1.4 C - Dynamic Memory Mar

- Recall: The definition of a **fixed-size** array
  - `int a[100];`
  - But now, let's consider a situation: no idea about th
- Runtime memory allocation and management
  -

Sr.No.	Function & Description
1	**void *calloc(int num, int size);** This function allocates an array of **num** elements each of which size in bytes will be **size**.
2	**void free(void *address);** This function releases a block of memory block specified by address.
3	**void *malloc(int num);** This function allocates an array of **num** bytes and leave them uninitialized.
4	**void *realloc(void *address, int newsize);** This function re-allocates memory extending it upto **newsize**.

https://www.tutorialspoint.com/cprogramming/c_memory_management.html

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {

   char name[100];
   char *description;

   strcpy(name, "Zara Ali");

   /* allocate memory dynamically */
   description = malloc( 30 * sizeof(char) );

   if( description == NULL ) {
      fprintf(stderr, "Error - unable to allocate required memory\n");
   } else {
      strcpy( description, "Zara ali a DPS student.");
   }

   /* suppose you want to store bigger description */
   description = realloc( description, 100 * sizeof(char) );

   if( description == NULL ) {
      fprintf(stderr, "Error - unable to allocate required memory\n");
   } else {
      strcat( description, "She is in class 10th");
   }

   printf("Name = %s\n", name );
   printf("Description: %s\n", description );

   /* release memory using free() function */
   free(description);
   return 0;
}
```

```
~/Desktop/cs97-test/testfolder ❯ gcc testMalloc.c -o testMalloc
~/Desktop/cs97-test/testfolder ❯ ./testMalloc
Name = Zara Ali
Description: Zara ali a DPS student.She is in class 10th
```

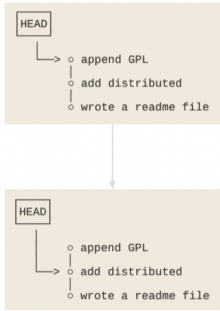
3. Git Basis

- Rollback
  - We want to rollback to previous versions
  - **`git reset --hard HEAD^`**
    - Reset current **`HEAD`** to its parent

```
* 1f3a18099a72266b46325e9a0c4faa47c1d453b5 (HEAD -> master) append GPL
* 63d49519f77488dff5a61a00a3544d5fe1c7374c add distributed
* 16c34ed1b23f58b28fc1819c70eced883619f3fd create a readme file
(END)
```

```
~/Des/c/learngit git master ❯ git reset --hard HEAD^
HEAD is now at 63d4951 add distributed
~/Des/c/learngit git master ❯ git log --graph --pretty=oneline
```

```
* 63d49519f77488dff5a61a00a3544d5fe1c7374c (HEAD -> master) add distributed
* 16c34ed1b23f58b28fc1819c70eced883619f3fd create a readme file
(END)
```



  - **`git reset --hard <commit_id>`**

```
~/Des/c/learngit git master ❯ git reset --hard 1f3a180
HEAD is now at 1f3a180 append GPL
```

  - **`git reflog`** ## checkout your previous command and commit ids

3. Git Basis

- Rollback
  - We want to rollback to the version **"add distributed"**
  - **`git reset`**
    - Reset current **`HEAD`** to the specified state
    - **`git reset --mixed <commit>`** (default option, ⇔ **`git reset <commit>`**)
      - **`master`** and **`head`** point to **`<commit>`**, the index is also modified to match the modified files (modifications are push back to the working directory, we need to do **`git add`**, then **`git commit`** to save changes after **`<commit>`**)
    - **`git reset --soft <commit>`**
      - Same as **`--mixed`**, but the index still has later changes saving in the staging area (we need to do **`git commit`** to save changes after **`<commit>`**)
    - **`git reset --hard <commit>`**
      - Same as **`--mixed`**, but remove all changes after **`<commit>`**

```
1f3a18099a72266b46325e9a0c4faa47c1d453b5 (HEAD -> master) append GPL
63d49519f77488dff5a61a00a3544d5fe1c7374c add distributed
16c34ed1b23f58b28fc1819c70eced883619f3fd create a readme file
(END)
```
reset

3. Git Basis

- Discard Changes
  - Modify our **`readme`**
  - We want discard changes in **`readme`**

1. **`Readme`** is in the current working directory
   - **`git status`**
   - **`git checkout -- readme`**
     - Discard the changes in the current working directory
2. **`Readme`** is in the staging area
   - **`git status`**
   - **`git reset HEAD readme`**
     - Unstage the current file, push it back to the working directory

