Homework 8

1 Part I: Using Stacks

1. Show the contents of two initially empty stacks, stack1 and stack2, after **each** of the following statements execute. Assume a, b, c, and d are objects. If you are writing the contents of your stack horizontally, make sure you specify which side is the top and which side is the bottom.

   - stack1.push(a)
   - stack1.push(b)
   - stack2.push(c)
   - stack2.push(d)
   - stack1.push(stack2.pop())
   - stack1.push(stack2.peek())
   - stack2.push(stack1.peek())
   - stack1.pop()

   The bottom of the stack will be the most left, and I will abbreviate stack1 as 1 and stack2 as 2.

   - stack1.push(a)
     1. a
     2.
   - stack1.push(b)
     1. a, b
     2.
   - stack2.push(c)
     1. a, b
     2. c
   - stack2.push(d)
     1. a, b
     2. c, d
   - stack1.push(stack2.pop())
     1. a, b, d
     2. c
   - stack1.push(stack2.peek())
     1. a, b, d, c
     2. c
   - stack2.push(stack1.peek())
     1. a, b, d, c
     2. c, c
   - stack1.pop()
     1. a, b, d
     2. c, c

2. Show the contents of the stack as you trace the algorithm to check for balanced parenthesis (given in the notes and in Section 9) for each of the following expressions. State the result of the algorithm (balanced or unbalanced).

   - (a (b + c) - [d + e]
   - a (b [c * { d + e }] - f)
   - (a { b + c } / [d * e] / f) - g)
   - a (b [c - d] * e + (f / g) } + h

   Since we are only checking for balanced parenthesis, I will gobble everything that is not (,), [,], {, or }. The stack that I will use will have its bottom displayed as the left side.

   - (a (b + c) - [d + e]
     - (
       push('(')
       stack: (
     - (
       push('(')
       stack: (, (
     -)
       pop() == '(' is true, continue
       stack: (
     - [
       push('[')
       stack: (, [
     -]
       pop() == '[' is true, continue
       stack: (
     - no more input, and stack is not empty, so expression uses unbalanced parenthesis.
   - a (b [c * { d + e }] - f)
     - (
       push('(')
       stack: (
     - [
       push('[')
       stack: (, [
     - {
       push('{')
       stack: (, [, {
     - }
       pop() == '{' is true, continue
       stack: (, [
     -]
       pop() == '[' is true, continue
       stack: (
     -)
       pop() == '(' is true, continue
     - no more input, stack is empty, so expression uses balanced parenthesis.
   - (a { b + c } / [d * e] / f) - g)

  - (
    push('(')
    stack: (
  - {
    push('{')
    stack: (, {
  - }
    pop() == '{' is true, continue
    stack: (
  - [
    push('[')
    stack: (, [
  -]
    pop() == '[' is true, continue
    stack: (
  -)
    pop() == '(' is true, continue
    stack:
  -)
    stack is empty, so there is no left parenthesis to pair with this right parenthesis. The expression uses unbalanced parenthesis.

- a (b [c - d] * e + (f / g) } + h
  - (
    push('(')
    stack: (
  - [
    push('[')
    stack: (, [
  -]
    pop() == '[' is true, continue
    stack: (
  - (
    push('(')
    stack: (, (
  -)
    pop() == '(' is true, continue
    stack: (
  - }
    pop() == '{' is false, break
    The expression uses unbalanced parenthesis

3. Convert the following infix expressions into postfix notation

- a / b * (c - d)
- a * b / (c - (d + e))
- (a + b * c) / (d * e * f - g)

- a / b * (c - d)

  - a b / c d - *
- a * b / (c - (d + e))
  - a b * c d e + - /
- (a + b * c) / (d * e * f - g)
  - a b c * + d e * f * g - /

4. Evaluate the following postfix expressions, using a = 1, b = 2, c = 3, d = 4, and e = 5.

   - a b - c * d +
   - a b * c a + / d e * -
   - a c + b * d -

   - a b - c * d + = 1
   - a b * c a + / d e * - = -19.5
   - a c + b * d - = 4

5. Evaluate the following infix expression using the algorithm given in Section 22 (`evaluteInfix`). Show the contents of `operatorStack` and `valueStack` as you trace through the evaluation. Assume that a = 2, b = 3, c = 4, and d = 5.

