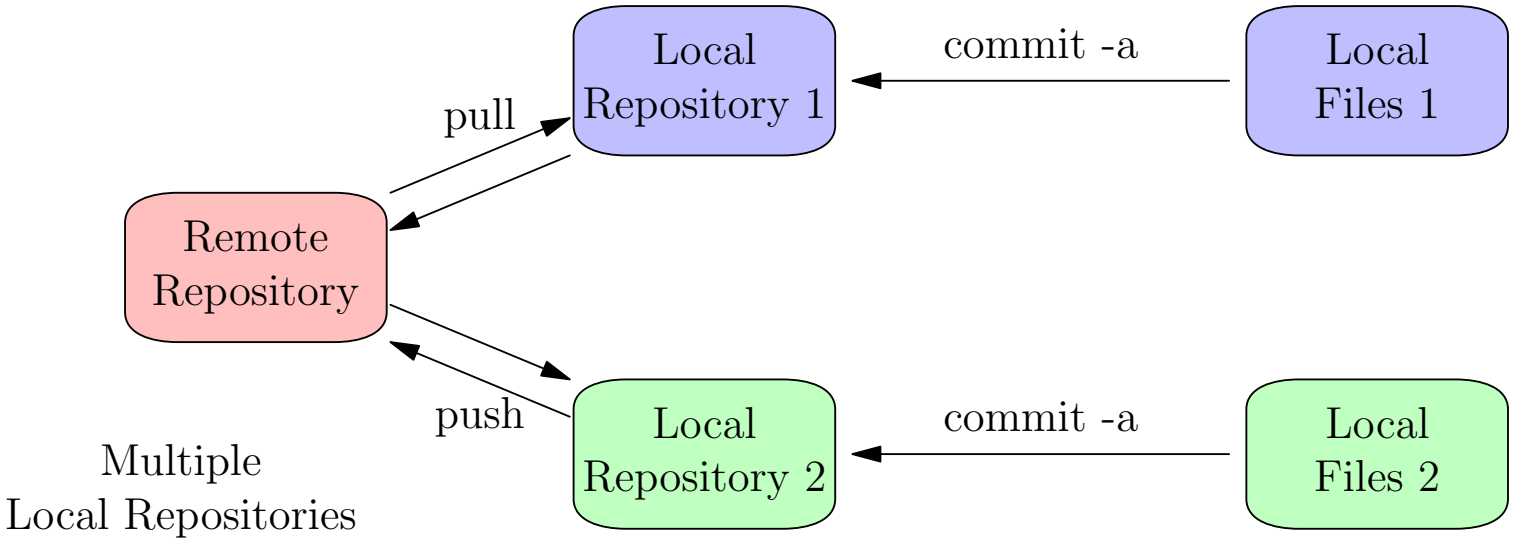
- Typically this would involve **ssh** or other internet protocol.
- But our "remote" repository is on the same computer, so not this time.
- Clone **without** the `--bare` option to get a working directory with files.
- The repository **git files** are in **.git** subdirectory (like Mercurial).

```
$ pwd
/home/bump
$ git clone /home/bump/remote/pi.git
Cloning into 'pi'...
done.
$ cd pi; pwd
/home/bump/pi
$ ls -a
. .. digits .git pi.txt rhymes
$ ls .git/
branches config description HEAD ...
```

1. `pull` from remote repository	**2. Work** on files in local working directory	**3.** `commit -a` to local repository (without staging)	**4.** `push` to remote repository

```
$ git pull
Already up-to-date.
```

Nothing has changed on remote, but if someone else had pushed a change, from **another local repository** pulling would update both the local repository and the local files. A merge conflict might need to be resolved.



Do some work.

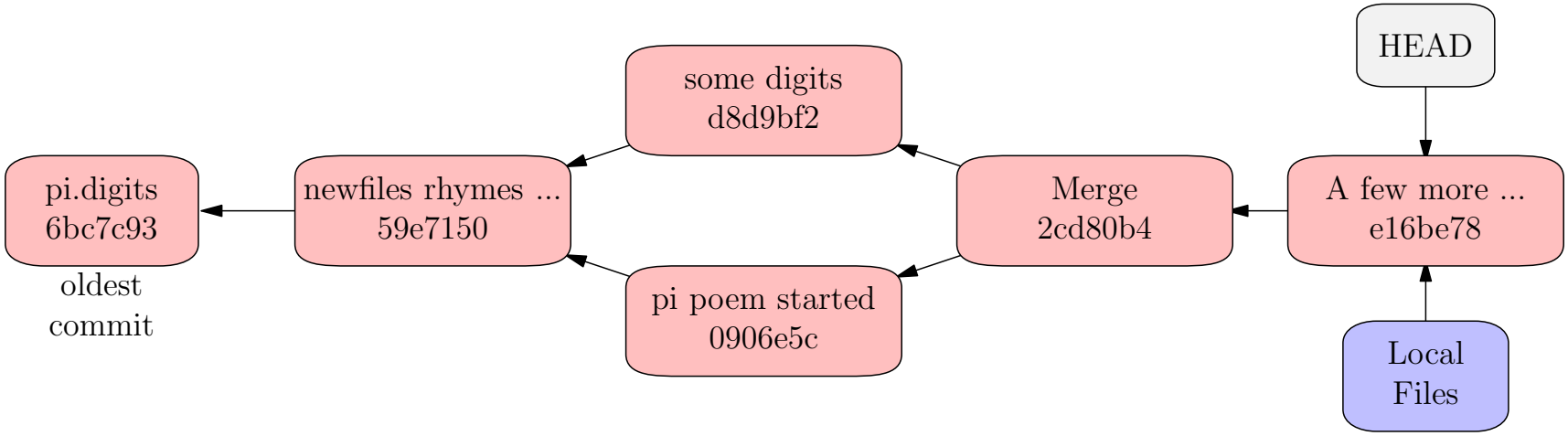
```
$ emacs pi.txt
$ git diff
diff --git a/pi.txt b/pi.txt
index 454909a..5f8cf88 100644
--- a/pi.txt
+++ b/pi.txt
@@ -1,5 +1,7 @@
-Memorize this little poem and you'll know
-ALL the digits of pi!
+                  PI
+
+Memorize this little poem and you'll know ALL the digits of pi!
+(But first we must finish writing the poem.)
Three point one four one five nine,
We eat pi all of the time.
```

Use `commit -a` to commit without staging

```
$ git commit -a -m "rewrote exordium"
```

The Repository

As with Mercurial, the repository contains a **directed acyclic graph (DAG)**, node a snapshot of the file system. Each node is created when you commit.



