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: Opeyemi Adesina, Sherif Aly, Guy-Vincent Jourdan and Marcel Turcotte

April 2017, duration: 3 hours

Identification

Last name: _____ First name: _____

Student #: _____ Seat #: _____ Signature: _____ Section: A or B or C or D

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	45	
3	20	
4	10	
Total	**90**	

Question 1 (15 marks)

A. Consider the class declaration below:

```
public class A extends B implements C {
        ...
}
```

Which ones of the following statements are valid, and which ones are invalid?

(a)

```
A var = new B();
```

Valid or **Invalid**

(b)

```
A var = new C();
```

Valid or **Invalid**

(c)

```
B var = new A();
```

Valid or **Invalid**

(d)

```
B var = new C();
```

Valid or **Invalid**

(e)

```
C var = new A();
```

Valid or **Invalid**

(f)

```
C var = new B();
```

Valid or **Invalid**

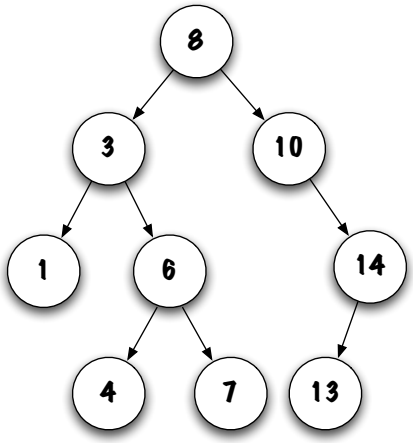
B. An exception is **checked** if it is caught by a try/catch block.
True or **False**

C. Once an exception is caught, the application resumes from the point where the exception was thrown.
True or **False**

D. Adding elements to a **BinarySearchTree** in **increasing** order will produce a **balanced** tree.
True or **False**.

E. The depth of a balanced binary tree of size n is $\lfloor \log_2 n \rfloor$.
True or **False**

F. The **postorder** traversal of the tree below is 1, 4, 7, 6, 3, 13, 14, 10 and 8.
True or **False**



G. A **deque** (double-ended queue) is an abstract data type similar to a queue, but where elements can be added and removed both from the front and the rear of the queue. Much like queues and stacks, deques can be implemented using arrays, circular arrays, singly or doubly linked lists, etc. Consider the following implementations:

- **ArrayDeque**: 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.
- **CircularArrayDeque**: 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.
- **LinkedDeque**: a (singly) linked list implementation, which has an instance reference variable to the current head Node of the deque.
- **DoublyLinkedDeque**: a doubly linked list implementation, which has an instance reference variable to the current head Node of the deque and an instance reference variable to the current tail Node of the deque, and doubly linked Nodes.

For each of the four 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.

	ArrayDeque)	**CircularArrayDeque**	**LinkedDeque**	**DoublyLinkedDeque**
void addFront(E elem)	Slow	Slow	Fast	Fast
void addRear(E elem)	Slow	Slow	Slow	Fast
E removeFront()	Slow	Fast	Fast	Fast
E removeRear()	Fast	Fast	Slow	Fast

Question 2 (45 marks)

Recall that the interfaces **Stack** and **Queue** are respectively defined as follows:

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

```
public interface Queue<E> {
    boolean isEmpty();
    void enqueue(E elem);
    E dequeue();
}
```

Assume that you have been provided a working implementation of the interface **Stack** called **StackImplementation**, and a working implementation of the interface **Queue** called **QueueImplementation**. In the following, you can use instances of both classes as you need, **but no other storage mechanisms**.

Our goal is to create a class **StacksAndQueues**, which will implement several **class** methods to do some basic manipulations of stacks and queues. To simplify the problem, we are going to assume that StacksAndQueues handles exclusively Stacks and Queues **of Strings**.

- In the following, you are asked to provide the code for the methods.
- For all of your methods, make sure to handle all exceptional situations appropriately.

The following example illustrates the use of the class **StacksAndQueues**:

```
Queue<String> queue;
queue = new QueueImplementation<String>();

queue.enqueue("a");
queue.enqueue("b");
queue.enqueue("c");
queue.enqueue("d");
queue.enqueue("e");

System.out.println(queue);

StacksAndQueues.reverseQueue(queue);

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

Executing the above Java program produces the following output.

