CS-102 Design Principles: Project 5

Due date: Monday, February 27 at 11:00pm

In this project, you will create a **circularly linked list**, and use it to model a simple stone-moving game (like Mancala, if that helps). A circularly linked list (CLL) has these characteristics:

- ☐ If the CLL is empty, then its `head` and `tail` are both null pointers;
- ☐ If the CLL has one node, then its `head` and `tail` both point at that single node's address;
- ☐ If the CLL has more than one node, then the pointers linking the nodes from `head` to `tail` operate like a (normal) linked list, and, in addition, the `tail`'s node's pointer points at the `head`'s node, creating a circular linked structure, sort of like a clock. Because the game we will play on this structure can be played both clockwise and anticlockwise, we recommend that your `Node` class have not only a `next` Node pointer but also a `back` Node pointer.

Think of each node in your CLL as a cup that can contain stones (of the kind used in a board game like Go or Mancala). At the start of the game, each cup should contain one stone. For example, if you printed the initial contents of the cups for a CLL that has eight nodes, you would get

```
1 1 1 1 1 1 1 1
```

Note that we are writing the output horizontally, with the head's node on the left and the tail's node on the right. The horizontal nature of this output hides the fact that, in fact, the structure is circular; thus, the node on the far right is linked to the node on the far left. We can think of each node as having an index from 0 to 7, left to right, as you'd expect in an array, although we will not use an array to store them.

There are only three *moves* in the game: you may `place` a stone in any cup; you may pick up all of the stones in a cup and `play` them clockwise; or you may pick up all of the stones in a cup and `play` them anticlockwise. To play is to drop one stone from your hand into each consecutive cup in the desired direction until you run out of stones.

And there is one *capture* rule: if the last stone dropped in a play lands in an empty cup, then that stone is discarded from the game.

For example, if we place a stone in the cup with index 3, then the CLL would look like this:

```
1 1 1 2 1 1 1 1
```

If we then played clockwise from that same cup, we would pick up the 2 stones and distribute them to the two cups to its right, resulting in the output:

```
1 1 1 0 2 2 1 1
```

Then if we place a stone in cup 5, we have:

```
1 1 1 0 2 3 1 1
```

Now let's play anticlockwise from cup 6:

```
1 1 1 0 2 4 0 1
```

Let's play clockwise from cup 4:

```
1 1 1 0 0 5 0 1
```

Note that, for the first time, our final stone placement landed in an empty cup (cup 6), so we discard that stone.

Finally, let's play clockwise from cup 5:

```
2 2 2 0 0 0 1 2
```

Here you see that we wrapped around from right to left, thanks to the circular nature of the linked list. A similar wrapping effect should occur if we play anticlockwise past the left end of the CLL.

Example input

The input will be in this form:

```
nodes 8
moves 6
place 3
clock 3
place 5
anti 6
clock 4
clock 5
```

Use the number by `nodes` for the number of nodes in your CLL. The second line shows how many moves will be played in the game. The integer by the string `place` gives the index of the cup into which we place one stone. The strings `clock` and `anti` are short for clockwise and anticlockwise, and the integers next to these strings give the index of the cup from which you will pull all stones in order to distribute them in the given direction. This example input above corresponds to the worked-out example in the previous section.

All input tests will have at least one node and at least one move.

Example output

The output should be a sequence of integers like

```
2 2 2 0 0 0 1 2
```

that shows the final distributions of stones in the cups. Each integer should be separated by a single space from its neighbor(s), and there should not be a final space at the end of the output.

Other notes

You should not use any arrays in this project. If you do, then your grade will be at most 50% even if you pass all of the tests. You may use the `iostream`, `string`, and `fstream` libraries only. Your program must use file i/o to read the input files.

Getting started

Please start by accepting the assignment at:

https://classroom.github.com/a/Mrx69HeO

which gives you a GitHub repository for your project.

How to submit:

- ☐ Add/commit/push your final code to your GitHub repository; this is the code that will be graded.
- ☐ On Gradescope, click on the assignment for this project, and submit your GitHub repository. (Select *master* as your branch.)

Grading

Your grade will come from:

75%	correctness + passing the tests
25%	style

There will be 13 tests:

- ☐ 2 tests will use `place` only
- ☐ 3 tests will use `place` and `clock`
- ☐ 2 tests will use `place` and `anti`
- ☐ 2 tests will use all commands, but have no captures
- ☐ 3 tests will use all commands and will have captures
- ☐ one test will be re-run using `valgrind` to check for memory leaks

Projects passing zero tests will receive at most 70% of the project grade (assuming perfect style). Passing tests increases the grade proportionally up from 70%; however, this assumes that your code isn't written to circumvent testing (for example, by "hard coding" answers to the tests), which is a violation of the honor code.

The elements of style that we will assess on this project are:

- ☐ Your helper functions should be no more than about 20 lines long (not including spacing for curly braces, and not counting comments); each function should have one purpose, and be named according to that purpose.
- ☐ Use appropriate programming constructs; e.g., don't use a loop when an if statement is sufficient, a while loop when a for loop is better, and so on.
- ☐ Prune your dead code: remove commented out code, unused variables, other code not contributing to the overall project.

We will give you one aggregate grade for the rest of these style elements:

- ☐ Include all file and function header comments, using the required format.
- ☐ Declare variables near their initial use.
- ☐ Indent properly and use correct placement of curly braces and parentheses.
- ☐ Use correct spacing around operators.
- ☐ Use reasonable variable names in proper naming style: using `lower_case` naming in local scope, and `UPPER_CASE` naming for global constants.