



# Lecture # 1.3

## Overview of VCSs

### git

**Course: Advanced Operating System**

**Instructor: Arif Butt**

**Punjab University College of Information Technology (PUCIT)**  
**University of the Punjab**



# Agenda

---

- Intro to Version Control
- Types of Revision Control Systems
  - Local Data Model (sccs, rcs)
  - Centralized Data Model (cvs, svn)
  - Distributed Data Model (bitkeeper, git, mercurial, darcs)
- Downloading, installing and configuring git
- Working with git
  - Initializing a git repository
  - Adding and committing files to git repository
  - Viewing logs and status
  - Deleting, renaming and comparing files
  - Ignoring files
  - Undoing changes and moving to old commits



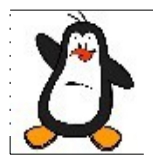


# Agenda

---

- Working with branches in **git**
  - Creating a new branch
  - Switching branches and working on them
  - Comparing two branches
  - Renaming and deleting a branch
  - Merging branches and handling merge conflicts
- Working with Remote Repositories in **git**
  - What is remote repository
  - Git hosting services
  - Creating a remote repository on bitbucket.org
  - Uploading a local project repo on bitbucket
  - Downloading a remote project repo from bitbucket





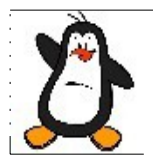
# Overview of Revision/Version Control System

- A Version Control System is a software tool that records changes to a file or a set of files over time, so that you can recall specific versions later
- Before VCSs exists we used different ways for maintaining versions of file(s) e.g. using **save as** for every new change made to file and making a copy and then giving a version number and date of update to that file
- A VCS allows us to
  - Maintain a history of different versions of a file
  - To move back and forth between these versions
  - Compare different versions
  - Merge multiple versions of same file to create a new version
  - Lock other users when one user is altering a file
  - Conflict resolution



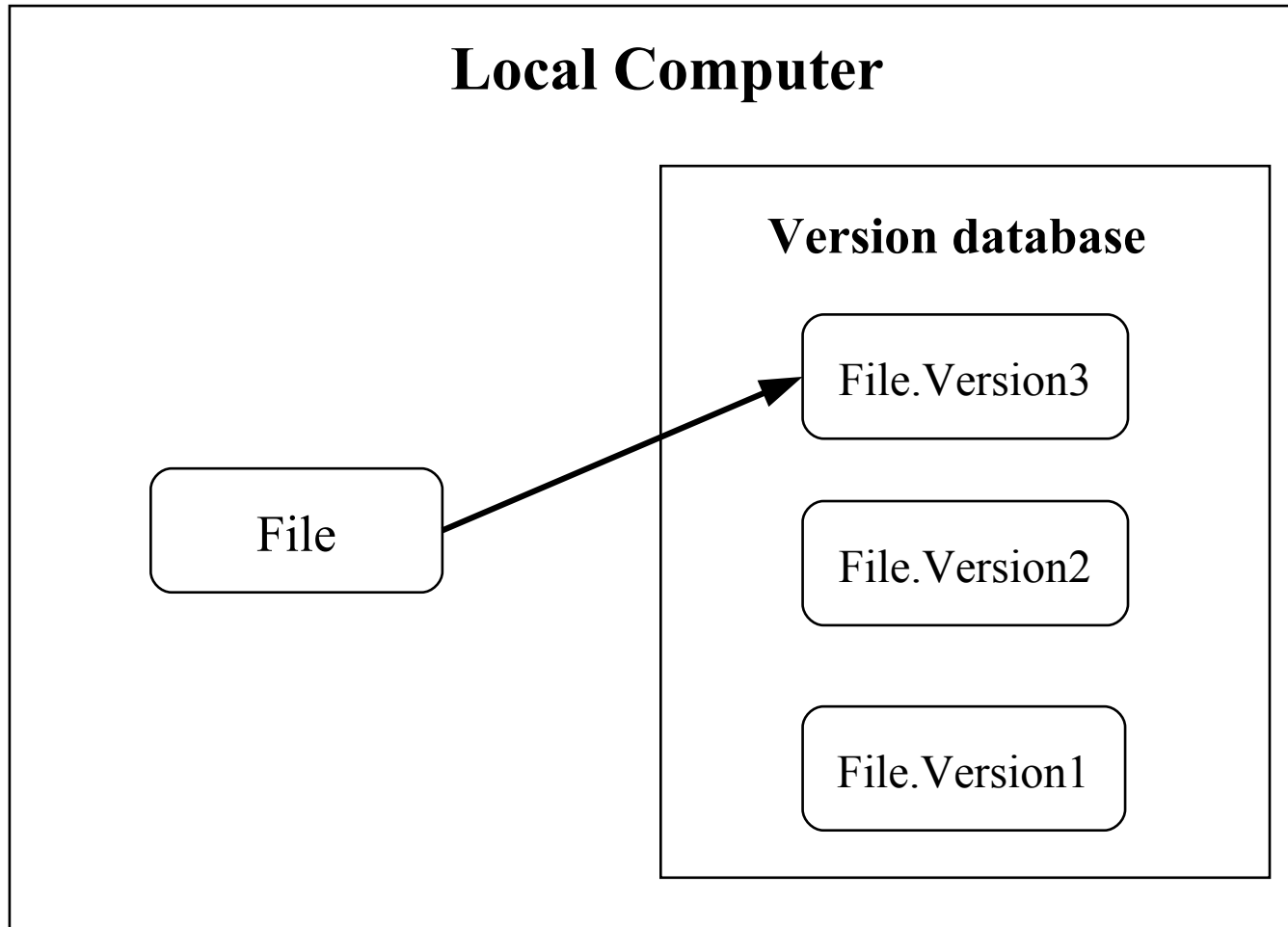
# Types of VCSs

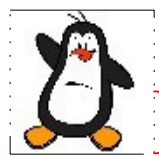
## I - Local Data Model



# Local Data Model

---



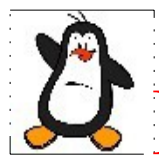


# Local Data Model (cont..)

---

## Source Code Control System: (SCCS-1972)

- Written in C, developed by AT&T and bundled free with UNIX
- It was not the first VCS, rather the first to become popular
- SCCs keeps the original file as it is and instead of saving the complete new version just save the snapshot of the changes
- If you want ver.3 of a file, you take ver.1 of the file and apply two set of changes to it to get to ver.3



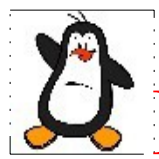
## Local Data Model (cont..)

---

### Revision Control System: (RCS-1982)

- Written in C, developed at Purdue University
- SCCS was for UNIX only, while RCS was for PCs as well
- RCS **keeps the most recent version of a file** in its whole form and if you want a previous version, you make changes to the latest version to re-create the older version
- This is faster than SCCS, as most of the time we need to work with the latest version of the file





# Local Data Model (cont..)

---

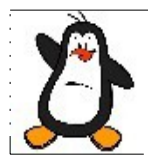
## Limitations of Local VCSs:

- You can track changes in a single file and not in a set of files or in a whole project
- Only one user can work with a file at a single time, therefore, multiple users / team members cannot collaborate and work on the same project