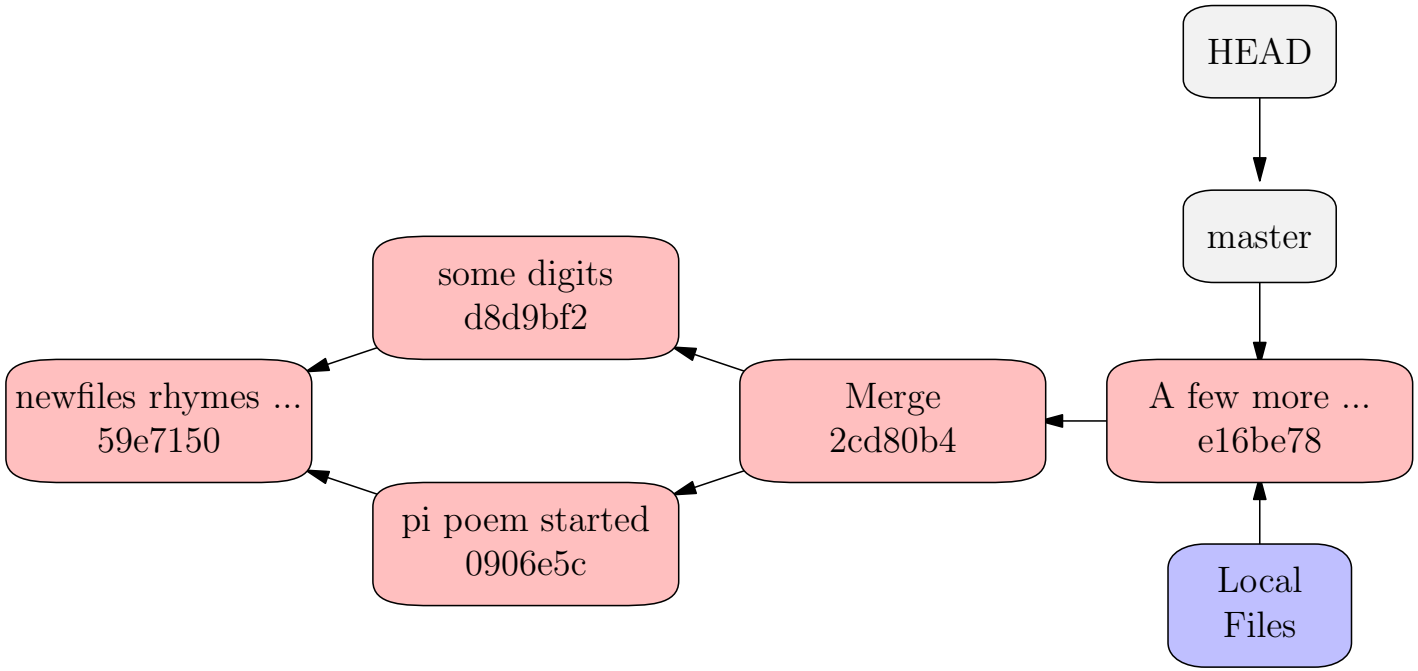
Commit objects get a **commit message** and a 39 hex digit **SHA1 hash**. Only the first few digits of the hash are needed to identify the node.

- **HEAD** points to the "current" node. If we commit, we attach a new node here. The local file system is derived from this snapshot.
- Use `gitk --all` to view the repository as a DAG.

Branches

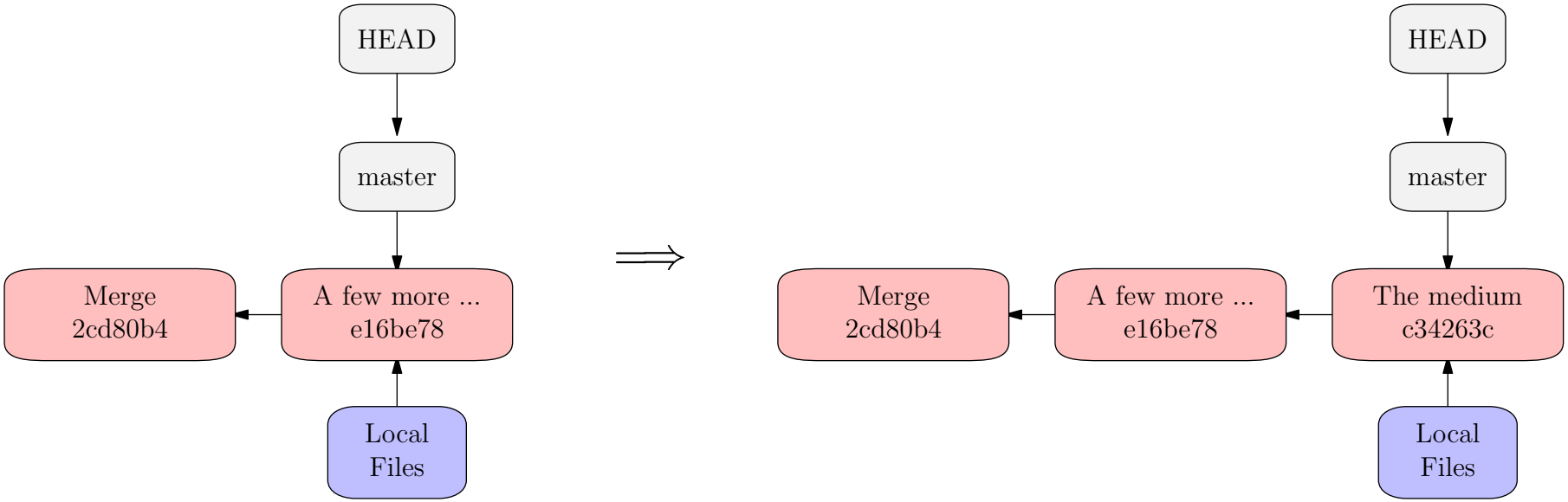
A **branch** is a pointer into the DAG. The "main" branch is called **master**.

The special pointer **HEAD** may point directly to a node. But usually it points to a branch which points to a node. The last picture was **incomplete** since it showed **HEAD** but not **master**. **More correctly**:



Committing

If **HEAD** points to a branch, say **master**, and we commit changes, a new node is created this point.

