4CCS1DST – Data Structures
Coursework 1

Last Name: Monte
First Name: Aaron Patrick
Student Number: 20059926

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.

signed: Aaron. 15/2/2021

Questions

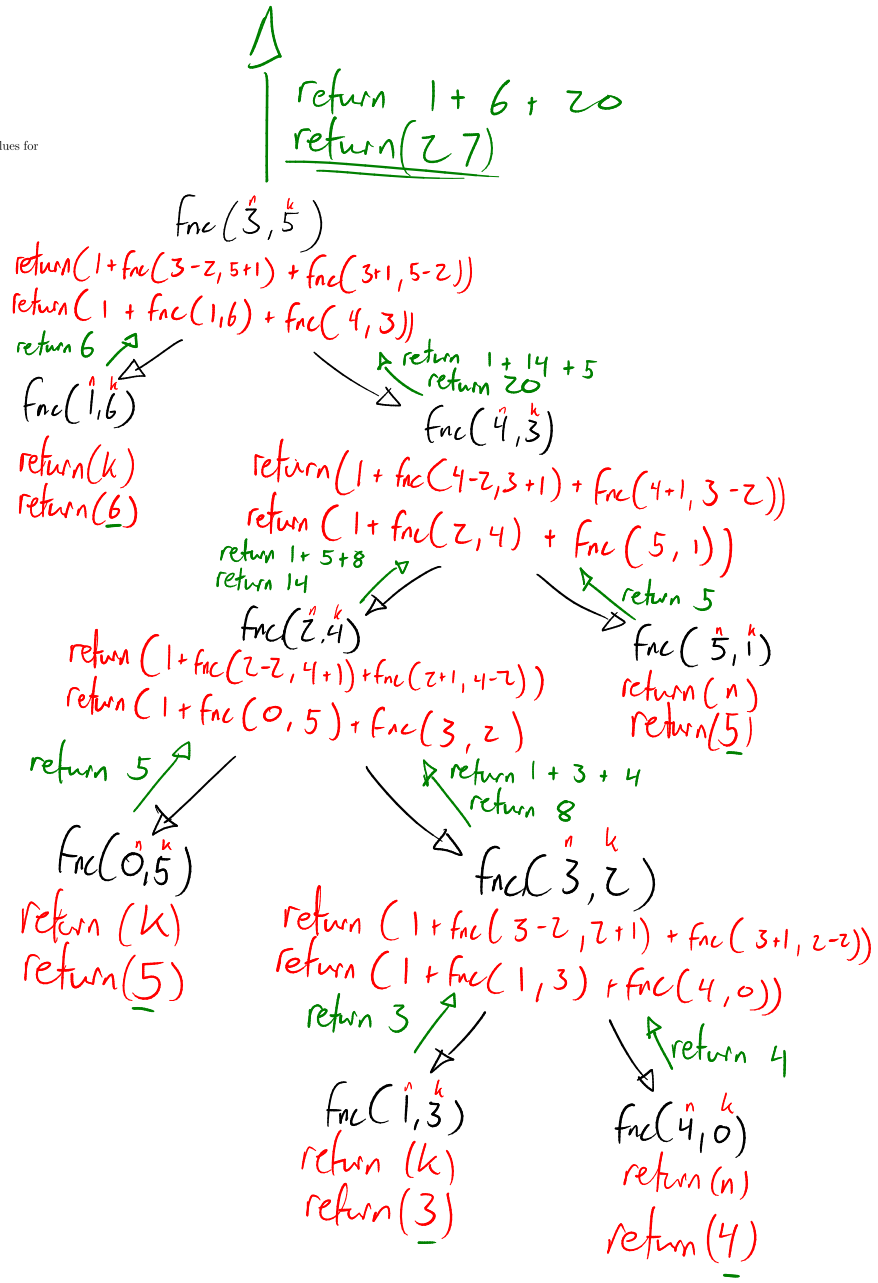
1. (10 marks)

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`.

assuming there is a closing bracket:



2. (10 marks)

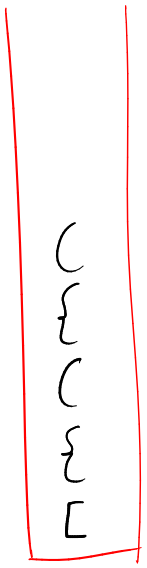
Apply the parentheses matching algorithm discussed in Lecture 4 (slide 15) to the following input sequence of tokens:

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

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.

top →



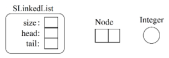
return False

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

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`.