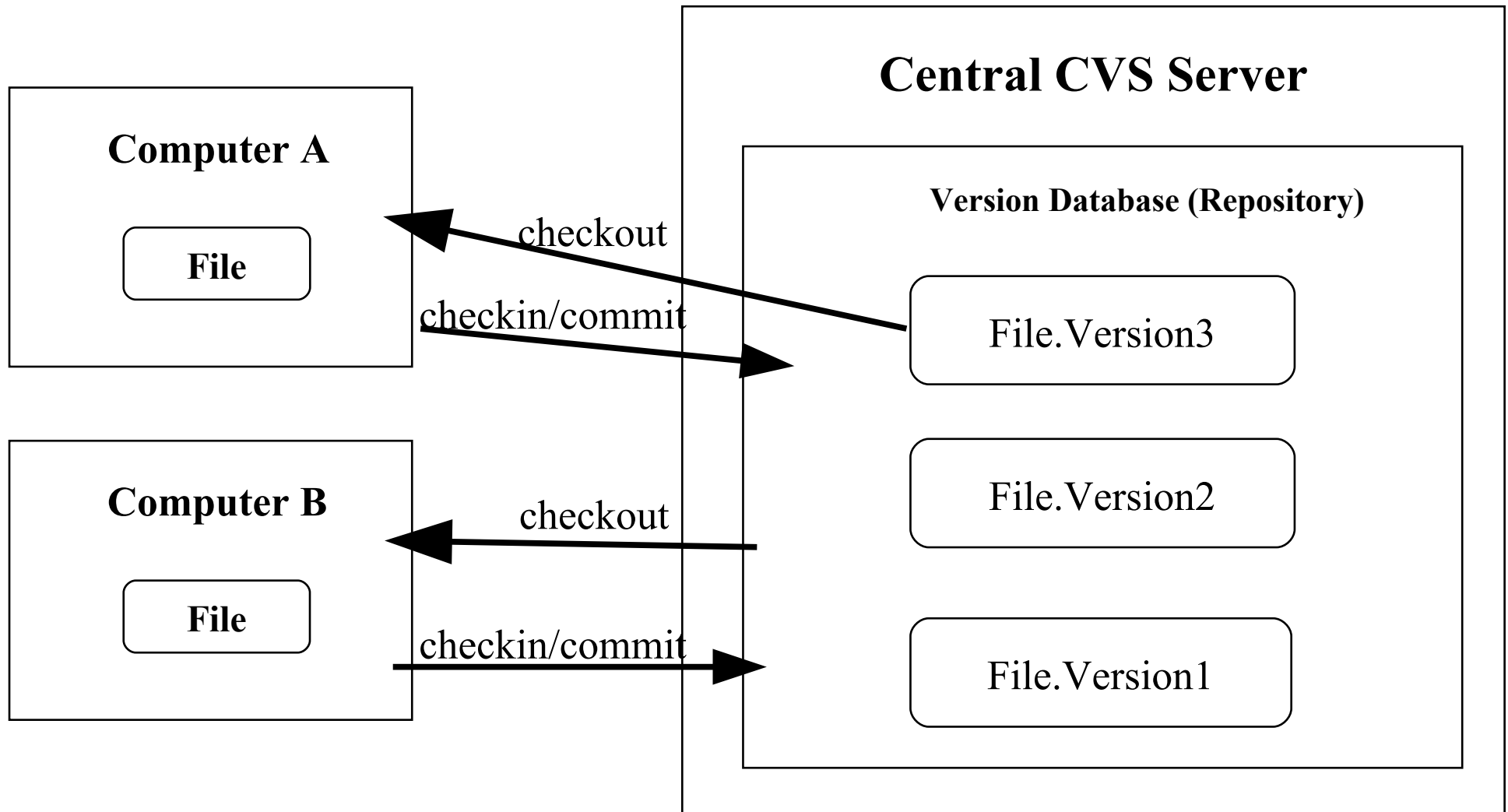


# Types of VCSs

## II - Centralized Data Model



# Centralized Data Model (cont..)



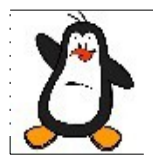


# Centralized Data Model (cont..)

---

## Concurrent Version System: (CVS-1990)

- Written in C and is open source
- Available for UNIX like OSs (UNIX, Linux, Solaris) as well as for MS Windows
- Introduced the idea of **branching**. A set of files may be branched at a point in time so that, from that time onward, two copies of those files may be developed in different ways independently of each other
- **Limitations**
  - CVS lacks atomic operations. Uses lock-modify-unlock model, allowing a user to place a lock on the checkout data in the repository, avoiding concurrency problems
  - No file renaming as cannot track directories



# **Centralized Data Model (cont..)**

---

## **Apache Subversion System: (SVN-2000)**

- Written in C, and is open source
- Cross platform and is faster than CVS
- Supports atomic commits
- Can track directories, so you can rename files within directories
- It can also track non-text files like images



# Centralized Data Model (cont..)

---

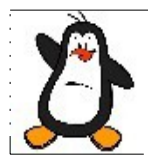
## Limitations of Centralized VCSs:

- Single point of failure as the centralized server containing the version database may crash
- No collaboration if server is down
- Developers do not have history of project on their local machines

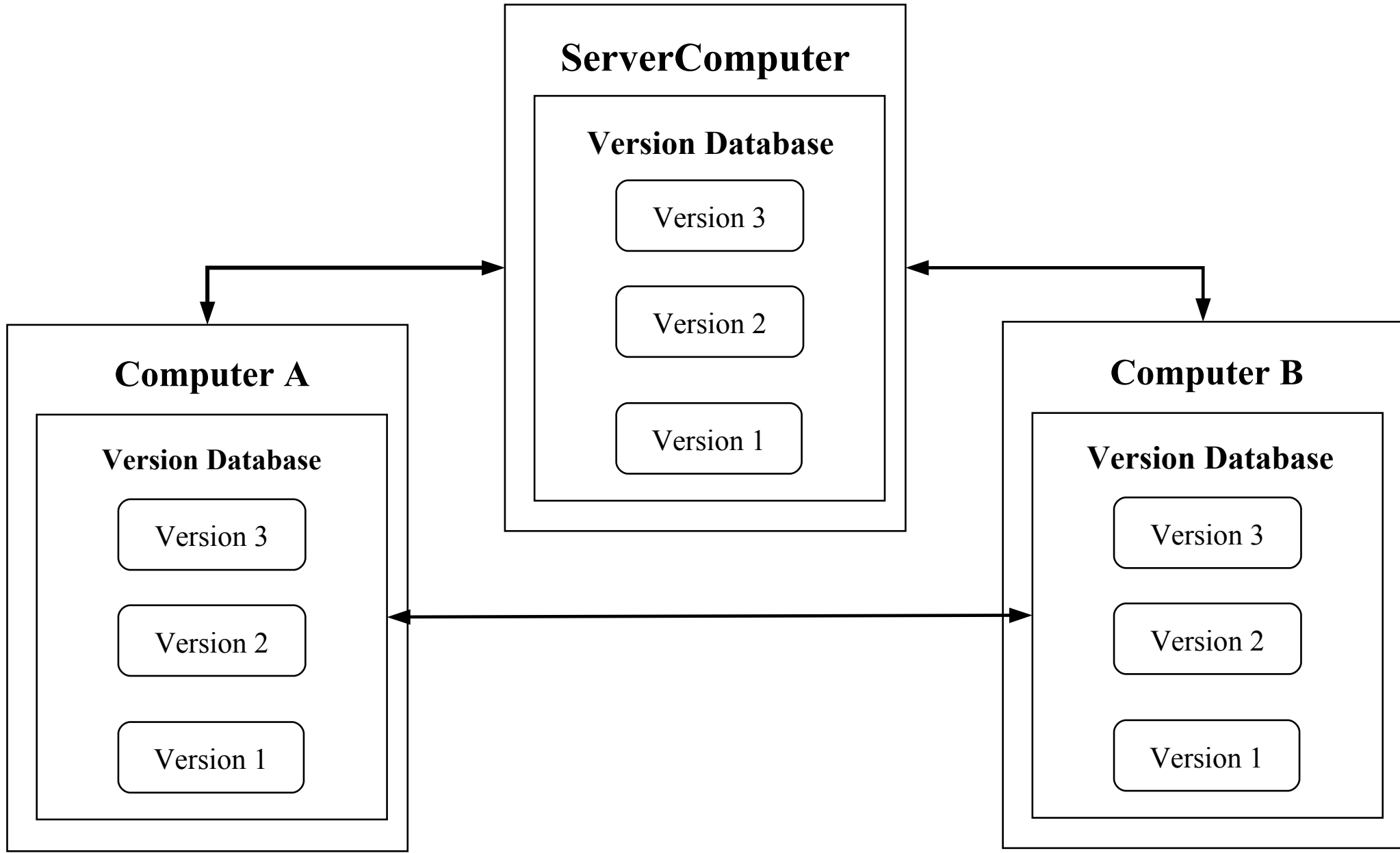


# Types of VCSs

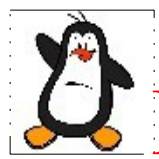
## III - Distributed Data Model



# Distributed Data Model





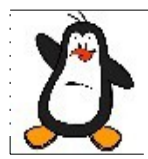


# Distributed Data Model (cont..)

---

## Bitkeeper -2000

- Written in C, and is proprietary and closed source
- A community version of bitkeeper with limited functionalities was free and that was used to manage Linux Kernel source from 2002 to 2005
- In April 2005, the “community version of bitkeeper” stopped being free and it was then **git** was born



# Distributed Data Model (cont..)

---

## git - 2005

- Developed by Linux Torvald in 2005,
  - Git is free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency
  - It is compatible with all UNIX-like systems (Linux, MacOS, Solaris, PCBSD, ...) and MS Windows
  - It is written mainly in C along with:
    - TCL: A general purpose interpreted dynamic programming language, which is embedded into C programs for rapid prototyping
    - Perl: A general purpose interpreted dynamic programming language and is popular for its string parsing abilities
    - Python: A general purpose interpreted dynamic programming language and supports multiple programming paradigms like procedural, object oriented, imperative and functional
-



