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 à l'Informatique II (ITI 1521)
EXAMEN FINAL

Professeurs: Sherif G. Aly, Nathalie Japkowicz, et Marcel Turcotte

Avril 2015, durée: 3 heures

Identification

Nom de famille : _____ Prénom : _____

d'Étudiant : _____ Siège # : _____ Signature : _____

Instructions

1. Examen à livres fermés.
2. L'utilisation de calculatrices, d'appareils électroniques ou tout autre dispositif de communication est interdit.
   (a) Tout appareil doit être éteint et rangé.
   (b) Toute personne qui refuse de se conformer à ces règles pourrait être accusée de fraude scolaire.
3. Répondez aux questions sur ce questionnaire.
   (a) Utilisez le verso des pages si nécessaire.
   (b) Aucune page supplémentaire n'est permise.
4. Ne retirez pas l'agrafe du livret d'examen.
5. Écrivez vos commentaires et hypothèses afin d'obtenir des points partiels.
6. Écrivez lisiblement, puisque votre note en dépend.
7. Attendez l'annonce de début de l'examen.

Barème

Question	Maximum	Résultat
1	10	
2	20	
3	15	
4	10	
5	10	
Total	**65**	

Question 1 (10 points)

A. **Vrai** ou **Faux**. Le code de Java suivant ne va pas compiler.

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

```
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. **Vrai** ou **Faux**. Le code de Java suivant ne va pas compiler.

```
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. Veuillez considérer l'implémentation de piles basée sur un tableau qui se trouve ci-dessous :

```
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;
    }
}
```

Vrai ou **Faux**. Le code souligné ci-dessus est nécessaire afin de se parer contre les fuites de mémoire.

D. Des valeurs sont ajoutées à un arbre binaire de recherche dans l'ordre suivant : **25**, **3**, **37**, **1**, **19**, **48**.

Vrai ou **Faux**. La traversée « pre-order » de cet arbre imprimera : **1**, **3**, **19**, **25**, **37**, **48**, alors que sa traversée « post-order » imprimera : **48**, **37**, **25**, **19**, **3**, **1**.

E. Veuillez considérer le programme suivant :

```
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);
    }
}
```

Qu'imprimera le programme ci-dessus ?

(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. Étant donné la définition de classe suivante :

```
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();
    }
}
```

Qu'imprimera le programme ci-dessus ?

(a) ABCD et il se terminera abruptement avec NullPointerException

(b) ABCBCB

(c) ABCDAB

(d) ABCDBC

(e) DCBCBC

G. Veuillez considérer la définition de classe suivante :

```
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());
    }
}
```

Qu'imprimera le programme ci-dessus ?

Question 2 (20 points)

Veuillez écrire une classe nommée **GeoCoordinate** qui emmagasine la longitude et la latitude d'un objet terrestre. Ces deux mesures sont chacune représentées par des valeurs de type **double** qui doivent être comprises dans l'intervalle [-90.0, 90.0].

Voici la description des constructeurs de cette classe :

- **GeoCoordinate()**, qui initialise la latitude à 0.0 (l'équateur) et la longitude à 0.0 (le premier méridien).
- **GeoCoordinate(double longitude, double latitude)**, qui initialise la longitude et la latitude à des valeurs comprises dans l'intervalle [-90.0, 90.0].

La classe contient les méthodes décrites comme suit :

- **double getLongitude()** et **double getLatitude()**, qui retournent la longitude et la latitude de la GeoCoordinate.
- **boolean equals(GeoCoordinate g)** qui retourne **true** si la GeoCoordinate de **g** représente la même GeoCordinate que celle de l'objet courant, et **false** dans tous les autres cas.
- **boolean isWithinBounds(GeoCoordinate sw, GeoCoordinate ne)** qui prend comme entrées exactement deux GeoCoordinates représentant le sud-ouest et le nord-est d'une surface rectangulaire, et retourne **true** si l'objet terrestre courant se trouve dans les limites de cette surface et **false** si ça n'est pas le cas. Veuillez assumer que les vertex de la surface sont toujours donnés dans l'ordre montré sur la figure ci-dessous. (Une application possible de cette méthode est de voir si, par exemple, un véhicule est sorti d'un certain périmètre géographique).

