HW2

- Due: Fri. Sep. 14, 2018, 11:59pm (PST)
- Directory name in your github repository for this homework (case sensitive): `hw2`
    - Once you have cloned your `hw-usc_username` repo, create this `hw2` folder underneath it (i.e. `hw-usc_username/hw2`).
    - If your `hw-usc_username` repo has not been created yet, please do your work in a separate folder and you can copy over relevant files before submitting.

Problem 1 (More git questions, 10%)

In this problem, we will be working with a fictional Sample Repository, which we pretend is located at https://github.com/usc-csci104-fall2016/SampleRepo. (There is no actual repo there, so don't be surprised if you cannot actually clone it or otherwise interact with it.) This exercise is designed to measure your understanding of the file status lifecycle in git. Please frame your answers in the context of the following lifecycle based on your interaction with the repository as specified below:

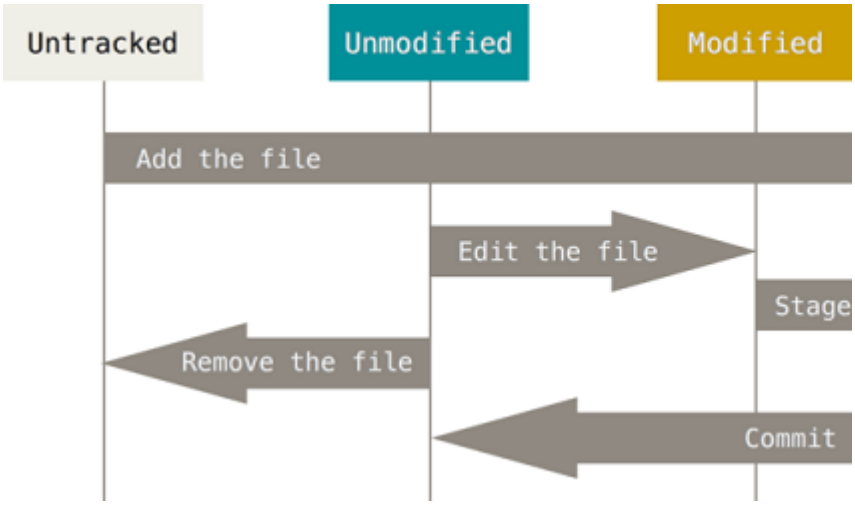

figure courtesy of the Pro Git book by Scott Chacon

Parts (a) through (f) should be done in sequence. In other words, when you get to part (f), you should assume that you already executed the earlier commands (a), (b), ..., (e). You **must** use the terminology specified in the lifecycle shown above, for example the output of `git status` is not accepted as a valid answer. For the purposes of this question, you can assume you have full access (i.e., read/write) to the repository.

Part (a):

What is the status of `README.md` after performing the following operations:

```
#Change directory to the home directory
cd
#Clone the SampleRepo repository
git clone git@github.com:usc-csci104-fall2016/SampleRepo.g
#Change directory into the local copy of SampleRepo
cd SampleRepo
```

Part (b):

What is the status of `README.md` and `fun_problem.txt` after performing the following operations:

```
#Create a new empty file named fun_problem.txt
touch fun_problem.txt
#List files
ls
```

```
#Append a line to the end of README.md
echo "Markdown is easy" >> README.md
```

Part (c):

What is the status of `README.md` and `fun_problem.txt` after performing the following operation:

```
git add README.md fun_problem.txt
```

Part (d):

What is the status of `README.md` and `fun_problem.txt` after performing the following operations:

```
git commit -m "My opinion on markdown"
echo "Markdown is too easy" >> README.md
echo "So far, so good" >> fun_problem.txt
```

Part (e):

What is the status of `README.md` and `fun_problem.txt` after performing the following operations:

```
git add README.md
git checkout -- fun_problem.txt
```

Also, what are the contents of `fun_problem.txt`? Why?

Part (f):

What is the status of `README.md` after performing the following operation:

```
echo "Fancy git move" >> README.md
```

Explain why this status was reached.

Problem 2 (Review Material, 0%)

Carefully review linked lists (Chapters 4, 9.2) and Recursion (Chapters 2, 5).

