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 (ITI 1121)
Final Examination

Instructors: Sherif G. Aly, Nathalie Japkowicz, and Marcel Turcotte

April 2015, 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	10	
2	20	
3	15	
4	10	
5	10	
Total	**65**	

Question 1 (10 marks)

A. **True** or **False**. The following Java code will not compile.

```java
public class Person {
    private String name;
    private int age;
}
```

```java
public class Child extends Person {
    private int grade;
    public Child(String name, int age, int grade) {
        this.name = name;
        this.age = age;
        this.grade = grade;
    }
}
```

B. **True** or **False**. The following Java code will not compile.

```java
public class Cell<E> {

    private E value;

    public Cell(E value) {
        if (value == null) {
            throw new NullPointerException("illegal value");
        }
        this.value = value;
    }

    public boolean isEqual(Cell<E> other) {
        if (other == null) {
            return false;
        } else {
            return value.equals(other.value);
        }
    }
}
```

C. Consider the following array-based implementation of a stack:

```
public class ArrayStack<E> implements Stack<E> {

    private E[] elems;
    private int top;

    public ArrayStack(int capacity) {
        elems = (E[]) new Object[capacity];
        top = 0;
    }

    public E pop() {
        if (top == 0) {
            throw new java.util.NoSuchElementException();
        }
        E saved;
        top = top - 1;
        saved = elems[top];
        elems[top] = null;
        return saved;
    }
}
```

True or **False**. The underlined code is needed to guard against memory leaks.

D. Values are added to a binary search tree in the following order: **25**, **3**, **37**, **1**, **19**, **48**.

True or **False**. The pre-order traversal of that tree will print: **1**, **3**, **19**, **25**, **37**, **48**, while its post-order traversal will print: **48**, **37**, **25**, **19**, **3**, **1**.

E. Consider the following program:

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

        Queue<Integer> q;
        q = new LinkedQueue();

        Stack<Integer> s;
        s = new LinkedStack();

        q.enqueue(1);
        s.push(1);

        int x = 0;

        for (int i = 1; i < 5; i++) {
            x = q.dequeue() + s.pop();
            s.push(x);
            while (x > 0) {
                q.enqueue(x);
                x = x - 1;
            }
        }

        int count;

        count = 0;
        while (!q.isEmpty()) {
            q.dequeue();
            count++;
        }

        System.out.println("Size of the queue is " + count);

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

        System.out.println("Size of the stack is " + count);
    }
}
```

What does the above program print?

(a) Size of the queue is 4; Size of the stack is 1.

(b) Size of the queue is 7; Size of the stack is 5.

(c) Size of the queue is 17; Size of the stack is 1.

(d) Size of the queue is 21; Size of the stack is 5.

(e) Size of the queue is 28; Size of the stack is 1.

F. Given the following class definition:

```
public class SinglyLinkedList<E> {

    private static class Node<T> {

        private final T value;
        private Node<T> next;

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

    public void addFirst(E elem) {
        head = new Node<E>(elem, head);
    }

    public void modify() {
        Node<E> p, q;
        p = head;
        q = p.next.next;
        q.next = p.next;
        for (int i = 0; i < 6; i++) {
            System.out.print(p.value);
            p = p.next;
        }
        System.out.println();
    }

    public static void main(String[] args) {
        SinglyLinkedList<String> l;
        l = new SinglyLinkedList();
        l.addFirst("D");
        l.addFirst("C");
        l.addFirst("B");
        l.addFirst("A");
        l.modify();
    }
}
```

What does the above program print?

(a) ABCD and then terminates abruptly with NullPointerException

(b) ABCBCB

(c) ABCDAB

(d) ABCDBC

(e) DCBCBC

G. Consider the following class definition:

```
public class SinglyLinkedList {

    public static class Node {

        public int value;
        public Node next;

        public Node(int value, Node next) {
            this.value = value;
            this.next = next;
        }
    }

    public Node first;

    public void addFirst(int elem) {
        first = new Node(elem, first);
    }

    public int mystery() {
        if (first == null) {
            return 0;
        }
        return mystery(first, 0);
    }

    private int mystery(Node p, int n) {
        int result;
        if (p.next == null) {
            result = n + p.value;
        } else {
            result = mystery(p.next, n + p.value);
        }
        return result;
    }

    public static void main(String[] args) {

        SinglyLinkedList l;
        l = new SinglyLinkedList();

        l.addFirst(3);
        l.addFirst(2);
        l.addFirst(1);