L'exécution du programme test ci-dessous, **GeoCoordinateTest**, doit produire les résultats suivants :

```
(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 continuée)

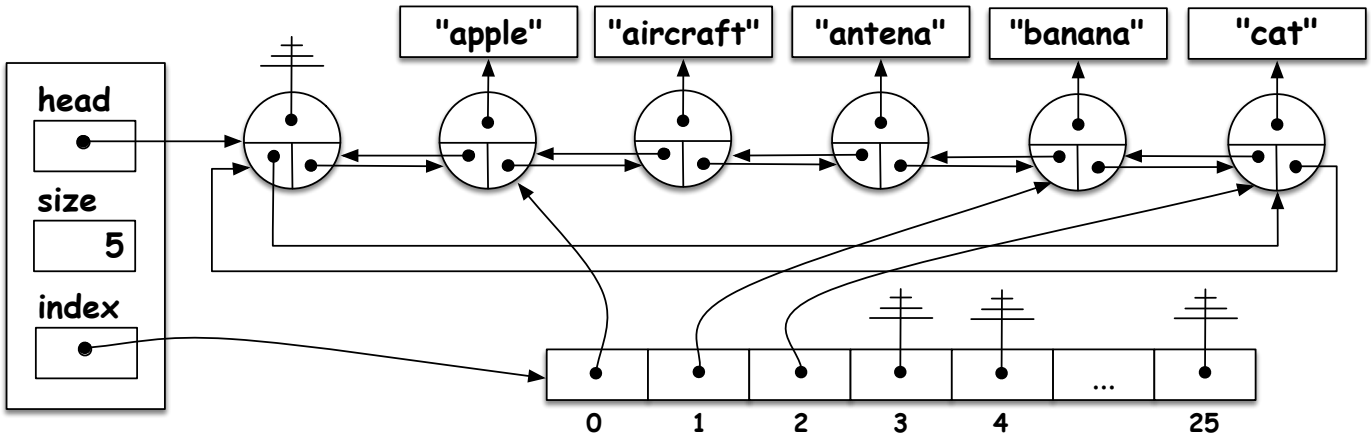
(Question 2 continuée)

Question 3 (15 points)

Une liste doublement chaînée qui emmagasine des chaînes de caractères est modifiée de manière à améliorer son efficacité. La liste chaînée sera maintenant une liste doublement chaînée « indexée » de manière à ce que l'emplacement des chaînes commençant par les lettres de l'alphabet 'a', 'b', 'c', ...'z' soient facilement localisés (dans cette question nous n'utilisons que des lettres minuscules).

Toutes les chaînes de caractères commençant avec une lettre donnée sont emmagasinées (ensembles) en séquence dans la liste doublement chaînée. Par exemple, "apple", "aircraft", "antenna", "banana", "ball", "baseball", "balloon", "cat", "caramel", "category", etc. est un exemple des chaînes qui peuvent être sauvegardées suivant une telle séquence dans la liste.

Afin d'accomplir la création de ce type d'index, un simple tableau unidimensionnel de taille 26 appelé « index » est utilisé. Chaque élément de ce tableau pointe vers l'emplacement dans la liste doublement chaînée où les chaînes de caractères qui commencent par la lettre donnée sont sauvegardées. Par exemple, index[0] pointe vers le premier noeud de la liste doublement chaînée contenant une chaîne de caractères débutant par la lettre 'a', et ainsi de suite. Une valeur **null** indique qu'il n'y a pas de chaînes de caractère de ce type dans la liste.



Le diagramme ci-dessus donne la représentation de la mémoire après avoir ajouté 5 éléments. L'exécution du programme test ci-dessous doit produire les résultats suivants :

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

