Université d'Ottawa
Faculté de génie

École de science informatique
et de génie électrique

uOttawa

University of Ottawa
Faculty of Engineering

School of Electrical Engineering
and Computer Science

Introduction to Computing II (ITI 1121)
FINAL EXAMINATION: SOLUTIONS

Instructors: Guy-Vincent Jourdan and Marcel Turcotte

April 2019, duration: 3 hours

Identification

Last name: ______________________ First name: ______________________________

Student #: ____________ Seat #: ____ Signature: ____________________ Section: A or B or C

Instructions

1. This is a closed book examination.
2. No calculators, electronic devices or other aids are permitted.
   (a) Any electronic device or tool must be shut off, stored and out of reach.
   (b) Anyone who fails to comply with these regulations may be charged with academic fraud.
3. Write your answers in the space provided.
   (a) Use the back of pages if necessary.
   (b) You may not hand in additional pages.
4. Do not remove pages or the staple holding the examination pages together.
5. Write comments and assumptions to get partial marks.
6. Beware, poor hand-writing can affect grades.
7. Wait for the start of the examination.

Marking scheme

Question	Maximum	Result
1	15	
2	15	
3	15	
4	15	
Total	**60**	

All rights reserved. No part of this document may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise without prior written permission from the instructors.

Question 1 (15 marks)

Implement the instance method **merge** for the class **LinkedList** that has the following characteristics.

- The list always starts with a dummy node, which is used as a marker for the beginning of the list. The dummy node is never used to store data. The empty list is made up of the dummy node only;
- In the implementation for this question, the nodes in the list are doubly linked;
- In this implementation, the list is circular, i.e. the reference from the last node in the list points to the dummy node, the reference **prev** of the dummy node points to the last node in the list. In the empty list, the dummy node is the first and last node of the list, its references **prev** and **next** point to the node itself;
- The last node is easily accessible, because it's always the node preceding the dummy node, the header of the list does not have a pointer to the tail element.

The method **merge** adds all the elements of the list designated by the parameter **other** to **this** list. After a call to the method **merge**, the list designated by the parameter **other** is empty.

- This question assesses your understanding of linked structures. For this reason, you cannot use the methods of the class **LinkedList**. In particular, you cannot use the methods **add()** or **remove()**.
- The method never throws an exception.

```
public class Test {
    public static void main(String[] args) {
        LinkedList<String> firstList, secondList;
        firstList = new LinkedList<String>();
        secondList = new LinkedList<String>();

        firstList.addLast("alpha");
        firstList.addLast("bravo");
        firstList.addLast("charlie");
        firstList.addLast("delta");

        secondList.addLast("echo");
        secondList.addLast("foxtrot");

        System.out.println(firstList);
        System.out.println(secondList);

        firstList.merge(secondList);

        System.out.println(firstList);
        System.out.println(secondList);
    }
}
```

Running the above program produces the following on the console.

```
[alpha, bravo, charlie, delta]
[echo, foxtrot]
[alpha, bravo, charlie, delta, echo, foxtrot]
[]
```

Complete the implementation of the method **merge**.

```
public class LinkedList<E> implements List<E> {

    private static class Node<T> {
       private T value;
       private Node<T> prev;
       private Node<T> next;
       private Node(T value, Node<T> prev, Node<T> next) {
          this.value = value;
          this.prev = prev;
          this.next = next;
       }
    }

    private final Node<E> head;
    private int size;

    public LinkedList() {
        head = new Node<E>(null, null, null);
        head.next = head.prev = head;
        size = 0;
    }

    public void merge(LinkedList<E> other) { // MODEL

        if (other != null&&other.size > 0*) {

            Node<E> thisLast, otherFirst, otherLast;

            thisLast = head.prev;

            otherFirst = other.head.next;

            otherLast = other.head.prev;

            otherFirst.prev = thisLast;

            thisLast.next = otherFirst;

            head.prev = otherLast;

            otherLast.next = head;

            other.head.next = other.head;

            other.head.prev = other.head;

            size = size + other.size;

            other.size = 0;

        }

    }
}
```

(*) `other.head.next != other.head`

Question 2 (15 marks)

Complete the implementation of the method **findAndReplace** for the class **LinkedList** on the next page. The method replaces all the occurrences of the object specified by the parameter **target** with the value of the parameter **replacement**. It also returns the total number of elements that were replaced by the method.

