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: Guy-Vincent Jourdan et Marcel Turcotte

Avril 2019, durée: 3 heures

Identification

Nom de famille : _____ Prénom(s) : _____

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 sur ce questionnaire.
   (a) Utilisez le verso des pages si nécessaire.
   (b) Aucune page supplémentaire n'est permise.
4. Écrivez vos commentaires et hypothèses afin d'obtenir des points partiels.
5. Écrivez lisiblement, puisque votre note en dépend.
6. Ne retirez pas l'agrafe du livret d'examen.
7. Attendez l'annonce de début de l'examen.

Barème

Question	Maximum	Résultat
1	15	
2	15	
3	15	
4	15	
Total	**60**	

Question 1 (15 points)

Implémentez la méthode d'instance **merge** pour la classe **LinkedList** qui a les caractéristiques suivantes.

- La liste commence toujours par un noeud factice, qui sert de marqueur pour le début de la liste. Le noeud factice n'est jamais utilisé pour enregistrer des données. La liste vide est constituée du noeud factice seulement ;
- Dans l'implémentation pour cette question, les noeuds de la liste sont doublement chaînés ;
- Dans cette implémentation, la liste est circulaire, c'est-à-dire que la référence **next** du dernier noeud de la liste pointe vers le noeud factice, la référence **prev** du noeud factice pointe vers le dernier noeud de la liste. Dans la liste vide, le noeud factice est le premier et le dernier noeud de la liste, ses références **prev** et **next** pointent vers le noeud lui-même ;
- Le dernier noeud est facilement accessible, parce que c'est toujours le noeud précédent le noeud factice, l'en-tête de la liste n'a pas de pointeur vers le dernier élément.

La méthode **merge** ajoute tous les éléments de la liste désignée par le paramètre **other** à **cette** liste. Après un appel à la méthode **merge**, la liste a désigné le paramètre **other** est vide.

- Cette question évalue votre compréhension des structures chaînées. Pour cette raison, vous ne pouvez pas utiliser les méthodes de la classe **LinkedList**. En particulier, vous ne pouvez pas utiliser les méthodes **add()** ou **remove()**.
- La méthode ne lance jamais d'exception.

```
public class Test {
    public static void main(String[] args) {
        LinkedList<String> firstList, secondList;
        firstList = new LinkedList<String>();
        secondList = new LinkedList<String>();

        firstList.addLast("alpha");
        firstList.addLast("bravo");
        firstList.addLast("charlie");
        firstList.addLast("delta");

        secondList.addLast("echo");
        secondList.addLast("foxtrot");

        System.out.println(firstList);
        System.out.println(secondList);

        firstList.merge(secondList);

        System.out.println(firstList);
        System.out.println(secondList);
    }
}
```

L'exécution du programme ci-dessus produit ceci sur la console.

```
[alpha, bravo, charlie, delta]
[echo, foxtrot]
[alpha, bravo, charlie, delta, echo, foxtrot]
[]
```

Complétez l'implémentation de la méthode **merge**.

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

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

    private final Node<E> head;
    private int size;

    public LinkedList() {
        head = new Node<E>(null, null, null);
        head.next = head.prev = head;
        size = 0;
    }

    public void merge(LinkedList<E> other) {

        if (_____________________________&&_____________________________) {

            Node<E> thisLast, otherFirst, otherLast;

            thisLast = _____________________________;

            otherFirst = _____________________________;

            otherLast = _____________________________;

            _____________________________ = thisLast;

            _____________________________ = otherFirst;

            _____________________________ = otherLast;

            otherLast.next = _____________________________;

            other.head.next = _____________________________;

            _____________________________ = _____________________________;

            _____________________________ = _____________________________;

            _____________________________ = _____________________________;

        }

    }
}
```

Question 2 (15 points)

Complétez l'implémentation de la méthode **findAndReplace** pour la classe **LinkedList** de la page suivante. La méthode remplace toutes les occurrences de l'élément spécifié par le paramètre **target** par la valeur du paramètre **replacement**. La méthode retourne aussi le nombre total d'éléments remplacés.

Il s'agit d'une méthode récursive. Veuillez noter que la liste est simplement chaînée et qu'il n'y a pas de noeud factice. Pour vous aider à bien comprendre les attentes, voici un programme test.

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

System.out.println(list);

System.out.println(list.findAndReplace("I", "she"));

System.out.println(list);

list.addLast("I");
list.addLast("said");
list.addLast("she");
list.addLast("said");
list.addLast("she");
list.addLast("said");
list.addLast("I");
list.addLast("said");

System.out.println(list);

System.out.println(list.findAndReplace("I", "she"));

System.out.println(list);

System.out.println(list.findAndReplace("I", "she"));

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

L'exécution du programme ci-dessous produira le résultat suivant sur la sortie.

