Department of EECS
York University

Term: Winter 2019
Instructor: Andy Mirzaian

EECS2011: Fundamentals of Data Structures
Sections M and Z
Assignment 2
Due: 10 pm, Friday, February 15, 2019

- **Read the course FAQ on how to submit assignments electronically.**
- **Print your name, eecs account, and student ID # on top of every file you submit.**

Assignment Objectives: This assignment consists of three problems and is designed to let you exercise on the following concepts related to Lecture Slides 3, 4, 5, and 6.

❑ **Problem 1:**

- **Recursion:** As a beginner, we tend to mentally trace the steps of a process iteratively (step 1, then step 2, then step 3, …). But to become a competent algorist we also need to be adept at a higher level of abstract mental process, namely, thinking recursively. There is a close connection between that process and mathematical induction. Induction is also applicable to iterative loops (e.g., Loop Invariant). So, we need to be good at induction too.
- **Analysis** of recursive algorithms by mathematical induction.

❑ **Problem 2:**

- **From abstract to concrete:** How to implement an ADT by a suitable concrete data structure that supports operations defined in the ADT, and satisfies a given set of additional specifications, restrictions, and resource requirements.

❑ **Problem 3:**

- **Developing apps:** How to select and effectively use suitable data structures to solve an application problem. In this assignment we want to see how to make effective use of linear data structures such as stacks in parsing expressions and related problems.

Problem 1: [35%]

Imagine a row of n lights , numbered 1 to n, that can be turned on or off only under certain conditions as follows. The first light can be turned on or off anytime. Each of the other lights can be turned on or off only when the preceding light is on and all other lights before it are off. If all the lights are on initially, how can you turn them all off? For three lights numbered 1 to 3, you can take the following steps, where 1 indicates a light that is ON and 0 indicates OFF:

1 1 1	3 lights ON initially
0 1 1	Turn OFF light 1
0 1 0	Turn OFF light 3
1 1 0	Turn ON light 1
1 0 0	Turn OFF light 2
0 0 0	Turn OFF light 1

We can solve this problem by indirect recursion using the two methods turnOff() and turnOn() that mutually call each other. The algorithm in pseudo-code for turnOff(n) is shown below:

```
Algorithm turnOff( n )
// Pre-Condition:   Lights 1 .. n are all currently on.
// Post-Condition:  Lights 1 .. n are all turned off.
1.   if ( n = 1 ) then Turn OFF light 1
2.   else {   // n ≥ 2
3.        if ( n > 2 ) then   turnOff( n – 2 )
4.        Turn OFF light  n
5.        if ( n > 2 ) then   turnOn( n – 2 )
6.        trunOff( n – 1 )
7.   }
end   // trunOff
```

a) Write a similar algorithm in pseudo-code for turnOn(n).

b) Implement these algorithms in Java. Use the results in a program to display the sequence of steps to turn off n lights that are initially on. Show the output for a few values of n. The output format should be similar to the 3-lights example shown above.

c) Obtain, with sufficient explanation, an accurate estimate of the number of times lights are turned off or on during the entire process. Express the answer as a function of n.

d) Combine the two algorithms turnOff(n) and turnOn(n) and replace them with a single recursive two parameter algorithm flipSwitches(n, s) written in pseudo-code. The boolean parameter s indicates which version is intended: switching n lights ON or OFF.

*[**Note:** turnOff() and turnOn() methods should no longer be used/invoked in your new algorithm —use only FlipSwitches() instead.]*

Problem 2: [30%]

Give a Java implementation of the deque ADT (see [GTG §6.3] for definition) using two stacks as the only instance variables. What are the running times of the methods?

*[**Note:** For the two instance variables you may use the Stack<E> class from the Java Standard Library, but you are only allowed to use its "pure" stack methods (see LS6, p. 5) and NOT the extra non-stack members such as those inherited from elsewhere.]*

Problem 3: [35%]

An expression with binary operators can be expressed unambiguously as a fully parenthesized infix expression (FPIE) , or un-parenthesized postfix expression (UPPE).

A FPIE ***exp*** is either an operand, or of the form ***(exp_1 op exp_2)*** , where ***exp_1*** and ***exp_2*** are FPIEs and ***op*** is a binary operator. The same expression as UPPE is either the said operand or of the form ***exp'_1 exp'_2 op*** , where ***exp'_1*** and ***exp'_2*** are UPPE equivalents of ***exp_1*** and ***exp_2*** , respectively. Here is an illustrative example:

((a + 20) / ((b – c) * (53.4 – d))) as FPIE,

a 20 + b c – 53.4 d – * / as equivalent UPPE.

Another way the same expression can be expressed unambiguously is the FPIE but with all left parentheses removed, i.e., we use only right parentheses. Let's call this version as right parenthesized infix expression (RPIE). The above example expression is:

a + 20) / b – c) * 53.4 – d))) as equivalent RPIE.

Write a Java program that takes from the standard input a valid RPIE and outputs the equivalent FPIE and UPPE.

Note:

- *Conversion from a FPIE to its equivalent RPIE is trivial: simply remove all left parentheses. However, the reverse conversion is interesting and non-trivial.*
- *An input string may include blank spaces, (right) parentheses, operands, and the four arithmetic operators +, –, *, /. Any character other than blank spaces, operators, and (right) parentheses will be assumed as part of an operand. Operands may be any numeric constants or variable identifiers but are not syntactically checked. Not withstanding that, IllegalArgumentException may be thrown if the input does not represent a valid RPIE.*

What files to submit:

- **a2sol.pdf :** Describe solutions to the three problems with detailed explanations of your algorithmic design ideas, pseudo-codes if asked for, illustrations, and convincing analysis.

- Java source codes and test I/O results:
  - Problem 1: **Lights.java** and **LightsTestIO.txt.**
  - Problem 2: **NewDeque.java** and **NewDequeTestIO.txt.**
  - Problem 3: **Expression.java** and **ExpressionTestIO.txt.**

  You may test each class using their main methods. Record their test results in the corresponding …TestIO.txt file.

  Remark: We won't discuss/require JUnit testing in this course. But those of you who are familiar with that framework may feel free to use it as an additional aid in the design and debugging process.