ITI 1121. Introduction to Computer Science

Laboratory 9 Winter 2016

July 2, 2016

1 Introduction

The purpose of the lab is to reinforce the concepts of the stack and queue data structures. You must solve the exercises and submit them as if they were midterm questions.

2 Question 1

For this question, you must implement the class (static) method:

```
boolean isSkipped(Stack s1, Stack s2)
```

- The method isSkipped returns true if and only if the stacks designated by s1 and s2 contain the same elements, in the same order, but with elements 2,4,6,. . . missing in s2. Given s1 designating a stack containing the elements 1,2,3,4,5,6,7, where 7 is the top element, and s2 designating a stack containing the elements 1,3,5,7, where 7 is the top element, isSkipped(s1, s2) returns true.
- Both stacks, designated by s1 and s2, must remain unchanged following a call to the method isSkipped. Specifically, given s1 and s2, two reference variables designating stacks, following the call isSkipped(s1, s2), s1 contains the same elements, in the same order, as it did before the call to isSkipped. Similarly, s2 contains the same elements, in the same order, as it did before the call to isSkipped.
- You can assume that both, s1 and s2, will not be null.
- An empty stack is considered the skipped version of another empty stack.
- The parameters of the method isSkipped are of type Stack, which is an interface.

For this question, there is an interface named Stack:

```
public interface Stack {
  public abstract void push(int item);
  public abstract int pop();
  public abstract boolean isEmpty();
}
```

- Notice that the parameter of the method push and the return value of the method pop are of type int.
- Assume the existence of DynamicStack, which implements the interface Stack. It has one constructor and its signature is DynamicStack().
- You cannot use arrays to store temporary data.
- You must use objects of the class DynamicStack() to store temporary data.
- You do not know anything about the implementation of DynamicStack. In particular, you do not know if it uses an array or not.
- You can assume that DynamicStack can store an arbitrarily large number of elements.

3 Question 2

Consider the implementation of the class **CircularQueue** below. Given a queue designated by q and containing the following elements: A, B, C, D, E, F, G, where A is the front element of the queue, following the call, q.magic(4), what will be the content of the queue?

(a) E, F, G

(b) A, B, C, D

(c) E, F, G, A, B, C, D

(d) E, F, G, A, B, C, D, A, B, C, D

(e) None of the above

Answer:

```
public class CircularQueue<E> {
    private E[] elems;
    private int front;
    private intr ear;

    public CircularQueue(int capacity) {
        if (capacity < 0) {
            throw new IllegalArgumentException("negative number");
```

```
        }
        elem s = new E[capacity];
        front = -1;
        rear = -1;
    }

    public void magic(int n) {
        if (rear != -1 && rear != front) {
            while (n > 0) {
                E current = elems[front];
                elems[front] = null;
                front = (front + 1) % elems.length;
                rear = (rear + 1) % elems.length;
                elems[rear] = current;
                n--;
            }
        }
    }
}
```

4 Question 3

Implement the class method:

```
public static <E> void swap(Stack<E> xs, Stack<E> ys)
```

- The method exchanges the content of two stacks, xs and ys.
- The method must work for any valid implementation of the interface Stack;
- You can assume the existence of the classes DynamicArrayStack and LinkedStack.

```
Stack<String> a, b;
a = new LinkedStack<String>();
a.push("alpha");
a.push("beta");
a.push("gamma");
b = new DynamicArrayStack<String>();
b.push("blue");
b.push("green");
b.push("yellow");
b.push("black");
System.out.println(a);
System.out.println(b);
```

```
swap(a, b);
System.out.println(a);
System.out.println(b);
```

In particular, the above statements should print the following.

```
[gamma, beta, alpha]
[black, yellow, green, blue]
[black, yellow, green, blue]
[gamma, beta, alpha]
```

Write the code for this method with the following signature:

```
public static <E> void swap(Stack<E> xs, Stack<E> ys) {}
```

5 Question 4

Complete the implementation of the instance methods **size()** and **swap()** within the class **LinkedStack** below.

- The method size() returns the number of elements that are currently stored into this stack
- The method swap exchanges the first two elements (not the values); the first element becomes the second and the second element becomes the first. The method returns false if there are less than 2 elements in the list. You cannot use the methods push and pop, instead the links of the structure (references) must be transformed.

```
public class LinkedStack<T> implements Stack<T> {

    private class Elem<E> { // Implements the nodes of the list
        private E info;
        private Elem<E> next;

        private Elem(E info, Elem<E> next) {
            this.info = info;
            this.next = next;
        }
    }

    private Elem<T> top; // Instance variable, designates the top element

    public int size() {}

    public boolean swap() {}
}
```

6 Question 5

Show the result that will be displayed on the screen for each of the following two calls to the method dump in the code fragment below. The class CircularQueue can be found below.

```
CircularQueue<Integer> q;
q = new CircularQueue<Integer>(4);
int i = 0;
while (!q.isFull()){
    i++;
    q.enqueue(new Integer(i));
}

if (!q.isEmpty()){
    q.dequeue();
}

if (!q.isEmpty()){
    q.dequeue();
}

while(!q.isFull()){
    i++;
    q.enqueue(new Integer(i));
}
q.dump();

while(!q.isEmpty()){
    q.dequeue();
}
q.dump();
```

Class CircularQueue

```
public class CircularQueue<E> implements Queue<E> {
    public static final int DEFAULT_CAPACITY = 100;
    private final int MAX_QUEUE_SIZE;
    private E[] elems;
    private int front, rear;

    public CircularQueue() {
        this(DEFAULT_CAPACITY);
    }

    public CircularQueue(int capacity) {
        if (capacity < 0) {
```

```
            throw new IllegalArgumentException( Integer.toString( capacity )
        }
        MAX_QUEUE_SIZE = capacity;
        elems = new E[MAX_QUEUE_SIZE];
        front = 0;
        rear = -1; // Represents the empty queue
    }

    public boolean isEmpty() {
        return (rear == -1);
    }

    public boolean isFull() {
        return !isEmpty() && nextIndex(rear) == front;
    }

    private int nextIndex(int index) {
        return (index + 1) % MAX_QUEUE_SIZE;
    }

    public void dump() {
        System.out.println("MAX_QUEUE_SIZE = " + MAX_QUEUE_SIZE);
        System.out.println("front = " + front);
        System.out.println("rear = " + rear);
        for(int i = 0; i < elems.length; i++){
            System.out.print( "elems["+i+"] = " );
            if (elems[i] == null) {
                System.out.println( "null" );
            } else {
                System.out.println( elems[ i ] );
            }
        }
        System.out.println();
    }

    public void enqueue(E o) {
        if (o == null) {
            throw new IllegalArgumentException( "null" );
        }
        if (isFull()){
            throw new QueueOverflowException();
        }
        rear = nextIndex(rear);
        elems[rear] = o;
    }
```

```
    public E dequeue() {
        if (isEmpty()) {
            throw new EmptyQueueException();
        }
        E result = elems[front];
        elems[front] = null; //      scrubbing
        if (front == rear) { // Following this call to dequeue
            front = 0; // the queue will be empty
            rear = -1;
        } else {
            front = nextIndex(front);
        }
        return result;
    }
}
```