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 (ITI1521)
EXAMEN FINAL

Instructeurs: Nour El-Kadri, Guy-Vincent Jourdan, et Marcel Turcotte

Avril 2016, 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	15	
2	15	
3	25	
4	15	
5	10	
Total	**80**	

Question 1 Répondez aux six (6) sous-questions [15 points]

A. Supposez que les déclarations de types ont été faites correctement, que **s** est non null et désigne une pile. La pratique illustrée par le programme ci-dessous devrait être évitée.
Vrai ou **faux**.

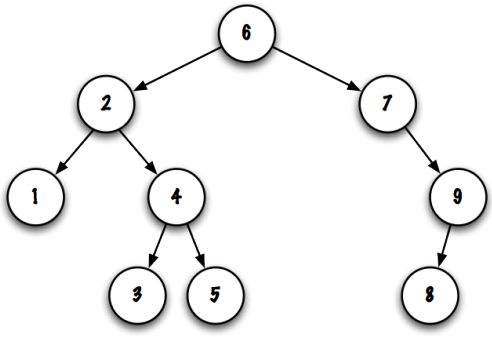
```
boolean done = false;

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

B. Une méthode qui lance une exception à déclaration non obligatoire **doit** la déclarer à l'aide de la clause **throws**, sinon une erreur sera signifiée lors de la compilation.
Vrai ou **faux**.

C. Les bonnes pratiques de la programmation orientée objet veulent que le programmeur attrape **toujours toutes** les exceptions lancées par les appels de méthodes à l'aide d'un bloc **try/catch**.
Vrai ou **faux**.

D. Étant donné deux instances initialement vides de la classe **BinarySearchTree** et la méthode **add** présentée en classe. Ajouter les mêmes éléments, mais dans un ordre différent, produira **toujours** des arbres de la même hauteur.
Vrai ou **faux**.

E. Consultez l'arbre binaire ci-dessous :



Pour chaque ordre des noeuds visités, encerclez le parcours qui visite les noeuds dans cet ordre (préfixe, infixe, postfixe, ou autre).

Ordre des noeuds visités : **6-2-1-4-3-5-7-9-8**	Préfixe	Infixe	Postfixe	Autre
Ordre des noeuds visités : **1-3-5-4-2-8-9-7-6**	Préfixe	Infixe	Postfixe	Autre
Ordre des noeuds visités : **1-2-3-4-5-6-7-8-9**	Préfixe	Infixe	Postfixe	Autre

F. Cette question porte sur quatre (4) implémentations de l'interface **List** :

ArrayList : une implémentation simple à l'aide d'un tableau, ayant une variable d'instance afin de désigner le tableau, ainsi qu'une variable d'instance pour le nombre d'éléments se trouvant dans le tableau (taille logique).

CircularArrayList : une implémentation à l'aide d'un tableau circulaire, ayant une variable d'instance afin de désigner le tableau, ainsi que des variables d'instance pour désigner les positions des éléments avant et arrière dans le tableau.

LinkedList : une implémentation à l'aide de noeuds simplement chaînés, ayant une variable d'instance afin de désigner le premier noeud de la liste.

DoublyLinkedList : une implémentation à l'aide de noeuds doublement chaînés, ayant une variable d'instance pour désigner le premier noeud de la liste, ainsi qu'une variable d'instance afin de désigner le dernier noeud de la liste.

