CS526
Homework Assignment 2

This assignment has two parts. Part 1 is a practice of designing and implementing small ***recursive*** methods and Part 2 is an implementation of an application using the stack data structure.

Part 1 (40 points)

Create a file named *RecursionPractice.java* and implement the following two methods within the file.

(1). Write a recursive Java method named *recursiveProduct* that computes the product of two integers, *m* and *n*, using only additions and subtractions, as described below:

> *recursiveProduct* (int *m*, int *n*)
> Input: integers *m* and *n*
> returns *m* * *n*

Note that *m* and *n* could be a negative number, 0, or a positive number. If you invoke the method with *m* = 10 and *n* = -20, the expected output is:

```
Recursive product: 10 times -20 is -200
```

(2). Write a recursive Java method named *findFixedSumPairs* that implements the following requirements:

> Given an array of positive integers *A* and an integer *k*, the program prints every pair of integers in *A*, sum of which equals *k*.

Assume that all integers in *A* are distinct and they are sorted in the increasing order.

For example, suppose that you have the following main method in your program:

```
public static void main(String[] args) {

      int[] a = {1, 5, 8, 11, 14, 15, 20, 23, 25, 28, 30, 34};
      int k;
      k = 43;
      System.out.println("k = " + k);
      findFixedSumPairs(a, k);
}
```

Then, the output of your program should be:

```
Fixed sum:
k = 43
a = [1, 5, 8, 11, 12, 14, 15, 20, 21, 22, 23, 25, 28, 30, 34, 36]

a[6] = 15, a[12] = 28
a[7] = 20, a[10] = 23
a[8] = 21, a[9] = 22
```

Note that the *findFixedSumPairs* itself is not a recursive method. You must write a separate, recursive method with additional parameters, which is invoked within the *findFixedSumPairs* method. Name this method *recursiveFixedSumPairs*. You need to determine appropriate input parameters of this method. The Section 5.4 in page 214 of the textbook discusses this issue. You may want to study this section.

An incomplete *RecursionPractice.java* code is posted on Blackboard. A *main* method is also written to test the above recursive methods. You need to complete the remaining part of *RecursionPractice.java*.

Part 2 (60 points)

You are required to write a Java program named *InfixEvaluation.java* that evaluates and produces the value of a given arithmetic expression using a stack data structure. For example, if the given expression is (2 + (5 * 3)), your program must produce 17. As another example, consider the expression ((4 – 2) * ((9 – 3) / 2)). Given this expression, your program must produce 6.

We make the following assumptions about the input expression:

- All operands are positive integers.
- Only the following operators are allowed: +, –, *, and /
- Expression is fully parenthesized. In other words, each operator has a pair of "(" and ")" surrounding its two operands.

Your program must read a number of arithmetic expressions from an input file, named *infix_expressions.txt*, and produce an output. A sample input file is shown below:

```
( ( 2 + 5 ) * 3 )
( ( ( 3 + 2 ) * 3 ) – ( 6 / 2 ) )
( ( ( 7 – 5 ) * 2 ) / ( 10 – 8 ) )
```

Each line has one arithmetic expression. Operands, operators, and parentheses are separated by a space(s). So, you can use a space, or spaces, as a delimiter when tokenizing an expression. The expected output for the above input is:

```
The value of ( ( 2 + 5 ) * 3 ) is 21
The value of ( ( ( 3 + 2 ) * 3 ) – ( 6 / 2 ) ) is 12
The value of ( ( ( 7 – 5 ) * 2 ) / ( 10 – 8 ) ) is 2
```

Note that a given expression is repeated in the corresponding output. You must write the output on the screen.

A pseudocode is given below. You must implement this pseudocode.

Read an input expression from the input file, tokenize it, and store the tokens in a linear data structure, such as an array. Then, do the following for each expression.

Create two stacks – one for operands and the other for operators

Scan the tokens in the linear data structure from left to right and perform the following:

- If a token is an operand, push it to the operand stack.
- If a token is an operator, push it to the operator stack.
- If a token is a right parenthesis, pop two operands from the operand stack and pop an operator from the operator stack and apply it to the operands. The result is pushed back to the operand stack.
- If a token is a left parenthesis, ignore it.
- After all tokens are processed, what is left in the operand stack is the result.

Repeat the same process for all expressions in the input file.

We assume that all expressions in the input file are valid (i.e., you don't need to check for invalid expressions).

For the two stacks, you must use the ***LinkedStack.java*** class that is included in the textbook's source code collection. You may not use Java's stack class or a stack class defined by somebody else. Note that you must not modify the ***LinkedStack.java*** class. Name your program as *InfixEvaluation.java*.

Documentation

No separate documentation is needed. However, you must include sufficient inline comments within your program.

Deliverables

You need to submit the following files:

- *RecursionPractice.java*
- *InfixEvaluation.java*
- All other necessary files, including:
  - *LinkedStack.java*
  - *Stack.java*
  - *SinglyLinkedList.java*

- *Other files, if any*

Combine all files that are necessary to compile and run your program into a single archive file. Name the archive file *LastName_FirstName_hw*2.*EXT*, where *EXT* is an appropriate file extension, such as *zip* or *rar*. Then, upload it to Blackboard.

Grading

Part 1 – (1): Your facilitator will run your program with 3 different input pairs and 5 points will be deducted for each wrong output.

Part 1 – (2): Your facilitator will run your program with three different *k* values and 5 points will be deducted for each wrong output.

Part 2: Your program will be tested with 4 input expressions and 5 points will be deducted for each wrong output.

Points will be deducted up to 20 points if your program does not have sufficient inline comments.