```
(front) -> [a, b, c, d, e] <- (rear)
(front) -> [e, d, c, b, a] <- (rear)
```

Question 2.1 ReverseQueue

The method **reverseQueue** is a **class** method of **StacksAndQueues** that takes a **Queue of String instances** as input parameter. After calling that method, the elements in the queue should be reversed.

For example, if the following queue is passed to **reverseQueue**:

```
(front) -> [a, b, c, d, e] <- (rear)
```

after the call, the queue should contain

```
(front) -> [e, d, c, b, a] <- (rear)
```

Provide your implementation of **reverseQueue** in the box below.

```
public class StacksAndQueues {

    // MODEL

    public static void reverseQueue(Queue<String> queue) {

        if (queue == null) {

            throw new NullPointerException("Empty queue"); //

        }
        // Stack<String> tmp = new StackImplementation<String>
        Stack<String> stack tmp = new StackImplementation<String>();

        while ( !queue.isEmpty() ) {

            tmp.push(queue.dequeue());

        }

        while ( !tmp.isEmpty() ) {

            queue.enqueue(tmp.pop());

        }

    }
```

Question 2.2 ReverseStack

The method **reverseStack** is a **class** method of **StacksAndQueues** that takes a **Stack of String instances** as input parameter. After calling that method, the elements in the stack should be reversed.

For example, if the following stack is passed to **reverseStack**:

```
(top) -> [a, b, c, d, e] <- (bottom)
```

after the call, the stack should contain

```
(top) -> [e, d, c, b, a] <- (bottom)
```

Provide your implementation of **reverseStack** in the box below

```
// MODEL

public static void reverseStack(Stack<String> stack) {

    if (stack == null) {

        throw new NullPointerException("Empty stack");

    }

    Queue < String > tmp = new QueueImplementation < String > ();

    while (!stack.isEmpty()){

        tmp.enqueue(stack.pop());

    }

    while(!tmp.isEmpty()){

        stack.push(tmp.dequeue();

    }

}
```

Question 2.3 RemoveAll (first method)

The first method **removeAll** is a **class** method of **StacksAndQueues** that takes two input parameters: a **Queue of String instances** and a **String instance**. After calling that method, every occurrence of that string has been removed from the queue. If that string was not in the queue, then the queue is left unchanged.

For example, if the following queue is passed to **removeAll**

```
(front) -> [a, b, c, a, b, c, a, b, c] <- (rear)
```

and the second parameter is the string "a", after the call, the queue should contain

```
(front) -> [b, c, b, c, b, c] <- (rear)
```

Provide your implementation of **removeAll** in the box below

```
// MODEL
public static void removeAll(Queue<String> queue, String toRemove) {
if ( queue == null || toRemove == null ) {
        throw new NullPointerException("null parameter");
}
Queue < String > tmp = newQueueImplementation < String > ();
while ( ! queue.isEmpty() ) {
        String s = queue.dequeue();
        if ( ! s.equals(toRemove) ) {
            tmp.enqueue(s);
        }
    }
    while ( ! tmp.isEmpty() ) {
        queue.enqueue(tmp.dequeue());
    }
}
```

Question 2.4 RemoveAll (second method)

The second method **removeAll** is a **class** method of **StacksAndQueues** that takes two input parameters: a **Stack of String instances** and a **String instance**. After calling that method, every occurrence of that string has been removed from the stack. If that string was not in the stack, then the stack is left unchanged.

For example, if the following stack is passed to **removeAll**

```
(top) -> [a, b, c, a, b, c, a, b, c] <- (bottom)
```

and the second parameter is the string "a", after the call, the stack should contain

```
(top) -> [b, c, b, c, b, c] <- (bottom)
```

Provide your implementation of **removeAll** in the box below