   - (a + b) / (c - d) - d

   - (a + b) / (c - d) - d == (2 + 3) / (4 - 5) - 5
     - (
       operatorStack.push(( )
       valueStack:
       operatorStack: (
     - 2
       valueStack.push(2)
       valueStack: 2
       operatorStack: (
     - +
       operatorStack.push(+)
       valueStack: 2
       operatorStack: (, +
     - 3
       valueStack.push(3)
       valueStack: 2, 3
       operatorStack: (, +
     -)
       topOperator = operatorStack.pop() == +
       operandTwo = valueStack.pop() == 3
       operandOne = valueStack.pop() == 2
       result = operandOne + operandTwo == 5
       valueStack.push(5)
       topOperator = operatorStack.pop() == (
       valueStack: 5
       operatorStack:

- /
  operatorStack.push(/)
  valueStack: 5
  operatorStack: /
- (
  operatorStack.push(( )
  valueStack: 5
  operatorStack: /, (
- c
  valueStack.push(4)
  valueStack: 5, 4
  operatorStack: /, (
- -
  operatorStack.push(-)
  valueStack: 5, 4
  operatorStack: /, (, -
- d
  valueStack.push(5)
  valueStack: 5, 4, 5
  operatorStack: /, (, -
-)
  topOperator = operatorStack.pop() == -
  operandTwo = valueStack.pop() == 5
  operandOne = valueStack.pop() == 4
  result = operandOne - operandTwo == -1
  valueStack.push(-1)
  topOperator = operatorStack.pop() == (
  valueStack: 5, -1
  operatorStack: /
- -
  topOperator = operatorStack.pop() == /
  operandTwo = valueStack.pop() == -1
  operandOne = valueStack.pop() == 5
  result = operandOne / operandTwo == -5
  valueStack.push(-5)
  operatorStack.push(-)
  valueStack: -5
  operatorStack: -
- 5
  valueStack.push(5)
  valueStack: -5, 5
  operatorStack: -
- infix has no more characters left to process
  topOperator = operatorStack.pop() == -
  operandTwo = valueStack.pop() == 5
  operandOne = valueStack.pop() == -5
  result = operandOne - operandTwo == -10
  valueStack.push(-10)
- operatorStack.isEmpty() == true
  (return valueStack.peek()) == -10

6. Write a method from the client perspective to display the contents of a stack in the order in which they were added. For example, if you push a, b, and c onto a stack, and invoke the method, the method would print a, b, c. Your method header should be: `public void printInAddOrder(StackInterface stack)`. The stack sent in as a parameter must not be destroyed. (Or, if you destroy it, you must recreate it.)

   Refer to Homework8Driver.java

7. Write a method from the client perspective to test whether a String is a palindrome using stacks. Your method header should be: `public boolean isPalindrome(String s)`. Your method should not be recursive. Use only the methods of `StackInterface`, defined in Section 3. (You can create a new object of type `LinkedStack`, such as: `StackInterface stack = new LinkedStack();`).

   Refer to Homework8Driver.java

2 Part II

2. Imagine a linked implementation of the ADT stack that places the top entry of the stack at the end of a chain of linked nodes. Describe how you can define the stack operations `push`, `pop`, and `peek` so that they do not traverse the chain.

   - Describe what references you would need in your linked structure. You can include references to particular nodes in the chain or you can include references within each node. Just be clear about what you are including. Then, using those references, described how you would implement the three methods.

   With a topNode reference at the end of a linked list, I can freely use the push and peek operations. To push a new piece of data onto the stack, I would create a new Node where the previous topNode references the new Node, and the current topNode becomes the new Node. To peek at the data in the topNode, I would just return the data in the node referenced by the topNode. However, to pop the topNode, I would need a reference to the Node that would become the new topNode. Let's say that I have a reference to the new topNode for pop. That means I can pop the current topNode and set the new topNode to the appropriate reference. However, I would need to traverse the list to get the next new topNode. To fix this problem, I would require all Nodes to have a reference to the Node that references them. This implies that I would require a doubly linked list. Hence, there is no avoiding the use of having the topNode be the head of the linked list, since a node on both ends of a doubly linked list are both a head and a tail.

   To accommodate the use of a doubly linked list, I will have to make slight changes to what I said above about push and peek.

   Refer to DoublyLinkedStack.java and DoublyNode.java

3. Consider adding an iterator to the ADT stack. Should a stack iterator support the `remove` operations?

   An iterator might be useful for reviewing the contents of the stack, but a remove operation would undermine the sole purpose of the stack. The purpose of the stack is to require a specific ordering of when an item can be removed from the list, LIFO. If we allowed removable of data in random positions of the stack, LIFO would be violated.

EC. Write a full linked stack implementation as described in #2. Implement a constructor, the three methods you described, `isEmpty` and `clear`.

Refer to DoublyLinkedStack.java and DoublyNode.java