        System.out.println(l.mystery());
    }
}
```

What does the above program print?

Question 2 (20 marks)

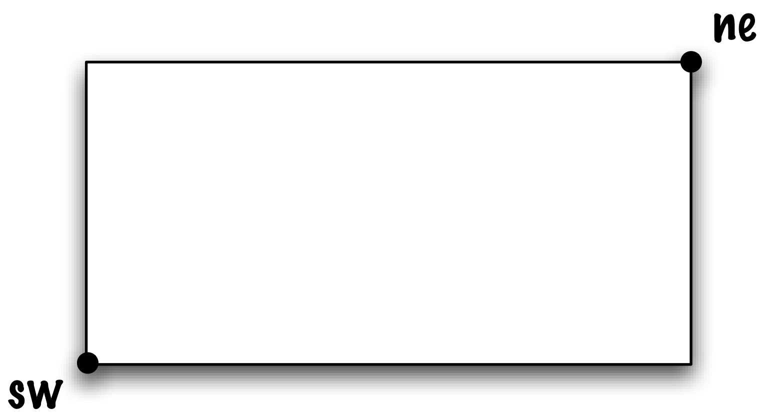
Write a class named **GeoCoordinate** that retains the longitude and latitude of an object on earth. Both longitude and latitude are represented as **double** values that must be in the range [-90.0, 90.0].

The class has the following constructors:

- **GeoCoordinate()**, which initializes the latitude to 0.0 (the equator), the longitude to 0.0 (the prime meridian).
- **GeoCoordinate(double longitude, double latitude)**: initializes the longitude and latitudes to values in [-90.0, 90.0].

The class has the following methods:

- **double getLongitude()** and **double getLatitude()**, that return the longitude and latitude of the GeoCoordinate.
- **boolean equals(GeoCoordinate g)** that returns **true** if the GeoCoordinate of **g** and the current object represent the same GeoCoordinate, and **false** otherwise.
- **boolean isWithinBounds(GeoCoordinate sw, GeoCoordinate ne)** that takes exactly two GeoCoordinates representing the south-west and north-est vertices of a rectangular area (see Figure below), and returns **true** if the current coordinate is within the bounds of the area, and **false** otherwise. (This can be used to check whether a given vehicle for example has gone outside of a certain geographic area).



Finally, the execution of the test program below, **GeoCoordinateTest**, should produce the following result:

```
(20.0,30.0)
false
true
caught IllegalArgumentException: Wrong argument(s) upon construction of GeoCoordinate
```

```
public class GeoCoordinateTest {

    public static void main(String[] args) {

        GeoCoordinate currentLocation;
        GeoCoordinate sw, ne;

        currentLocation = new GeoCoordinate(20.0, 30.0);

        sw = new GeoCoordinate(10.0, 10.0);
        ne = new GeoCoordinate(30.0, 30.0);

        System.out.println(currentLocation);
        System.out.println(currentLocation.equals(sw));
        System.out.println(currentLocation.isWithinBounds(sw, ne));

        try {
            GeoCoordinate wrongCoordinate = new GeoCoordinate(-100.0, 20.0);
        } catch (IllegalArgumentException e) {
            System.out.println("caught IllegalArgumentException: " + e.getMessage());
        }

