Final 2016 and solutions

Introduction to Computing II (University of Ottawa)

Scan to open on Studocu

Studocu is not sponsored or endorsed by any college or university
Downloaded by Nexus Tech (nexus.w124rd225@gmail.com)

Université d'Ottawa
Faculté de génie

École de science informatique
et de génie électrique

University of Ottawa
Faculty of engineering

School of Electrical Engineering
and Computer Science

Introduction to Computing II (ITI1121)
Final Examination: Solutions

Instructors: Nour El-Kadri, Guy-Vincent Jourdan, and Marcel Turcotte

April 2016, 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	25	
4	15	
5	10	
Total	**80**	

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.

Downloaded by Nexus Tech (nexus.w124rd225@gmail.com)

Question 1 Multiple-choice questions (15 marks)

A. Practices such as the one depicted by the following program should be avoided. Assume that **s** designates a valid instance of some stack implementation and that **value** is of the correct type.
[✓ **True**] or **False**

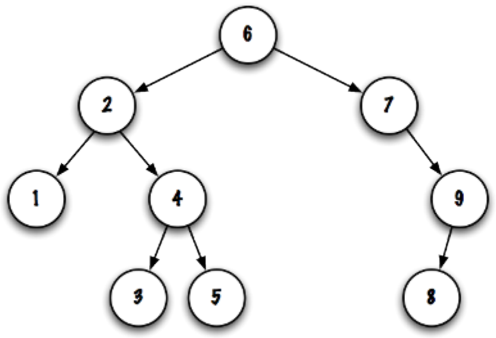
```
boolean done = false;

while (! done) {
    try {
        value = s.pop();
    } catch (EmptyStackException e) {
        done = true;
    }
}
```

B. A method that throws unchecked exceptions **must** declare them using the keyword **throws**, otherwise a compile-time error will be produced.
True or [✓ **False**]

C. Good object oriented programming practices suggest that a programmer must always catch all the exceptions generated by the methods being called.
True or [✓ **False**]

D. Consider two empty **BinarySearchTree** objects and the method **add** presented in class. Adding the same elements, but in different order into the two trees, will always lead to trees of the same height.
True or [✓ **False**].

E. Consider the binary search tree below.



For each series of nodes, circle the corresponding traversing strategy (pre-order, in-order, post-order, other):

Order of traversal: **6-2-1-4-3-5-7-9-8**	[✓ Pre-order]	In-order	Post-order	Other
Order of traversal: **1-3-5-4-2-8-9-7-6**	Pre-order	In-order	[✓ Post-order]	Other
Order of traversal: **1-2-3-4-5-6-7-8-9**	Pre-order	[✓ In-order]	Post-order	Other

Downloaded by Nexus Tech (nexus.w124rd225@gmail.com)

F. Consider 4 implementations of a list:

- **ArrayList**: a simple array implementation, which has an instance reference variable to the array, as well as an instance variable containing the number of elements in the array.
- **CircularArrayList**: a circular array implementation, which has an instance reference variable to the array, as well as instance variables for the current front and rear elements in the array.
- **LinkedList**: a (singly) linked list implementation, which has an instance reference variable to the current head Node of the list.
- **DoublyLinkedList**: a doubly linked list implementation, which has an instance reference variable to the current head Node of the list and an instance reference variable to the current tail Node of the list, and doubly linked Nodes.

For each of the six methods below, indicate for each implementation if the method can be **Fast** (that is, its execution time is constant) or if it will be **Slow** (that is, proportional to the number of elements in the list).

