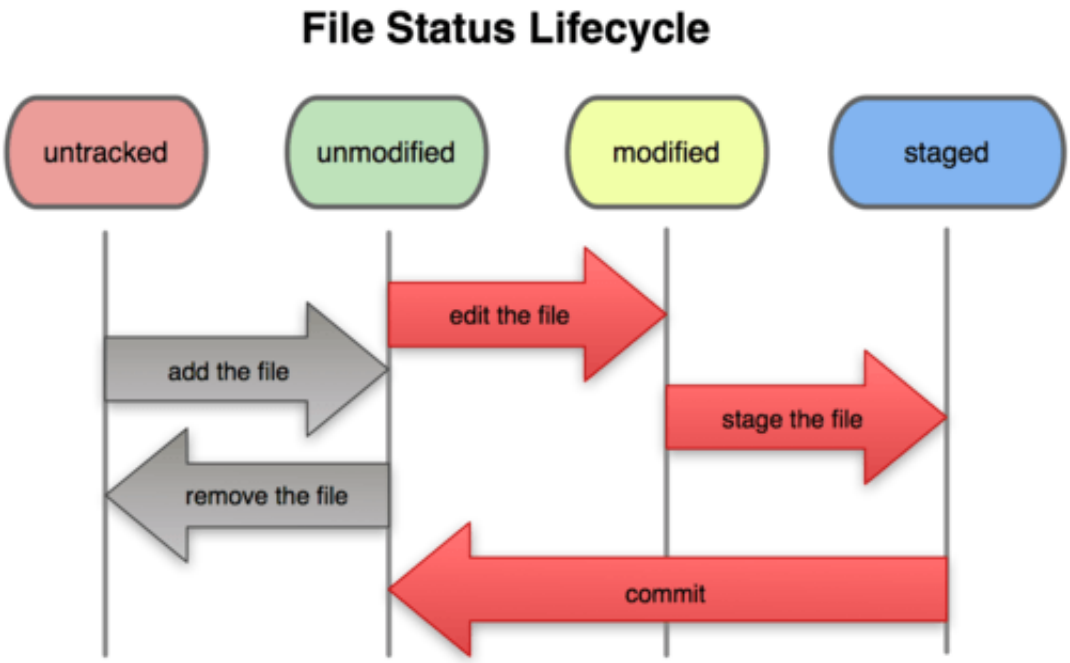
CSCI 104 - Fall 2015 Data Structures and Object Oriented Design

HW2

- Due: Monday, September 14th, 11:59pm PST
- Directory name in your github repository for this homework (case sensitive): `hw2`

Problem 1 (More git questions, 10%)

In this problem, we will be working with a Sample Repository 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 terminalogy 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-fall2015/SampleRepo.git
#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 (Runtime Analysis, 15%)

In Big-Θ notation, analyze the running time of the following three pieces of code/pseudo-code. Describe it as a function of the input size (here, `n`).

Part (a)

```
for (int i = 0; i < n; i ++)
    if (i % (int) sqrt(n) == 0) {
       for (j = 0; j < n; j ++)
          { /* do something that takes O(1) time */ }
    }
```

Part (b)

Notice that this code is very similar to what will happen if you keep inserting into an ArrayList (which you will see once we have covered ArrayLists in class).

```
int f (int n)
{
  int *a = new int [10];
  int size = 10;
  for (int i = 0; i < n; i ++)
     {
        if (i == size)
          {
             int newsize = 3*size/2;
             int *b = new int [newsize];
             for (int j = 0; j < size; j ++) b[j] = a[j];
             delete [] a;
             a = b;
             size = newsize;
          }
        a[i] = i*i;
     }
}
```

Part (c)

This is the exact same code as in part (b), except for changing the definition of `newsize`.

```
int f (int n)
{
  int *a = new int [10];
```

```
  int size = 10;
  for (int i = 0; i < n; i ++)
     {
        if (i == size)
          {
             int newsize = size + (int) sqrt(n);
             int *b = new int [newsize];
             for (int j = 0; j < size; j ++) b[j] = a[j];
             delete [] a;
             a = b;
             size = newsize;
          }
        a[i] = i*i;
     }
}
```

Problem 4 (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, i.e., 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. That way, when you give a wrong answer, 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 for a single web page all the words that appear on that web page.
2. a data type that stores for each student all classes that the student is enrolled in, and lets you check easily whether a particular student is enrolled in a particular class.
3. a data type that stores the names of countries in the order in which they will march in the Special Olympics opening ceremony.