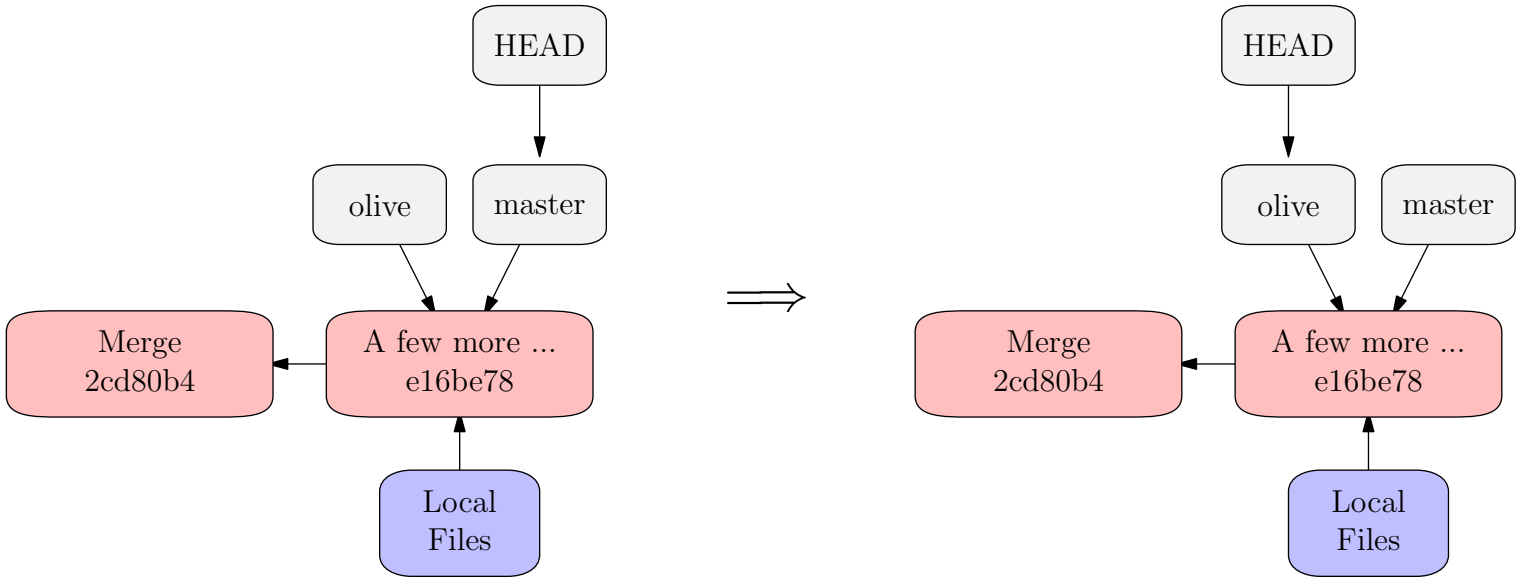



```
$ git commit -a -m "The medium"
```

git checkout

To switch to a different branch, use

```
$ git checkout [branch name].
```

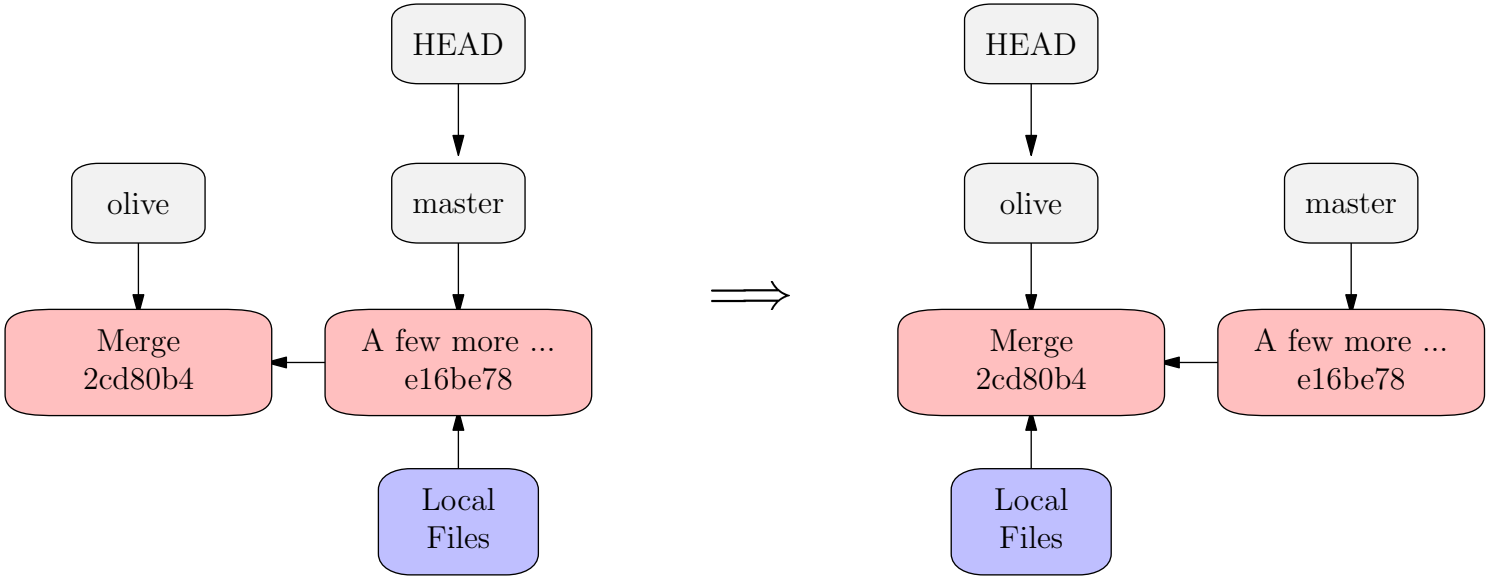



```
$ git checkout olive
```

git checkout

To switch to a different branch, use

```
$ git checkout [branch name].
```

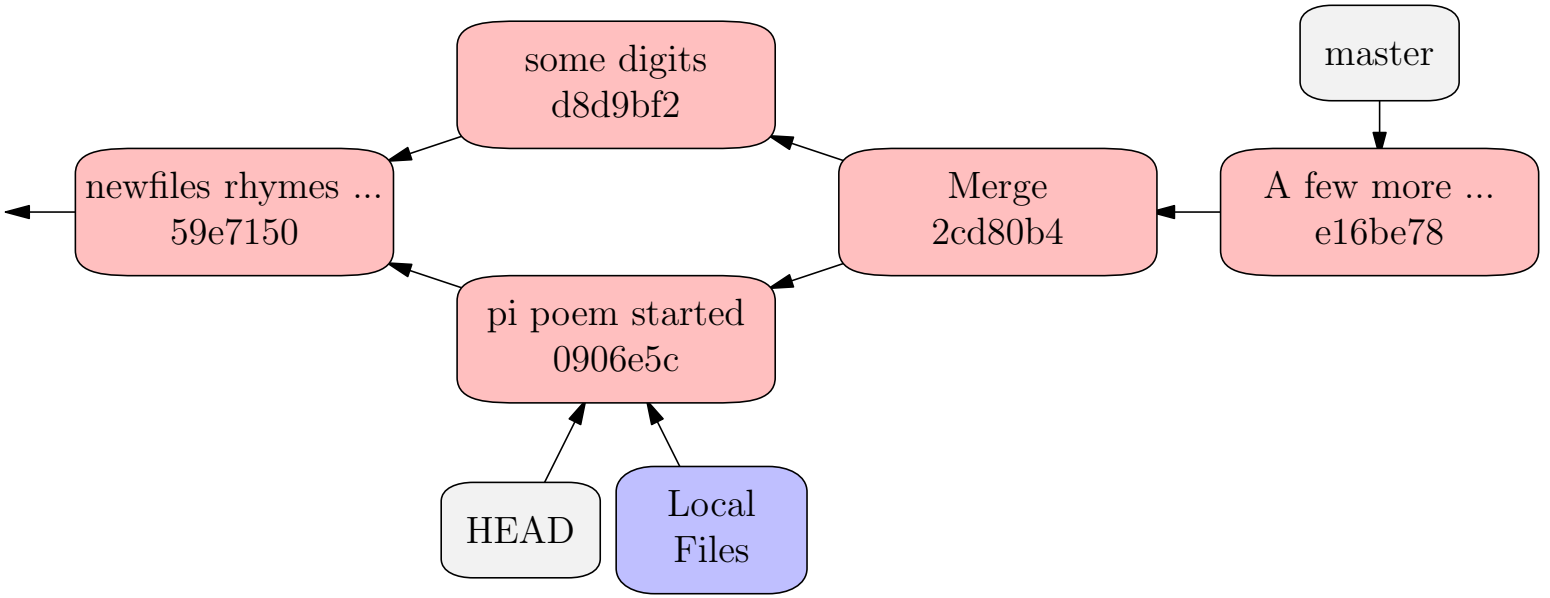



```
$ git checkout olive
```

HEAD can change the active snapshot. We recover the (**tracked**) local files.

