Lab8: Using Stack & Queues
CSSSKL162: Programming Methodology Skills

Recursive & Iterative Algorithms Using Stacks, Queues, and Lists

Summary
This lab presents toy problems that model programming dilemmas that can be addressed using various Data Structures such as the Stack, Queue, and List. These solutions to the problems at hand can be coded iteratively or recursively, and we'll practice with both. We'll start by examining palindromes, which are strings that are identical both forward and in reverse ("bob" or "racecar"), and then consider the issues inherent in dating royalty.

Warming Up With Reversing Strings
Palindromes are interesting strings in that they are the same when displayed both left-to-right and right-to-left. For example, "Do geese see God" or "A man a plan a canal: panama!" are both palindromes, not considering punctuation or case sensitivity. We'll write multiple methods to determine whether a string is a palindrome, and we can make use of Stacks to accomplish this. To get started, download the driver code and first practicing printing a String in reverse using a Stack:

- Download the driver file UsingStacksSuitorsLab.java and read it
- Find the main method and comment out everything below the printReverse() function calls in main
- Find the printReverse() method and declare a Stack inside of it
    - Use your Stack from the previous labs (the Linked Stack is preferable here)
        - If your Stack is broken, you can consider using Java's Stack
            - Stack<Integer> foo = new Stack<Integer>();
    - Write a loop that iterates over the input string
        - In the loop, push each char from the input string on the stack
    - Write a second loop that iterates until the stack is empty
        - In the loop, pop off a char from the stack and print it.
            - This output should be the reverse of the input string
- Execute the code and observe your output; does it match the sample output below?
    - If not, see if your Stack is the culprit by commenting it out and trying Java's built-in Stack class (java.util).

Recursively Reversing Strings Using the Method Call Stack
In this section, we'll rewrite the iterative code above as a recursive function. In doing so, we'll remove the looping constructs (for/while is dropped in place of a recursive call), and we'll remove the explicit

stack variable (since we'll be using the already available Stack that holds Stack Frames). Lots of code seemingly goes away, and yet the output of our recursive function will be the same.

- In main, uncomment out all calls to recPrintReverse()
- Find the method recPrintReverse() and consider the following cases
    - If the string has many characters, your code should:
        - Print that character
        - Remove it from the string
        - Call this method again with the new substring //recursive case
    - If the string has only one character
        - Print that character and do nothing else //base case
- Write the base and recursive case in recPrintReverse and execute the driver code; does it match the previous output?

Palindromes & Stacks Iteratively

In this section, we'll determine that the String "bob" is a palindrome while "food" is not, both iteratively and recursively. Start by uncommenting the lines of code that call isPalindrome() in main, and…

- In main, uncomment out all calls to isPalindrome()
- Find the isPalindrome() function in the driver and look at how it's called from main
- Inside of isPalindrome, do the following:
    - Declare a Stack to use from the previous labs (the Linked Stack is preferable here)
        - If your Stack is broken, you can consider using Java's Stack
            - Stack<Integer> foo = new Stack<Integer>();
    - Write a loop that iterates over the input string
        - In the loop, push each char from the input string on the stack
    - Write a second loop that iterates until the stack is empty
        - In the loop, pop off a char from the stack and append it to a string.
            - This string is now the reverse of the original input
    - Now that you have two strings (the input and it's reverse), return true if they're equal or false otherwise
- Execute the code and observe your output; does it match the sample output below?

Palindromes & Stacks Recursively

In this section, we'll repeat what we did with printReverse() and create a recursive version of the iterative isPalindrome() function we just made. This function should call itself repeatedly to determine whether a given String is a palindrome, and will not use any explicit loop construct (while/for) nor will it

use an explicitly declared stack. By handing data to and from function calls (using input parameters or return values), you are storing variables for use by future function calls in an **Activation Record** or **Stack Frame** stored on the **Call Stack.** This will shorten the code significantly, while still producing correct output.

- In main, uncomment all calls to isPalindromeRec ()
- Find the method isPalindromeRec () and consider the following cases
    - If the string is of size 0 or 1, it's implicitly a palindrome //base case: success
        - For example, the substring "I" in the song title "I palindrome I" is a palindrome.
    - If the string is larger, just look at the first and last characters
        - If different, this isn't a palindrome //base case: fail
        - If the same, chop those letters off and call this function again //recursive case
- Write the two base cases at the top of recPrintReverse(), and add the third recursive case beneath them.
- Execute your code; does the output match the previous output?

Queues of Thread Objects

Thread pools are often employed by an Operating System to accelerate the handling of multi-threaded applications. Rather than "new" a Thread Object each time something becomes ***runnable,*** we can just build a bunch of threads (n == 4 here) ahead of time, and just dispatch them (or start() them here) when they are required. In this section, we'll build a small pool of threads and put them in a Queue. We'll then use that Queue to remove(dequeue) a Thread and call start() on it. We can print out the FIFO ordering of our new Thread pool once we've built the threads and enqueued them,

Introduction to The Royal Suitors

In an ancient land, the beautiful princess Eve (or handsome prince Val) had many suitors. Being royalty, it was decided that a special process must be used to determine which suitor would win the hand of the prince/princess. First, all of the suitors would be lined up one after the other and assigned numbers. The first suitor would be number 1, the second number 2, and so on up to the last suitor, number *n*. Starting at

the suitor in the first position, she/he would then count three suitors down the line (because of the three letters in his/her name) and that suitor would be eliminated and removed from the line. The prince/princess would then continue, counting three more suitors, and eliminate every third suitor. When the end of the line is reached, counting would continue from the beginning.

For example, if there were 6 suitors, the elimination process would proceed as follows:

123456	initial list of suitors, starting count from 1
12456	suitor 3 eliminated, continue counting from 4
1245	suitor 6 eliminated, continue counting from 1
125	suitor 4 eliminated, continue counting from 5
15	suitor 2 eliminated, continue counting from 5
1	suitor 5 eliminated, 1 is the lucky winner

A Solution Using Queues

Write a program/method called findPlaceToStand (int n) that uses a queue to determine the positions that you should stand in to marry the prince/princess if there are *n* suitors. When counting, traverse the queue. For the first and second suitors (numbers) on the queue, put them at the back of the queue. For the third suitor, discard the number. When there is only one number left, the suitor is determined.

- In the driver, find the commented out code near the bottom of main() to get started.
- The last two lines invoke the function findPlaceToStand()
    - This takes an integer and determines where to stand to win in the game of love.
        - Should you use a List, a Stack, or a Queue to solve this?
            - Does the selection of structure affect your results?

Use the Queue interface, but allocate each queue as a LinkedList or use your own Linked List implementation, if finished:

```
Queue<Integer> myQueue = new LinkedList<Integer>();
```

Both Queue and LinkedList are defined in java.util.

Using Stacks

Change your code such that when the end of the line is reached, the direction is reversed and the counting proceeds backwards toward the beginning of the line. Similar, upon reaching the first person in line, the

counting reverses again. The suitor at the beginning or end of the line will be counted twice when reversing, once for the forward count and once for the backward count.

Implement your change using the Stack class (not an interface). Create a stack as follows:

```
Stack<Integer> myStack = new Stack<Integer>();
```

Stack is also in java.util.

It is best to implement this version with two stacks (and no queues).

For Extra Challenge (otional)

Change your code such that the prince/princess randomly chooses whether the use the same order through the line as the previous traversal or go in reverse. To determine whether to go in reverse use:

```
Math.random() < 0.5
```

This one is a bit tricky (depending on how you solve it)…