This is a recursive method. Please note that the list is simply linked and that there is no dummy node. To help you understand the expectations, here is a test program.

```
LinkedList<String> list;
list = new LinkedList<String>();

System.out.println(list);

System.out.println(list.findAndReplace("I", "she"));

System.out.println(list);

list.addLast("I");
list.addLast("said");
list.addLast("she");
list.addLast("said");
list.addLast("she");
list.addLast("said");
list.addLast("I");
list.addLast("said");

System.out.println(list);

System.out.println(list.findAndReplace("I", "she"));

System.out.println(list);

System.out.println(list.findAndReplace("I", "she"));

System.out.println(list);
```

Running the program above will produce the following result on the output.

```
[]
0
[]
[I, said, she, said, she, said, I, said]
2
[she, said, she, said, she, said, she, said]
0
[she, said, she, said, she, said, she, said]
```

Complete the implementation of the method **findAndReplace**.

```
public class LinkedList<E> implements List<E> {

    private static class Node<T> {

        private T value;
        private Node<T> next;

        private Node(T value, Node<T> next) {
            this.value = value;
            this.next = next;
        }
    }

    private Node<E> head;
    private int size;

    public int findAndReplace(E target, E replacement) {

        if (target == null || replacement == null) {

            throw new NullPointerException();

        }

        return findAndReplace(head, target, replacement);
    }

    private int findAndReplace(Node<T> current, E target, E replacement) {

        int number;

        if (current == null) {

            number = 0;

        } else { // general case

            number = findAndReplace(current.next, target, replacement);

            if (current.value.equals(target)) {

                current.value = replacement;

                number++;

            }

        }

        return number;
    }

}
```

Question 3 (15 marks)

For this question, our goal is to implement five (5) methods for a **dynamic circular array list**. That list has the following instance variables:

- **array** is a reference variable to the array containing the elements of the list;
- **size** is the logical (current) size of the list;
- **first** is the index of the first element of the list in **array**;
- **last** is the index of the last element of the list in **array**;
- **capacity** is the capacity of the array.

By convention, when the list is empty, both **first** and **last** are zero. We have two constructors for our class: the first one, without parameter, creates an array of the default capacity, while the second one uses the value of its parameter as the initial size of the array. The size must be at least 1.

Here is a partial implementation for our class:

```
public class CircularDynamicArrayList<E> {

    private E[] array;
    private int size;
    private int first, last;
    private int capacity = 100;

    @SuppressWarnings("unchecked")
    public CircularDynamicArrayList() {
        array = (E[]) new Object[capacity];
        first = last  = 0;
        size = 0;
    }

    @SuppressWarnings("unchecked")
    public CircularDynamicArrayList(int capacity) {
        if (capacity < 1) {
            throw new IllegalArgumentException("Minimum capacity is 1");
        }
        this.capacity = capacity;
        array = (E[]) new Object[capacity];
        first = last = 0;
        size = 0;
    }

    public boolean isEmpty() {
        return size == 0;
    }

    public boolean isFull() {
        return size == capacity;
    }

    // More code

}
```

Question 3.1 *ensureSpace*

The first method we want to write is **ensureSpace**. This method makes sure that the list will be able to accommodate a new element. If necessary it **doubles** the size of the array. Complete the source code for the implementation below.

```
public class CircularDynamicArrayList<E> {

    // (...)

    private void ensureSpace() {

        if (isFull()) {

            Object[] newArray = new Object[2*capacity];

            for (int i = 0; i <capacity; i++) {

                newArray[i] = array[(first+i)%capacity];

            }

            first = 0;

            last = capacity - 1;

            capacity = 2*capacity;

            array = (E[]) newArray;

        }
    }
}
```