Revisiting Old Snapshots

We can also `git checkout [SHA1 hash]`.
Moving HEAD the local filesystem will instantly change to an older version.




```
$ git checkout 0906e5c
```

The local filesystem has reverted to an earlier state. HEAD points to the node where a new node will be attached if you commit now. But ...

Detached HEAD

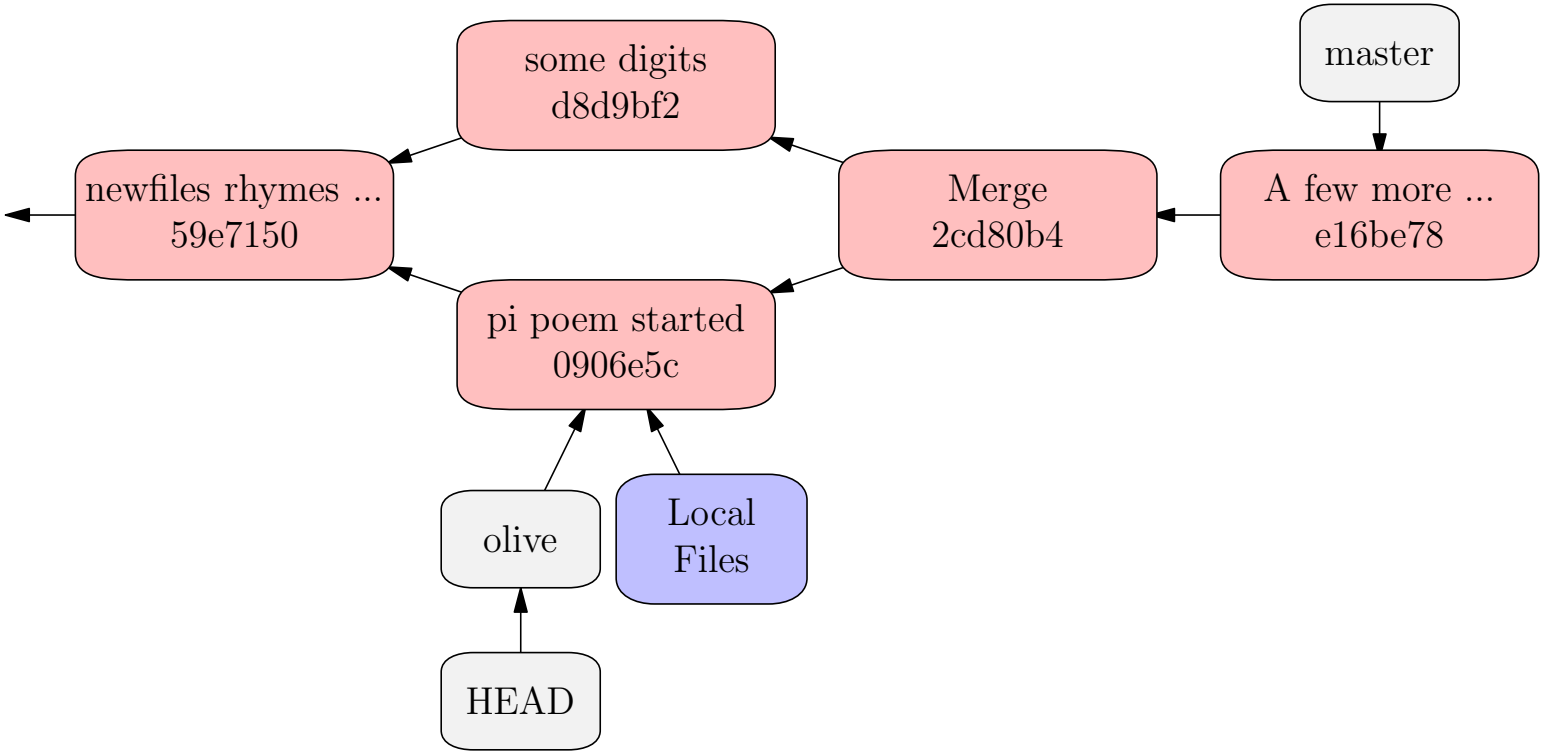
Checking out an older snapshot gives us the files, but **if we want to commit we will need to create a branch**. (We'll get to this ...)

```
$ git checkout 0906e5c

Note: checking out '0906e5c'.

You are in 'detached HEAD' state. You can look around, make
experimental changes and commit them, and you can discard any
commits you make in this state without impacting any branches
by performing another checkout.

If you want to create a new branch to retain commits you create,
you may do so (now or later) by using -b with the checkout
command again. Example:

git checkout -b new_branch_name

HEAD is now at 0906e5c... pi poem started
```

Checkout and branch

Unless we just want to look at the file system, or if there is already a branch at the given node, we will need to create a branch there before we do any real work.

So we may prefer to `git checkout -b [new branch name] [SHA1 hash]`




```
$ git checkout -b olive 0906e5c
```

The Staging Area (index)

In Git, `add` and `commit` are **separate commands** though they can be combined in `commit -a`. **Why?**

- There are two reasons to commit a file.
- **Reason 1:** to **save** correct (but unfinished) work.
- **Reason 2:** to add what is finished to the local or remote repository.

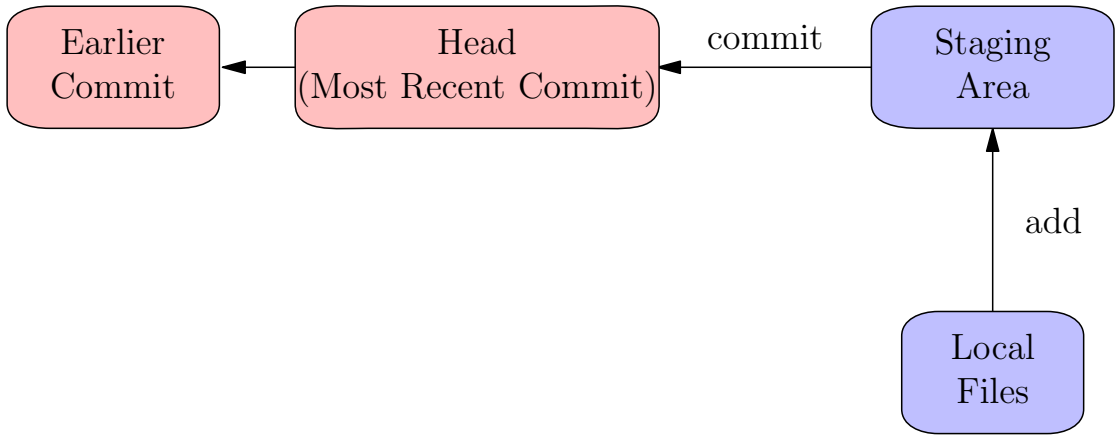
Reason 1 suggests we should save frequently. But if we use **commit -a** too often we get **snapshots of unfinished work** in the repository.