```
~/Des/c/learngit git master !1 ❯ cat readme
Git is a distributed version control system.
Git is free software under the GPL(General Public Lisence).
Git has a mutable index called stage.
My stupid boss still prefers SVN.
```

```
~/Des/c/learngit git master !1 ❯ git status
On branch master
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:   readme

no changes added to commit (use "git add" and/or "git commit -a")
```

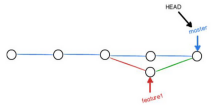
```
~/Des/c/learngit git master !1 ❯ git add readme
~/Des/c/learngit git master +1 ❯ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   readme
```

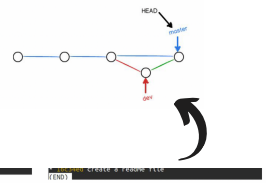
```
~/Des/c/learngit git master +1 ❯ git reset HEAD readme
Unstaged changes after reset:
M       readme
~/Des/c/learngit git master !1 ❯ git checkout -- readme
~/Des/c/learngit git master ❯ git status
On branch master
nothing to commit, working tree clean
~/Des/c/learngit git master ❯ cat readme
Git is a distributed version control system.
Git is free software under the GPL(General Public Lisence).
Git has a mutable index called stage.
```

5. Branch

- Solve Conflicts - 3
  - What if we have conflicts when merging two branches
    - Use `git log --graph --pretty=oneline --abbrev-commit` to see the branch merging status



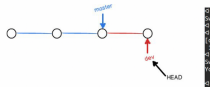

```
*   ff484ee (HEAD -> master) fix conflict
|\
| * 20fd55f (feature) add and easy
* | 4335e46 add & easy
|/
* 0f1caa8 branch dev test
* e8f3dc5 (origin/master) remove tmp
* 5811aa8 add test
* 30b788f understand how stage works
* 1f3a180 append GPL
* 63d4951 add distributed
* 16c34ed create a readme file
(END)
```

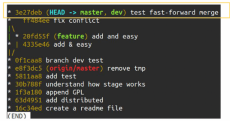


- Recursive Merge
  - `git merge --no-ff -m "message" <branch>`

5. Branch

- Merging Strategy
  - Fast-forward Merge
    - `git merge <branch>`





5. Tag

- Give your commit a nickname
- Create a tag
  - `git tag <tag_name>` ## default commit: HEAD
  - `git tag <tag_name> <commit_id>`
  - `git tag <tag_name> -m "tag message"`
- List all tags
  - `git tag`
- List tag info
  - `git show <tag_name>`
- Delete a tag
  - `git tag -d <tag_name>`
- Push a tag
  - `git push origin <tag_name>`
- Push all tags
  - `git push origin --tags`
- Delete tag in remote repo
  - `git push origin :refs/tags/<tagname>`

```
Makefile   car.hpp   car.cpp   sensor.cpp   engine.hpp
c_eg > car_eg > Makefile
CC=g++
CPPFLAGS=-Wall -std=c++17

default: car

car: car.cpp car.hpp engine.o sensor.o navigation.o
	$(CC) $(CPPFLAGS) engine.o sensor.o navigation.o car.cpp -o car