    }

}
```

(Question 2 continued)

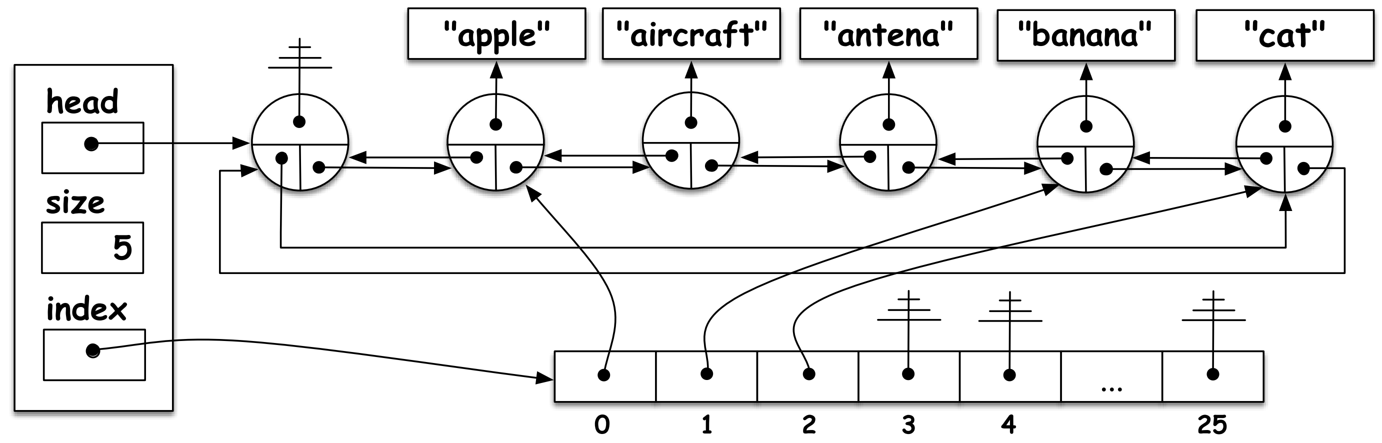
(Question 2 continued)

Question 3 (15 marks)

A doubly linked list that specifically stores strings is modified to enhance its efficiency. The linked list will now be an "indexed" doubly linked list so that the location of strings starting with the letters of the alphabet 'a', 'b', 'c', ... 'z' are easily located (lower case letters only for this question).

All strings starting with a given letter are stored (together) in sequence in the linked list. For example, "apple", "aircraft", "antenna", "banana", "ball", "baseball", "balloon", "cat", "caramel", "category", etc. is an example of some strings that can be stored in such sequence in the linked list.

To achieve this kind of indexing, a one dimensional array called "index" of size 26 is used. Each element in the array points to the starting location in the linked list that holds the corresponding character. For example, index[0] points to the first node in the linked list that has strings that start with the letter 'a', and so on. A **null** value indicates there are no strings of this kind in the list.



The diagram above shows the memory representation after inserting the first 5 elements. The execution of the test program below must produce the following result:

```
{apple,aircraft,antena,banana,cat}
{apple,aircraft,antena,banana,ball,baseball,balloon,cat,caramel,case}
{apple,aircraft,antena,banana,ball,baseball,balloon,caramel,case}
{apple,aircraft,antena,ball,baseball,balloon,caramel,case}
```

```
IndexedLinkedList l;

l = new IndexedLinkedList();

l.add("apple"); l.add("aircraft"); l.add("banana"); l.add("antena"); l.add("cat");

System.out.println(l);

l.add("ball"); l.add("baseball"); l.add("caramel"); l.add("balloon"); l.add("case");

System.out.println(l);

l.delete('c');

System.out.println(l);

l.delete('b');

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

Complete the implementation of the class **IndexLinkedList** on the next pages.

A. Complete the implementation of the constructor below. (3 marks)

```
public class IndexedLinkedList {

    private final String ALPHA = "abcdefghijklmnopqrstuvwxyz";

    private static class Node {

        private final String value;
        private Node previous;
        private Node next;

        private Node(String value, Node previous, Node next) {
            this.value = value;
            this.previous = previous;
            this.next = next;
        }
    }

    private final Node head;

    private final Node[] index;

    private int size;

    public IndexedLinkedList() {


        head = ________________________________________;


        ______________________________;


        ______________________________;



        index = ________________________________________;


        size = 0;
    }

    // Continues on the next page...
```

B. Write a method **boolean delete(char c)** that deletes the first occurring node whose strings starts with the character **c**. The method returns **true** for a successful deletion, and **false** if nothing was found to delete. (12 marks)

```
    public boolean delete(char c) {

        int position;

        position = ________________________________________________;

        Node toRemoved;

        toRemoved = ______________________________;

        if (____________________________________) {

            Node before, after;

            before = _______________________________;

            after = _______________________________;

            _____________________________ = _________________________;

            _____________________________ = _________________________;

            if (___________________________ && __________________________________________) {

                index[position] = after;

            } else {

                index[position] = ________________________;

            }

            size--;
        }

        return toRemoved != null;
    }

} // IndexedLinkedList
```

Question 4 (10 marks)

Complete the implementation of the static method **int remove(Queue<E> q, E e, int n)**, which removes the first **n** occurrences of **e** in **q**. The method must work for any implementation of the interface **Queue**:

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

- Following a call to the method **remove**, the elements in the queue must remain in the same order, except that the first **n** occurrences of **e** have been removed.
- The method throws **NullPointerException** if either **q** or **e** are **null**. It throws **IllegalArgumentException** if **n** is a negative number.
- If the queue had less than **n** occurrences of **e**, the returned value represents the excess of elements that could not be removed. Consider the example below.
- Since you do not know the size of the queue, you cannot use an array for temporary storage. Instead, you must either use a queue or a stack (or both). You can assume the existence of the class **LinkedQueue**, which implements the interface **Queue**, as well as **LinkedStack**, which implements the interface **Stack**.

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

Executing the test program below produces the following output:

```
LinkedQueue: {A,B,R,A,C,A,D,A,B,R,A}
0
LinkedQueue: {B,R,C,D,A,B,R,A}
2
LinkedQueue: {B,R,D,A,B,R,A}
```

```
Queue<String> q;
q = new LinkedQueue<String>();

q.enqueue("A");
q.enqueue("B");
q.enqueue("R");
q.enqueue("A");
q.enqueue("C");
q.enqueue("A");
q.enqueue("D");
q.enqueue("A");
q.enqueue("B");
q.enqueue("R");
q.enqueue("A");

System.out.println(q);
System.out.println(remove(q,"A",3));
System.out.println(q);
System.out.println(remove(q,"C",3));
System.out.println(q);
```

```
public class Remove {
    public static <E> int remove(Queue<E> q, E e, int n) {














    } // End of remove
} // End of Remove
```

Question 5 (10 marks)

A **binary search tree** is a flexible and efficient data structure. However, as we discussed in class, the structure of the tree depends on the order in which the elements are inserted into the tree. In the worse cases, the binary search tree is no more efficient than linked lists. A **left linear** tree is such degenerated case.

Definition: A binary search tree is **left linear** if it contains at least one node and all the nodes of the tree have no right child.

A. Give (draw) an example of a **left linear** binary search tree having exactly 4 nodes.

B. Implement the instance method **boolean isLeftLinear()** that returns **true** if the instance is left linear, and **false** otherwise.

```
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 = null;
```

(Question 5 continued)

```
} // End of BinarySearchTree
```

A String and characters

Characteristics of the class **String** that might be useful.

- **char charAt(int index)** returns the char value at the specified index.
- **int indexOf(int ch)** returns the index within this string of the first occurrence of the specified character.
- **int length()** returns the length of this string.

Useful fact about characters.

- You can get the ASCII value of a character using a type cast to int.
- The execution of `System.out.println( (int) 'a' );` displays 97.
- The execution of `System.out.println( (int) 'b' );` displays 98.

B Stack

```
/**
 * Stack Abstract Data Type. A Stack is a linear data structure
 * following last-in-first-out protocol, i.e. the last element
 * that has been added onto the Stack, is the first one to
 * be removed.
 *
 * @param <E> the type of elements in this stack
 */

public interface Stack<E> {

    /**
     * Tests if this Stack is empty.
     *
     * @return true if this Stack is empty; and false otherwise.
     */

    boolean isEmpty();

    /**
     * Returns a reference to the top element; does not change
     * the state of this Stack.
     *
     * @return The top element of this stack without removing it.
     */

    abstract E peek();

    /**
     * Removes and returns the element at the top of this stack.
     *
     * @return The top element of this stack.
     */

    abstract E pop();

    /**
     * Puts an element onto the top of this stack.
     *
     * @param element the element be put onto the top of this stack.
     */

    void push( E element );

}
```

C Queue

```
/**
 * Queue Abstract Data Type. A Queue is a linear data structure
 * following first-in-first-out protocol, i.e. the first element that
 * has been added to the Queue, is the first one to be removed.
 *
 * @param <E> the type of elements in this queue
 */

public interface Queue<E> {

        /**
         * Tests if this Queue is empty.
         *
         * @return true if this Queue is empty; and false otherwise.
         */

        boolean isEmpty();

        /**
         * Removes and returns the front element of the Queue.
         *
         * @return the front element of the Queue.
         */

        E dequeue();

        /**
         * Puts an element at the rear of this Queue.
         *
         * @param element the element be put at the rear of this Queue.
         */

        void enqueue( E element );

}
```