- Git has both **short term** and **long term** solutions to this dilemma.

Short term: Git has a **staging area (index)** where the commit is prepared.
Long term: **branches** can be used where Mercurial would use **mq**.

Staging

The **staging area (index)** contains a snapshot of the local file system that will be copied into the repository if a commit is done **now**.



- Not all files are tracked. Only tracked files go into the repository.
- Even if a file is tracked, it may not be in the same state in the local file system as in the staging area.
- To copy a file into the staging area use **git add**.
- **git commit** copies staged files into the repository creating a new node.

More on `git add`

- Roughly `git commit -a` is the same as `git add` followed by `git commit`.
- But not quite since `git add` must specify a file or files to add.

Not all files are **tracked**. A file is untracked until you explicitly add it to the repository using `git add`.

- `git commit -a` only commits tracked files.
- `git commit -a` copies all tracked files into the staging area then commits.
- `git add [file]` copies the file into the staging area and (if necessary) begins tracking it.

Staging Demonstration.

```
$ git status
# On branch master
# (use "git add <file>..." to update what will be committed)
# (use "git checkout -- <file>..." to discard changes in working
directory)
#
# modified: pi.txt
no changes added to commit (use "git add" and/or "git commit -a")
```

Do some work.

```
$ emacs pi.txt
```

No files staged yet so **index and HEAD contain the same files.**

`$ git diff` shows the difference between **local files and index**.

Stage your work

```
$ add pi.txt
```

We have staged one file. Now **index differs from HEAD**. But local files are the same as **index** so:

`$ git diff`	without arguments shows no changes.
`$ git diff --staged`	shows the staged changes that will be committed.

Work some more.

```
$ emacs pi.txt
```

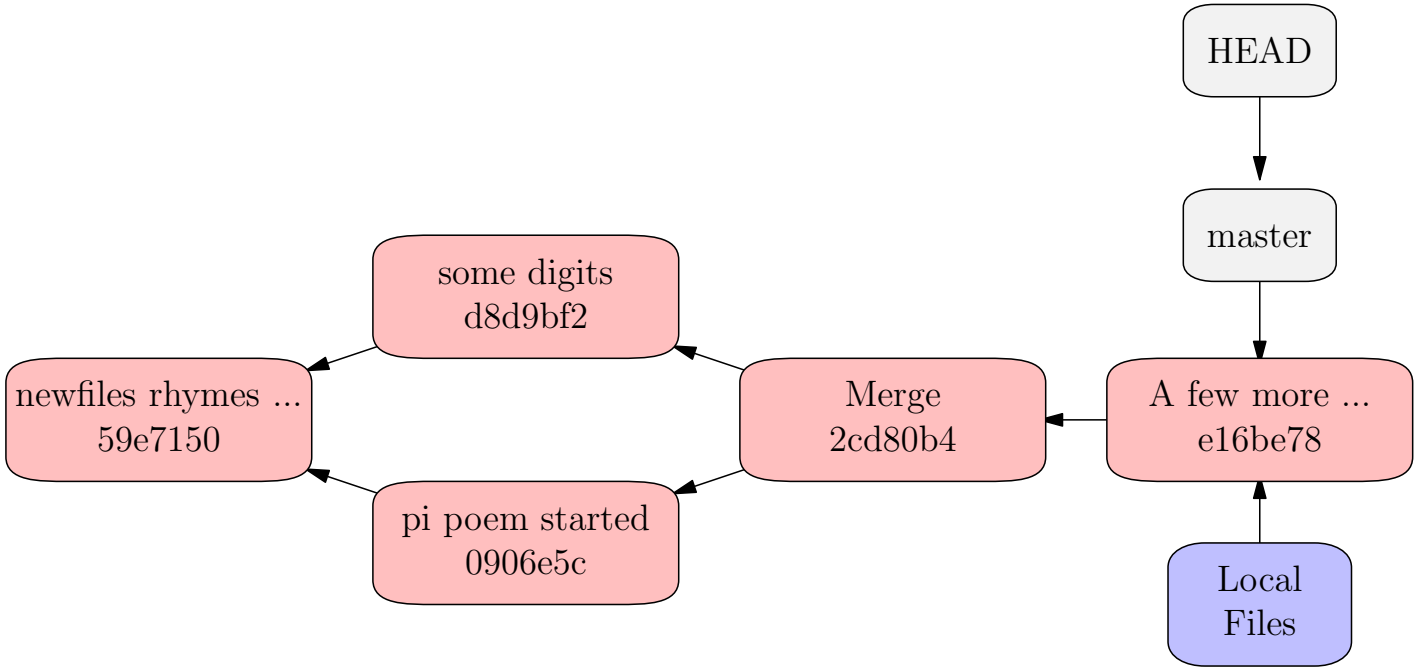
Now **HEAD** and **index** and **local files** all differ. Use `git diff` with or without `--staged` to see the changes.

```
$ git status
# On branch master
# Changes to be committed:
# (use "git reset HEAD <file>..." to unstage)
#
# modified: pi.txt
#
# Changes not staged for commit:
# (use "git add <file>..." to update what will be committed)
# (use "git checkout -- <file>..." to discard changes in working
directory)
#
# modified: pi.txt
```

- `git add` moves the local changes into the staging area.
- `git commit` followed by `git add` commits the staged changes, then moves the local files into the staging area.
- `git add` followed by `git commit` commits all changes.

Branches (Review)

- **HEAD** is a pointer to the place where a new commit will be attached.
- Usually **HEAD** points to a **branch** which points to a node.
- There is a **default branch** called **master**.