Notez que les deux implémentations à l'aide d'un tableau implémentent un tableau dynamique afin de sauvegarder un nombre arbitrairement grand d'éléments. Cependant, la taille physique du tableau n'est pas réduite automatiquement.

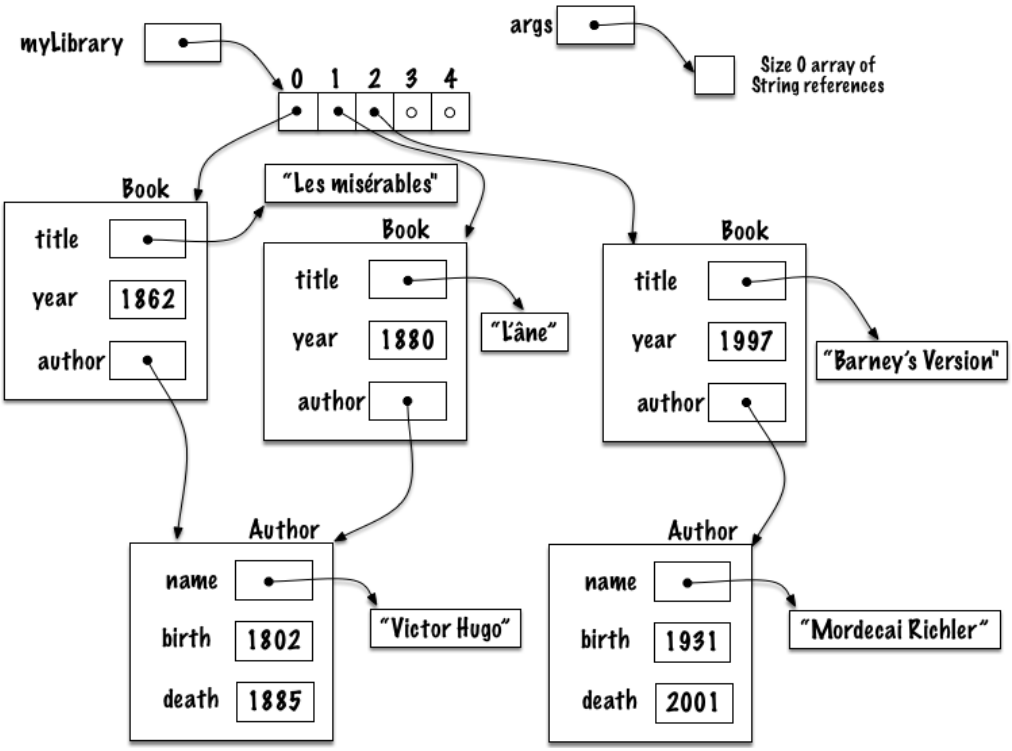
Pour chacune des six méthodes ci-dessous et pour chaque implémentation, indiquez si la méthode est **rapide** (son temps d'exécution est constant) ou **lente** (son temps d'exécution dépend du nombre d'éléments dans la liste).

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

Question 2 [15 points]

Comme vous l'avez fait pour la question 1 du devoir 4, effectuez l'ingénierie inverse du diagramme de mémoire ci-dessous. Vous devez fournir l'implémentation de toutes les classes, variables d'instance, constructeurs, méthodes, et méthode principale, de sorte que l'exécution de votre programme produise le diagramme de mémoire ci-dessous.

Indice : il faut une classe **Book** et une classe **Author**, ainsi qu'une méthode principale.



Question 3 [25 points]

Nous avons vu en classe deux techniques afin d'implémenter l'interface **List** : à l'aide d'un tableau ou à l'aide de noeuds chaînés. Chaque technique a ses forces et ses faiblesses. Dans certains cas, il pourrait être avantageux de passer d'une implémentation à l'autre, selon les besoins de l'application.

Pour cette question, nous proposons une implémentation **hybride**. Cette classe fournira une implémentation à l'aide d'un tableau **et** une implémentation à l'aide de noeuds chaînés, quoique pas en même temps. L'implémentation à l'aide de noeuds chainés utilise des noeuds simplement chaînés, sans noeud factice. L'implémentation à l'aide d'un tableau utilise un simple tableau qui n'est pas circulaire.

Afin de supporter les deux implémentations, notre classe, **HybridList**, aura les variables d'instance suivantes :

- **headNode** désigne le premier noeuds de la liste ou la valeur **null** si la liste est vide.
- **headArray** désigne un tableau.
- **currentSize** est une variable d'un type primitif représentant le nombre d'éléments dans la liste (taille logique).
- **currentCapacity** est une variable d'un type primitif représentant la taille actuelle du tableau (taille physique).
- finalement, la variable **isArray** a la valeur **true** si l'implémentation actuelle utilise le tableau et **false** si les éléments sont sauvegardés dans une liste chaînée.

Voici l'implémentation partielle de la classe.

