Data Structures and Algorithms I: Stacks Worksheet

Answer questions A2,3, all of part B, and questions C2, 3.

Part A: Theory

The first part of this worksheet focuses on the use of dry runs to understand how Stacks work. For each question, you will be given a sequence of Stack operations. You should show the impact of this sequence of operations on both of the Stack implementation strategies (arrays and links) discussed in the course. In both cases, assume that the Stack is initially empty.

1) Show how the state of both an array-based and a link-based stack changes after each of the following operations:
   push(15), push(20), push(5), pop(), push(17), push(12), pop(), push(7), push(13), push(2)
   After the last operation, work out the sum of the values stored in the stack and the sum of the values output from the stack.

2) Show how the state of both an array-based and a link-based stack changes after each of the following operations:
   push('e'), push('s'), push('c'), pop(), push('u'), push('a'), pop(), push('o'), push('t'), pop(), push('h')
   After the last operation, work out the word held in the stack and the word that was output via the pop operations.

3) Show how the state of both an array-based and a link-based stack changes after each of the following operations:
   push('Ireland'), pop(), push('England'), pop(), push('Wales'), pop(), push('Scotland'), pop(), push('France'), push('Germany')
   After the last operation, list the countries that were popped from the stack and the countries held in the stack.

4) Show how the state of both an array-based and a link-based stack changes after each of the following students are pushed onto a stack:

01020304 Rem Collier	*01020305 David Leonard*
01020306 Joe Carthy	*01020307 John Dunnion*
01020308 Michela Bertolotto	*01020309 Fintan Costello*
01020310 Henry McLoughlin	*01020311 Damian Dalton*

   Now, pop the last three students off the stack.
   After the last operation, list the names of the students that were popped from the stack and the names of the students held in the stack.

   NOTE: For the purposes of showing how the state of the stack changes, please use the student number only (first column).

Part B: Stack Implementations

In this second part of the worksheet, you will be expected to write Java code that corresponds to the two Stack implementation strategies discussed in the course. Much of this work will require that you transform pseudo code given in the course into Java code.

1) **ArrayStack**: Create a new Java class called `ArrayStack`. As was discussed in the lecture, this class should have 3 fields, two constructors and 5 methods (one for each stack operation).

   In order to support visualisation of the stack state, you need to write an additional (sixth) method for the `ArrayStack` class, which has the following declaration:

```
public String toString() {
      return "an Array Stack instance";
}
```

   Basically, your implementation should create a string representation of the state of the stack that corresponds (somewhat) to the dry run format discussed in the course. Specifically, you should get the string to print out: i) the size of the stack; and ii) each value stored in the array that underpins the stack, so that the bottom of the stack appears first, and the top of the stack appears last.

   For example, a stack that has three values 'A', 'B', and 'C' pushed onto it would generate output of the form:

```
3     A B C
```

   Note that the size and the array contents are separated by a tab. As a hint, I will show how to create the above output in a way that is reasonably easy to adapt to work with the actual state of an array-based stack:

```
public String toString() {
      int size = 3;
      String output = "" + size + "\t";
      output += 'A' + " ";
      output += 'B' + " ";
      output += 'C' + " ";

      return output;
}
```

Your version of the above method will allow you to do the following:

```
public class StackTest {
      public static void main(String[] args) {
            ArrayStack s = new ArrayStack();
            System.out.println(s);
            s.push("A");
            System.out.println(s);
            s.push("B");
            System.out.println(s);
            s.push("C");
            System.out.println(s);
      }
}
```

This would produce the following output:

```
0
1     A
2     A B
3     A B C
```

Include the above main method as part of the ArrayStack class and run it to check that you get the expected output.

2) **LinkedStack**: Create a new Java class called `LinkedStack`. As was discussed in the lecture, this class should have 2 fields, one constructor and 5 methods (one for each stack operation).

   In addition, you need to create an inner class to model the idea of a `Node`. As this is the first of this style of implementation that you have covered, an outline of the code is given below (the push(o) operation is also implemented for you as an example):

```
public class LinkedStack {
      public class Node {
            Object element;
            Node next;

            public Node(Object element) {
                  this.element = element;
            }

            public String toString() {
                  return element.toString();
            }
      }

      int size;
      Node top;

      public LinkedStack() {
            top = null;
            size = 0;
      }
      public void push(Object o) {
        Node node = new Node(o);
        node.next = top;
        top = node;
        size++;
      }

      public Object pop() {
      }

      public Object top() {
      }

      public int size() {
      }

      public boolean isEmpty() {
      }

      // state visualisation
      public String toString() {
        String output = "";
        Node node = top;
        while (node != null) {
          output = node.element.toString() + " " + output;
          node = node.next;
        }

        return "" + size + "\t" + output;
      }
}
```

Please study this outline code and make sure that you understand what is going on before you complete the implementation.

Copy the test main method from question 1 and adapt it to work with the `LinkedStack` class. Make sure that you get the same output.

Part C: Evaluation

The final part of this worksheet requires that you test your implementations against the dry runs that you performed in part A. For each question, you should write 2 program start classes: one for each implementation strategy. As a naming convention you should use the following: PCQxA.java for the ArrayStack implementation strategy and PCQxL.java for the LinkedStack implementation strategy. x should be the question number.

NOTES:

- Because your Stack implementations work with objects, you cannot push primitive data types directly onto the stack. Instead, you have to convert them to their object equivalents. For example, if you had an integer value , `15`, then the object equivalent of this is `new Integer(15)`. Similarly, the char value `'a'` has an object equivalent `new Character('a')`.
- When you pop objects that correspond to primitive data types from a Stack, you need to convert them back to their primitive data types before you can use them in calculations. To do this, you use a XXXValue() method that is associated with the object equivalent. For example, to convert an Integer object referenced by the variable, `intObj`, into an integer value, you do the following: `intObj.intValue()`,. Similarly, to convert a Character object, `ch`, into a char, you do the following `ch.charValue()`.

1) Write two main methods that correspond to the series of push and pop operations outlined in question A1. You do not need to work out the totals in the program, only print out the state after each operation (also print out the state before the first operation).
2) Write two main methods that correspond to the series of push and pop operations outlined in question A2. You do not need to work out the words in the program, only print out the state after each operation (also print out the state before the first operation).
3) Write two main methods that correspond to the series of push and pop operations outlined in question A3. You do not need to print out the countries in the program, only print out the state after each operation (also print out the state before the first operation).
4) Write two main methods that correspond to the series of push and pop operations outlined in question A1. You do not need to print out the student names in the program, only print out the state after each operation (also print out the state before the first operation). Remember, the state is represented in terms of the student numbers.

   NOTE: While you only need to print out the state, you should still try to push all the student information into the stack. For this, you will need to create a `Student` class that models the student information (student number and name). This class will need a `toString()` method, which returns the student number and a constructor that takes 2 arguments: the student number and the student name.