Problem 3 (Abstract Data Types, 15%)

For each of the following data storage needs, describe which abstract data types you would suggest using. Natural

choices would include `list`, `set`, `map`, but also any simpler data types that you may have learned about before.

Try to be specific, e.g., rather than just saying "a list", say "a list of integers" or "a list of structs consisting of a name (string) and a GPA (double)". Also, please give a brief explanation for your choice: we are grading you at least as much on your justification as on the correctness of the answer. Also, if you give a wrong answer, when you include an explanation, we'll know whether it was a minor error or a major one, and can give you appropriate partial credit. Also, there may be multiple equally good options, so your justification may get you full credit.

1. a data type that stores voting statistics for each zip code, and allows for speedy access of these statistics when given a zip code.
2. a data type that stores all of the students that solved a chocolate problem in CSCI 104.
3. a data type that stores all of the collectible pokemon in Pokemon Go, ordered by their pokedex index (an integer from 1 to 151)
4. a data type for the database of the website rottentomatoes: given a movie name and the year it was released, it brings up all of the reviews associated with that movie.

Problem 4 (Linked Lists, Recursion, 10%)

Consider the following C++ code. What linked list is returned if funcA is called with the input linked list `1,2,3,4,5`? All of the points for this problem will be assigned based on your explanation, since we have full faith in your ability to run this program and copy down the answer. We **strongly** recommend solving this by hand, and only using a compiler to verify your answer.

```
struct Node {
    int value;
    Node *next;
};

void funcB (Node* in1, Node* in2);

Node* funcA (Node* in)
```

```
{
   if (in == NULL) return NULL;
   Node* out = NULL;
   if (in->next != NULL)
   {
       out = funcA (in->next);
       funcB (in, out);
       in->next = NULL;
       return out;
   }
   return in;
}

void funcB (Node* in1, Node* in2)
{
   if (in2->next != NULL)
   {
       funcB (in1, in2->next);
       return;
   }
   in2->next = in1;
}
```

Problem 5: Selection Sort (15%)

For this problem, you will be implementing selection sort on a doubly-linked list.

A skeleton that defines the item struct and the function signatures has been provided for you in `selection_sort.h` in the homework resources repository. Please copy it to your homework folder.

Write your code in `selection_sort.cpp`, and include `selection_sort.h` to get the Item struct and function signatures. You may create additional helper functions inside `selection_sort.cpp` if you want, but do not add them to the header.

Part A: findMin()

First, you will need to implement `cpp Item* findMin(Item * head)`

This function will search through the given linked list and return a pointer to the `Item` with the smallest value. If there

are multiple `Items` that contain the smallest value, then your function may return any of them.

If and only if the list is empty should this function should return nullptr. It should run in linear time.

Part B: LLSelectionSort()

Next, you will implement the actual selection sort routine: `cpp Item* LLSelectionSort(Item* head);`

This function will sort a list in ascending order using selection sort. Recall that selection sort first moves the smallest item to the front, then the 2nd smallest after that, until the list is sorted. We highly recommend using your `findMin()` function in `LLSelectionSort()`.

If `head` is not nullptr, `LLSelectionSort()` will return the pointer to the new head of the sorted list. If `head` is nullptr, `LLSelectionSort()` will return null. It must run in O(n²).

Problem 6: Circular Linked List (25%)

For this problem, you will be using your skills with linked lists to create a class that stores a doubly-linked list of ints. The list will also be circular, meaning that the head and tail of the list link to each other. The list supports all operations in the List ADT, including insert, remove, get, and set, as well as a simple iterator interface which is explained below.

A class skeleton has been provided in `circular_list_int.h`. Copy this header to your homework folder and, in `circular_list_int.cpp`, implement the functions in `CircularListInt` according to their descriptions in the header file.

Circular Indicies

Since this class is a circular list, it is meant to store data that will be accessed continuously in a circle, and it would be nice if we could move around the circle just by incrementing the index. So, indicies past the end of the list should "wrap around" back to the front of the list. For example, if a list has 5 members, index 5 wraps back to element 0, index 6 maps to element 1, and index 10 maps to element 0. This rule applies to `get()`, `set()`, and `remove()`.