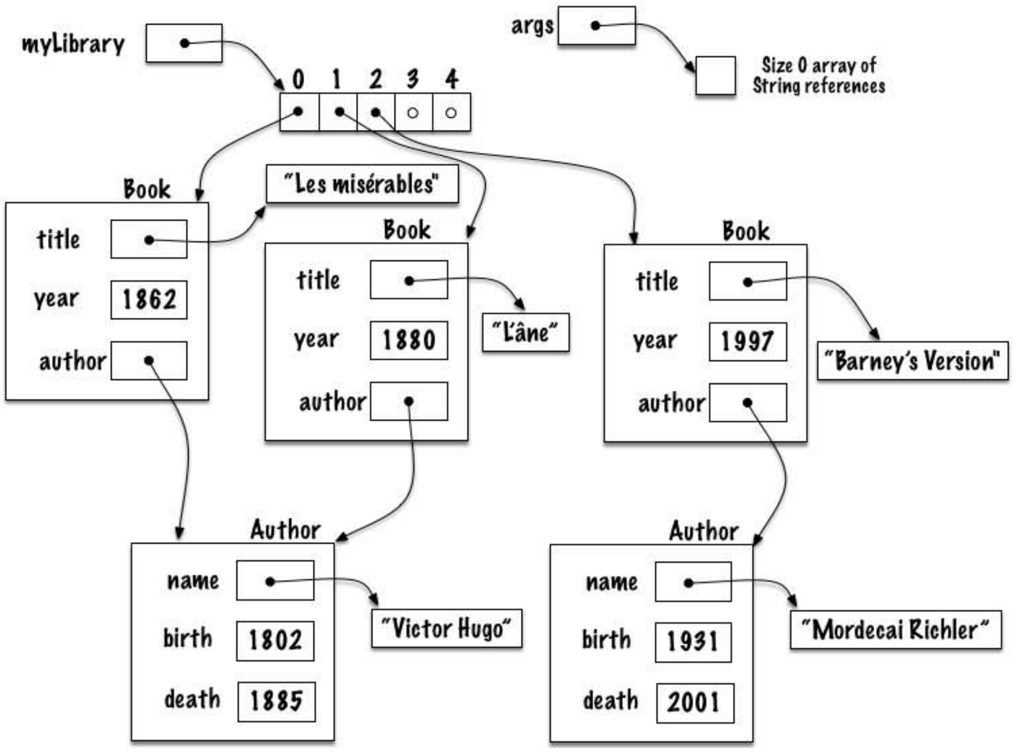
Note that both array implementations are based on **dynamic arrays**, and thus can accommodate any number of elements. However, the array is not automatically shrunk.

	ArrayList	**CircularArrayList**	**LinkedList**	**DoublyLinkedList**
void addFirst(E elem)	Slow	Slow	Fast	Fast
void addLast(E elem)	Slow	Slow	Slow	Fast
void add(E elem, int pos)	Slow	Slow	Slow	Slow
E get(int pos)	Fast	Fast	Slow	Slow
void removeFirst()	Slow	Fast	Fast	Fast
void removeLast()	Fast	Fast	Slow	Fast

Downloaded by Nexus Tech (nexus.w124rd225@gmail.com)

Question 2 (15 marks)

As we have done for Question 1 of Assignment 4, reverse engineer the memory diagram below: you need to provide the implementation of all the classes, instance variables, constructors and the main method that will lead to that memory state.
Hint: You need a class **Book** and a class **Author**, as well as a **main** method.



```
// Model

public class Author {

    private String name;
    private int birth;
    private int death;

    public Author(String name, int birth , int death){
                this.name = name;
                this.birth = birth;
                this.death = death;
    }

}
```

Downloaded by Nexus Tech (nexus.w124rd225@gmail.com)

```
// Model

public class Book {

    private int year;
    private String title;
    private Author author;

    public Book(String title, int year, Author author) {
        this.year = year;
        this.title = title;
        this.author = author;
    }

    public Author getAuthor() {
           return author;
    }

    public static void main(String[] args) {

        Book[] myLibrary = new Book[5];

        myLibrary[0] = new Book("Les miserables", 1862, new Author("Victor Hugo", 1802, 1885));
        myLibrary[1] = new Book("L'ane", 1880, myLibrary[0].getAuthor());
        myLibrary[2] = new Book("Barney's Vesion", 1997, new Author("Mordecai Richler", 1931,

    }

}
```

Downloaded by Nexus Tech (nexus.w124rd225@gmail.com)

Question 3 (25 marks)

We have seen in class two main ways of implementing a list: using an array or using linked elements. We have also seen that both approaches each have strengths and weaknesses. In some cases, it could be useful to be able to switch between the two approaches, based on what the application is currently doing.

Our goal is to provide a **hybrid** implementation. That implementation provides **both** an array implementation and a linked-list implementation, albeit not at the same time. The linked-list implementation uses a singly-list approach (with no dummy node). The array implementation uses a simple array (no circular array).

In order to support both implementations, our **HybridList** class will have the following instance variables:

- **headNode** is a reference pointing at the first **Node** of a non-empty list.
- **headArray** is a reference variable pointing to the array.
- **currentSize** is a primitive variable which holds the current number of elements in the list.
- **currentCapacity** is a primitive variable which holds the current capacity for the array.
- Finally, **isArray** is primitive variable used to record whether the object is currently using an array or a linked-list to store the list's elements.