```
// MODEL
public static void removeAll(Stack<String> stack, String toRemove) {
    if ( stack == null || toRemove == null ) {
        throw new NullPointerException("null parameter");
    }
    Stack<String> tmp = StackImplementation<String>();
    while ( !stack.isEmpty() ) {
        String s = stack.pop();
        if ( !s.equals(toRemove) ) {
            tmp.push(s)
        }
    }
    while ( !tmp.isEmpty() ) {
        stack.push(tmp.pop());
    }
}
```

Question 2.5 RemoveFirst (first method)

The first method **removeFirst** is a **class** method of **StacksAndQueues** that takes two input parameters: a **Queue of String instances** and a **String instance**. After calling that method, **the first** occurrence of that string has been removed from the queue. If that string was not in the queue, then the queue is left unchanged.

For example, if the following queue is passed to **removeFirst**

```
(front) -> [a, b, c, a, b, c, a, b, c] <- (rear)
```

and the second parameter is the string "a", after the call, the queue should contain

```
(front) -> [b, c, a, b, c, a, b, c] <- (rear)
```

If the second parameter had been the string "b", after the call, the queue should have contained

```
(front) -> [a, c, a, b, c, a, b, c] <- (rear)
```

Provide your implementation of **removeFirst** in the box below

```
// MODEL
public static void removeFirst(Queue<String> queue, String toRemove) {
    if ( queue == null || toRemove == null ) {
        throw new NullPointerException("null parameter");
    }
    Queue < String > tmp = new QueueImplementation < String > ();
    boolean remove = true;
    while ( ! queue.isEmpty() ) {
        String s = queue.dequeue();
        if ( !s.equals(toRemove) || !remove ) {
            tmp.enqueue(s)
        } else {
            remove = false;
        }
    }
    while ( ! tmp.isEmpty() ) {
        queue.enqueue(tmp.dequeue());
    }
}
```

Question 2.6 RemoveFirst (second method)

The second method **removeFirst** is a **class** method of **StacksAndQueues** that takes two input parameters: a **Stack of String instances** and a **String instance**. After calling that method, **the first** occurrence of that string has been removed from the stack. If that string was not in the stack, then the stack is left unchanged.

For example, if the following stack is passed to **removeFirst**

```
(top) -> [a, b, c, a, b, c, a, b, c] <- (bottom)
```

and the second parameter is the string "a", after the call, the stack should contain

```
(top) -> [b, c, a, b, c, a, b, c] <- (bottom)
```

If the second parameter had been the string "b", after the call, the stack should have contained

```
(top) -> [a, c, a, b, c, a, b, c ]<- (bottom)
```

Provide your implementation of **removeFirst** in the box below

```
// MODEL

public static void removeFirst(Stack<String> stack, String toRemove) {

    if ( stack == null || toRemove == null ) {

        throw new NullPointerException("null parameter");

    }

    Stack < String > tmp = new StackImplementation < String > ();

    boolean remove = true;

    while ( remove && ! stack.isEmpty() ) {
        Strings = stack.pop();

        if ( ! s.equals(toRemove) ) {

            tmp.push(s);

        } else {

            remove = false;

        }

    }

    while ( ! tmp.isEmpty() ) {

        stack.push(tmp.pop());

    }

}
```

Question 2.7 RemoveLast (first method)

The first method **removeLast** is a **class** method of **StacksAndQueues** that takes two input parameters: a **Queue of String instances** and a **String instance**. After calling that method, **the last** occurrence of that string has been removed from the queue. If that string was not in the queue, then the queue is left unchanged.

For example, if the following queue is passed to **removeLast**

```
(front) -> [a, b, c, a, b, c, a, b, c] <- (rear)
```

and the second parameter is the string "a", after the call, the queue should contain

```
(front) -> [a, b, c, a, b, c, b, c] <- (rear)
```

Provide your implementation of **removeLast** in the box below

```
   // MODELS

   public static void removeLast(Queue<String> queue, String toRemove){

         reverseQueue(queue);
         removeFirst(queue,toRemove);
         reverseQueue(queue);