Problem 5 (Linked Lists, Recursion, 25%)

Write a **recursive** function to split the elements of a singly-linked list into two sublists, one containing the elements less than or equal to a given number, the other containing the elements larger than the number. At the same time, the original list should **not** be preserved (see below). Your function must be recursive - you will get **NO** credit for an iterative solution.

You should use the following `Node` type from class:

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

Here is the function you should implement:

```
void split (Node*& in, Node*& smaller, Node*& larger, int pivot);
 /* When this function terminates, the following holds:
    - smaller is the pointer to the head of a new singly linked list containing
      all elements of "in" that were less than or equal to the pivot.
    - larger is the pointer to the head of a new singly linked list containing
      all elements of "in" that were (strictly) larger than the pivot.
    - the linked list "in" no longer exists (should be set to NULL).
```

Hint: by far the easiest way to make this work is to not `delete` or `new` nodes, but just to change the pointers.

While we will only test your `split` function, you will probably want to write some `main` code to actually test it.

Problem 6 (Linked Lists, 25%)

We have provided you an incomplete implementation of a doubly-linked list in the `homework-resources/hw2` folder. You can update/pull the `homework-resources` folder to obtain it and then copy it to your own hw2 directory in your own `hw_usc-username` repo. For visual reference it is linked here: llistint.h and llistint.cpp.

1. You need to examine the code provided and complete the `insert`, `remove`, and `getNodeAt` member functions in `llistint.cpp`. `getNodeAt` is a private helper function which will return a pointer to the i-th node and is used in several other member functions (and may help with `insert` and `remove`). Valid locations for insertion are 0 to SIZE (where SIZE is the size of the list and indicates a value should be added to the back of the list). Valid locations for removes are 0 to SIZE-1. Any invalid location should cause the function to simply return without modifying the list.

2. After completing the two functions above, you should write a separate program to test your implementation. You should allocate one of your `LListInt` items and make calls to `insert()` and `remove()` that will exercise the various cases you've coded in the functions. For example, if you have a case in `insert()` to handle when the list is empty and a separate case for when it has one or more items, then you should make a call to `insert()` when the list is empty and when it has one or more items. It is important that when you write code, you test it thoroughly, ensuring each line of code is triggered at some point. You need to think about how you can test whether it worked or failed as well. In this case, calls to `get()`, `size()`, and others can help give you visibility as to whether your code worked or failed.

We have provided a sample test program for you here testAddToEmptyList.cpp. Use it as a template and good example for how to write tests for individual features of your class.

Problem 7 (Using STL sets, 10%)

Using the STL `set` implementation, write a program that reads a text file, and then lets a user query interactively to test if certain words are in the file. Specifically, you should read a text file (whose name is given at the command line) that consists of arbitrary characters. Words consist only of letters (upper case or lower case); anything else separates words. (This is similar to HW1, except you don't have to worry about links now.)

Once you have read the file and stored it in a convenient format (STL sets of a suitable type), you should interactively query the user to enter strings. If the user enters the empty string (presses return), your program should terminate. For each non-empty string that the user enters, you should output `in the file` or `not in the file`. In looking for the string, you should ignore case. So if the file contains the word `Pokemon`, and the user types `poKEMon`, you should output `in the file`.

As a small piece of advice, the relevant STL `set` functions are `insert`, `erase`, and `count` (which will return 0 or 1, depending on whether the element is in the set). The function `find` can also work, but it relies on the concept of iterators, which we have not yet covered.