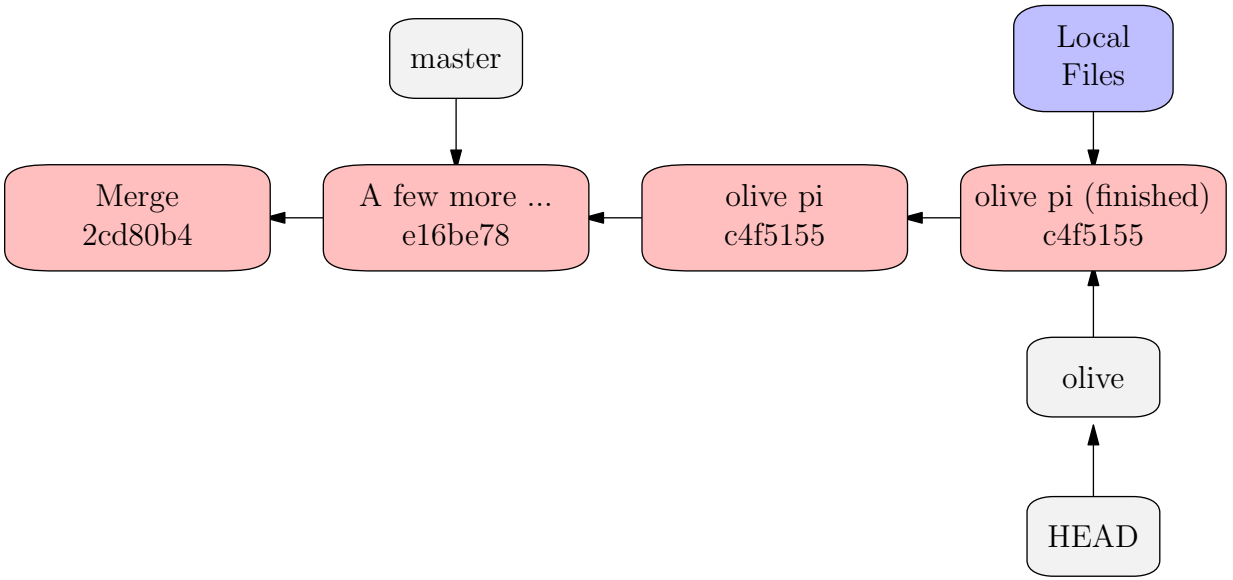


- There can be other branches pointing to other nodes. You can move **HEAD** to an existing branch using `git checkout [branch name]` .
- `gitk --all` is a good way to see the branches.

Use Branches For Experimental Features

Create a new branch, and add a couple of commits.

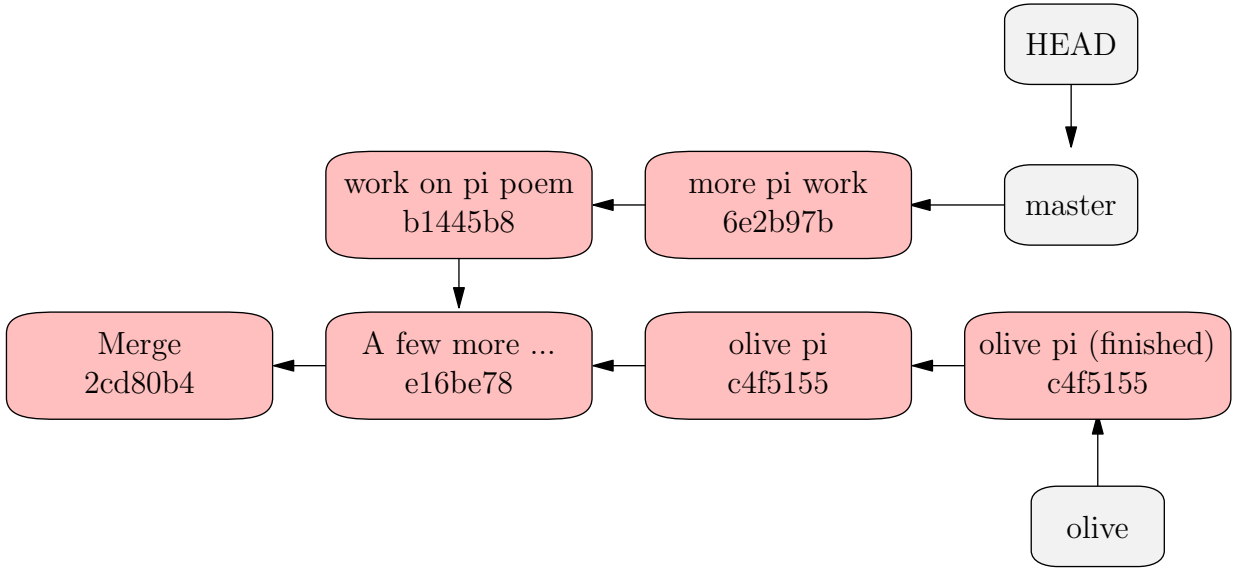
```
$ git checkout -b olive
$ emacs olive_pi.txt
$ git add olive_pi.txt
$ git commit -a -m "olive pi"
$ emacs olive_pi.txt
$ git commit -a -m "olive pi (finished)"
```



Perhaps better:
Create the branch
on the remote repository.

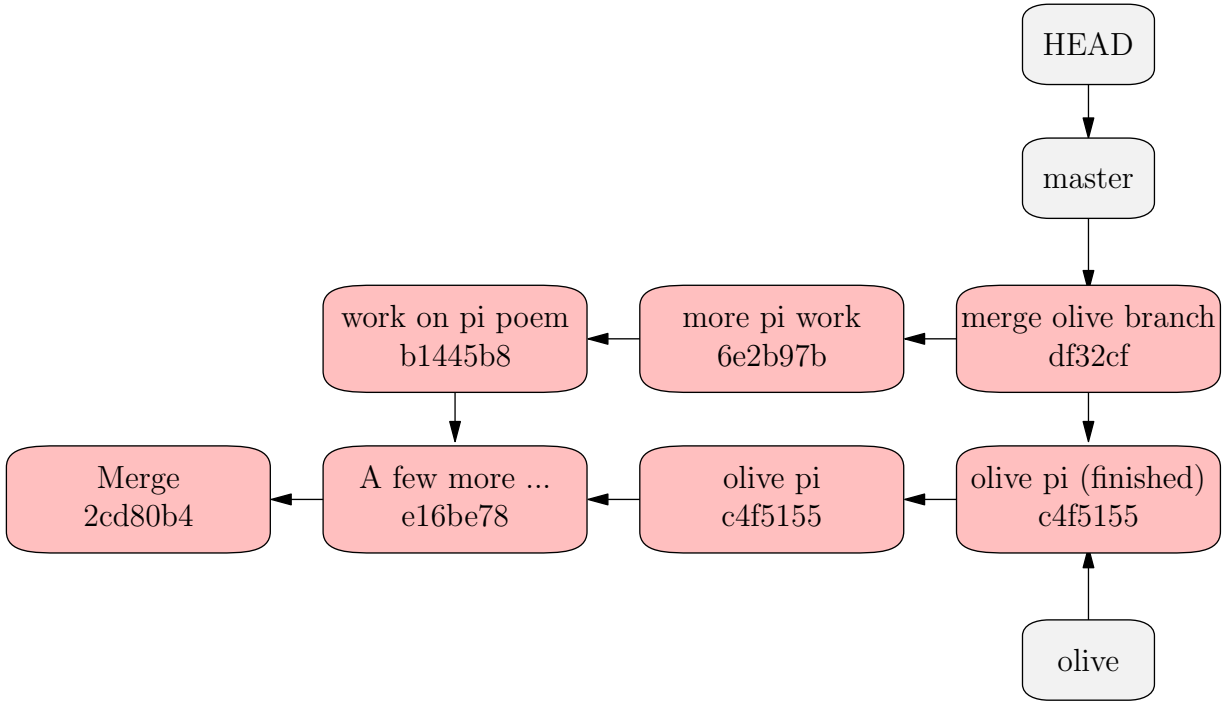
Use master for permanent changes

`$ git checkout master` (Then add a couple of commits.)



