CSCI 104 – Homework 2

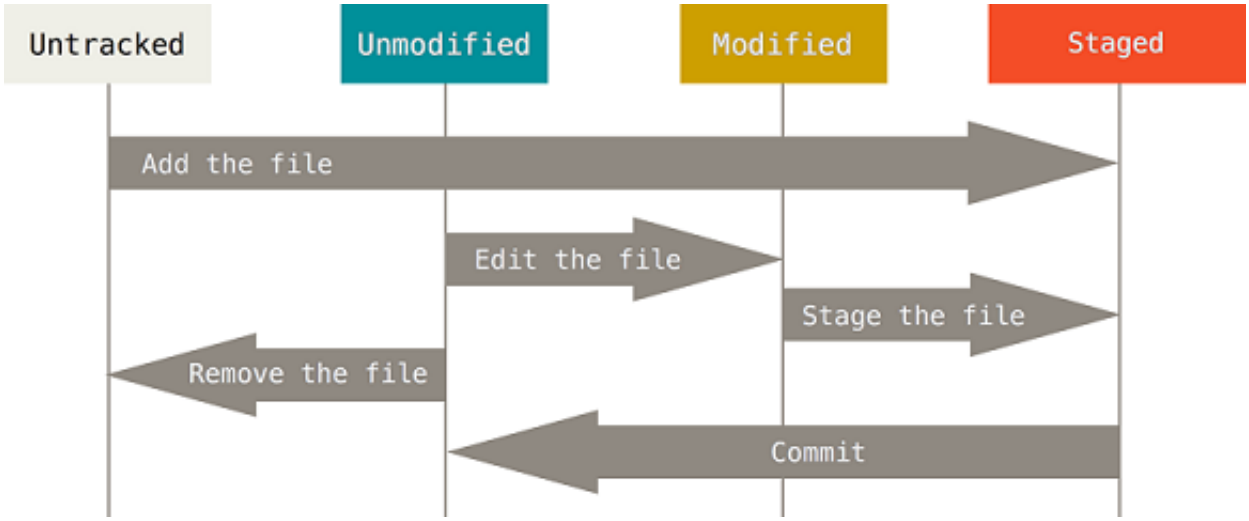
Data Structures and Object Oriented Design

HW2

- Due: Friday, February 5th, 11:59pm PST
- Directory name in your github repository for this homework (case sensitive): `hw2`
- Place your answers to problems 1,3,4,5 in a file name `hw2.txt`

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-spring2016/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, 20%)

In Big-Θ notation, analyze the running time of the following pieces of code/pseudo-code. Describe it as a function of the input size (here, `n`). You should **always** explain your work when solving mathematics problems.

Part (a)

```
for (int i = 0; i < n; i ++)
    if (A[i] == 0) {
       for (int j = 1; j < n; j *= 2)
          { /* do something that takes O(1) time */ }
    }
```

Part (b)

```
for (int i = 1; i < n; i ++)
{
   for (int j = i; j < n; j ++)
   {
        if (j % i == 0)
        {
            for (int k = 1; k < n; k *= 2)
               { /* do something that takes O(1) time */ }
        }
   }
}
```

Part (c)

```
int *a = new int [10];
int size = 10;
for (int i = 0; i < n; i ++)
{
```

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

Part (d)

Notice that this code is very similar to what happens if you keep inserting into a vector.

```
int *a = new int [10];
int size = 10;
for (int i = 0; i < n; i ++)
{
   if (i == size)
   {
       int newsize = 2*size;
       int *b = new int [newsize];
       for (int j = 0; j < size; j ++) b[j] = a[j];
       delete [] a;
       a = b;
       size = newsize;
   }
   a[i] = sqrt(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: we are grading you as much on your answer as for your justification. If 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 all of the past and present presidents of the U.S., and the order in which they served.
2. a data type that stores population estimates for each zip code, and allows for speedy access of the population when given a zip code.
3. a data type that stores all of the students that solved a tricky homework problem in CSCI 170.
4. a data type that stores a gradebook: given a student ID and assignment number, it lets you easily check the student's numeric grade for that assignment.

Problem 5 (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. 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)
{
   Node* out = NULL;
   if (in->next)
   {
       out = funcA(in->next);
       funcB(in, out);
       in->next = NULL;
       return out;
   }
   return in;
}

void funcB(Node* in1, Node* in2)
{
   if (in2->next)
   {
       funcB(in1, in2->next);
```

```
        return;
    }
    in2->next = in1;
}
```

Problem 6 (Linked Lists, Recursion, 20%)

Write a **recursive** function to merge the elements of two sorted (in increasing order), singly-linked lists into a single sorted, singly-linked list. The original lists 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:

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

Here is the function you should implement:

```
Node* merge (Node*& first, Node*& second);
```

When your merge function terminates, both first and second should be set to `NULL` (the original lists are not preserved).

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 `merge` function, you will probably want to write some `main` code to actually test it.

Problem 7 (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.

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 functions above, you should write a separate program to test your implementation. Please note that these tests **will** be graded. 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 in the `homework-resources` folder, `testAddToEmptyList.cpp`. Use it as a template and good example for how to write tests for individual features of your class.

Commit then Re-clone your Repository

Be sure to add, commit, and push your code in your `hw2` 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-spring2016/hw_usc-username.git`
5. Go into your hw2 folder `$ cd hw_username/hw2`
6. Recompile and rerun your programs and tests to ensure that what you submitted works.

•