```
[]
0
[]
[I, said, she, said, she, said, I, said]
2
[she, said, she, said, she, said, she, said]
0
[she, said, she, said, she, said, she, said]
```

Complétez l'implémentation de la méthode **findAndReplace**.

```
public class LinkedList<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> head;
    private int size;

    public int findAndReplace(E target, E replacement) {

        if (_____________________________________________________________) {

            _____________________________________________________________;

        }

        return findAndReplace(_____________________, target, replacement);
    }

    private __________ findAndReplace(_____________________, E target, E replacement) {

        int number;

        if (_____________________________) { // Cas de base

            _____________________________________________________________;

        } else { // Cas général

            __________ = findAndReplace(_____________________, target, replacement);

            if (_____________________________________________________________) {

                ___________________________________________________________

                ___________________________________________________________

            }

        }

        return _____________________________;
    }

}
```

Question 3 (15 points)

Pour cette question, notre but est de fournir cinq (5) méthodes pour l'**implémentation d'une liste à l'aide d'un tableau circulaire et dynamique**. Voici les variables d'instance de cette classe :

- **array** est une variable référence désignant le tableau contenant les éléments de la liste ;
- **size** est la taille logique (actuelle) de la liste ;
- **first** est l'index du premier élément de la liste dans le tableau désigné par la variable d'instance **array** ;
- **last** est l'index du dernier élément de la liste dans le tableau désigné par la variable d'instance **array** ;
- **capacity** est la capacité (taille physique) du tableau.

Par convention, lorsque la liste est vide, les variables **first** et **last** ont toutes les deux la valeur zéro. Nous avons deux constructeurs pour notre classe : le premier, sans paramètre, crée un tableau de la capacité par défaut, tandis que le second utilise la valeur de son paramètre comme taille initiale du tableau. La taille doit être d'au moins un (1). Voici une implémentation partielle pour notre classe :

```
public class CircularDynamicArrayList<E> {

    private E[] array;
    private int size;
    private int first, last;
    private int capacity = 100;

    @SuppressWarnings("unchecked")
    public CircularDynamicArrayList() {
        array = (E[]) new Object[capacity];
        first = last  = 0;
        size = 0;
    }

    @SuppressWarnings("unchecked")
    public CircularDynamicArrayList(int capacity) {
        if (capacity < 1) {
            throw new IllegalArgumentException("Minimum capacity is 1");
        }
        this.capacity = capacity;
        array = (E[]) new Object[capacity];
        first = last = 0;
        size = 0;
    }

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

    public boolean isFull() {
        return size == capacity;
    }

    // Les autres méthodes seraient ici

}
```

Question 3.1 *ensureSpace*

La première méthode que nous voulons écrire est **ensureSpace**. Cette méthode permet de s'assurer que la liste pourra accueillir un nouvel élément. Si nécessaire, la méthode **double** la taille du tableau. Complétez le code source de l'implémentation ci-dessous.

```
public class CircularDynamicArrayList<E> {

    // (...)

    private void ensureSpace() {

        if (_____________________________________________) {

             ____________________ newArray = _____________________________________________

            for (int i = ___________; i <___________; i++) {

                newArray[___________] = _____________________________________________

            }

            first = _____________________________________________

            last = _____________________________________________

            capacity = _____________________________________________

            array = _____________________________________________

        }
    }
}
```

Question 3.2 *addFirst*

La méthode **addFirst** ajoute un nouvel élément à la liste à la première position. Tous les autres éléments sont conservés. Nous devons nous assurer de traiter les conditions qui ne sont pas valides. Vous pouvez utiliser les méthodes de la classe écrites jusqu'à présent dans cette nouvelle méthode.

```
public class CircularDynamicArrayList<E> {

   // (...)

   public void addFirst(E newElement) {

        if (_____________________________________________) {

            throw ____________________________________________________________

        }

        ___________________________________________

        if (isEmpty()) {

            ___________________________________________

        } else {

            first = ____________________________________________________________

        }

        array[__________] = _____________________________

        size = _____________________________

    }
}
```

Question 3.3 *removeFirst*

La méthode **removeFirst** supprime et retourne l'élément à la première position. Tous les autres éléments sont conservés. Nous devons nous assurer de traiter les conditions qui ne sont pas valides. Vous pouvez utiliser les méthodes de cette classe écrites jusqu'à présent dans cette nouvelle méthode.

