CSL 201 Data Structures
Lab 2 Due on August 28, 11.55pm

Instructions

- You are to use Java as the programming language. Use Eclipse as the IDE.
- You have to work individually for this lab.
- You are not allowed to share code with your classmates nor allowed to use code from the internet. You are encouraged engage in high level discussions with your classmates; however ensure to include their names in the report. If you refer to any source on the Internet, include the corresponding citation in the report. If we find that you have copied code from your classmate or from the Internet, you will get a straight F grade in the course.
- The submission must be a zip file with the following naming convention - rollnumber.zip. The Java files should be contained in a folder named after the question number.
- Include appropriate comments to document the code. Include a `read me` file containing the instructions on for executing the code. The code should run on institute linux machines.
- Upload your submission to moodle by the due date and time. Do not email the submission to the instructor or the TA.

This lab will improve your understanding of linked lists, stacks, recursion and queues. This is a lengthy lab, so get started early to complete it on time.

1 Extended Integer Arithmetic (5 points)

The integer type in Java supports integer values between -32768 to 32768. While one could use the long type to perform operations on numbers that can be represented by 4 bytes of storage, this is insufficient to represent numbers with 100 digits or more. Design and Implement a new data type called ExtendedInt that can represent an integer with over 100 digits. You will have to use a linked list to store the digits of the numbers and implement the following standard operations $+, -, *, /, <, >, ==, \leq, \geq$. The input to your program will be of the following format `ExentdedInt1 op ExtendedInt2`. Your program should output the result of the operation `op` performed on `ExtendedInt1` and `ExtendedInt2`.
Input
number1 operation number2
Output
result

2 Unrolling Recursion (15 points)

The objective of this problem is to simulate recursion using stacks and loops. A synthetic linear recursive procedure for this problem is provided in Code Fragment 1. A recursive function such as the one described is intuitive and easily understandable. On calling myRecursion(input), the execution first checks for the stopping condition. If the stopping condition is not met, then operations inside the recursive call are performed. This can include operations on local variables. The operations inside the function can result in changes to the input. The recursive function is again called using the modified input.

```
1   public static int myRecursion(input)
2   {
3     if (recursion stopping condition)
4       return value;
5     //perform some operations
6     ...
7     //modified input  = make changes to input
8     ...
9     return myRecursion(modified input);
10  }
```

Code Fragment 1: myRecursion

Programming languages internally execute recursion using stacks.
To understand the call stack operations performed during execution, let us consider a call to myRecursion(input). Assuming that the recursion stopping criteria is satisfied at a recursive depth of 2, the recursion trace during execution is illustrated in Figure 1

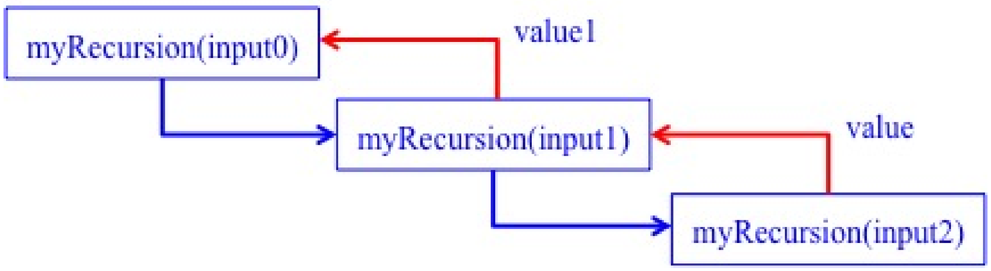

Figure 1: recursion trace on myRecursion

Before calling the first level recursion using input1, the current state of level 0 recursion is pushed onto a stack for future processing. Similarly before calling the second level recursion using input2, the current state of level 1 recursion is pushed onto the stack for future processing. After execution of second level of recursion, when the stopping condition is satisfied, value is returned. Following which recursion at level 1 continues to execute using the returned value.
The depth of the recursion can sometimes exceed the allowed capacity resulting in "Stack Overflow Error". You can verify this by executing a recursive procedure to compute the sum of first n integers with $n = 10000$. Our goal in this problem to re-write recursive functions using stacks and while loops. The idea is to simulate the stacked recursive function call.
In order to create the user defined stack, we need to first define a class whose object stores the current values of local variables used inside the recursion function (lines 5 through 8 in Code Fragment 1. We also need to store the stage of the current recursion if a recursive call can start more than one recursions (multiple recursions). Thus we would need only one stack to perform even multiple recursions. A sample snapshot class for the myRecursion function is illustrated in Code Fragment 2

```
/**
 * Sample class to store the values of local variables of myRecursion function.
 *
 * @author ckn
 *
 */
public class myRecursionSnapShot {
  /**
  * define all local variables as private members
  */
  private int stage;

  public myRecursionSnapShot() {
  }

  public myRecursionSnapShot(local variables, int stage) {
    //local variable assignments
    this.stage = stage;
  }
  /**
  * public get functions to obtain the value of the variables.
  */

  /**
  * public get functions to obtain the value of stage
  */

  public int getStage() {
    return stage;
  }
  /**
  * public set functions to assign values of the variables.
  */

  /**
  * public set functions to assign the value of the stage
  */
  public void setStage(int stage) {
    this.stage = stage;
  }
}
```

Code Fragment 2: Snapshot

You should create an object containing the current status values of local variables and push it to the stack before executing the deeper recursive function as discussed in the class. Implement the following problems first in the recursive fashion and then simulate the recursion using stacks and loops.

