Project 2: Stacks and Queues

Handed out: **September 9, 2012**
Due: **September 23, 2012 at 11:59pm**

Description

The project is divided into two parts. Part 1 will cover stacks and part 2 will cover queues. In both parts, you will need to implement a data structure that extends upon the basic functionality of stacks and queues. You will then be asked to use your data structure to solve a particular problem. In addition, you will need to provide a brief description of your implementation and justify the decisions made.

Part 1 - DoubleStack

Overview

Create a data type **`DoubleStack`** that implements a two-color double-stack (named red and blue, respectively) and has as its operations color-coded versions of the regular **`Stack`** API. For example, this data type should allow for both a red push operation and a blue push operation. Give an efficient implementation of this API using a single array whose capacity is set at some value `N` that is assumed to always be larger than the size of the red and blue stacks combined.

A) Implementation

Implement the methods of the generic class `DoubleStack` as defined below.

```
public class DoubleStack<T> {
   public enum Color { RED, BLUE }
   public DoubleStack(int N);          // Construct empty DoubleStack of capacity N
   public int size(Color c);           // Return the number of items in a stack
   public void push(T item, Color c); // Add item to a stack.
   public Item pop(Color c);           // Pop most recent item from a stack
   public Item peek(Color c);          // Look at object on top of stack w/o removing it
   public boolean isEmpty(Color c);    // Test if a stack is empty
   public boolean isFull();            // Test if DoubleStack is full
}
```

Throw a **`EmptyStackException`** if the client attempts to remove an item from an empty Stack and a **`RuntimeException`** exception if the client attempts to push an item on either stack when the array is already full.

Testing

We will be checking the correctness of your implementation using test files in a format described below. For these tests, we will use **`String`** types as stack items. The first line of the input file indicates the capacity of the DoubleStack to be used. This line is then followed by a combination of keywords and strings to be added to the DoubleStack. There are 4 different keywords: “--red” (resp. “--blue”) selects the red (resp. blue) stack for subsequent operations; “-” (resp. “?”) indicates that a string should be popped (resp. peeked) from the currently selected stack (red or blue). Any other string in the input file is simply pushed onto the currently selected stack. By default, the DoubleStack must be initialized to the color “red”.

Write a `main` function in the implementation of DoubleStack that reads input files of strings in this format and outputs the correct results. The elements popped from DoubleStack should be separated by a space character in output. At the end of the file, your routine should indicate on a new line how many elements remain in both red and blue stack.

For example, following input file

```
40
Computer Science
--blue
Purdue University
--red West Lafayette
?
- - - ?
```

processed through

```
% java -cp .:stdlib.jar DoubleStack < input1.txt
```

should produce following output if your implementation is correct:

```
Lafayette Lafayette West Science Computer
(1 left on RED stack, 2 left on BLUE stack)
```

Additional test files and corresponding solution files are available: input2.txt ➟ output2.txt and input3.txt ➟ output3.txt.

B) Infix to Prefix Conversion and Prefix Evaluation

Description

This task is split in two parts: in the first part, you will write a **`DoubleStack`** client that converts an expression written in Infix notation to its equivalent Prefix notation. In the second part you will use the **`DoubleStack`** data structure to evaluate the resulting Prefix expressions.

Background

Infix, Postfix and Prefix notations are three different but equivalent ways of writing arithmetic expressions. The Infix notation is the standard way to write an expression, with the operator inserted between its operands. In contrast, the Prefix notation (also known as "Polish notation"), writes operators before their operands. For example in prefix expression:

```
/ * A + B C D
```

the division acts on the result of the multiplication, and so the multiplication has to happen before the division (and similarly the addition has to happen before the multiplication). This expression can also be written as follows.

```
/ (* A (+ B C)) D
```

Here the Infix equivalent is

```
(A * (B + C)) / D
```

B1) Infix to Prefix

Write a program Infix2PreFix.java that reads infix arithmetic expressions from standard input, converts the expression to prefix notation, and outputs the result to standard output. Your implementation should use a **`DoubleStack`** to perform the conversion. It should also assume that each line in input corresponds to a different expression.

```
% cat input.txt
( A + B ) - C
( X + Y ) * ( W + Z )
% java -classpath .:stdlib.jar Infix2PreFix < input.txt
- + A B C
* + X Y + W Z
```

Your program needs to handle the following operators: *** / + -** (with ***** and **/** having higher precedence than **+** and **-**) and Parentheses **( )**. You shall assume that operands and operators in input are always separated by space characters. You must follow the same convention in output.