```
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;
    }

    // ...
```

Donnez l'implémentation des méthodes **public void toArray()** et **public void toLinkedList()**. La méthode **public void toArray()** change la façon dont les éléments sont sauvegardés, du mode où les éléments sont sauvegardés dans une liste chaînée vers le mode où ils sont sauvegardés dans un tableau. La méthode **public void toLinkedList()** fait l'opération inverse, elle change la façon dont sont sauvegardés les éléments, du tableau vers une liste chaînée. Consultez le programme ci-dessous pour un exemple d'utilisation.

- Dans les deux cas, si les données sont déjà sauvegardées dans le bon mode (les données sont dans un tableau lors de l'appel à la méthode **toArray()** ou les données sont dans une liste chaînée lors de l'appel à la méthode **toLinkedList()**), la méthode n'a rien à faire.
- Pour votre implémentation des méthodes **toArray()** et **toLinkedList()**, vous n'avez pas à vous préoccuper de la gestion de la mémoire associée à l'ancien mode lors du passage au nouveau mode de sauvegarde.
- Pour vos implémentations de ces méthodes, vous ne pouvez pas utiliser des appels de méthode de la classe **HybridList**, en conséquence, ces méthodes ne sont pas présentées dans l'implémentation partielle [1]. Votre code source doit manipuler explicitement les variables d'instance **headNode**, **headArray**, **currentSize**, **currentCapacity** et **isArray**.
- Supposez que **currentSize** contient toujours le bon nombre d'éléments. Par contre, ne supposez pas que **currentCapacity** est valide lors d'un appel à la méthode **toArray()**, vous devez vous même vous assurer que le tableau ait une taille adéquate.
- Veuillez noter que les méthodes **toArray()** et **toLinkedList()** ne retournent aucune valeur. Elles ne font que changer la représentation interne des données.

Le programme ci-dessous illustre l'utilisation de la classe **HybridList**. Dans cet exemple, la liste est d'abord en mode liste chaînée. Dans ce mode, des éléments sont ajoutés à la structure de données. Par la suite, la représentation des données passe au mode tableau, on accède alors aux éléments à l'aide de la méthode **get()**.

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

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

        hList.toLinkedList(); // hList is now a linked list

        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));
        }

}
```

Donnez votre implémentation des méthodes **toArray()** et **toLinkedList()** sur les deux pages qui suivent.

1. La méthode **isEmpty()** est présente et vous pouvez l'utiliser

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

    public void toArray() {









    } // End of toArray
```

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

    public void toLinkedList() {









    } // End of toLinkedList
```

Question 4 [15 points]

Veuillez écrire la méthode de classe **roll(Stack<E> s, int n)** qui transforme la pile désignée par **s** tel que les **n** éléments du dessous sont maintenant sur le dessus de la pile. L'ordre relatif des éléments du haut et du bas de la pile doit rester le même.

```
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);
```

L'exécution du programme Java ci-dessous affiche ceci :

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

- Assurez-vous de traiter toutes les situations exceptionnelles de manière appropriée.
- Puisque vous ne connaissez pas la taille de **s**, vous ne pouvez pas utiliser de tableaux pour la sauvegarde temporaire des données. Utilisez plutôt des piles. Supposez l'existence de la classe **LinkedStack**, qui implémente l'interface **Stack**.
- **Stack** est une interface.

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

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









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

Question 5 [10 points]

Consultez la classe **SinglyLinkedList** ci-dessous.

```
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
```

Consultez aussi l'implémentation de la méthode **mystery1**.

```
   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;
        }
    }
```

A. Supposez que la variable **list** désigne une instance de la classe **SinglyLinkedList** contenant les éléments suivants, dans cet ordre : **[A,B,C,A,B,C,A,B,C]**. Donnez le résultat affiché suite à l'exécution du programme ci-dessous. (Supposez que la méthode **toString()**, non montrée ici, affiche simplement les éléments de la liste ; voir l'exemple ci-dessous.)

```
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);
```

Donnez vos réponses dans la boite ci-dessous. Le résultat du premier appel **System.out.println(list)** vous est donné.

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

Consultez maintenant le code source de la méthode **mystery2** de la classe **SinglyLinkedList**. Assurez-vous que vous avez bien identifié la différence subtile entre **mystery1** et **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;
    }
}
```

B. Supposez que la variable **list** désigne une instance de la classe **SinglyLinkedList** contenant les éléments suivants, dans cet ordre : **[A,B,C,A,B,C,A,B,C]**. Donnez le résultat affiché suite à l'exécution du programme ci-dessous.

```
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);
```

Donnez vos réponses dans la boite ci-dessous. Le résultat du premier appel **System.out.println(list)** vous est donné.

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