Question 3.2 *addFirst*

The method **addFirst** adds a new element to the list at the first position. All the other elements are kept. We must ensure to check for invalid conditions. You can use the methods of the class written so far in this new method.

```
public class CircularDynamicArrayList<E> {

   // (...)

   public void addFirst(E newElement) {

        if (newElement == null) {

            throw new NullPointerException();

        }

        ensureSpace();

        if (isEmpty()) {

            first = last = 0;

        } else {

            first = (first+capacity-1) % capacity;

        }

        array[first] = newElement;

        size = size + 1;

    }
}
```

Question 3.3 *removeFirst*

The method **removeFirst** removes and returns the element at the first position. All the other elements are kept. We must ensure to check for invalid conditions. You can use the methods of the class written so far in this new method.

```
public class CircularDynamicArrayList<E> {

    // (...)

    public E removeFirst() {

        if (isEmpty()) {

            throw new IllegalStateException();

        }

        E resultat = array[first];

        array[first] = null;

        size =  size - 1;

        if (size == 0) {

            first = last = 0;

        } else {

            first = (first+1)%capacity;

        }

        return result;

    }
}
```

Question 3.4 *add*

The method **add** adds a new element to the list at the specified position. Indexes are 0-based. All the other elements are kept. We must ensure to check for invalid conditions. You can use the methods of the class written so far in this new method.

```
public class CircularDynamicArrayList<E> {

    // (...)

    public void add(E newElement, int index) {

        if (newElement == null) {

            throw new NullPointerException();

        }

        if (index < 0 || index > size) {

            throw new IndexOutOfBoundsException();

        }

        if (isEmpty()) {

            addFirst(newElement);

        } else {

            ensureSpace();

            int currentIndexLocation =  (first+index) % capacity;

            int i =  (last+1) % capacity;

            while (i != i != currentIndexLocation) {

                array[i] = array[(i+capacity-1) % capacity];

                i = (i+capacity-1) % capacity;

            }

           size++;

           last = (last+1) % capacity;

           array[currentIndexLocation] = newElement;

        }
    }
}
```

Question 3.5 *remove*

Finally, the method **remove** removes the element at the specified index in the list, and returns it. All the other elements are kept. We must ensure to check for invalid conditions. You can use the methods of the class written so far in this new method.

```
public class CircularDynamicArrayList<E> {

    // (...)

    public E remove(int index) {

        if(isEmpty()) {

             throw new IllegalStateException();

        }

        if(index < 0 || index >= size) {

            throw new IndexOutOfBoundsException();

        }

        E resultat;

        if (size == 1) {

            result = removeFirst();

        } else {

            int currentIndexLocation = (first+index) % capacity;

            resultat = array[currentIndexLocation];

            int i = currentIndexLocation;

            while (i != last) {

                array[i] = array[(i+1) % capacity]

                i =  (i+1) % capacity;

            }

           array[last] = null;

           size--;

           last = (last+capacity-1) % capacity;

        }

        resultat;
    }
}
```

Question 4 (15 marks)

In this question, we are using **lists** and **binary search trees** together. Specifically, we will work with the following interface **List**:

```
public interface List<E> {

    // Returns the current size of the list
    int getSize();

    // Returns true if and only if the list is empty
    boolean isEmpty();

    // Adds elem as the first element of the list
    void addFirst(E elem);

    // Adds elem as the last element of the list
    void addLast(E elem);

    // Adds elem at (0-based) index ''index'' in the list
    void add(int index,E elem);

    // Returns the reference of the first element of the list
    E getFirst();

    // Returns the reference of the last element of the list
    E getLast();

    // Returns the reference of the element at index ''index''  in the list
    E get(int index);

    // Removes and returns the reference of the first element of the list
    E removeFirst();

    // Removes and returns the reference of the last element of the list
    E removeLast();

    // Removes returns the reference of the element at  index ''index'' in the list
    E remove(int index);
}
```

In the following, we have access to two implementations of the interface **List**. We do not specify which one to use yet. We also have a class **BinarySearchTree**. Some parts of its implementation is provided below.