# git

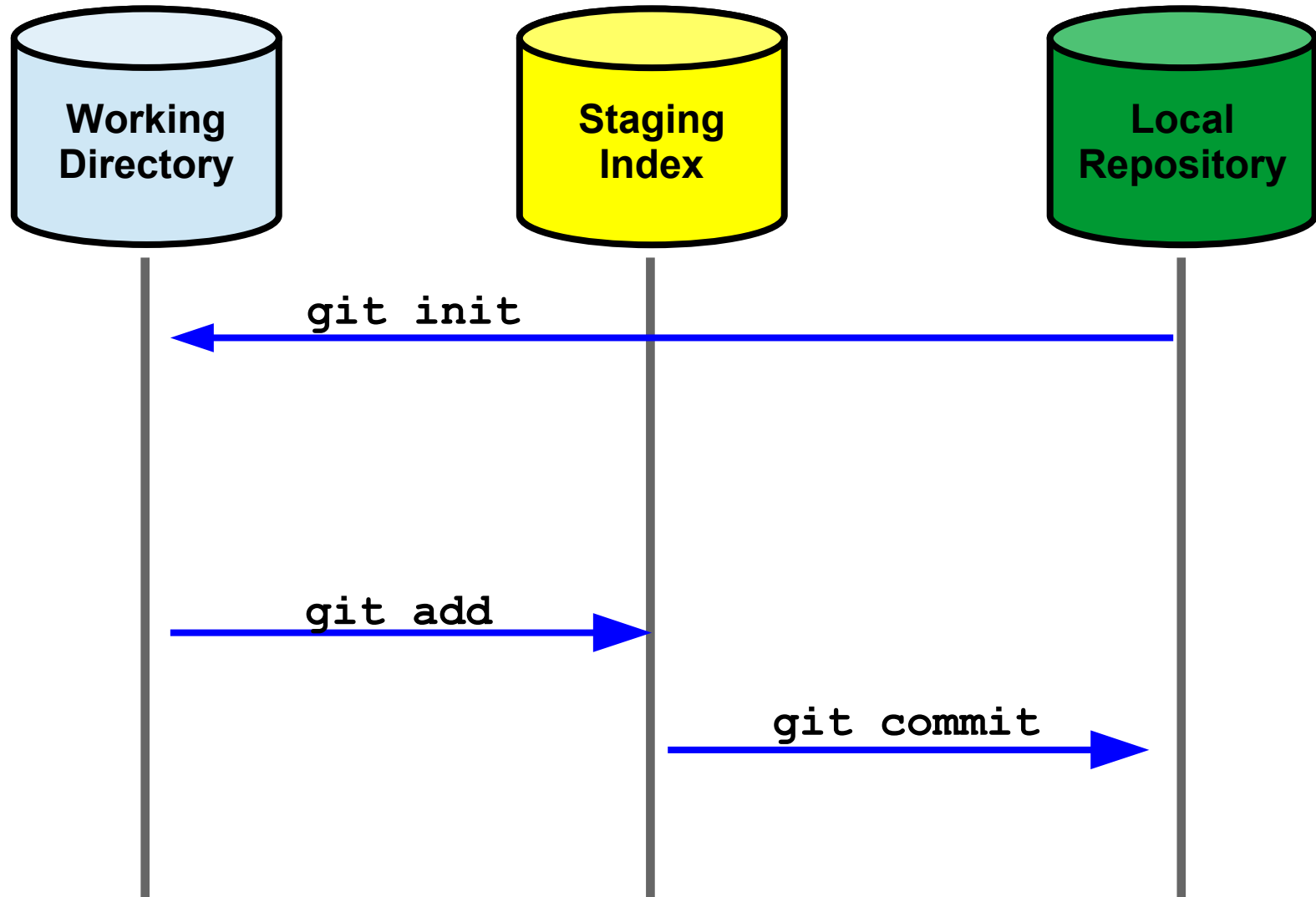
## Installation & Configuration

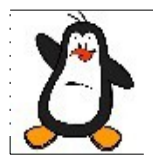
### A helloworld with git



# Basic Workflow of git

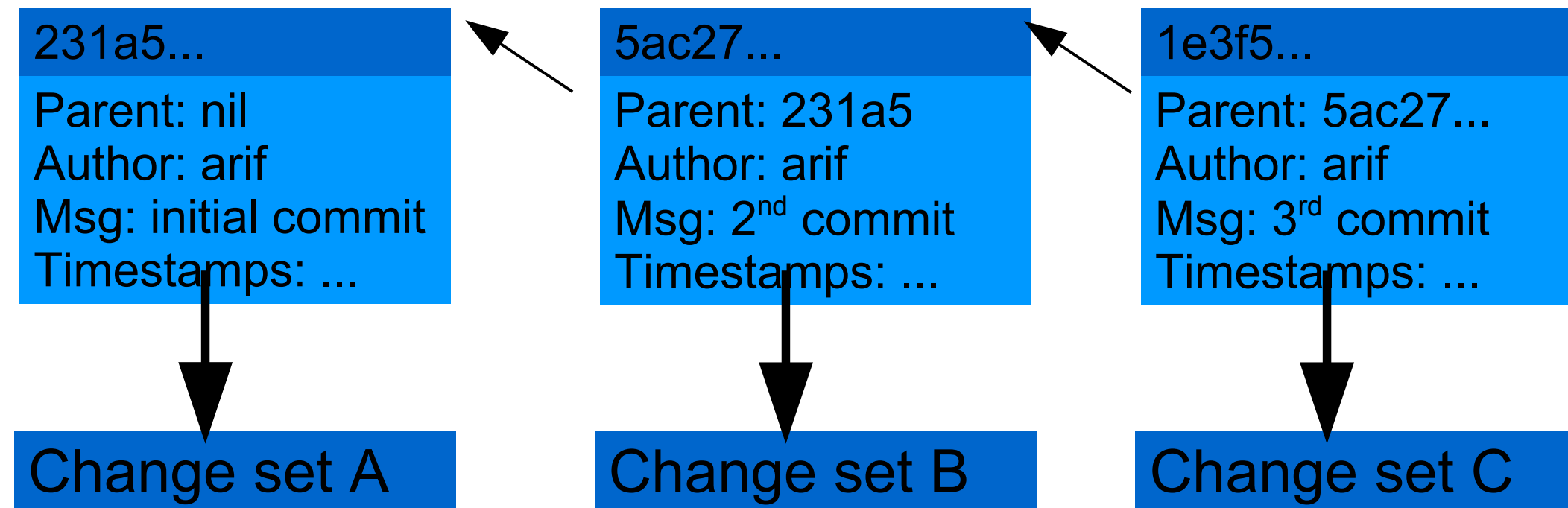
---





# The commit Objects

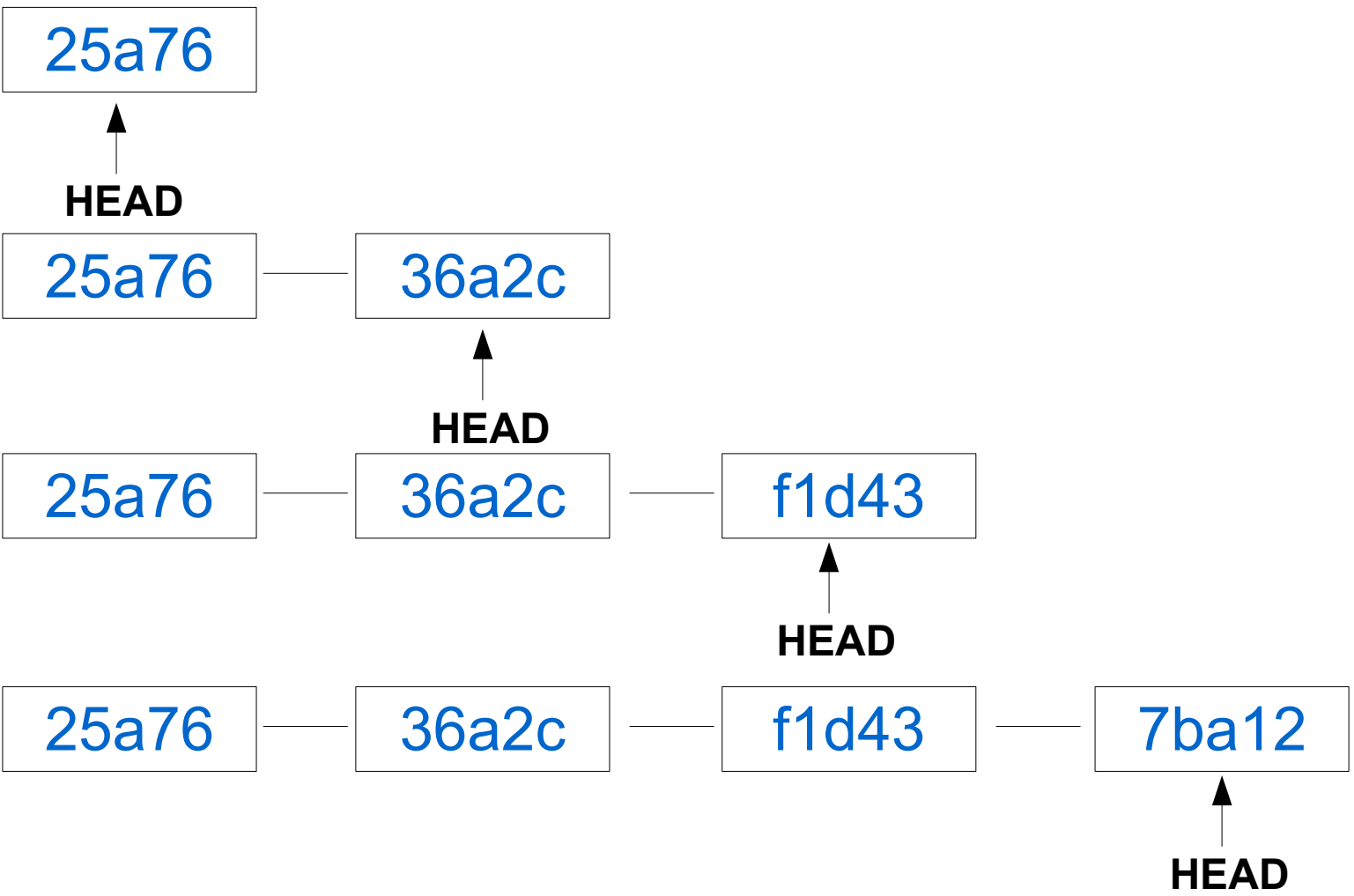
Suppose you have made three commits in your project, that means there are three change sets. Each commit object refers to a change set. Following figure illustrates how the series of commits are linked together. Note that the parent of each refers to a previous commit. We can see who has committed, when, why and with what change

