Programming Laboratory 8
CSCI 1913: Introduction to Algorithms,
Data Structures, and Program Development
March 21–22, 2017

0. Introduction.

In this assignment, you will implement a stack as a Java class, using a linked list of nodes. Unlike the stack discussed in the lectures, however, your stack will be designed to efficiently handle repeated pushes of the same element. This shows that there are often many different ways to design the same data structure, and that a data structure should be designed for an anticipated pattern of use.

1. Theory.

The most obvious way to represent a sequence of objects is simply to list them, one after the other, like this.

a a b b b c a a d d d d

Note that the same objects often appear many times in a row. This is called a *run* of those objects. In the example sequence, there is a run of 2 *a*'s, a run of 3 *b*'s, a run of 1 *c*, a run of 2 *a*'s, and a run of 4 *d*'s. You can represent a sequence with runs by listing its objects, along with the number of times each object appears. For example, you can represent the sequence shown above like this.

a b c a d
2 3 1 2 4

Representing a sequence in this way is called *run-length encoding*. If a sequence has long runs, or many runs, then run-length encoding will represent it more efficiently than simply listing its objects. However, if a sequence has short runs, or few runs, then run-length encoding may represent it *less* efficiently, because extra space is needed to store the lengths of the runs.

Since a stack is just a simple kind of sequence, you can use run-length encoding to implement it. In this assignment, you will write a Java class called `RunnyStack` that implements a stack which uses run-length encoding. Here are some examples of how it works. Suppose you push an object *a* on an empty `RunnyStack`. Then the stack will look like this, with a run of 1 *a*.

a 1

Now suppose you push *b*. The stack now looks like this, with a run of 1 *b*, and a run of 1 *a*.

b 1
a 1

If you push another *b* on the `RunnyStack`, then the length of the run on top of the stack is incremented, so the stack looks like this.

b 2
a 1

If you push yet another *b*, then the length of the run at the top of the stack would increase to 3. However, suppose that you pop the `RunnyStack` instead. Then the length of the run at the top is decremented, so that the stack looks like this.

b 1
a 1

If you pop the `RunnyStack` one more time, then the length of the run on top of the stack is decremented to 0. However, a run of 0 objects is like no run at all, so it vanishes, and the stack looks as it did after the first push.

a 1

Stacks with run-length encoding are used internally by some compilers and interpreters, because they often push the same objects over and over again.

2. Implementation.

You must write a class called `RunnyStack` that represents a stack of objects. Your class must implement run-length encoding, as described previously. It must also hold objects of type `Base`, so it will look like this.

```
class RunnyStack<Base>
{
  ⋮
}
```

Your class must define at least the following methods, as described below. To simplify grading, your methods must have the same names as the ones shown here.

- `public RunnyStack()`

  Constructor. Make a new, empty instance of `RunnyStack`.

- `public int depth()`

  Return the depth of the stack: the sum of the lengths of all the runs it holds. This is not necessarily the same as the number of runs it holds, which is returned by the method `runs`.

- `public boolean isEmpty()`

  Test if the stack holds no objects.

- `public Base peek()`

  If the stack is empty, then throw an `IllegalStateException`. Otherwise, return the object at the top of the stack.

- `public void pop()`

  If the stack is empty, then throw an `IllegalStateException`. Otherwise, decrement the length of the run on top of the stack. If this leaves a run of zero objects on top of the stack, then remove that run.

- `public void push(Base object)`

  If the stack is empty, then add a new run of one `object` at the top of the stack. If the stack is not empty, then test if `object` is equal to the object in the run at the top of the stack. If it is, then increment the length of that run. If it isn't, then add a new run of one `object` at the top of the stack. Note that `object` may be `null`.

- `public int runs()`

  Return the number of runs in the stack. This is not necessarily the same as its depth, which is returned by the method `depth`.

Here are some hints, requirements, and warnings. First, all these methods must work using $O(1)$ operations, so they are not allowed to use loops or recursions. That means you may need to introduce extra private variables. *You will receive no points for this assignment if you use loops or recursions in any way!*

Second, your `RunnyStack` class must have a private nested class called `Run`. You must use instances of `Run` to implement your stack. Each instance of `Run` represents a run of objects. *You will receive no points for this assignment if you use arrays in any way!*

The class `Run` must have three private slots that have the following names and types. The slot `object` points to the `Base` object that appears in the run. The slot `length` is an `int` that is the length of the run. The slot `next` points to the instance of `Run` that is immediately below this one on the stack, or to `null`. It must also have a private constructor that initializes these slots.

Third, your `push` method must test non-`null` objects for equality using their `equals` methods. It must use the Java '`==`' operator only for testing `null` objects. It is helpful to define an extra private method called `isEqual` that takes two objects as arguments, and tests if they are equal. If either object is `null`, then `isEqual` uses '`==`'. If neither object is `null`, then `isEqual` uses `equals`.

Fourth, `RunnyStack`'s methods are not allowed to print things. If you were writing `RunnyStack` in the Real World, then it might be part of some larger program. You don't know if that larger program should print things.

3. Deliverables.

The file **`tests.java`** contains Java code that performs a series of tests. Each test calls a method from your class `RunnyStack`. If it prints what the method returns, then it is also followed by a comment that shows what must be printed if the method works correctly.

Run the tests, then turn in the Java source code for the class `RunnyStack`. Your lab TA will tell you where and how to turn it in. If your lab is on **Tuesday, March 21**, then your work must be turned in by **11:55 PM** on **Tuesday, March 28**. If your lab is on **Wednesday, March 22**, then your work must be turned in by **11:55 PM** on **Wednesday, March 29**. To avoid late penalties, please be careful not to confuse these two dates.