Problem: how to merge the two lines of development.

```
git merge -m "merge olive branch" olive
```

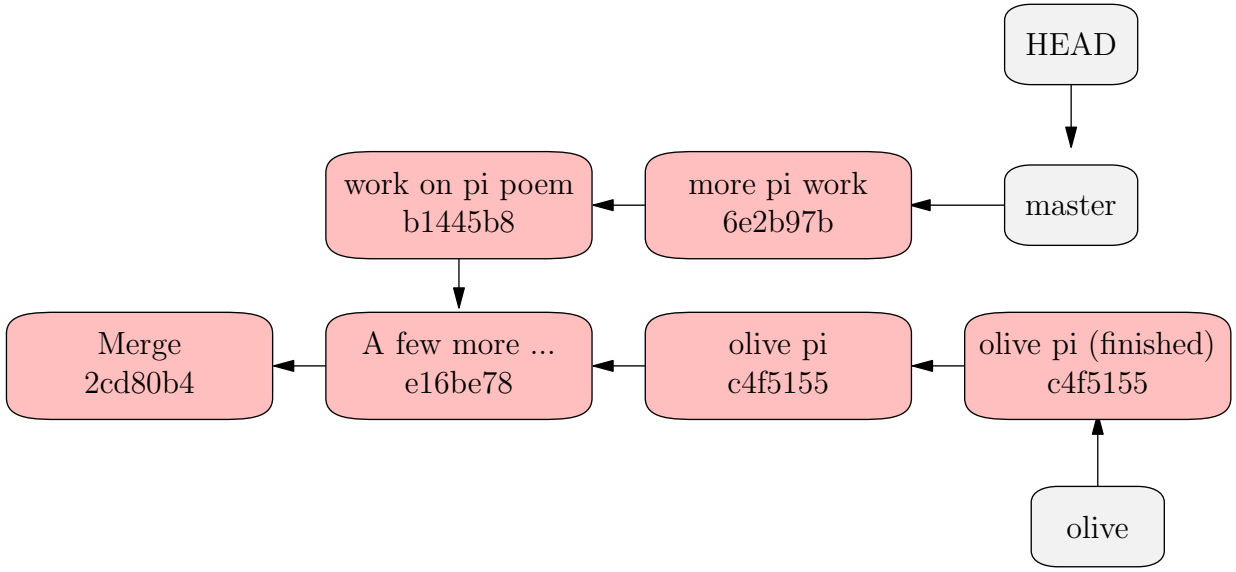


The branch **olive** is not needed now, so we may delete it.

```
$ git branch -d olive
```

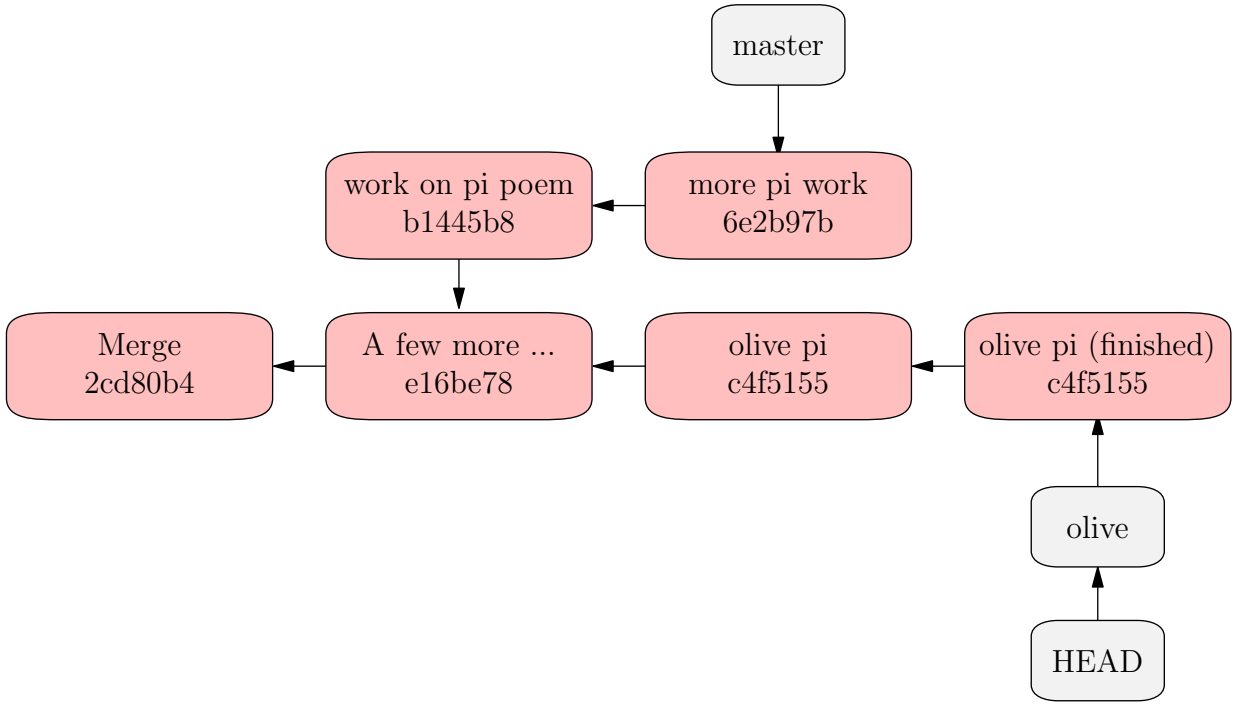
git rebase

Starting with the **same situation** we can use **git rebase** to obtain a merge with a **better topology**. This is useful in **realistic workflows** so we cover it.