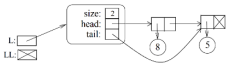
Using the following graphical representation of objects of classes `SLinkedList`, `Node` and `Integer`:



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:

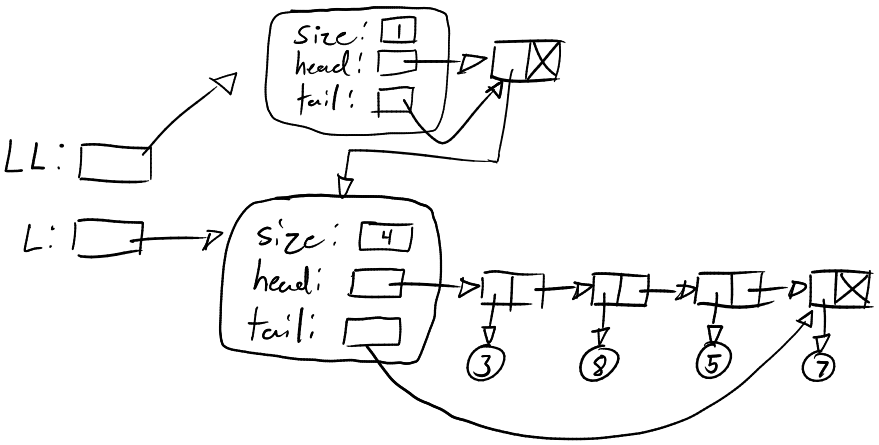
```
SLinkedList<Integer> L = new SLinkedList<Integer>();
L.insertAtHead(5);
L.insertAtHead(8);
SLinkedList<SLinkedList<Integer>> LL;
```

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);
```



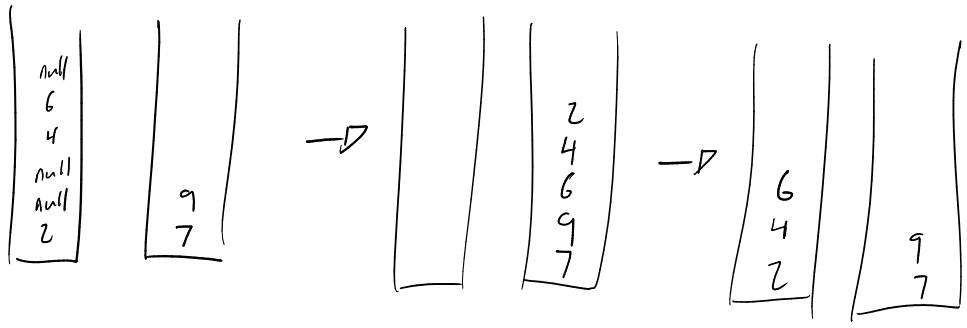
```
public static <E> void compress(Stack<E> s1, Stack<E> s2) {

        // record initial size to know where to stop
        int initialSize = s2.size();

        // remove nulls and push not nulls into s2
        while(!s1.isEmpty()) {

                if(s1.top() != null) {
                        s2.push(s1.pop());
                } else {
                        s1.pop();
                }

        // push s1 values that are currently in s2 back into s1
        while(s2.size() != initialSize){
                s1.push(s2.pop());
        }
}
```

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.

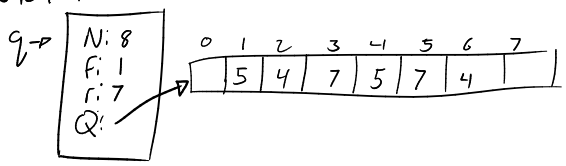
   See the method `main` below for an example of the expected behaviour of the method `compress`.

   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.

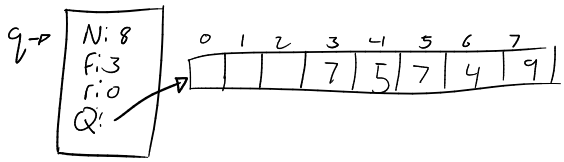
```
package labeSGTeCoursework.cw1;

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]"
   }

}
```



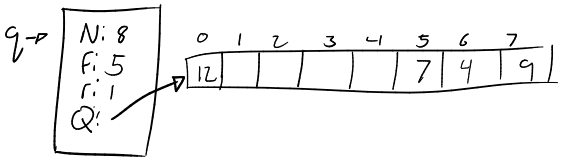
initial state:



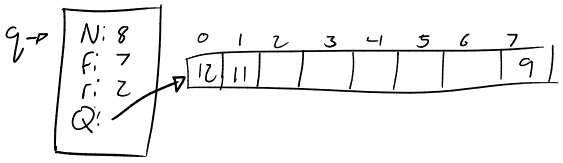
round 1:



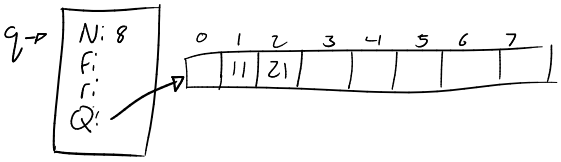
round 2:



round 3:



round 4:



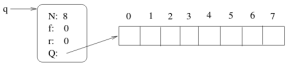
STOP

5. (10 marks)

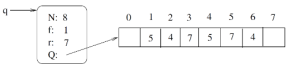
This question refers to the array-based "Circular Queue" implementation of the Queue data structure included in Lecture 4. Assume that class `CircularQueue<E>` is this implementation in Java. The statement:

```
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>`.