CPSC 250L Lab 13
Stacks and Queues

Fall 2017

1 Introduction

In this lab, we will introduce two more data structures namely, Last in First out (LIFO) stacks and First in First out (FIFO) queues.

2 Exercises

Fork and clone the `cpsc250l-lab13` repository from the CPSC 250 student group for this semester.

The only imports that you are allowed to use for this lab are the following.

- `import java.io.File`
- `import java.io.PrintWriter`
- `import java.io.IOException`

You are not allowed to use `ArrayList` or any other Java container in any classes you create.

You are expected to commit after each method in each exercise. Specifically, you need to commit at least 5 times in order to get full credit.

2.1 Stacks

Before attempting this exercise read Section 4.1.

Exercise 1

Create a class called `StackMethods` and implement the following methods.

1. `public static Stack<Integer> getEvenNumbers(Stack<Integer> stack)`

   This method returns a new `Stack` that contains copies of the even numbers in the `stack` parameter. After execution, the `stack` parameter should be unchanged.

2. `public static boolean checkParentheses(String input)`

   This method checks a string for mismatched parentheses. To do so, you iterate through each character in the string. If you encounter any opening parenthesis (that is `'('`, `'['`, and `'{'`), you push that character to your stack. When you encounter a closing parenthesis (`')'`, `']'`, and `'}'`), you pop a character from stack. If the current character in the string does not match the one that you popped from the stack, then you have encountered a mismatch. It returns `true` if you **do not** encounter a mismatch and `false` if you do encounter a mismatch.

   Hint: Use `String.charAt(i)` to get the `i`th character in the string. Alternatively, you could use `String.toCharArray()` to extract the characters as an array.

3. `public static void reverse(Node<Integer> reverseMe, File out)`

   This method outputs the reverse of the linked list to the specified output file. To do so, traverse the linked list while storing each value of the list in a stack. Afterwards, pop values from the stack and output them to the file.

Exercise 1 Complete

Run:

```
git add .
git commit -m "Completed exercise 1"
git push origin master
```

2.2 Queues

Before attempting this exercise read Section 4.2.

Exercise 2

Create a class called `QueueMethods` and implement the following methods.

1. `public static Queue<Candy> getCandyOrder(Queue<Candy> dispenser, int day)`

   Pippi Longstocking has a PEZ dispenser, and everyday she eats all the candy in the dispenser. However, each day she eats the candy in a different way.

   - On Mondays she eats them one by one in the order they come out from the dispenser.
   - On Tuesdays she eats one candy and adds the next candy back to the bottom of the dispenser, and continues to do this until all candies are gone.
   - On Wednesdays, she eats one candy and adds the next two back to the bottom of the dispenser (in the same order they were taken off), and continues to do this until all the candies are gone.
   - On Thursdays she eats one candy and puts the next 3 candies back at the bottom of the dispenser, and so on, until Sunday.

- On Sundays, she eats one candy and puts back the next 6 candies.

She puts candy from the top of the dispenser onto the bottom of the dispenser one and a time

This method receives a queue candy dispenser (implemented in the class `Queue`) and a day of the week (where Monday is 0, Tuesday 1, and so on until 6 for Sunday), and returns a new queue with the candy Pippi ate, in the order she ate it, until the time the dispenser is empty.

2. `public static void getRemainingCandy(Queue<Candy> dispenser, Candy piece)`

   Sometimes Pippi shares her candy with her friends. Each of her friends likes only one flavor of candy, and Pippi allows her friends to eat all the candy of the particular flavor they like in her dispenser.

   This method receives a queue candy dispenser (implemented in the class `Queue`) and a flavor of candy and alters the dispenser so that all candies of the particular flavor are removed. All other pieces should be in the same order.

   Example dispenser is Cherry → Orange → Grape → Peppermint → Cherry → Cherry and Pippi's friend likes Cherry, the queue is altered to be Orange → Grape → Peppermint.

Exercise 2 Complete

Run:

```
git add .
git commit -m "Completed exercise 2"
git push origin master
```

3 Common Mistakes

The solutions to some common mistakes are as follows.

1. Pay close attention to methods that require the stack to be unchanged at the end of the method.
2. Before popping from the stack or dequeueing from the queue, be sure that it is not empty.
3. In `checkParentheses`, `'('` and `')'` match, `'['` and `']'` match, and `'{'` and `'}'` match. An example of parentheses that do not match are `'('` and `'}'`.

4 Tutorial

4.1 Stacks

Consider the scenario where you turn in a test to your lecture professor. When you finish the test, you place your test on top of a pile of other tests. When another student finishes, he or she places a test on top of yours. After the test is finished, your professor takes the top test off of the pile and grades it. Your professor will then take the next test off of the pile and grade that. The pile of tests is called a *last in, first out stack*. When you place your test on the pile, you are *pushing* your test to the stack. When your professor grades your test, your professor *pops* your test off of the stack.

In a stack, the only element that you can see is the top element of the stack. Furthermore, the only way to see the next element of the stack is to pop off the top element. This can be problematic when doing simple operations such as finding the size of the stack. To do so, consider the following code.

Code:

```
public static int getSize(Stack<Integer> s) {
    Stack<Integer> tmp  = new Stack<Integer>();
    int            size = 0;
    while (!s.isEmpty()) {
        tmp.push(s.pop());
        ++size;
    }
    while (!tmp.isEmpty())
        s.push(tmp.pop());
    return size;
}
```

In order to determine the size of the stack, we had to pop all elements from the stack and store them in a temporary stack. By doing so, we modified the stack. Therefore, since getting the size of an object should not modify it, we had to rebuild the stack. This means we effectively had to iterate through the stack twice in order to determine the size. This is a common theme in problems that involve looking at each element in a stack without modifying the stack.

4.2 Queues

Suppose you are in a line at a bank. The teller handles the request of the person at the front of the line and then the next person is at the front of the line. If someone arrives to the line, they start out in the back of the line. This is an example of a *first in, first out queue* (otherwise known as a FIFO queue). In a FIFO queue, you *enqueue* things at the end of the queue and you *dequeue* things at the front of the queue. Like with stacks, you can only see the front element and to get the next element you need to dequeue the first. Thus, we encounter the same problem we have with stacks when we try to compute the length of a queue.

Code:

```
public static int getSize(Queue<Integer> q)
    Queue<Integer> tmp  = new Queue<Integer>();
    int            size = 0;
    while (!q.isEmpty) {
        tmp.enqueue(q.dequeue());
        ++size;
    }
    while (!tmp.isEmpty)
        q.enqueue(tmp.dequeue());
    return size;
}
```

Namely, when we try to determine the size of the queue, we have to empty it and then refill it.