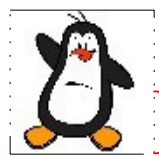




# HEAD Pointer in git

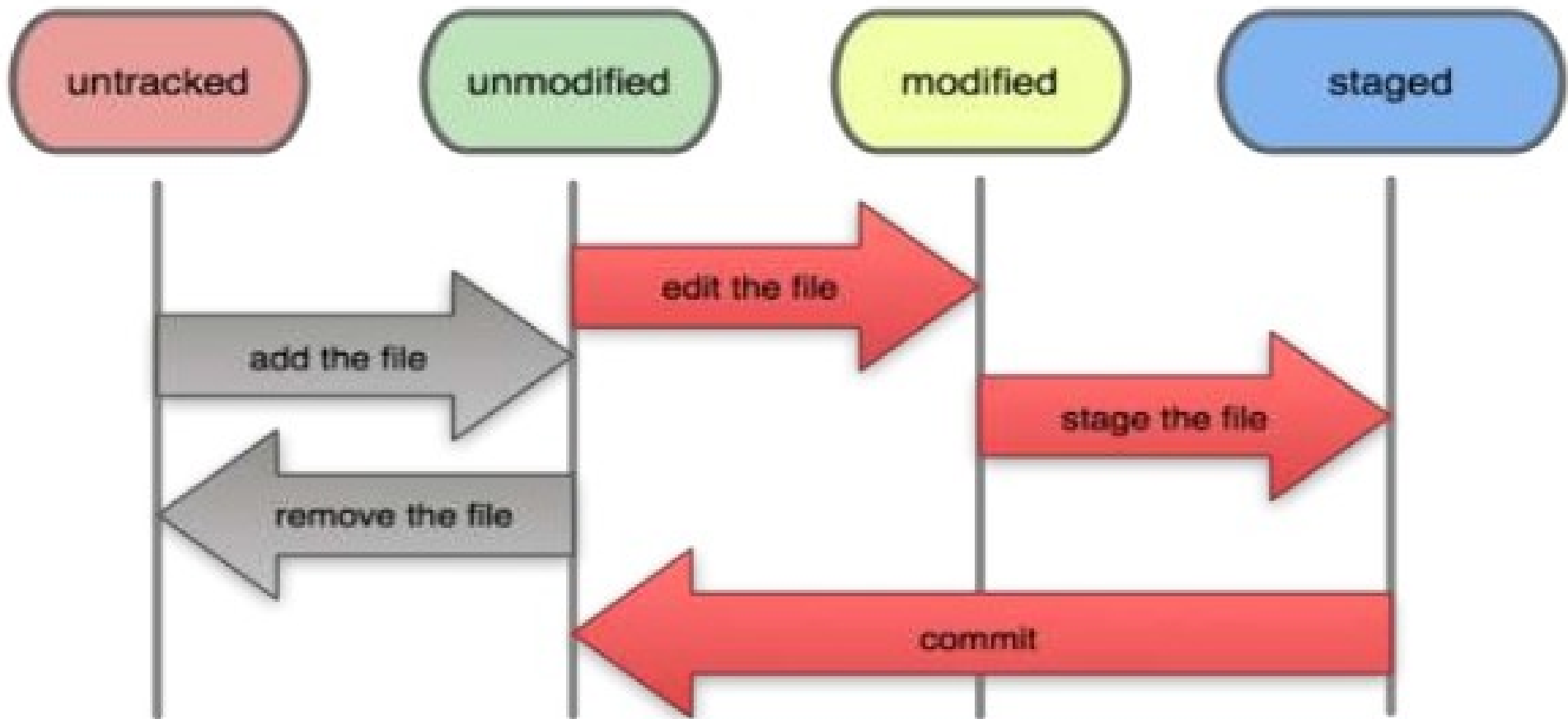
---

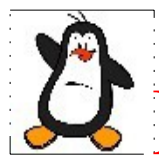




# Life cycle of a file in git

---





## Move to an old commit

We can move the head pointer to some previous commit and start recording the commits from that commit object onwards. There are three ways of doing this

**Soft Reset:** `$ git reset --soft <ID>`

- Head is moved to the specified commit ID
- No changes are made in the staging index or working directory

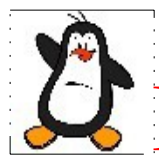
**Mixed Reset:** `$ git reset --mixed <ID>`

- Head is moved to the specified commit ID
- Staging index is also changed to match the local repository
- No changes are made in the working directory

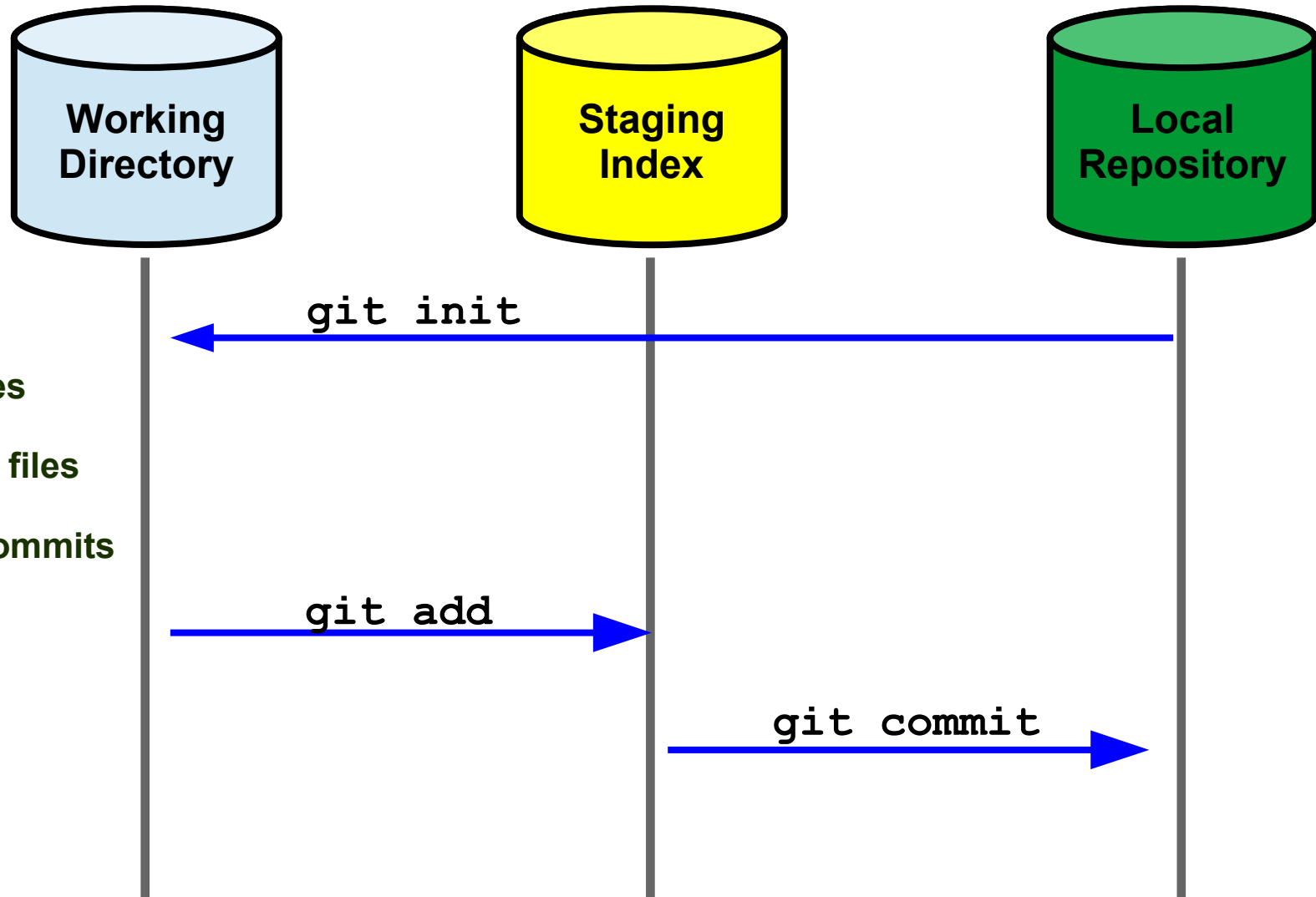
**Hard Reset:** `$ git reset --hard <ID>`

- Head is moved to the specified commit ID
- Staging index and working directory both match the local repository





# Basic Workflow of git (Review)



Tracked Files (Unmodified, Modified, Staged)