```
public class CircularDynamicArrayList<E> {

    // (...)

    public E removeFirst() {

        if (_____________________________________________) {

            throw ____________________________________________________________

        }

        E resultat = ____________________________________

        array[__________] = ___________________________

        size =  ________________________________________

        if (size == 0) {

            ____________________________________________________________

        } else {

            ____________________________________________________________

        }

        return ____________________________________

    }
}
```

Question 3.4 *add*

La méthode **add** ajoute un nouvel élément à la liste à la position spécifiée. Les index débutent avec la valeur zéro (0). Tous les autres éléments sont conservés. Nous devons nous assurer de traiter les conditions qui ne sont pas valides. Vous pouvez utiliser les méthodes de la classe écrites jusqu'à présent dans cette nouvelle méthode.

```
public class CircularDynamicArrayList<E> {

    // (...)

    public void add(E newElement, int index) {

        if (_____________________________________) {

            throw ____________________________________________________

        }

        if (_____________________________________) {

            throw ____________________________________________________

        }

        if (isEmpty()) {

            addFirst(newElement);

        } else {

            ___________________________________

            int currentIndexLocation =  ___________________________________

            int i =  ______________________________________

            while (i != ____________________________________) {

                array[_____________________] = ____________________________________

                i = ____________________________________

            }

           ____________________________________________________________

           ____________________________________________________________

           ____________________________________________________________

        }
    }
}
```

Question 3.5 *remove*

Enfin, la méthode **remove** supprime l'élément à l'index spécifié dans la liste, et le renvoie. Tous les autres éléments sont conservés. Nous devons nous assurer de traiter les conditions qui ne sont pas valides. Vous pouvez utiliser les méthodes de la classe écrites jusqu'à présent dans cette nouvelle méthode.

```
public class CircularDynamicArrayList<E> {
    // (...)

    public E remove(int index) {

        if(_____________________________________) {

             throw ____________________________________________________

        }

        if(_____________________________________) {

            throw ____________________________________________________

        }

        E resultat;

        if (size == 1) {

            ____________________________________________________

        } else {

            int currentIndexLocation = _____________________________________

            resultat = _____________________________________

            int i = _____________________________________

            while (i != _____________________________) {

                array[__________] = _____________________________

                i =  _____________________________

            }

           _____________________________________

           _____________________________________

           _____________________________________

        }

        _____________________________________
    }
}
```

Question 4 (15 points)

Dans cette question, nous utilisons des **listes** et des **arbres binaires de recherche** ensemble. Plus précisément, nous allons travailler avec l'interface **List** suivante :

```
public interface List<E> {

    // Retourne la taille actuelle (logique) de la liste
    int getSize();

    // Renvoie true si et seulement si la liste est vide
    boolean isEmpty();

    // Ajoute un élément comme premier élément de la liste
    void addFirst(E elem);

    // Ajoute l'élément elem comme dernier élément de la liste
    void addLast(E elem);

    // Ajoute un élément à l'index «index» dans la liste
    void add(int index,E elem);

    // Renvoie la référence du premier élément de la liste
    E getFirst();

    // Renvoie la référence du dernier élément de la liste
    E getLast();

    // Retourne la référence de l'élément à l'index «index» dans la liste
    E get(int index);

    // Supprime et renvoie la référence du premier élément de la liste
    E removeFirst();

    // Supprime et renvoie la référence du dernier élément de la liste
    E removeLast();

    // Supprime la référence de l'élément à l'index «index» dans la liste
    E remove(int index);
}
```

Dans ce qui suit, nous avons accès à deux implémentations de l'interface **List**. Nous ne précisons pas encore laquelle utiliser. Nous avons aussi une classe **BinarySearchTree**. Certaines parties de son implémentation sont présentées ci-dessous.

Question 4.1 *buildSortedList*

Notre premier but est d'ajouter une méthode **buildSortedList** à la classe **BinarySearchTree**. Cette méthode reçoit en paramètre la référence d'une liste vide (objet dont la classe réaliste l'interface **List**), et elle remplit cette liste avec tous les éléments actuellement dans l'arbre binaire de recherche, en ordre croissant.

Par exemple, le code test suivant :

```
public static void testBuildSortedList(List<String> list) {

    BinarySearchTree<String> tree;
    tree = new BinarySearchTree<String>();

    tree.add("F"); tree.add("A"); tree.add("C");
    tree.add("X"); tree.add("M"); tree.add("N");
    tree.add("O"); tree.add("B"); tree.add("L");