engine.o: engine.cpp engine.hpp
	$(CC) $(CPPFLAGS) -c engine.cpp

sensor.o: sensor.cpp sensor.hpp
	$(CC) $(CPPFLAGS) -c sensor.cpp

nevigation.o: navigation.cpp navigation.hpp
	$(CC) $(CPPFLAGS) -c navigation.cpp

clean:
	rm -f *.o car
```

Pros and Cons Comparison Table

Merging

(+) Only one commit is created per merge.

(+) Does not change existing commit history, so you're less likely to screw over others working on the same branch.

(+) Your steps can be fully retraced because you know when each merge was performed.

(-) If you have to merge often, these merge commits may pollute your history and make it harder to understand.

(-) Your history will still have interweaving branches that may make navigation (`git log`, `git bisect`, etc.) harder.

Rebasing

(-) A new commit is made for every commit you rebase.

(-) Changes existing commit history. If you misuse this command, you could mess up an important branch like `main` for everyone else. See the Golden Rule of Rebasing.

(-) It is difficult to see when a rebase actually occurred.

(+) No unnecessary merge commits polluting your history, making it easier to understand.

(+) Keeps your history as linear as possible, making it easier to navigate with such commands.

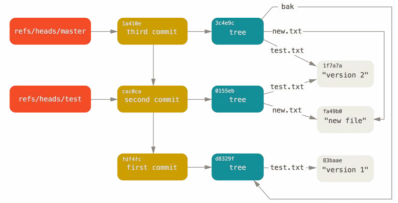
4 C - Dynamic Memory Management

Some common issues

- Using Before Writing
  - Malloc memory contents initialize to garbage
  - We need to write to it before trying to use it
- Forgetting to free
  - Causes memory leak
- Double free (freeing the same address twice)
  - Causes undefined behavior and maybe a seg-fault
- Using after free (dangling pointer)
  - Causes undefined behavior

1.5 C - Libraries

- Standard I/O and File I/O in C: <stdio.h>
  - https://www.tutorialspoint.com/c_standard_library/stdio_h.htm
- Standard I/O
  - `printf`: send formatted output to stdout
    - `%d`: int; `%f`: float, `%p`: pointer addrs; `%s`: string; `%c`: char ...
  - `getchar`: read a character from stdin
  - `gets`: read a line from stdin
  - `puts`: write a string to stdout
- File I/O:
  - Use pointers to access files in various modes
    - `r`: reading mode; `w`: writing mode; `a`: appending mode
  - `FILE *fopen(const char *filename, const char *mode)`: Opens the filename pointed to by filename using the given mode.
  - `int fclose(FILE *stream)`: Closes the stream. All buffers are flushed.
  - `fread`: Reads data from the given stream into the array
  - `fwrite`: Writes data from the array pointed to by ptr to the given stream.



Git Overall Picture

1.0 Git Internal

- What happens after running `git init`?
  - Initialize an empty git repo in current directory;
  - Everything inside this repo(directory) will be tracked
  - A new folder `.git` : contain version control information
- Let's take a further look into this `.git` folder
  - Core parts:
    - **HEAD** (file): points to the branch you currently check out
    - **index** (file): where Git stores your staging area information
    - **objects** (dir): stores all the content for your database
    - **refs** (dir): stores pointers into commit objects (branches, tags, remotes and more)
  - Other parts:
    - **description** (file): used only by the GitWeb program, so don't worry about it.
    - **config** (file): contains your project-specific configuration options
    - **info** (dir): keeps a global exclude file for ignored patterns that you don't want to track in a .gitignore file
    - **hooks** (dir): contains your client- or server-side hook scripts (https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks#_git_hooks)

1.1 Git Object - tree

c36d4..

tree		size
blob	5b1d3	README
tree	03e78	lib
tree	cdc8b	test
blob	cba0a	test.rb
blob	911e7	xdiff

- `tree`: a simple object that has a bunch of pointers to blobs and other trees
  - Generally, it represents the contents of a directory or subdirectory
- Check the contents of any tree:
  - `git show <SHA hash>`

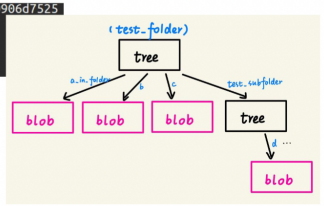
```
tree f15f9667f2afed857635e83e42cc17c0906d7525    test_folder
~/Des/c/learngit  master ?3 ❯ ls test_folder
a_in_folder  b_in_folder  c_in_folder  test_subfolder
~/Des/c/learngit  master ?3 ❯ git show f15f9667f2afed857635e83e42cc17c0906d7525
```

```
tree f15f9667f2afed857635e83e42cc17c0906d7525

a_in_folder
b_in_folder
c_in_folder
test_subfolder/
(END)
```

  - `git ls-tree <SHA hash>`

```
~/Des/c/learngit  master ?3 ❯ git ls-tree f15f9667f2afed857635e83e42cc17c0906d7525
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391    a_in_folder
100644 blob 61780798228d17af2d34fce4cfbdf35556832472    b_in_folder
100644 blob f2ad6c76f0115a6ba5b00456a849810e7ec0af20    c_in_folder
040000 tree 723f4575a99cfed82508fa44242a7b0012c0f576    test_subfolder
```



GIT is a directed acyclic graph

Show git log in a graph way with git log --graph --pretty=oneline --abbrev-commit

1.1 Git Object Model

- All the history information of a project is stored in files referenced by a 40-digit "object name" like this:

```
1f3a18099a72266b46325e9a0c4faa47c1d453b5
63d49519f77488dff5a61a00a3544d5fe1c7374c
16c34ed1b23f58b28fc1819c70eced883619f3fd
```

- The name is calculated by taking the SHA1 hash of the **contents** of the object
  - SHA1: a cryptographic hash function
  - It is virtually impossible to find two different objects with the same name
- The git objects: (type, size, content)
  - size = size of the content
  - content: depend on what type of the object is
- Object types
  - `blob`: used to store file data - it is generally a file
  - `tree`: basically like a directory - it references a bunch of other trees and/or blobs
  - `commit`: points to a single tree, marking it as what the project looked like at a certain point in time
  - `tag`: a way to mark a specific commit as special
- List all objects:
  - `git rev-list --objects --all`
  - `git cat-file --batch-check='%(objectname) %(objecttype)' --batch-all-objects`
- Checkout the type of a git object
  - `git cat-file -t <SHA hash>`

```
~/Des/c/learngit  master ?3 ❯ git cat-file --batch-check='%(objectname) %(objecttype)' --batch-all-objects
0a4fcb855cecb952023cebb946677f0f3d6aa08c tree
230ee42402690ab966327172559a760dc7fe71e9 blob
925fd2cef6c1d7eb0ecb28a9da797b20280cf2a9 commit
9d926bb2dda97b9bc2d4af65B177293b6a2750e0 tree
e1b795c9ac7614f8181f53cd8076d763f4860343 commit
```

```
~/Des/c/learngit  master ?3 ❯ git cat-file 9d926bb -p
100644 blob 230ee42402690ab966327172559a760dc7fe71e9    testfile
~/Des/c/learngit  master ?3 ❯ git cat-file 0a4fcb855cecb952023cebb946677f0f3d6aa08c -p
100644 blob 230ee42402690ab966327172559a760dc7fe71e9    testfile
100644 blob 230ee42402690ab966327172559a760dc7fe71e9    testfile_copy
```

```
~/Des/c/learngit  master ?1 ❯ git cat-file -t 16c34ed
commit
~/Des/c/learngit  master ?1 ❯ git cat-file -t 90fb4f10
blob
```

1.1 Git Object - commit

ae668..

commit	size
tree	c4ec5
parent	a149e
author	Scott
committer	Scott
my commit message goes here and it is really, really cool	

- `commit`: links a physical state of a tree with metadata
  - Description of how we got there and why
  - It contains:
    - Tree: The SHA1 name of a tree object, representing the contents of a directory at a certain point in time
    - Parent commit: The SHA1 name of some number of commits which represent the immediately previous step(s) (eg: merging commits may have multiple parent nodes)
    - Author: person responsible for this change (originally wrote the patch)+ date
    - Committer: person who actually created the commit on behalf of the author(last applied the patch) + date
    - Comment: describing this commit
  - Difference between author and commiter: https://ivan.bessarabov.com/blog/git-author-committer
- Check the contents of any commit:
  - `git log`
  - `git show --pretty=raw <SHA hash>`
  - `git show -s --pretty=raw <SHA hash>`

```
git show --pretty=raw eda302b62bb676c04c77bc46c1726b20eda155cd 111x16
commit eda302b62bb676c04c77bc46c1726b20eda155cd
tree bbf3e3b5efa7a146393ac3f25fdf7c0e0581b392
parent da67befb9c0fa57ee084ee29b368cb3b810f926
author YuxingQiu <yuxqiu@gmail.com> 1614325739 -0800
committer YuxingQiu <yuxqiu@gmail.com> 1614325739 -0800

    add test_subfolder

diff --git a/test_folder/test_subfolder/d_in_folder b/test_folder/test_subfolder/d_in_folder
new file mode 100644
index 0000000..4bcfe98
--- /dev/null
+++ b/test_folder/test_subfolder/d_in_folder
@@ -0,0 +1 @@
+d
(END)
```