Question 4.1 buildSortedList

Our first goal is to add a method **buildSortedList** to the class **BinarySearchTree**. That method receives a reference to an initialized, empty **List**, as an input parameter, and it populates that list with all the elements currently in the binary search tree, in increasing order.

For example, the following test code:

```
public static void testBuildSortedList(List<String> list) {

    BinarySearchTree<String> tree;
    tree = new BinarySearchTree<String>();

    tree.add("F"); tree.add("A"); tree.add("C");
    tree.add("X"); tree.add("M"); tree.add("N");
    tree.add("O"); tree.add("B"); tree.add("L");

    System.out.println(list);
    tree.buildSortedList(list);
    System.out.println(list);

}
```

produces the following output:

```
[]
[A, B, C, F, L, M, N, O, X]
```

If the tree is currently empty, then the lists remains unchanged. For example, the following test code:

```
public static void testBuildSortedList2(List<String> list) {

    BinarySearchTree<String> tree;
    tree = new BinarySearchTree<String>();

    System.out.println(list);
    tree.buildSortedList(list);
    System.out.println(list);

}
```

produces the following output:

```
[]
[]
```

A partial, incomplete listing of the class **BinarySearchTree** is provided next:

```
public class BinarySearchTree <E extends Comparable<E>> {

    private static class Node<T> {
        private T value;
        private Node<T> left;
        private Node<T> right;
        private Node(T value) {
            this.value = value;
            left = null;
            right = null;
        }
    }

    private Node<E> root;
    private int size;

    public BinarySearchTree() {
        root = null;
        size = 0;
    }

    public int getSize() {
        return size;
    }

    public boolean contains(E obj) {
        // hidden to save space
      }

    public boolean add(E obj) {
        // hidden to save space
    }

    public String toString() {
        // hidden to save space
    }

    // More code not shown

}
```

In the space below, provide the code for the method **buildSortedList**. As you can see, the method uses a private method, suggesting a recursive implementation.

```
public class BinarySearchTree <E extends Comparable<E>> {

    // Lots of code not shown

    public void buildSortedList(List<E> list) {

        if (list == null) {
            throw new NullPointerException("List must not be null");
        }

        if (list.getSize() != 0) {
            throw new IllegalArgumentException("List must be empty");
        }

        // MODEL

        if (root != null) {
            buildSortedList(list, root);
        }

    }

    private void buildSortedList( List<E> list, Node<E> current) {

        // MODEL

        if (current.left != null) {
            buildSortedList(list, current.left);
        }

        list.addLast(current.value);

        if (current.right != null) {
            buildSortedList(list, current.right);
        }

    }

}
```

Question 4.2 Constructing a balanced tree

We now use the sorted list to construct a new binary search tree. In this section, we need to add a new constructor to the class **BinarySearchTree**. That constructor receives as parameter a reference to a **List**, which is assumed to be sorted. The goal of the constructor is to create a new binary search tree containing all the elements of the list. The resulting tree should be as balanced as possible.

For example, assume that **list** designates a list with the following elements: **[1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]**. This list contains 13, consecutive, sorted Fibonnacci numbers.