Test files and corresponding outputs are available online. infix1.txt ➟ prefix1.txt and infix2.txt ➟ prefix2.txt.

B2) Prefix Evaluation

For the second part, you should now **write a program EvalPrefix.java that evaluates prefix expressions**. The same operators and precedence rules are considered here as well. Your program should read prefix expressions from standard input and outputs the corresponding value to standard output. Note that while the previous task did not require the operands of infix expressions to be numbers, we are here only concerned with prefix expressions with integer operands.

For instance

```
+ * 1 2 3
```

should yield

```
5
```

A test file is available online with corresponding solution. prefix2.txt ➟ eval2.txt

Part 2 - Queue

For the second part of this project, you must Implement the standard **`queue`** data structure that we studied in the class with all supported operations such as *enqueue, dequeue, size, isEmpty.* Along with this you also need to implement a **O(1)** random access peek method which will tell what object is in *i-th* position in the lineup of queued elements and throw a **`RuntimeException`** otherwise.. Additionally, your iterator implementation should support each iteration operation (including construction) in constant worst-case time, and use a constant amount of extra space per iterator.

Specifically, you must **implement MyQueue.java to support following API**:

```
public class MyQueue implements Iterable<T> {
   public MyQueue();                  // construct an empty Queue
   public int size();                 // return number of items in Queue
```

```
    public boolean isEmpty ();          // test if queue is empty
    public boolean isFull ();           // test if queue is full
    public void enqueue (T item);       // Add item to the queue
    public T dequeue ();                // Remove element from the queue
    public T peek ();                   // Return element at front of the queue w/o removing it
    public T lookup (int i);            // Look up i-th element in queue if available,
                                        // throw exception otherwise
    public Iterator<T> iterator();      // return an iterator that examines the
                                        // items in order from front to back
}
```

Note that the constructor does not prescribe a specific capacity to the queue so your implementation must support dynamic resizing. **Your implementation must be time and memory efficient!**

Testing

To check the correctness of your implementation we will be using test files in a format described below. For these tests, we will use **`Integer`** types as queue items. The files contain values to be added to the queue (separated by space characters). In addition, the symbol '-' indicates that a dequeue must be performed, the symbol '*' indicates that a peek must be performed, and '?' indicates that the next number in input should be interpreted as the location where a lookup query must be performed. The result of each of these queries must be printed to standard output, separated by a space character.

Write a `main` function in the implementation of MyQueue that reads input files of strings in this format and outputs the correct results. The result of each query must be printed to standard output, separated by space characters. At the end of the file, your routine should indicate on a new line how many elements remain in the queue.

For instance, following input

```
3 4 5 * - * ? 2
```

should yield following result

```
3 3 4 5
2
```

Additional test files qinput2.txt and qinput3.txt with corresponding solutions qoutput2.txt and qoutput3.txt are available.

Report

Submit a report (PDF file) answering the following questions:

1. Explain briefly how you implemented the DoubleStack. Explain how your implementation guarantees an efficient usage of the allocated memory.
2. Describe the algorithm you implemented for the conversion from infix to prefix and justify its correctness.
3. Explain the algorithm you implemented for the evaluation of prefix expressions.
4. Explain how you implemented MyQueue and justify the constant time complexity of your "lookup" function.

Deliverables

You should submit 4 java files: **DoubleStack.java, Infix2Prefix.java, EvalPrefix.java, MyQueue.java**, and your PDF file.

Submission

Submit your solution before **September 23, 2012 11:59pm**. turnin will be used to submit this assignment. The submission procedure is the same as for the first project.

Inside your working directory for this project on *data* (e.g., ~/cs251/project2), create a folder in which you will include all source code used and libraries needed to compile and run your code. Please DO NOT use absolute paths in your files since they will become invalid once submitted. Optionally, you can include a README file to let us know about any known issues with your code (like errors, special conditions, etc).

After logging into data.cs.purdue.edu, please follow these steps to submit your assignment:

- Enter the working directory for this project

```
% cd ~/cs251/project2
```

- Make a directory named <your_first_name>_<your_last_name> and copy all the files needed to compile and run your code there.
- While still in the working directory of your project (e.g., ~/cs251/project2) execute the following turnin command

```
% turnin -c cs251 -p project2 <your_first_name>_<your_last_name>
```

Keep in mind that old submissions are overwritten with new ones whenever you execute this command. You can verify the contents of your submission by executing the following command:

```
% turnin -v -c cs251 -p project2
```

Do not forget the -v flag here, as otherwise your submission would be replaced with an empty one.