Untracked Files



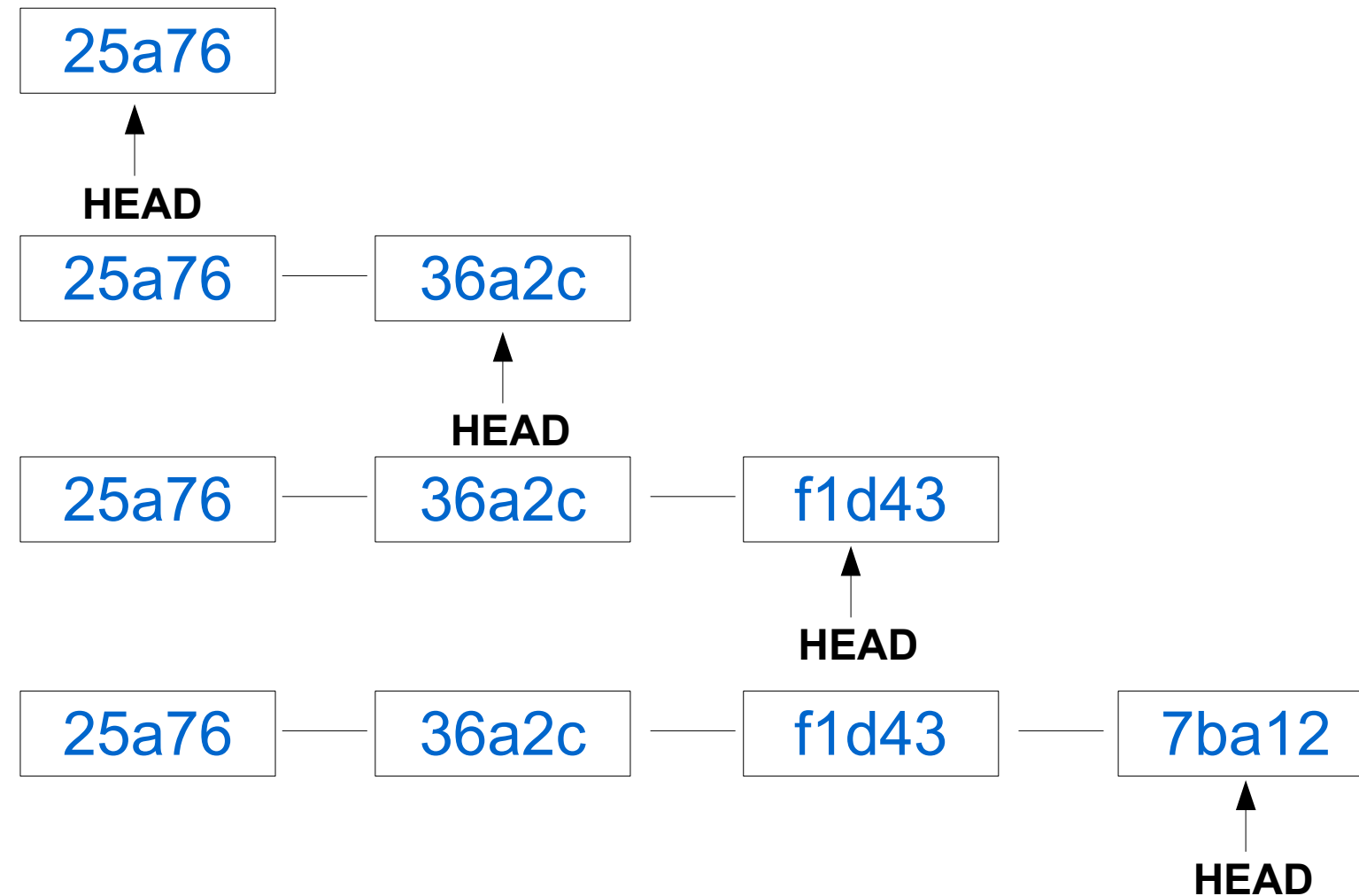
# git

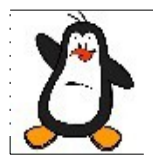
## Branches



# Overview of git Branches

A `git` branch represents an independent line of development. Every `git` repository has at least one branch called the master branch. An illustration of **master branch** is shown below:





# Overview of git Branches (cont...)

25a76

36a2c

master

HEAD



Suppose you are working on a project and have done some commits on the master branch which is the main line of your project development as shown above. You think of adding a new feature to your project but you are not sure whether it will work or not

- **OPTION 1:** You continue working on the same branch. If the new feature is a success, its GR8 and the development continues as shown below:

25a76

36a2c

f1d43

7ba12

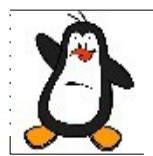
7ba12

master

HEAD



However, if the new feature is a failure you roll back to commit with SHA 36a2c using a `git reset`, and your master branch again becomes similar to the one shown at the top



# Overview of git Branches (cont...)

25a76

master

36a2c

HEAD



## OPTION 2:

25a76

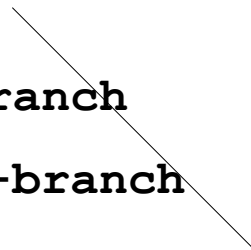
master

36a2c

f1d43

7ba12

HEAD



```
$ git branch new-branch
```

```
$ git checkout new-branch
```

```
$ git checkout master
```

234d12

new-branch

348cd

ac12f

HEAD



If the new-branch is a success, then you need to merge your new-branch with the master branch, otherwise, you can delete the new-branch and the master branch continue growing



# git

## Merging the Branches

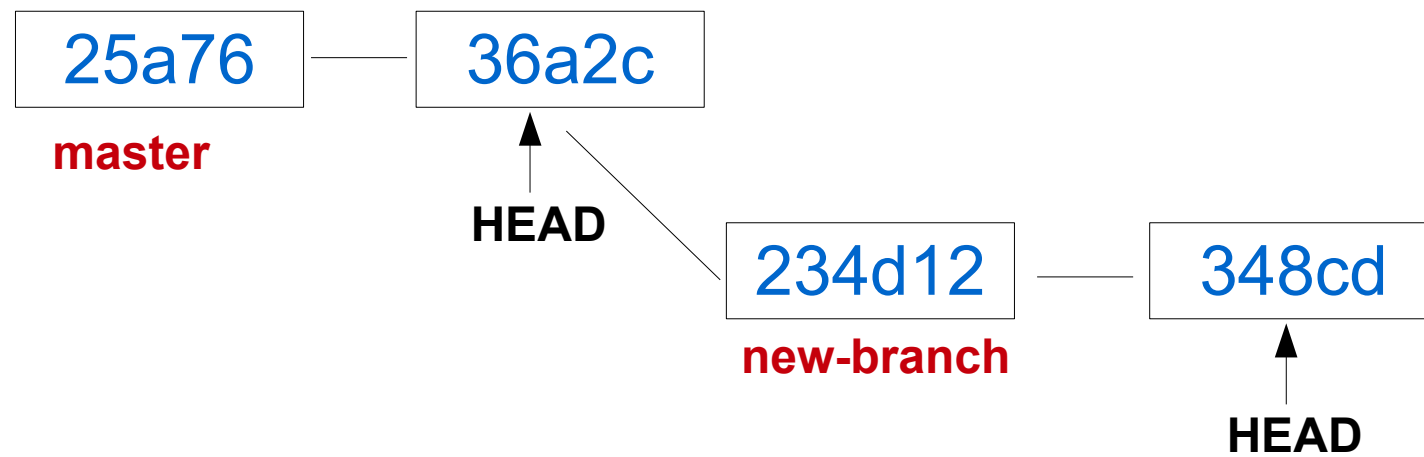


# Merging Branches in git

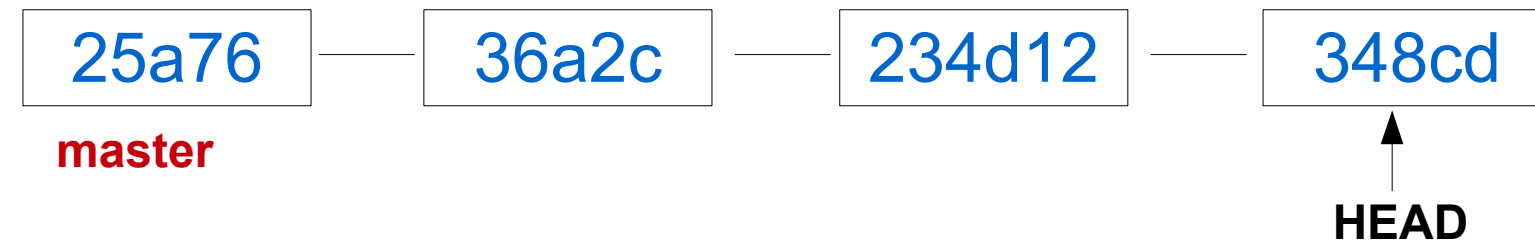
- Now we know how to create a new branch and how to perform development on that branch. After we are done developing and testing the new feature, it is time to bring those changes back to the master branch. For this we need to do a merge
- There can be two types of merges
  - Fast Forward Merge
  - Real Merge



# Fast Forward Merge



After fast forward merge:

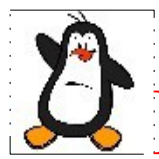


**Before you give merge command, your current branch should be the receiving branch**

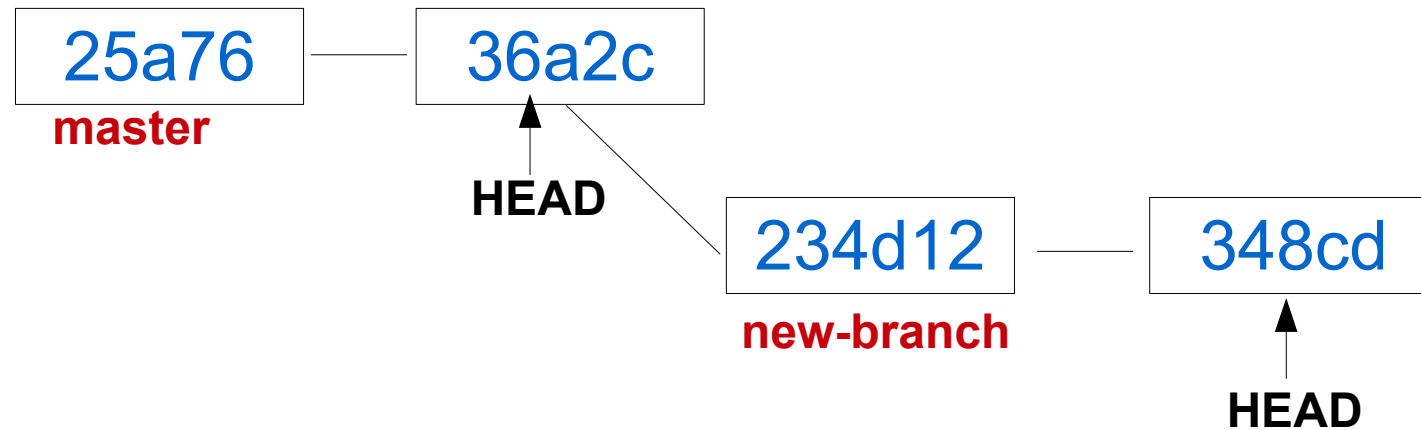
```

$ git checkout master
$ git merge new-branch
  
```





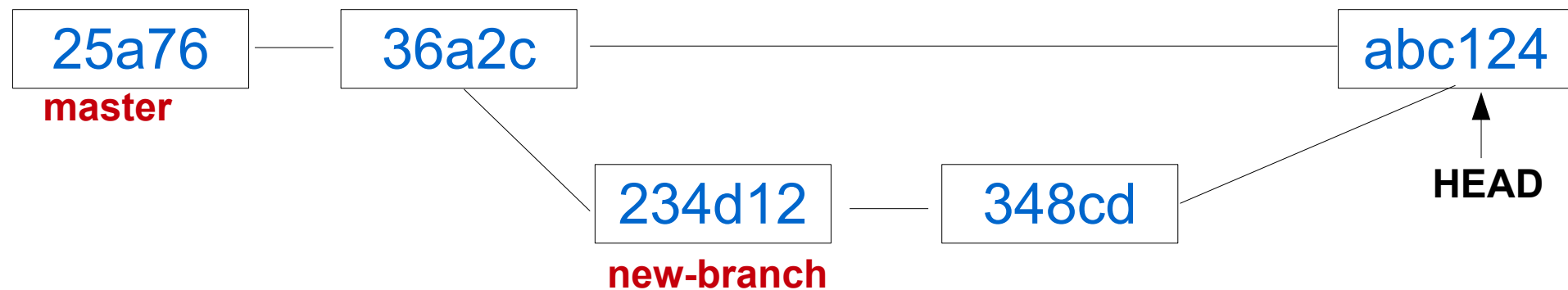
# Real Merge

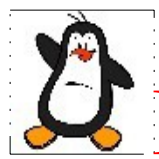


You can always force `git` not to do a fast forward merge, rather do an additional commit merge

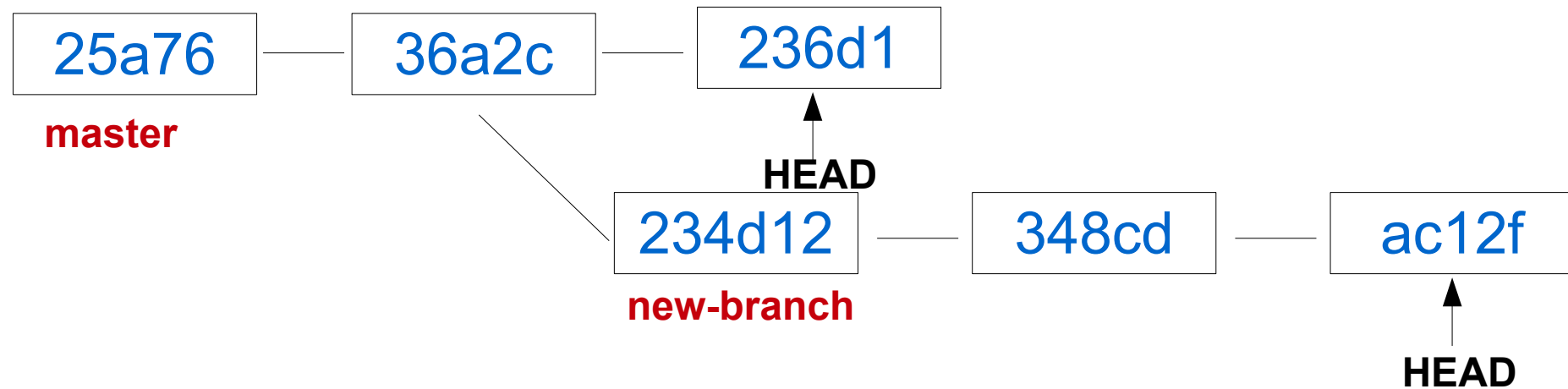
```

$git checkout master
$git merge --no-ff newbranch
  
```



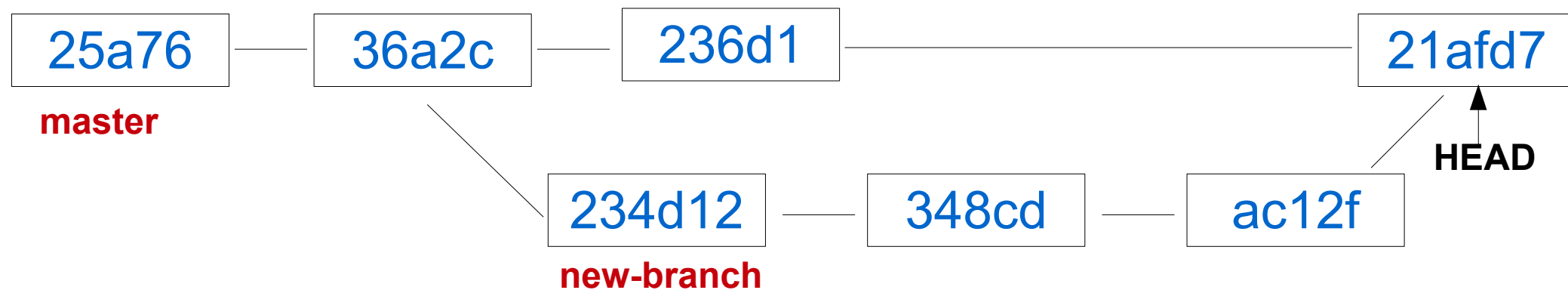


# Real Merge



In this scenario a fast forward merge is not possible. So when you do a merge, git will perform a real merge

```
$git checkout master
$git merge newbranch
```





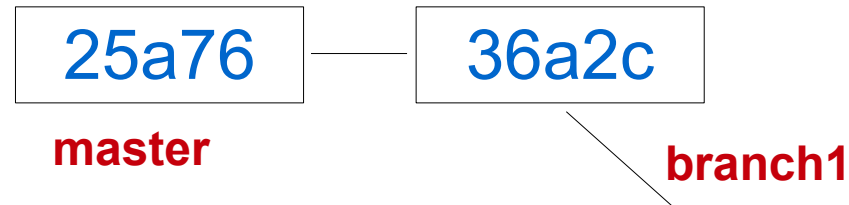
# git

## Handling Merge Conflicts

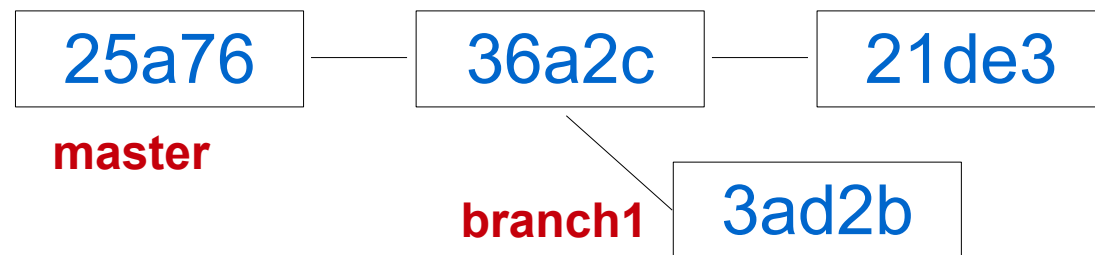


# What is a Merge Conflict?

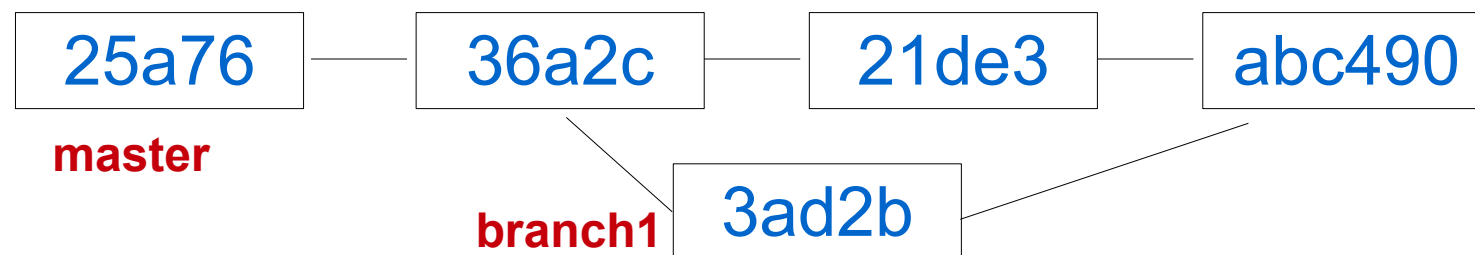
Suppose there are two branches **master** and **branch1**, as shown:

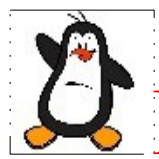


Both have a file suppose `file1.txt`, which is of course similar in both. A developer on **master** branch edit line#25 of `file1.txt` and do a commit. Another developer on **branch1** edit line#50 of `file1.txt` and do a commit



Now if you merge, it will be a success, because both have made changes to same file, but to different lines





# Handling Merge Conflicts

---

However, if both the developers have made changes to same line or set of lines a conflict will occur, which `git` cannot handle and it will give a message that auto-merging failed. In case of a merge conflict we have three choices to resolve the conflict:

## Abort merge:

```
$ git merge --abort
```

## Resolve manually:

Open the file in some editor and perform the changes manually, add, commit, and finally perform merge

```
$ git merge <branchname>
```

## Use merge tools:

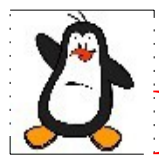
You can use different tools to automate this process like `araxis`, `diffuse`, `kdiff3`, `xxdiff`, `diffmerge`

```
$ git mergetool --tool=diffuse
```

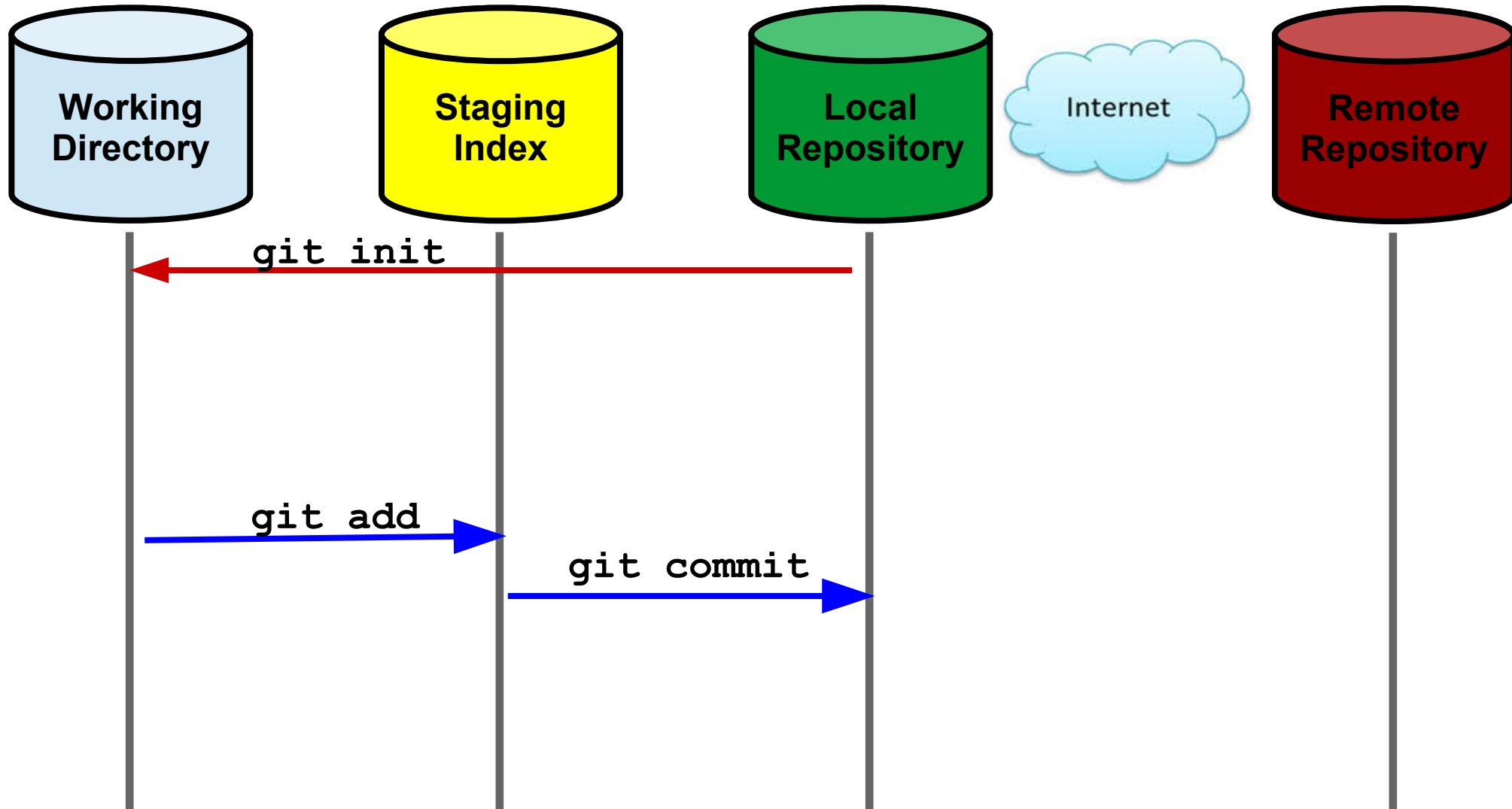
---

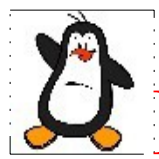


# Working of `git` Remote Repositories



# Remote Repository





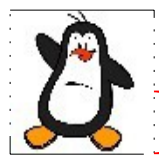
# Hosting Services for Versioning Systems

---

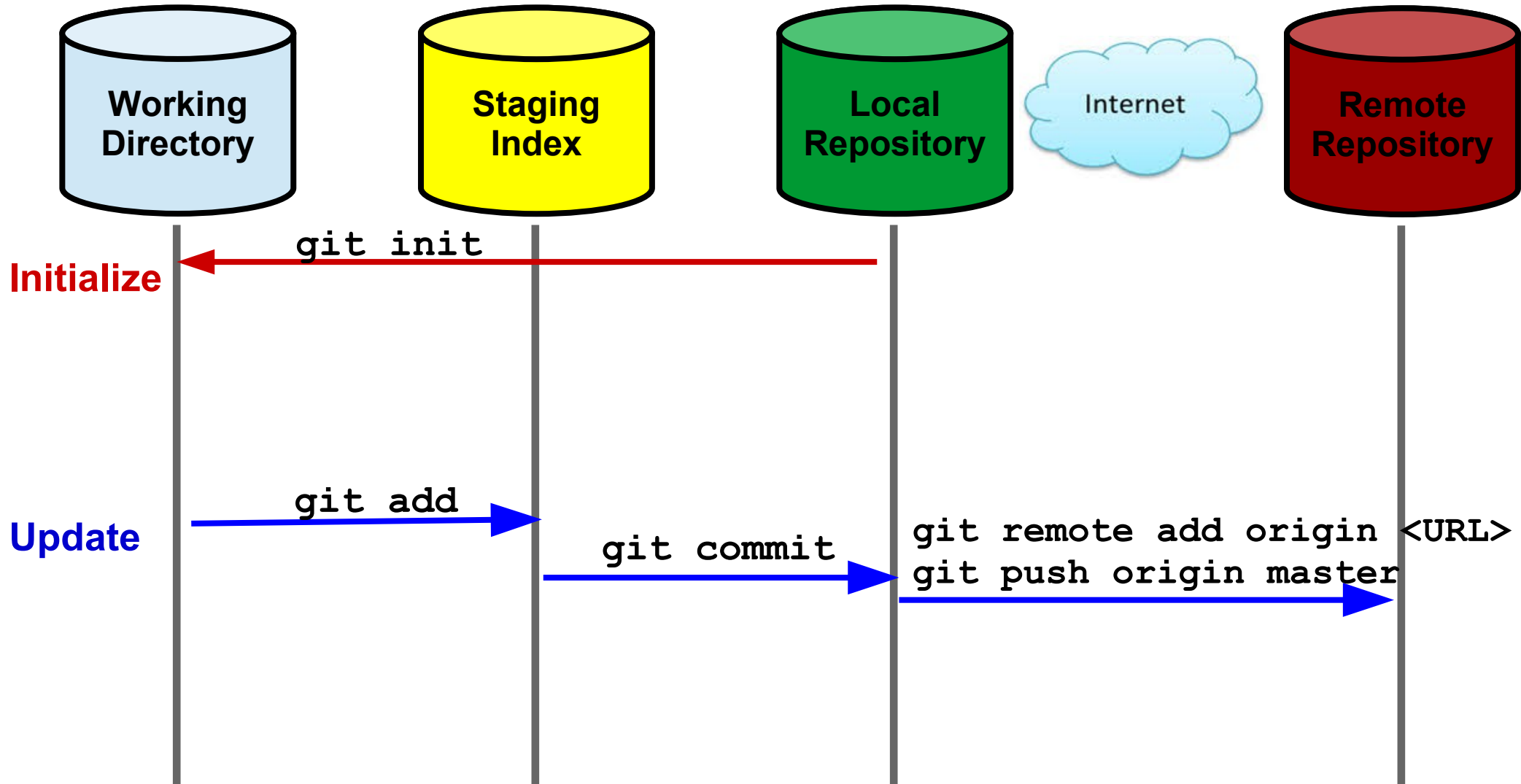
Some of the most famous hosting services are:

- <https://bitbucket.org>
- <https://github.com>
- <https://gitlab.com>



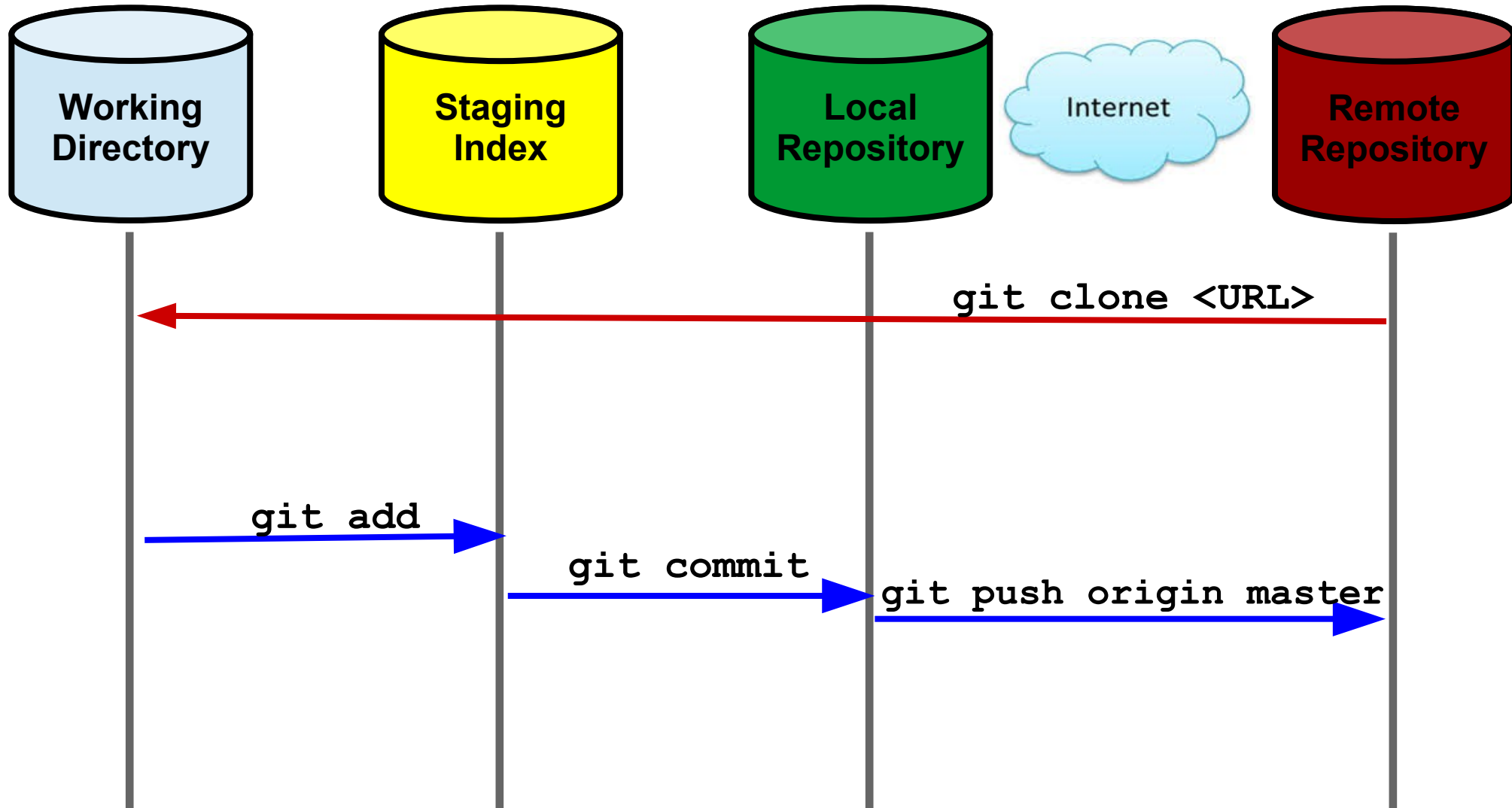


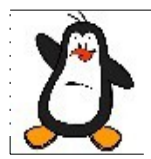
# Pushing a Local Repo to Remote Repo



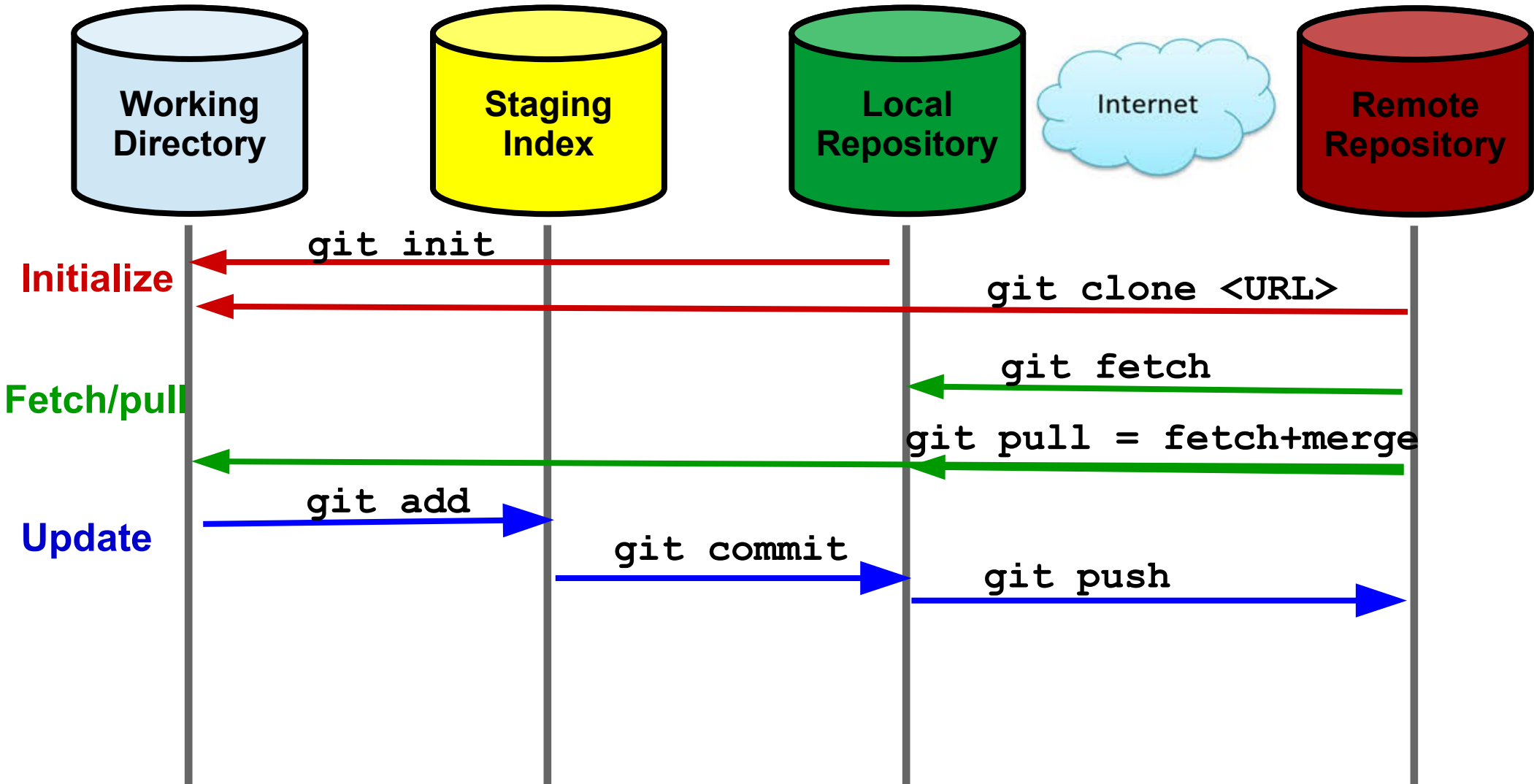


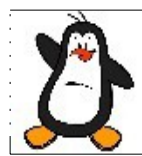
# Cloning Remote Repo to Local





# Workflow of git (Review)





# Things To Do

---

O.k., and now you'll do exactly what I'm telling you !



If you have problems visit me in counseling hours. . . .

---