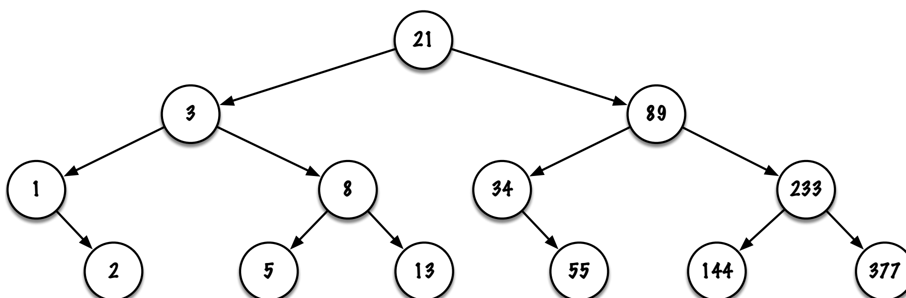
The following code snippet:

```
System.out.println(list);
BinarySearchTree<Integer> tree;
tree = new BinarySearchTree<Integer>(list);
```

produces the following output:

```
[1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]
        null
      377
        null
    233
        null
      144
        null
  89
        null
      55
        null
    34
      null
21
        null
      13
        null
    8
        null
      5
        null
  3
        null
      2
        null
    1
      null
```

In other words, it builds the following tree:



In the space below, provide your implementation of the new constructor. Here again, the method uses a private method.

```
public class BinarySearchTree <E extends Comparable<E>> {

    // Lots of code not shown

    public BinarySearchTree(List<E> list) {

        if(list == null || list.getSize() == 0) {
            return;
        }

        root = buildBalanced(0, list.getSize()-1, list);

    }

    private Node<E> buildBalanced(int from, int to, List<E> list) {

        // MODEL

        int mid = from + (to-from)/2;

        Node<E> node = new Node<E>(list.get(mid));

        if (mid > from) {
            node.left = buildBalanced(from, mid-1,list);
        }

        if (mid < to) {
            node.right = buildBalanced(mid+1, to,list);
        }

        return node;

      }

}
```

Question 4.3 Efficient implementation

The ultimate goal of our two methods is to be able to create a balanced binary search tree from an existing possibly unbalanced binary search tree, using code similar to the following:

```
    public BinarySearchTree<E> createBalanced(BinarySearchTree<E> unbalanced) {

        List<E> list;

        list  = ......  // initialization goes here

        unbalanced.buildSortedList(list);

        BinarySearchTree<E> balanced;

        balanced = new BinarySearchTree<E>(list);

        return balanced;
    }
```

We need that code to be as efficient as possible[1], which means that our method **buildSortedList** and our new binary search tree constructor must be efficient.

For our list implementation, we have two choices: our first choice is a circular doubly-linked list with a dummy node. That implementation has a single constructor, without parameter:

```
public class DoublyLinkedListDummyNode<E> implements List<E> {

        public DoublyLinkedListDummyNode() {
            // some code
        }

        // more code
}
```

Our second choice is a circular dynamic array list implementation similar to the one of Question 3. In this implementation, the size of the array is doubled when an element is added to a full array. That implementation has two constructors: a constructor without parameter, and another one which received a *int* value as input parameter. With the first constructor, some default value is used for the initial size of the array. With the second constructor, the input parameter is used as the initial size of the array:

```
public class CircularDynamicArrayList<E> implements List<E> {

        public CircularDynamicArrayList() {
            // Some code. Uses some default value for the
            // initial size of the array
        }

        public CircularDynamicArrayList(int initialSize) {
            // Some code. Uses initialSize for the
            // initial size of the array
        }

        // More code
}
```

[1]Specifically, it must be linear in the size of the unbalanced binary search tree.

You need to decide which implementation are we going to use in the method **createBalanced** in order to have a fast implementation. Add your initialization of the variable **list** in the code below:

```
    public BinarySearchTree<E> createBalanced(BinarySearchTree<E> unbalanced) {

        List<E> list;

        list = new ArrayList<E>(unbalanced.getSize());

        unbalanced.buildSortedList(list);

        BinarySearchTree<E> balanced;
        balanced = new BinarySearchTree<E>(list);

        return balanced;

    }
```