```
git show --pretty=raw -s eda302b62bb676c04c77bc46c1726b20eda155cd
commit eda302b62bb676c04c77bc46c1726b20eda155cd
tree bbf3e3b5efa7a146393ac3f25fdf7c0e0581b392
parent da67befb9c0fa57ee084ee29b368cb3b810f926
author YuxingQiu <yuxqiu@gmail.com> 1614325739 -0800
committer YuxingQiu <yuxqiu@gmail.com> 1614325739 -0800

    add test_subfolder
(END)
```

3.2 Khan's Algorithm

Basic principle:

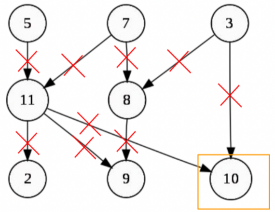
- If we have an edge (u => v), then u should comes before v in the ordering
- Topological ordering is possible iff. the graph has no directed cycles

- First, find a list of "start nodes" which have no incoming edges and insert them to a set ***s*** (at least one such node mush exist in a non-empty acyclic graph)

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edge

while S is not empty do
    remove a node n from S
    add n to L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S

if graph has edges then
    return error   (graph has at least one cycle)
else
    return L   (a topologically sorted order)
```



- S = (10, 9) => (9), n = 10
- L = [3, 5, 7, 11, 8, 2] => [3, 5, 7, 11, 8, 2, 10]
- E = [ ]

GIT BASICS

`git init <directory>`	Create empty Git repo in specified directory. Run with no arguments to initialize the current directory as a git repository.
`git clone <repo>`	Clone repo located at `<repo>` onto local machine. Original repo can be located on the local filesystem or on a remote machine via HTTP or SSH.
`git config user.name <name>`	Define author name to be used for all commits in current repo. Devs commonly use `--global` flag to set config options for current user.
`git add <directory>`	Stage all changes in `<directory>` for the next commit. Replace `<directory>` with a `<file>` to change a specific file.
`git commit -m "<message>"`	Commit the staged snapshot, but instead of launching a text editor, use `<message>` as the commit message.
`git status`	List which files are staged, unstaged, and untracked.
`git log`	Display the entire commit history using the default format. For customization see additional options.
`git diff`	Show unstaged changes between your index and working directory.

UNDOING CHANGES

`git revert <commit>`	Create new commit that undoes all of the changes made in `<commit>`, then apply it to the current branch.
`git reset <file>`	Remove `<file>` from the staging area, but leave the working directory unchanged. This unstages a file without overwriting any changes.
`git clean -n`	Shows which files would be removed from working directory. Use the `-f` flag in place of the `-n` flag to execute the clean.

REWRITING GIT HISTORY

`git commit --amend`	Replace the last commit with the staged changes and last commit combined. Use with nothing staged to edit the last commit's message.
`git rebase <base>`	Rebase the current branch onto `<base>`. `<base>` can be a commit ID, branch name, a tag, or a relative reference to HEAD.
`git reflog`	Show a log of changes to the local repository's HEAD. Add `--relative-date` flag to show date info or `--all` to show all refs.

GIT BRANCHES

`git branch`	List all of the branches in your repo. Add a `<branch>` argument to create a new branch with the name `<branch>`.
`git checkout -b <branch>`	Create and check out a new branch named `<branch>`. Drop the `-b` flag to checkout an existing branch.
`git merge <branch>`	Merge `<branch>` into the current branch.

REMOTE REPOSITORIES

`git remote add <name> <url>`	Create a new connection to a remote repo. After adding a remote, you can use `<name>` as a shortcut for `<url>` in other commands.
`git fetch <remote> <branch>`	Fetches a specific `<branch>`, from the repo. Leave off `<branch>` to fetch all remote refs.
`git pull <remote>`	Fetch the specified remote's copy of current branch and immediately merge it into the local copy.
`git push <remote> <branch>`	Push the branch to `<remote>`, along with necessary commits and objects. Creates named branch in the remote repo if it doesn't exist.

`git stash`	**`git stash pop`**
Save modified and staged changes	write working from top of stash stack
`git stash list`	**`git stash drop`**
list stack-order of stashed file changes	discard the changes from top of stash stack

To record changes to a few *parts* of some of your changed files (i.e., patch chunks), you have a few options.