The exception to this is when the list has 0 length, and there are no elements at all. No amount of wrapping would produce a valid index here! Instead, `get()` should return 0,

and `set()` and `remove()` should just return without modifying the list.

What's all this "size_t" business?

You might notice that all functions with parameters for an index accept it as a `size_t`. You haven't really seen it before, but `size_t` is the the correct type to use for array indicies. It is defined as a type large enough to hold the size of any object in memory, up to and including the size of your entire PC's memory. It's also an unsigned type, so it can't hold negative values (there are no negative sizes), and you don't have to worry about handling them. More then that, it also semantically indicates that a variable is supposed to hold the size of something, not just some random integral quantity. `size_t`s are used all over the standard library for indices and sizes of collections, so it's best to start getting used to them now.

You might notice that you get compiler warnings when you try to compare a `size_t` to a regular integer, for example in code like this:

```
std::vector<int> foo;

for(int index = 0; index < foo.size(); ++index)
{
    /* do stuff */
}
```

The issue is that `std::vector::size()` returns an (unsigned) `size_t`, and it's being compared to a (signed) int in the for-loop statement. Due to how C type conversion rules work, this can lead so strange and unpredictable results like one of the operands being truncated or converted from negative to positive. The easiest fix is to just delare all variables containing indices and counts as `size_t`s. Failing that, you can just cast the result of size to an integer to an int, like this: `index < ((int)foo.size())`. That will work fine as long as your vector doesn't have more than 2^{31}-1 items in it, and that won't happen in this class!

Problem 7: Duck Duck Goose (25%)

For this problem, you will implement a program that simulates a game of duck duck goose. You'll model the flow

of the game according to the rules, and print out what's going at on at each step. Finally, your program will determine and print the game's winner.

You will store the ID number and order of players in a `CircularListInt` that models how the players are sitting in a circle. There will also be one player, "it" who is outside the circle and will compete for a spot in the next round. Each round of the game will be played as follows:

Gameplay

Here's how we will play duck duck goose.

First, the "it" player will go around the circle of players calling "duck" until they choose a player to be "goose". Then, the goose will chase "it" around the circle. If the goose catches "it", then the "it" player is out, and a new "it" is randomly selected from the players other than the goose. If "it" manages to run around the circle without being tagged, then "it" takes the goose's spot and the goose becomes the new "it".

Here's exactly how you'll model this game:

1. Let's say that the game has **n** players not including "it". Each round, the "it" player will generate **m**, a random number between 0 and 4**n-1 inclusive. Starting at the head of the list, "it" will then advance ***m** places, printing "<player id> is a Duck." to the output for each player passed. Finally, after skipping past **m** players, "it" will reach the chosen player and and print "<player id> is a Goose!".
2. We will simulate this chase using a random number. Both the goose and "it" choose a random number between 1 and 49 inclusive. In the case of a tie, re-pick both random numbers until there is no longer a tie.
3. If "it" has a higher number, the it player takes the goose's spot in the circular list and the goose becomes the new it. Output "<it player id> took <goose player id>'s spot!".
4. If the goose has a higher number, the goose keeps its spot in the circular list, and the it player is out and is removed from the game. Output "<it player id> is out!".

5. If there are now no remaining players apart from the goose, then the goose player wins. The program should output "Winner is <goose player id>!".
6. Otherwise, choose a new "it" randomly from the remaining players other than the goose, then output "<new it player id> was chosen as the new it."

Output Instructions

The procedure above gives several places where you are supposed to output text. Output all text to the `std::ostream` passed to the `simulateDDGRound()` function, with a newline after each message. Make sure to print messages exactly as they are written in the section above, including correct case and puctuation.

Where we give placeholders, like "<it player id>", make sure to substitute the ID of the relevant player, as a decimal integer without any extra spaces or padding. Don't literally output `<player id> is a Duck.`

Game Configuration File

Information about the game will be read from a simple configuration file, with one value on each line. The format is as follows:

```
line 1: random seed (unsigned integer)
line 2: number of players in the game (1-indexed)
line 3: ID of "it" player
lines 4 to n+3: IDs of each seated player, from player 0 (
```

The first line is the seed that you will use for all random numbers generated throughout the program. Don't forget that this value is an **unsigned** int, so it might be too large to fit in a standard **int**.