The code below shows the beginning of the implementation.

```java
public class HybridList<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> headNode;
    private E[] headArray;
    private int currentSize = 0;
    private int currentCapacity = 50;

    private boolean isArray = false;

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

    // ...
```

Downloaded by Nexus Tech (nexus.w124rd225@gmail.com)

You need to provide the implementation of 2 methods: the method **public void toArray()**, which changes the storage of the current list elements from linked list to array, the method **public void toLinkedList()**, which changes the storage of the current list elements from array to linked list.

- In both cases, if the implementation is already of the right kind (already using an array when **toArray()** is called, or already using a linked list when **toLinkedList()** is called), the method does nothing.
- In these methods, you do not worry about memory management. In particular, you do not need to "scrub" the memory of the "old" storage mechanism when switching to the new one.
- For your implementation of the methods, you cannot use the methods of the class **HybridList**, accordingly these methods are not shown on the listing[1]. Your code needs to manipulate the instance variables directly, **headNode**, **headArray**, **currentSize**, **currentCapacity** and **isArray**.
- You can assume that **currentSize** is always correct. However, you cannot assume that the value stored in **currentCapacity** is up-to-date when the method **toArray()** is called, so you have to take care of this.
- Note that neither **toArray()** to **toLinkedList()** return any value. They simply change the internal representation of the list.

The following sample code shows a program using that class. In this sample program, the hybrid list is first used as a **LinkedList**. In that form, a series of elements are added. It is then switched to an **ArrayList**, and the elements are accessed through the method **get()**.

```
public static void main(String[] args) {

        HybridList<Integer> hList = new HybridList<Integer>();

        hList.toLinkedList(); //hList is now a LinkedList

        for (int i=0; i<100; i++) {
            hList.addFirst(i);
        }

        hList.toArray(); //hList is now an Array

        for(int i = 0 ; i < hList.size(); i ++) {
            System.out.println(hList.get(i));
        }

}
```

In the next 2 pages, provide your code for both methods.

[1]The method **isEmpty()** is shown, so you can use this one.

Downloaded by Nexus Tech (nexus.w124rd225@gmail.com)

```
// continuing class HybridList<E> implements List<E>

    public void toArray() {

        // Model

        if (isArray) {
            return;
        }

        while (currentCapacity < currentSize) {
            currentCapacity = 2*currentCapacity;
        }

        headArray = (E[]) new Object[currentCapacity];

        Node<E> p = headNode;
        for (int i =0; i < currentSize; i++) {
            headArray[i] = p.value;
            p = p.next;
        }

        isArray = true;









   } // End of toArray
```

Downloaded by Nexus Tech (nexus.w124rd225@gmail.com)

```
// continuing class HybridList<E> implements List<E>

    public void toLinkedList() {


        // Model

        if (!isArray) {
            return;
        }

        if (currentSize == 0) {
            headNode = null;
        } else {
            headNode = new Node<E>(headArray[0], null);
            Node<E> p = headNode;
            for (int i = 1; i < currentSize; i++) {
                p.next = new Node<E>(headArray[i], null);
                p = p.next;
            }
        }

        isArray = false;

    } // End of toLinkedList
```

Downloaded by Nexus Tech (nexus.w124rd225@gmail.com)

Question 4 (15 marks)

Write the static method **roll(Stack<E> s, int n)** that transforms the stack designated by **s** such that the **n** bottom elements are now on the top of the stack. The relative order of the top and bottom elements must remain unchanged.

```
Stack<String> s;
s = new LinkedStack<String>();

s.push("a");
s.push("b");
s.push("c");
s.push("d");
s.push("e");

System.out.println(s);

roll(s, 2);

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

Executing the above Java test program displays the following:

```
[a, b, c, d, e]
[c, d, e, a, b]
```

- Make sure to handle all exceptional situations appropriately.
- Since you do not know the size of the stack, you cannot use arrays for temporary storage. Instead, you must use stacks. You can assume the existence of the class **LinkedStack**, which implements the interface **Stack**.
- **Stack** is an interface.