   }

   public static void removeLast(Queue<String> queue, String toRemove) {
        if (queue == null || toRemove == null) {
            throw new NullPointerException();
        }
        QueueImplementation<String> tmp;
        tmp = new QueueImplementation<String>();
        int count = 0;
        while (! queue.isEmpty()) {
            String s;
            s = queue.dequeue();
            if (s.equals(toRemove)) {
                count++;
            }
            tmp.enqueue(s);
        }
        while (! tmp.isEmpty()) {
            String s;
            s = tmp.dequeue();
            if (s.equals(toRemove)) {
                if (count > 1) {
                    queue.enqueue(s);
                    count--;
                }
            } else {
                queue.enqueue(s);
            }
        }
    }
}
```

Question 2.8 RemoveLast (second method)

The second method **removeLast** is a **class** method of **StacksAndQueues** that takes two input parameters: a **Stack of String instances** and a **String instance**. After calling that method, **the last** occurrence of that string has been removed from the stack. If that string was not in the stack, then the stack is left unchanged.

For example, if the following stack is passed to **removeFirst**

```
(top) -> [a, b, c, a, b, c, a, b, c] <- (bottom)
```

and the second parameter is the string "a", after the call, the stack should contain

```
(top) -> [a, b, c, a, b, c, b, c]<- (bottom)
```

Provide your implementation of **removeLast** in the box below

```
    // MODELS

    public static void removeLast(Stack<String> stack, String toRemove) {
        reverseStack(stack);
        removeFirst(stack,toRemove);
        reverseStack(stack);
    }


    public static void removeLast(Stack<String> stack, String toRemove) {
        if (stack == null || toRemove == null) {
            throw new NullPointerException();
        }
        StackImplementation<String> tmp;
        tmp = new StackImplementation<String>();
        while (! stack.isEmpty()) {
            tmp.push(stack.pop());
        }
        boolean removed = false;
        while (! tmp.isEmpty()) {
            String s;
            s = tmp.pop();

            if (! removed && s.equals(toRemove)) {
                removed = true;
            } else {
                stack.push(s);
            }
        }
    }

}
```

Question 3 (20 marks)

You have been provided with a working implementation of a doubly linked list. It 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 uses doubly linked Nodes. The relevant part of that implementation is shown below:

```
public class DoublyLinkedList<T> implements List<T> {

    private static class Node<E> {

        private E value;
        private Node<E> previous;
        private Node<E> next;

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

    }

    private Node<T> head;
    private Node<T> tail;

    public boolean isEmpty(){
        return head == null;
   }
    ...
}
```

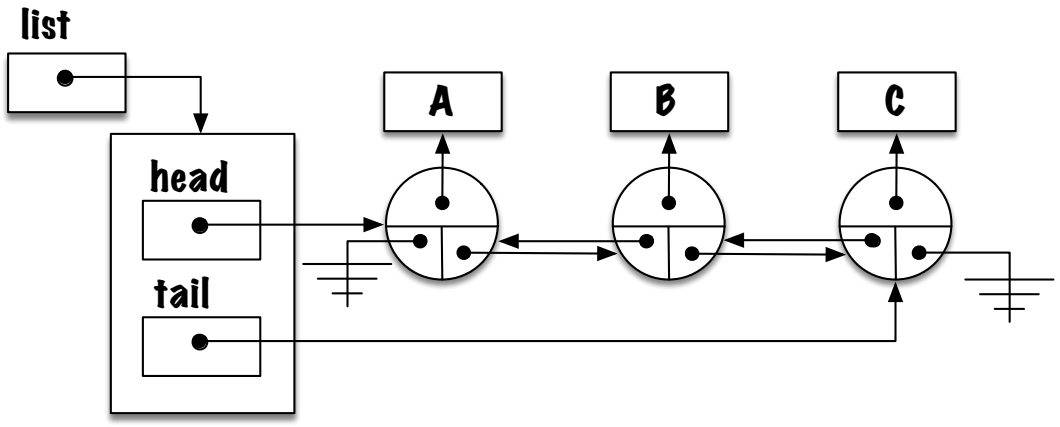
We want to add a **two-way, looping iterator** to that implementation. By ”two-way”, we mean that the iterator can go either **forward** or **backward** , thanks to two methods, **next** (which moves the iterator forward and returns the next value) and **prev** (which moves the iterator backwards and returns the previous value). By “looping”, we mean that when the iterator reaches one end of the list, it continues from the other end of the list.

The interface **Iterator** is defined as follows:

```
public interface Iterator<E> {
    E next ();
    boolean hasNext ();
    E prev ();
    boolean hasPrev ();
}
```

The method **hasNext** (respectively **hasPrev**) returns true if and only if the next call to **next** (respectively to **prev**) will return an element (of type **E**).

Assume that “list” is a reference variable of type “List” of “Strings”, which contains the values “[A, B, C]”.



The following code illustrates the use of our iterator: it declares a reference variable “iterator” of type **Iterator** on “list”, it then calls the method **iterator()** on the list, and assigns the returned reference to the variable “iterator”. The iterator is then moved forward 5 times (it loops back to the front on the fourth move) and backward 3 times (it loops back to the rear on the second move backward):

```
//``list'' is a reference variable of type ``List<String>'',
// that contains the values ``[A, B, C]''.

Iterator<String> iterator;
iterator = list.iterator();

System.out.println(iterator.next()); // prints A
System.out.println(iterator.next()); // prints B
System.out.println(iterator.next()); // prints C
System.out.println(iterator.next()); // loops back to the front and prints A
System.out.println(iterator.next()); // prints B
System.out.println(iterator.prev()); // prints A
System.out.println(iterator.prev()); // loops back to the rear and prints C
System.out.println(iterator.prev()); // prints B
```

Our goal is to provide the implementation for the method **iterator()**, as well as all the necessary code required for it to work[1].

Provide all the code in the box that follows.

[1] Our implementation should be correct. In particular, it should be such that several iterators can be used concurrently.

```
// this code is added to the class DoublyLinkedList<T>

public Iterator<T> iterator(){
    // your code for method Iterator<T> goes here

    return new DoublyLinkedListIterator();

}

private class DoublyLinkedListIterator implements Iterator<T> {

    private Node<T> currentIterator = null;

    public  T next () {
        if (isEmpty()) {
            throw new IllegalStateException("Empty list");
        }
        if (currentIterator == null || currentIterator.next == null) {
            currentIterator = head;
        } else {
            currentIterator = currentIterator.next;
        }
        return currentIterator.value;
    }

    public  boolean hasNext () {
        return !isEmpty();
    }

    public  T prev () {
        if (isEmpty()) {
            throw new IllegalStateException("Empty list");
        }
        if (currentIterator == null || currentIterator.previous == null) {
            currentIterator = tail;
        } else {
            currentIterator = currentIterator.previous;
        }
        return currentIterator.value;

    }

    public  boolean hasPrev() {
        return !isEmpty();
    }
}
```

Question 4 (10 marks)

Consider the class **SinglyLinkedList** outlined below

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

     private static class Node {
        private Boolean value;
        private Node next;
        private Node( Boolean value, Node next ) {
            this.value = value;
            this.next = next;
        }
    }

    private Node head;

    public boolean isEmpty(){
        return head == null;
    }

    // class continues after that
```

We want to add the method **and()** to the class SinglyLinkedList. The method **and()** is a **recursive** method that returns true if and only if **all** the elements of the list instance are **true**. The behaviour of the method is undefined if the list is empty.

For example, the code below prints **true** then **false**

```
SinglyLinkedList test = new SinglyLinkedList();

test.add(true);
test.add(true);
test.add(true);
System.out.println(test.and()); // prints "true"
test.add(false);
test.add(true);
System.out.println(test.and()); // prints "false"
```

Provide the recursive implementation of the method in the following box.

Warning: make sure that your implementation is as efficient as possible. Two marks will be removed for an inefficient implementation!

```
    public boolean and() {
        if(isEmpty()) {
            throw new IllegalStateException("Empty List");
        }
        return and(head);
    }

    private boolean and(Node p)
        if (p.next == null) {
            return p.value;
        } else if (p.value == false) {
            return false;
        } else {
            return and(p.next);
        }
    }
```