The second line is the 1-indexed number of players who will be taking part in the game. This includes the "it" player, and will always be at least two. Line 3 has the ID of the player who will start as "it".

Subsequent lines have the IDs of each player who will be playing the game, in order. The player on line 4 is player 0 and starts at the head of the list, and the last player is player **n**-1 and starts at the tail of the list.

Player IDs will be unique numbers between 1 and 2^{31}-1, and will not follow any ordering in the configuration file. Player IDs will represent each player in all printouts, and can be imagined to correspond 1:1 to each player's name. Also, you may assume that the config file will have correct syntax. There will always be at least two players, and there will never be more players than will fit in a signed int.

Randomness

For this assignment, you will have to generate random numbers to model the players' decisions. For this, you'll be using the C function `rand()`. Each time you call it, `rand()` returns a random number in the range of [0, RAND_MAX], where RAND_MAX is at least 2^16-1 (=32767). To scale that number into a range where you can use it, just use the modulo operator with an integer 1 higher than the largest result you want. For example, the following snippet will return random numbers between 0 and 1000.

```
int random_int = rand() % 1001;
```

Now, rand is not a true random number generator, but rather a pseudorandom sequence generator. Internally it works by performing some mathematical operation on a *seed value* to produce the next value in the sequence, which is returned and also stored for the next time `rand()` is called. The mathematical operation done on the seed is such that the sequence jumps unpredicatably around its range, and successive calls to `rand()` approximate a sequence of random integers. The important takeaway here is that the numbers returned from `rand()` depend **only** on the initial random seed. If you set the random seed to the same value, and your program makes calls to `rand()` in exactly the same order, then you will get the **exact** same sequence of random numbers each time, and your program's behavior will be consistent.

This is why the random seed is specified in the config file. When your program first starts, it should call `srand()` with the given seed. Then, the output will be the same each time your program is run with that configuration file, and it will be much easier for you to debug and test.

NOTE: If you have any questions about how `rand()` works, you can look at its Linux manual page here.

Fun fact: while rand() is plenty random for our needs, the numbers it generates turn out to be biased by a small amount. For things like cryptography and scientific simulations, people need much higher-quality sources of randomness. C++11 includes a (more complicated but) much improved random generator library that provides several well-defined, uniform sources of randomness. Check out this neat video for more info.

For even more sensitive (read: paranoid) cryptography applications, there are actually hardware chips that generate truly random numbers. They use an arrangement of semiconductors where the circuit's output voltage is determined by quantum probability fields, and is thus completely random.

Setting Up your Program

Write your code in `duck_duck_goose.cpp`, and in it include `duck_duck_goose.h` from the homework resources repository. This header contains the `Game` struct that you will use in your code.

Your code should be split into at least two functions. First, you must have a `main()` function that sets up the game. It should take two arguments on the command line, the paths to the configuration and output files. Your program will be executed like this: `./duck_duck_goose config.txt output.txt`

`main()` should open the config file and read it, and create the game struct with the given player names, then, set the initial random seed by calling the function `srand()` with the random seed from the config file as an argument.

`main()` should then call the `simulateDDGRound()` function until a player wins, writing the output of the game to the output file. This function is defined as follows:

```
void simulateDDGRound(GameData * gameData, std::ostream &
```

This function should run through the procedure in the Gameplay section to simulate one round of duck duck goose. It does not need to return anything, but should set the members of `gameData` appropriately to reflect the new state of the game. It does, however, need to run in $O(n^2)$ for the number of players in the game.

In `duck_duck_goose.cpp` you may add as many helper functions as you wish, but make sure to implement and use `simulateDDGRound()` exactly as described, because we will be calling it both inside and outside of your main function.

You may **not** use any containers from the standard library on this assignment. Instead, you should use your CircularListInt class to store the player data! It would probably be prudent to get that class tested and working first, before you start on your Duck Duck GOose assignment. Good luck, we're all counting on you!

