University of Puerto Rico – Mayagüez
Department of Computer Science and Engineering

CIIC 4020 / ICOM 4035 – Data Structures
Prof. Juan O. López Gerena
Spring 2019-2020
Laboratory #3 – Stacks and Queues

Exercise #1 – Palindromes (Stacks & Queues)

A palindrome is a string that reads the same forward as backwards. Some simple examples are: civic, eve, radar, level, rotor, refer, madam, noon, and kayak. If we ignore case, spaces and punctuation, we obtain a much richer pool of examples:

- Race car
- Step on no pets
- Madam, I'm Adam.
- Was it a car or a cat I saw?
- Eva, can I stab bats in a cave?
- Mr. Owl ate my metal worm
- Do geese see God?
- A nut for a jar of tuna
- Go hang a salami, I'm a lasagna hog

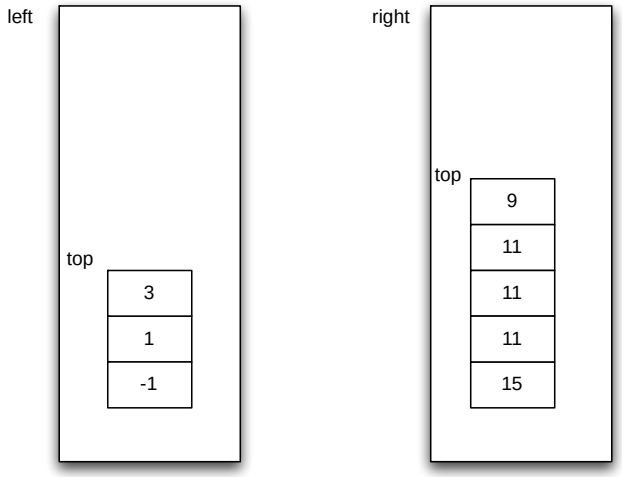
Write a program named `Palindromes.java` that prompts for a string and uses a stack and a queue to determine whether the string is a palindrome (ignoring case, spaces and punctuation). Implement a method named `isPalindrome` that receives a string as input and returns true if the string is a palindrome and false otherwise. Your program should print out (within the `main` method) either "Palindrome" or "Not a palindrome", depending on the user's input.

Note: You must use a stack and a queue, **DO NOT** solve the exercise by merely comparing positions within the string.

Think: How can you solve this problem using *only* stacks (two or more if needed)?

Exercise #2 – Stack Sort (Stacks)

In order to sort values we will use two stacks which will be called the left and right stacks. The values in the stacks will be sorted (in non-descending order) and the values in the left stack will all be less than or equal to the values in the right stack. The following example illustrates a possible state for our two stacks. Notice that the values in the left stack are sorted so that the smallest value is at the bottom of the stack. The values in the right stack are sorted so that the smallest value is at the top of the stack. If we read the values up the left stack and then down the right stack, we get -1, 1, 3, 9, 11, 11, 11, 15, which is in sorted order.



Suppose that we have a new value that we want to put into our sorted collection. We will want to put it on the top of one of the two stacks, but we may have to first move values around so that we maintain the sorted order.

Consider the following cases, using the example shown above as point of reference, to help you design your algorithm:

- If we were to insert the value 5, it could be added on top of either stack and the collection would remain sorted. What other values, besides 5, could be inserted in the example without having to move any values?
- If we were to insert the value 0, some values must be moved from the left stack to the right stack before we could actually insert 0. How many values must actually be moved?
- If we were to insert the value 11, first some values must be moved from the right stack to the left stack. How many values must actually be moved?
- What condition should we use to determine if enough values have been moved in either of the previous two cases?

Write a program named `StackSort.java` that continues to prompt the user for integers until the user finishes (pressing Enter without entering a number). The program should then print the values in sorted order.

Think: What is the worst-case running time of inserting a number? What is the worst-case running time of starting the program with both empty stacks and inserting *n* numbers?

Exercise #3 – Singly Linked Queue (Queues)

Implement the Queue ADT using a Singly Linked List-type structure **without dummy nodes**. In this case, the head node refers to the first data node and the tail node refers to the last data node. If the list is empty, then both the head and the tail must be *null*. Name your class `SinglyLinkedQueue.java`.

Note: Because there are no dummy nodes, you must be especially careful with the edge cases and test accordingly. For example, when enqueueing in an empty queue, when dequeueing the last element, etc.

Think: Is it more convenient to have the tail node represent the front of the queue or the rear of the queue? How are the worst-case running times of enqueue and dequeue affected by this decision?

Exercise #4 – Deque (Queues)

Using the provided Deque interface, implement the Deque ADT using a Circular Doubly Linked List-type structure with **only** a dummy header (***no*** *dummy trailer*). Name your class `CircularDoublyLinkedDeque.java`.

Exercise #5 – Fully Parenthesized Expressions (Stacks)

Write a program named `Fully_Paren.java` that uses stacks to evaluate *fully parenthesized* expressions with variables. You should use 3 stacks: one for the operators, one for the parenthesis (or brackets or braces), and one for the operands. For this exercise, every time you see the word "parenthesis", keep in mind that it may well be a bracket ("[") or a brace ("{").

There will only be two types of expressions:

- variable name = integer constant
- variable name : expression

Implementation details:

Every time you see an operand, push it into the operand stack.
Every time you see an operator, push it into the operator stack.
Every time you see an opening parenthesis, push into the parenthesis stack.

Every time you see a closing parenthesis, then it is time to evaluate a sub-expression! First, make sure the closing parenthesis matches the opening one. Then, to evaluate the sub-expression inside the parenthesis, you will need to pop the operand stack twice to obtain the two operands, and pop the operator stack once to know what operation will be performed with those two operands. If the parenthesis don't match, or if the operator stack is empty, or if the operand stack does not have at least two elements, then the expression is invalid and you should indicate so *immediately* by printing a message to the screen (see sample output below).

Your program should continually prompt for input and finish when no new input is entered. At the end, print the final values for the variables (use a tab character '\t' to separate the variable name from the variable value).

You may assume that:

- Variable names will consist of a capital letter, so that you'll have at most 26 variables.
- We will be using only integer values.
- There will be no spaces in the expressions.

Sample input:

```
L=5
W=10
A:(L*W)
X=100
L:[X-W]
L:(A-X))
L:((A-X)
J:([X-(A/W])*L)
J:([X-(A/W)]*L)
W:A+X
L:((L+A)*W)
```

Sample output:

```
L:(A-X)) is invalid
L:((A-X) is invalid
J:([X-(A/W])*L) is invalid
W:A+X is invalid

The final symbol table is:

variable        value
A               50
J               8550
L               1400
W               10
X               100
```

Hint 1: You may use an array of type Integer to store the variable values, where the variable A is at index 0, variable B is at index 1, etc. Unlike int, Integer allows you to use null to initialize the array entries, so that you may easily keep track of which variables actually exist.

Hint 2: Variables may be considered as characters or strings of length 1. However, when you evaluate a sub-expression, it will result in an integer value, which will have to be pushed into the operand stack. The problem is you can't push an Integer into a Character/String stack or vice-versa. Hence, you should declare your operand stack to be of type Integer, and when you encounter a variable, push its value (Integer) onto the stack instead of the variable name (Character/String).

Think: What type of changes would you need to implement so that an expression like A+X would be considered valid (i.e. so that the expression doesn't have to be *fully* parenthesized)?