- `git commit --interactive`. This will bring up an interactive prompt that will allow you to choose what you'd like to commit; to get started, try typing "`status`" at the "`What now?`" prompt to see what's changed, and then "`patch`" to interactively make choices. (Resist the urge to shout "*whatnow?!*" whenever you see the prompt.)
- `git add -i`. This is like `commit --interactive`, but it doesn't create any commits. This command just tells git that on the next `commit`, include these changes.
- `git add -p`. This command allows you to only add patches of a file. Use this if you have edited a large number of files, but you don't want all of the modifications to be one large commit. You will be asked about each chunk of diff in every file about whether it should be added or not. Some useful commands are: `s` - split the current diff up into smaller diffs to accept/reject or `y/n` - yes or no to accept the current diff.

To get a graphical view of your repository's history, **run gitk**

git-help - Display help information about Git

`git rm [file]`	
delete the file from project and stage the removal for commit	
`git mv [existing-path] [new-path]`	
change an existing file path and stage the move	
`git log --stat -M`	
show all commit logs with indication of any paths that moved	

`git config --global user.name "[firstname lastname]"`	
set a name that is identifiable for credit when review version history	
`git config --global user.email "[valid-email]"`	
set an email address that will be associated with each history marker	
`git config --global color.ui auto`	
set automatic command line coloring for Git for easy reviewing	

`git log`	
show the commit history for the currently active branch	
`git log branchB..branchA`	
show the commits on branchA that are not on branchB	
`git log --follow [file]`	
show the commits that changed file, even across renames	
`git diff branchB...branchA`	
show the diff of what is in branchA that is not in branchB	
`git show [SHA]`	
show any object in Git in human-readable format	

Both `~` and `^` on their own refer to the parent of the commit (`~~` and `^^` both refer to the grandparent commit, etc.) But they differ in meaning when they are used with numbers:

- `~2` means **up two levels in the hierarchy**, via the first parent if a commit has more than one parent
- `^2` means **the second parent** where a commit has more than one parent (i.e. because it's a merge)

These can be combined, so `HEAD~2^3` means `HEAD`'s grandparent commit's third parent commit.

`user.name <name>`	Define the author name to be used for all commits by the current user.
`git config --global user.email <email>`	Define the author email to be used for all commits by the current user.
`git config --global alias. <alias-name> <git-command>`	Create shortcut for a Git command. E.g. `alias.glog "log --graph --oneline"` will set "`git glog`" equivalent to "`git log --graph --oneline`.
`git config --system core.editor <editor>`	Set text editor used by commands for all users on the machine. `<editor>` arg should be the command that launches the desired editor (e.g., vi).
`git config --global --edit`	Open the global configuration file in a text editor for manual editing.

GIT LOG

`git log -<limit>`	Limit number of commits by `<limit>`. E.g. "`git log -5`" will limit to 5 commits.
`git log --oneline`	Condense each commit to a single line.
`git log -p`	Display the full diff of each commit.
`git log --stat`	Include which files were altered and the relative number of lines that were added or deleted from each of them.
`git log --author="<pattern>"`	Search for commits by a particular author.
`git log --grep="<pattern>"`	Search for commits with a commit message that matches `<pattern>`.
`git log <since>..<until>`	Show commits that occur between `<since>` and `<until>`. Args can be a commit ID, branch name, HEAD, or any other kind of revision reference.
`git log -- <file>`	Only display commits that have the specified file.

`git diff --cached`	Show difference between staged changes and last commit

GIT RESET

`git reset`	Reset staging area to match most recent commit, but leave the working directory unchanged.
`git reset --hard`	Reset staging area and working directory to match most recent commit and **overwrites all changes** in the working directory.
`git reset <commit>`	Move the current branch tip backward to `<commit>`, reset the staging area to match, but leave the working directory alone.
`git reset --hard <commit>`	Same as previous, but resets both the staging area & working directory to match. **Deletes** uncommitted changes, and **all commits after** `<commit>`.

GIT REBASE

`git rebase -i <base>`	Interactively rebase current branch onto `<base>`. Launches editor to enter commands for how each commit will be transferred to the new base.

GIT PULL

`git pull --rebase <remote>`	Fetch the remote's copy of current branch and rebases it into the local copy. Uses git rebase instead of merge to integrate the branches.

GIT PUSH

`git push <remote> --force`	Forces the `git push` even if it results in a non-fast-forward merge. Do not use the `--force` flag unless you're absolutely sure you know what you're doing.
`git push <remote> --all`	Push all of your local branches to the specified remote.