Note: A Testable Design

You might be wondering why we bothered defining an interface for `simulateDDGRound()` instead of just giving you a command-line interface for your program. The reason is, that it makes your code easier to test.

Being able to call `simulateDDGRound()` outside of your config file parsing and setup logic lets us grade you more fairly. If all we tested was your complete executable, a simple mistake in your `main()` function that caused your configuration file to be read wrong could potentially dock you a huge number of points. However, now that your game logic and parsing logic are separated internally with a known interface, we can check each one on its own, and give you a more accurate grade.

The other reason is to check randomness. In order to see that you've set up the probabilities correctly, we'd like to do something like simulate the same round 1000 times and check that the outcomes are what they're supposed to be. To do that efficiently, we need to be able to execute each round by itself outside of your larger program. So, we had you split the logic up like this.

Tests like this, that divide up a codebase into smaller pieces and analyze them seperately, are called *unit tests*. They are considered good programming practice since they help detect bugs in complex programs, where it can be hard to trace an issue back to its source. Going forward, keep an eye out for places in your code where you can make things testable. It will help ensure that your programs work correctly, and get you more points!

Testing your Code

To do a decent job on this assignment, you will need to rigorously test your code, and verify that everything is behaving how it's supposed to. Eventually, over the course of this class, you will learn how to use Google Test to create your own tests, and soon it will be up to you to verify that your homework works inside the given specifications.

However, this is only homework 2, so we're going to give you a little help on testing. Inside the `homework_resources` repository, you'll find a `hw2-check` folder which has some testing code to get you started. We've created some utility functions for common tasks in testing, like building a circular list from a vector of elements, or checking that a linked list has been properly sorted. They should make the task of creating test cases much more straightforward. We've also gotten you started with a few simple test cases in the `basic_tests.cpp` file in the subfolder for each problem. Make sure to read through all our comments to learn how to use the testing utility functions.

First, you will need to set up the checker. Copy the `hw2-check` folder into your `hw2` directory, and open a new terminal inside `hw2-check`. Now, run the command `cmake .` to initialize the test suite code. You only have to do this once; once `cmake` completes successfully the checker is set up forever. Lastly, run the command `make check`. This will actually compile your code and the test code, and run the tests.

To get creating your own tests, just create a new .cpp file in one of the problems' subfolders, include the relevant headers, and start writing test cases! Next time you run `make check`, cmake will see the new cpp file and build it automatically. Your goal is to verify that your code behaves correctly in all situations, and you can best do that by accounting for normal and abnormal input. Try to think of boundary and edge cases that might cause trouble for your code, like a 0-element list, or a game with only two players. Always look through your code as you're writing it and debugging it, and think of situations that might cause it to break. Then, write a test case that sets up each situation, and verifies that code behaves correctly. Once you've covered every base you can think of, you can feel confident about your code!

Since the test cases aren't graded, we encourage you to submit your test case .cpp files on Piazza for your classmates to utilize. By all means, work together to create the most bulletproof test suite possible!

One last thing: The `hw2-check` build system is the **same** as what we use on our test suite, and the tests we run are a **superset** of the basic_tests files we gave you. So, if your code fails to compile with `hw2-check` or fails some of the default tests, you will be losing significant points on your assignment. It's not hard though, just don't forget to use hw2-check!

Commit then Re-clone your Repository

You can submit your homework here. Please make sure you have read and understood the submission instructions.

WAIT! You aren't done yet. Complete the last section below to ensure you've committed all your code.

Commit then Re-clone your Repository

Be sure to add, commit, and push your code in your directory to your `hw-usc-username` repository. Now double-check what you've committed, by following the directions below (failure to do so may result in point deductions):

1. Go to your home directory: `$ cd ~`
2. Create a `verify` directory: `$ mkdir verify`
3. Go into that directory: `$ cd verify`
4. Clone your hw-username repo: `$ git clone git@github.com:usc-csci104-fall2018/hw-username.git`
5. Go into your folder `$ cd hw-username/`
6. Recompile and rerun your programs and tests to ensure that what you submitted works.

USC > UCLA Muahaha