1. Sum of first n integers
2. n^{th} number in the Fibonacci series.
3. checks if a given string s is a palindrome.

```
/*
 * Copyright 2014, Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser
 *
 * Developed for use with the book:
 *
 *    Data Structures and Algorithms in Java, Sixth Edition
 *    Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser
 *    John Wiley & Sons, 2014
 *
 * This program is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 */

/**
 * A collection of objects that are inserted and removed according to the last-in
 * first-out principle. Although similar in purpose, this interface differs from
 * java.util.Stack.
 *
 * @author Michael T. Goodrich
 * @author Roberto Tamassia
 * @author Michael H. Goldwasser
 */
public interface Stack<E> {

  /**
   * Returns the number of elements in the stack.
   * @return number of elements in the stack
   */
  int size();

  /**
   * Tests whether the stack is empty.
   * @return true if the stack is empty, false otherwise
   */
  boolean isEmpty();

  /**
   * Inserts an element at the top of the stack.
   * @param e   the element to be inserted
   */
  void push(E e);

  /**
   * Returns, but does not remove, the element at the top of the stack.
   * @return top element in the stack (or null if empty)
   */
  E top();

  /**
   * Removes and returns the top element from the stack.
   * @return element removed (or null if empty)
   */
  E pop();
}
```

Code Fragment 3: Stack Interface

3 Stack Permutations (10 points)

A stack permutation is a ordering of numbers from 1 to n that can be obtained from the initial ordering $1, 2, ..., n$ by a sequence of stack operations. The stack permutations are performed using two queues and one stack. Elements from the input queue I can only be dequeued, elements can only be enqueued to the output queue O and push and pop operations are allowed on the stack S.The operations allowed are

- If I is not empty, remove the element at the head of I and place it at the end of O.
- If I is not empty, remove the element at the head of I and push it onto S.
- If S is not empty, pop the element at the top and place it at the end of O

For example, suppose we have the numbers 1, 2, 3, 4, 5 in the input queue I. We perform the following sequence of operations.

enqueue(O, dequeue(I))
enqueue(O, dequeue(I))
push(S, dequeue(I))
push(S, dequeue(I))
enqueue(O, dequeue(I))
enqueue(O, pop(S))
enqueue(O, pop(S))

The output queue O will now have the sequence 1, 2, 5, 4, 3. This is a stack permutation of the input sequence. Note that 1, 2, 5, 3, 4 is not an example for stack permutation.
In this problem implement a Java program to check whether a given sequence of numbers is a stack permutation. You will implement the stack and queue interfaces as described in Code Fragments 1 and 2. Implement these Interfaces using the singly linked list. You are not allowed to use the default stack and queue implementations available in Java.

```java
/*
 * Copyright 2014, Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser
 *
 * Developed for use with the book:
 *
 *    Data Structures and Algorithms in Java, Sixth Edition
 *    Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser
 *    John Wiley & Sons, 2014
 *
 * This program is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 */

/**
 * Interface for a queue: a collection of elements that are inserted
 * and removed according to the first-in first-out principle. Although
 * similar in purpose, this interface differs from java.util.Queue.
 *
 * @author Michael T. Goodrich
 * @author Roberto Tamassia
 * @author Michael H. Goldwasser
 */
public interface Queue<E> {
  /**
   * Returns the number of elements in the queue.
   * @return number of elements in the queue
   */
  int size();

  /**
   * Tests whether the queue is empty.
   * @return true if the queue is empty, false otherwise
   */
  boolean isEmpty();

  /**
   * Inserts an element at the rear of the queue.
   * @param e  the element to be inserted
   */
  void enqueue(E e);

  /**
   * Returns, but does not remove, the first element of the queue.
   * @return the first element of the queue (or null if empty)
   */
  E first();

  /**
   * Removes and returns the first element of the queue.
   * @return element removed (or null if empty)
   */
  E dequeue();
}
```

Code Fragment 4: Queue Interface

The input to your program will be the length of the sequence n, followed by the sequence. For example
5 1 2 5 4 3
6 2 1 3 4 6 5
If the input sequence is a stack permutation, the program should output the sequence of stack operations that will result in the sequence, else output the reason why the input sequence is not a stack permutation.