4CCS1DST Data Structures, 2019/20
Coursework 1

Coursework posted on 7 February 2020.

Submit your answers on **KEATS**.

Submission deadline: 19 February 2020, 16:00.

Submission Instructions

Submit your answers on KEATS as a single pdf (recommended) or MS Word file. No other formats will be accepted. The name of your file has to include your name and your KCL id number (the number on your college ID card) in the following format (beginning with the last name):

`LastName_FirstName_KCLid_CW1`

with the extension indicating the type of file. The submission deadline is given above. Late submissions will not be accepted (unless a formal deadline extension request has been accepted by the Chair of the Exam Board). The maximum size of a submission file is 2MB. Your submission must be typeset in such a way that it can be properly printed on A4 paper in black and white. For example, lines in source codes should be formatted so that they fit within the page width. If the answer requires that you show diagrams, create them using tools in the document typesetting system which you use or draw by hand, scan or make a picture, and include the image in your submission file. In either case, your diagram has to be clear an readable.

Check before submitting your file that it prints properly in black and white on A4 paper. Do not use colours – the print-out of your submission from a black and white printer will be used for marking.

It is not required that you run and test any codes, but it may be helpful for you to do so.

This Coursework 1 assignment contributes 7.5% to your final mark for the 4CCS1DST module, so your submission must be your own individual work. The College Academic Regulations and procedures apply, in particular the College's statement and strategy on plagiarism and related forms of cheating.

You must include the following information and declaration in the beginning of your submission file:

> 4CCS1DST – Data Structures, 2018/19
> Coursework 1
>
> Last Name:
> First Name:
> Student Number:
>
> By submitting this coursework, I declare that I understand the nature of plagiarism as defined in the Department Handbook and that the content of this coursework submission is entirely my own work.

Questions

1. (10 marks)

   Consider the recursive method `fnc(n,k)` defined below.

```
public static int fnc(int n, int k) {
   if ( n <= 1 ) { return k; }
   else if ( k <= 1 ) { return n; }
   else { return (1 + fnc(n-2,k+1) + fnc(n+1,k-2));
}
```

   Draw the recursion tree of the call `fnc(3,5)`. Your diagram should include the return values for all calls to method `fnc`.

2. (10 marks)

   Apply the parentheses matching algorithm discussed in Lecture 4 (slide 15) to the following input sequence of tokens:

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

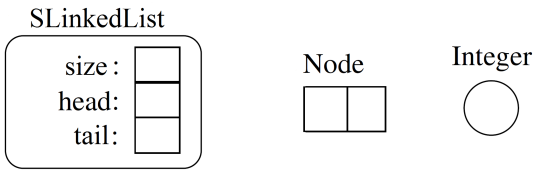
   There are three types of parehtheses in this sequence: ( ), [ ], and { }. This algorithm checks, using a stack, if all parentheses in the input sequence properly match. Show the content of the stack when the algorithm terminates, indicating clearly which end of the stack is the top.

   You do not have to show intermediate steps – show only the final stack.

3. (10 marks)

   This question refers to the `SLinkedList<E>` class from Lecture 2 (discussed in lectures and LGT). We are assuming that this class has methods `size`, `elementAtHead`, `elementAtTail`, `insertAtHead`, `insertAtTail`, and `removeAtHead`.

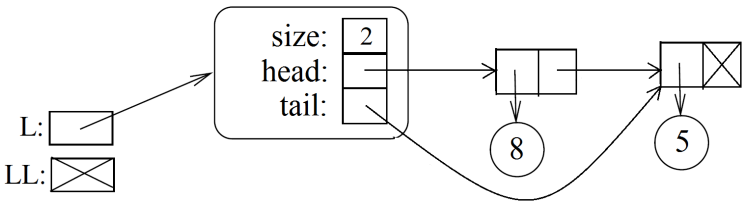
   Using the following graphical representation of objects of classes `SLinkedList`, `Node` and `Integer`:



   and indicating references to objects using arrows and `null` references using crosses, the state of the variables and objects after the following code is executed:

```
SLinkedList<Integer> L = new SLinkedList<Integer>();
L.insertAtHead(5);
L.insertAtHead(8);
SLinkedList<SLinkedList<Integer>> LL;
```

   can be represented by the following diagram:



   Draw the full diagram representing the state of all variables and objects after the following additional code is executed:

```
LL = new SLinkedList<SLinkedList<Integer>>();
LL.insertAtHead(L);
LL.insertAtHead(new SLinkedList<Integer>());
LL.elementAtTail().insertAtTail(7);
LL.elementAtHead().insertAtHead(3);
```

4. (10 marks) Give code for the method `compress` in the class `Stacks` shown below. This method has two arguments, the "main" stack `s1` and an auxiliary stack `s2`, both of the generic type `Stack<E>`. This generic type `Stack<E>` is the interface discussed in Lecture 4 (and is included in the `net.datastructures` package for the Goodrich-Tamassia-Goldwasser textbook). The objective of the method `compress` is to remove all `null` elements from the stack `s1`. The remaining (non-`null`) elements should be kept on `s1` in their initial order. The stack `s2` should be used as a temporary storage for the elements from `s1`. At the end of the computation of this method, stack `s2` should have the same content as at the beginning of the computation. The method should not use any arrays.

   See the method `main` below for an example of the expected behaviour of the method `compress`.

   Remark. The method `compress` only knows that `s1` and `s2` are implementations of the interface `Stack<E>`, but does not know anything about the details of those implementations. This means that method `compress` can access `s1` and `s2` only by using the interface methods.

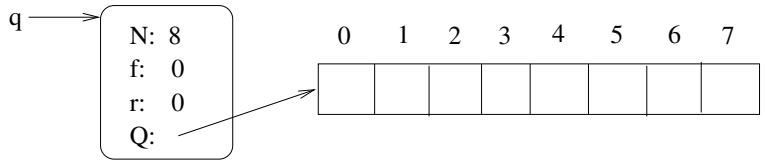
```
package labsSGTsCoursework.cw1;

import net.datastructures.Stack;
import net.datastructures.ArrayStack;

public class Stacks {

   public static <E> void compress(Stack<E> s1, Stack<E> s2) {
      ... // YOUR CODE REPLACES DOTS HERE
   }

   public static void main(String[] args) {

      // test method compress

      Stack<Integer> S = new ArrayStack<Integer>(10);
      S.push(2); S.push(null); S.push(null); S.push(4); S.push(6); S.push(null);
      Stack<Integer> X = new ArrayStack<Integer>(10);
      X.push(7);  X.push(9);
      System.out.println("stack S: " + S);
                              // prints: "stack S: [2, null, null, 4, 6, null]"
      System.out.println("stack X: " + X);
                              // prints: "stack X: [7, 9]"
      compress(S, X);
      System.out.println("stack S: " + S);
                              // should print: "stack S: [2, 4, 6]"
      System.out.println("stack X: " + X);
                              // should print: "stack X: [7, 9]"
   }

}
```

5. (10 marks)

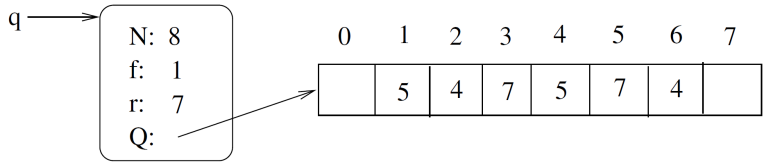
   This question refers to the array-based "Circular Queue" implementation of the Queue data structure included in Lecture 4. Assume that class `CircularQueue<E>` is this implementation in Java. The statement:

```
Queue<Integer>  q = new CircularQueue<Integer>(8);
```

   creates the empty queue `q`, which has the internal representation shown in the following diagram:



   After seven `enqueue` operations and one `dequeue` operation the internal representation of the queue `q` is:



   Trace now the computation of the following code:

```
while ( (q.size() > 1) && (q.front() < 10) ) {
    q.enqueue(q.dequeue() + q.dequeue());
}
```

   and show the **internal** representation of the queue `q` (in the form as in the above diagrams) after this computation.

   Remark: you do not need to implement the class `CircularQueue<E>`.