```
IndexedLinkedList l;
l = new IndexedLinkedList();

l.add("apple"); l.add("aircraft"); l.add("banana"); l.add("antena");
l.add("cat"); l.add("ball"); l.add("baseball"); l.add("caramel");
l.add("balloon"); l.add("category");

System.out.println(l);

l.delete('c');
System.out.println(l);

l.delete('b');
System.out.println(l);
```

Veuillez compléter l'implémentation de la classe **IndexLinkedList** sur les pages qui suivent.

A. Complétez l'implémentation du constructeur ci-dessous. (3 points)

```
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;
    }

    // Continue sur la page suivante...
```

B. Veuillez écrire une méthode **boolean delete(char c)** qui efface le premier noeud apparaissant dans la liste dont la chaîne de caractères commence par le caractère **c**. La méthode retourne **true** dans le cas où l'effacement a pu se faire, et **false** si rien n'a pu être effacé. (12 points)

```
    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 points)

Veuillez compléter l'implémentation de la méthode statique **int remove(Queue<E> q, E e, int n)**, qui retire les **n** premières apparitions de **e** dans la file **q**. La méthode doit fonctionner pour n'importe quelle implémentation de l'interface **Queue** :

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

- Suivant un appel à la méthode **remove**, les éléments de la file doivent rester dans le même ordre à part le fait que les **n** premières apparitions de **e** ont été retirées.
- La méthode lance l'exception **NullPointerException** lorsque **q** ou **e** est **null**. Elle lance l'exception **IllegalArgumentException** lorsque **n** est un nombre négatif.
- Si la file a moins de **n** apparitions de **e**, la valeur retournée représente le nombre d'éléments qui n'ont pas pu être retirés. Veuillez consulter l'exemple ci-dessous.
- Puisque vous ne connaissez pas la taille de la file, vous ne pouvez pas utiliser des tableaux pour sauvegarder temporairement les éléments. À la place de cela, vous devez utiliser soit une file ou une pile (ou les deux). Vous pouvez supposer l'existence de la classe **LinkedQueue**, qui implémente l'interface **Queue**, ainsi que **LinkedStack**, qui implémente l'interface **Stack**.

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

L'exécution du programme test ci-dessous produit la sortie suivante :

```
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 points)

Un **arbre binaire de recherche** est une structure de données flexible et efficace. Cependant, comme discuté en classe, la structure de l'arbre dépend de l'ordre dans lequel les éléments sont insérées dans l'arbre. Dans les pires cas, l'arbre binaire de recherche n'est pas plus efficace qu'une liste chaînée. Un arbre **linéaire à gauche** est l'un de ces cas dégénérés.

Définition : Un arbre binaire de recherche est **linéaire à gauche** s'il contient au moins un noeud et si tous les noeuds de l'arbre n'ont pas d'enfant droit.

A. Veuillez montrer (dessiner) un exemple d'arbre binaire de recherche **linéaire à gauche** ayant exacterment 4 noeuds.

B. Veuillez implémenter la méthode d'instance **boolean isLeftLinear()** qui retourne **true** si l'instance est linéaire à gauche, et **false** dans tous les autres cas.

```
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;
```

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

A Classe String et les caractères

Ces caractéristiques de la classe **String** pourraient être utiles.

- **char charAt(int index)** retourne le caractère se trouvant à la position désignée par l'index.
- **int indexOf(int ch)** retourne l'index de l'occurrence la plus à gauche du caractère spécifié.
- **int length()** retourne la longueur de la chaine.

Information utile sur les caractères.

- On obtient la valeure ASCII d'un caractère à l'aide d'un forçage de type vers int.
- L'exécution de `System.out.println( (int) 'a' );` affiche 97.
- L'exécution de `System.out.println( (int) 'b' );` affiche 98.

B Pile

```
/**
 * 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 File

```
/**
 * 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 );

}
```