```
public interface Stack<E> {
    boolean isEmpty();
    E peek();
    E pop();
    void push(E elem);
}
```

Downloaded by Nexus Tech (nexus.w124rd225@gmail.com)

```
public class Roll {
    public static <E> void roll(Stack<E> s, int n) {

        // Model

        if (s == null) {
            throw new NullPointerException();
        }

        if (n < 0) {
            throw new IllegalArgumentException("n is negative: " + n);
        }

        Stack<E> reverse, forward;

        reverse = new LinkedStack<E>();
        forward = new LinkedStack<E>();

        int count = 0;
        while (! s.isEmpty()) {
            reverse.push(s.pop());
            count++;
        }

        if (n > count) {
            throw new IllegalArgumentException("n is too large");
        }

        for (int i=0; i<n; i++) {
            forward.push(reverse.pop());
        }

        while (! reverse.isEmpty()) {
            s.push(reverse.pop());
        }

        while (! forward.isEmpty()) {
            reverse.push(forward.pop());
        }

         while (! reverse.isEmpty()) {
            s.push(reverse.pop());
        }











    } // End of roll
} // End of Roll
```

Downloaded by Nexus Tech (nexus.w124rd225@gmail.com)

Question 5 (10 marks)

Consider the class **SinglyLinkedList** outlined below:

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

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

    }

    private Node<E> head;

    // class continues after that
```

Consider the following code for the method **mystery1** in the class **SinglyLinkedList**

```
   public boolean mystery1(E o) {

        if(o == null) {
            throw new NullPointerException("Null parameter");
        }

        if (head == null) {
            return false;
        }

        boolean result = mystery1(head, o);

        if (head.value.equals(o)) {
            head = head.next;
            return true;
        } else {
            return result;
        }
    }

    private boolean mystery1(Node<E> p, E o) {
        if (p.next == null) {
            return false;
        }

        boolean result = mystery1(p.next, o);

        if (p.next.value.equals(o)) {
            p.next = p.next.next;
            return true;
        } else {
            return result;
        }
    }
```

Downloaded by Nexus Tech (nexus.w124rd225@gmail.com)

A. Let **list** be an instance of the class **SinglyLinkedList** containing the following list: **[A,B,C,A,B,C,A,B,C]**. What will the following code print out? (Note: the method **toString()** of the class **SinglyLinkedList** , not shown here, simply prints the values of the elements in the list).

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

list.add("A");   list.add("B"); list.add("C");
list.add("A");   list.add("B"); list.add("C");
list.add("A");   list.add("B"); list.add("C");

System.out.println(list);

System.out.println(list.mystery1("A"));

System.out.println(list);

System.out.println(list.mystery1("C"));

System.out.println(list);

System.out.println(list.mystery1("A"));

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

Write your answer below. The answer to the first call to **System.out.println(list);** is already provided.

```
[A,B,C,A,B,C,A,B,C]

Solution:

true

[b  c  b  c  b  c]

true

[b  b  b]

false

[b  b  b]
```

Downloaded by Nexus Tech (nexus.w124rd225@gmail.com)

Now consider the following code for the method **mystery2** in the class **SinglyLinkedList**. Make sure to spot the subtle differences between the methods **mystery1** and **mystery2**.

```
  public boolean mystery2(E o) {

      if(o == null) {
          throw new NullPointerException("Null parameter");
      }

      if (head == null) {
          return false;
      }

      boolean result = mystery2(head, o);

      if (head.value.equals(o)) {
          if (result) {
              head = head.next;
          }
          return true;
      } else {
          return result;
      }
  }

  private boolean mystery2(Node<E> p, E o) {
      if (p.next == null) {
          return false;
      }

      boolean result = mystery2(p.next, o);

      if (p.next.value.equals(o)) {
          if (result) {
              p.next = p.next.next;
          }
          return true;
      } else {
          return result;
      }
  }
```

Downloaded by Nexus Tech (nexus.w124rd225@gmail.com)

B. Let **list** be an instance of **SinglyLinkedList** containing the following list: **[A,B,C,A,B,C,A,B,C]**. What will the following code print out?

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

list.add("A");  list.add("B"); list.add("C");
list.add("A");  list.add("B"); list.add("C");
list.add("A");  list.add("B"); list.add("C");

System.out.println(list);

System.out.println(list.mystery2("A"));

System.out.println(list);

System.out.println(list.mystery2("C"));

System.out.println(list);

System.out.println(list.mystery2("A"));

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

Write your answer below. The answer to the first call to **System.out.println(list);** is already provided.

```
[A,B,C,A,B,C,A,B,C]

Solution:

true

[b c b c a b c]

true

[b b a b c]

true

[b b a b c]
```

Downloaded by Nexus Tech (nexus.w124rd225@gmail.com)

(blank space)

Downloaded by Nexus Tech (nexus.w124rd225@gmail.com)