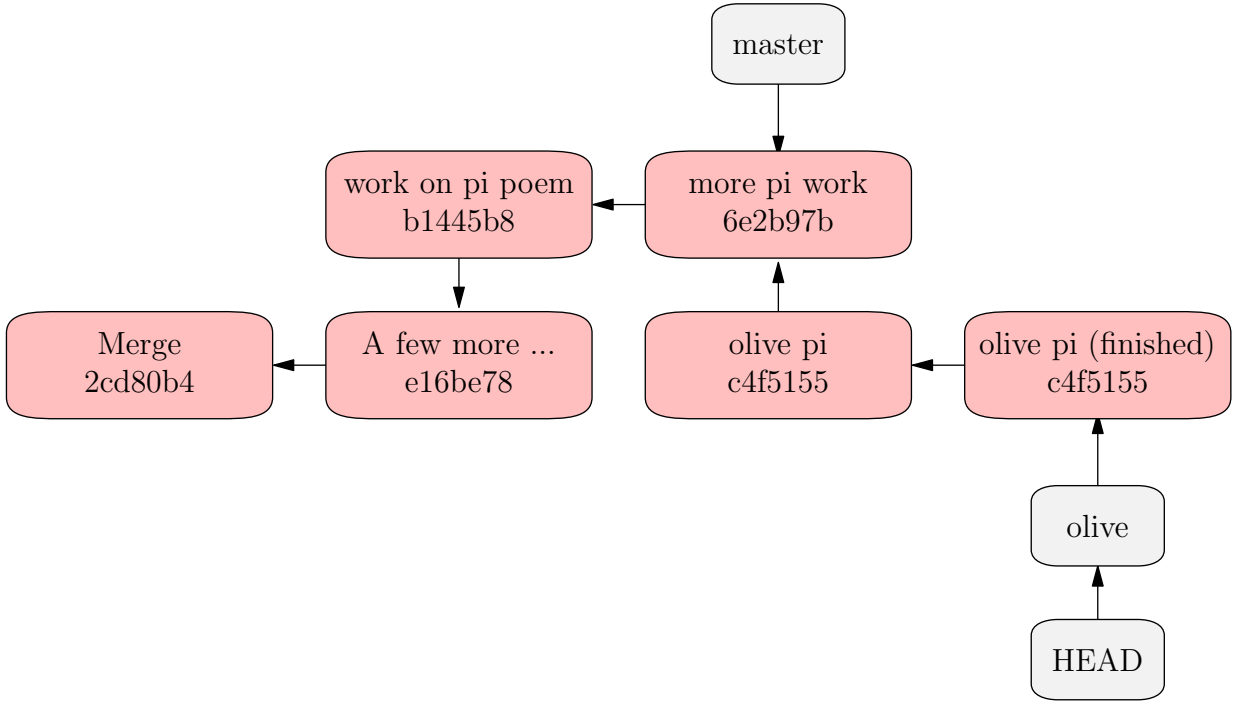


Make sure you are on the right branch.

```
$ git checkout olive
```



```
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: olive pi
Applying: olive pi (finished)
```



This involves **rewriting history** to move the patches from the point where the branches **olive** and **master** diverge **to apply after master**.

`git rebase -i`

- Git rebase **interactive** gives you functionality like Mercurial queues.
- Like **mq** **git rebase is all about rewriting history**.
- A commit plays the role of a queue patch.
- You can reorder the commits that are about to be rebased.
- (Or already rebased!)
- You may also **squash** commits, which is the same as **folding** patches.

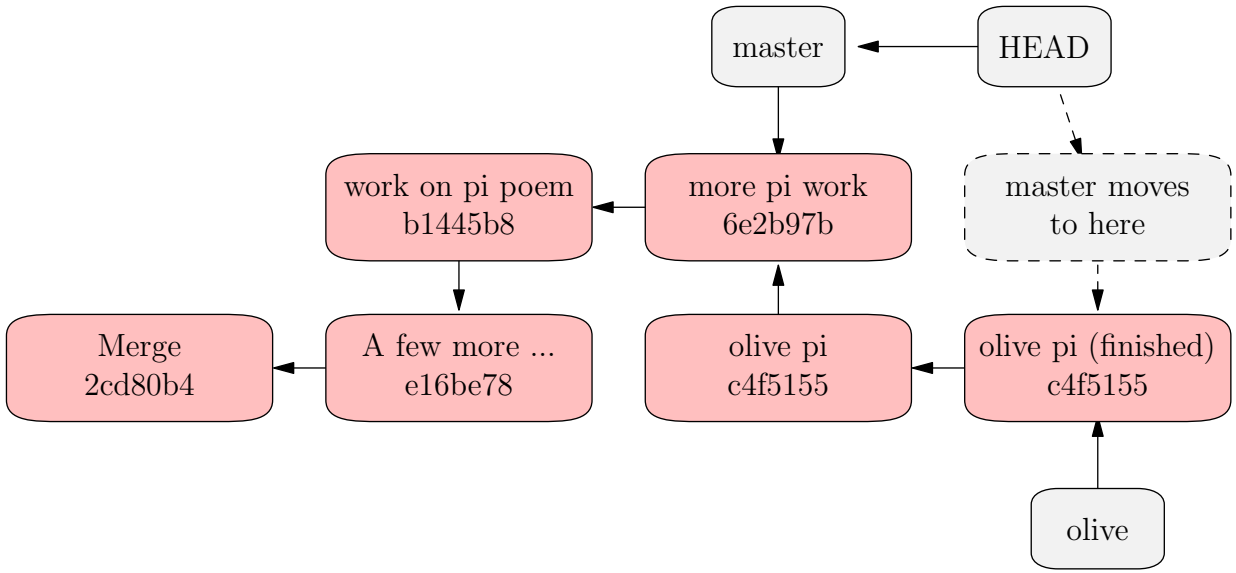
(**demo**)

Fast Forward

If you now want to move **master** up to the end of the tree, you may do this.

You probably don't want to do this unless you are release manager because master should parallel the remote branch origin/master.

```
$ git checkout master
$ git merge olive
```



Remote branches

If either `$ gitk --all` or `$ git branch -a`

we will see any remote branches that have been pulled. Let us create a new branch and push it to remote. We can work on the local repository for this.

```
$ git checkout -b olive
Switched to a new branch 'olive'
$ git push origin olive
Total 0 (delta 0), reused 0 (delta 0)
To /Users/bump/remote/pi.git/
* [new branch] olive -> olive

$ git branch -a
master
* olive
remotes/origin/HEAD -> origin/master
remotes/origin/master
remotes/origin/olive
```

Remote branches

- If we clone a repository, we get all the remote branches.
- If new branches are created on remote, we get them when we pull.
- The remote branch **olive** is identified as **origin/olive**.

To see the remote branche I can use `gitk --all` or

```
$ git branch -a
* master
remotes/origin/HEAD - origin/master
remotes/origin/master
remotes/origin/newbranch
remotes/origin/olive
```

The * indicates that **master** is the active branch, but the other remote branches are shown. If I want to work on **olive** I create a corresponding local branch.

Working on a remote branch

```
$ git checkout -b olive origin/olive
Branch olive set up to track remote branch olive from origin.
Switched to a new branch 'olive'
```

Now I can work on the local branch **olive** and push and pull changes to the remote repository.

To pull changes use

```
$ git pull origin olive
```

To push changes, it is a good idea to configure git so that only changes on the active branch are pushed. Assuming you have configured git this way:

```
$ git config --global push.default simple
```

git will push the current branch to the one it would pull from. This is probably what you want.

Making patches

Used `git format-patch`

Automation

Git can do tasks automatically. For example, we wanted to update web pages automatically every time someone pushed to a repository.

We added a script to **hooks/post-recieve** in the bare repository:

```
#!/bin/sh

cd /var/www/icerm; GIT_DIR='.git'; git pull; git update-server-info
```