    System.out.println(list);
    tree.buildSortedList(list);
    System.out.println(list);

}
```

produit la sortie suivante :

```
[]
[A, B, C, F, L, M, N, O, X]
```

Si l'arbre est actuellement vide, les listes restent inchangées. Par exemple, le code test suivant :

```
public static void testBuildSortedList2(List<String> list) {

    BinarySearchTree<String> tree;
    tree = new BinarySearchTree<String>();

    System.out.println(list);
    tree.buildSortedList(list);
    System.out.println(list);

}
```

produit la sortie suivante :

```
[]
[]
```

Une déclaration partielle et incomplète de la classe **BinarySearchTree** est fournie ci-dessous :

```
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;
    private int size;

    public BinarySearchTree() {
        root = null;
        size = 0;
    }

    public int getSize() {
        return size;
    }

    public boolean contains(E obj) {
        // Caché pour sauver de l'espace
      }

    public boolean add(E obj) {
        // Caché pour sauver de l'espace
    }

    public String toString() {
        // Caché pour sauver de l'espace
    }

    // Les autres méthodes apparaitraient ici

}
```

Dans l'espace ci-dessous, fournissez le code de la méthode **buildSortedList**. Comme vous pouvez le voir, la méthode utilise une méthode privée, suggérant une implémentation récursive.

```
public class BinarySearchTree <E extends Comparable<E>> {

    // Beaucoup de code caché

    public void buildSortedList(List<E> list) {

        if (list == null) {
            throw new NullPointerException("List must not be null");
        }

        if (list.getSize() != 0) {
            throw new IllegalArgumentException("List must be empty");
        }









    }

    private void buildSortedList( List<E> list, Node<E> current) {

















    }

}
```

Question 4.2 Construire un arbre balancé

Nous utilisons maintenant la liste triée pour construire un nouvel arbre binaire de recherche. Dans cette section, nous devons ajouter un nouveau constructeur à la classe **BinarySearchTree**. Ce constructeur reçoit comme paramètre la référence à une liste (objet dont la classe réalise l'interface **List**), qui est censée être triée. Le but du constructeur est de créer un nouvel arbre binaire de recherche contenant tous les éléments de la liste. L'arbre résultant devrait être aussi équilibré que possible.

Par exemple, supposons que **list** désigne une liste avec les éléments suivants : **[1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]**. Cette liste contient les 13 premiers nombres de Fibonnacci consécutifs et triés.

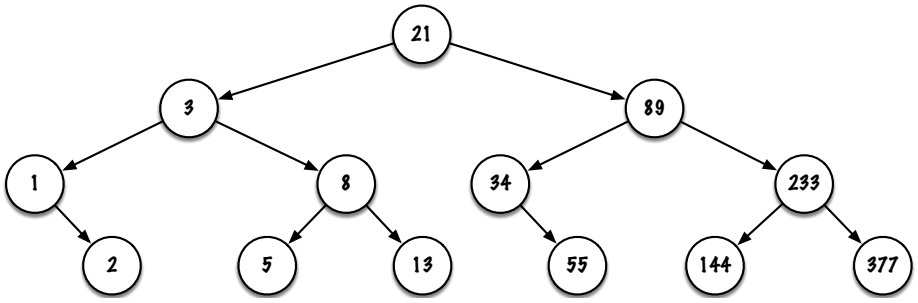
L'extrait de code suivant :

```
System.out.println(list);
BinarySearchTree<Integer> tree;
tree = new BinarySearchTree<Integer>(list);
```

produit la sortie suivante :

```
[1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]
        null
      377
        null
    233
        null
      144
        null
  89
        null
      55
        null
    34
      null
21
        null
      13
        null
    8
        null
      5
        null
  3
        null
      2
        null
    1
      null
```

Autrement dit, il construit l'arbre suivant :



Dans l'espace ci-dessous, donnez votre implémentation du nouveau constructeur. Là encore, la méthode utilise une méthode privée.

```
public class BinarySearchTree <E extends Comparable<E>> {

    // Beaucoup de code caché

    public BinarySearchTree(List<E> list) {

        if(list == null || list.getSize() == 0) {
            return;
        }

        root =

    }

    private Node<E> buildBalanced(int from, int to, List<E> list) {









     }
}
```

Question 4.3 Implémentation efficace

Le but ultime de nos deux méthodes est de pouvoir créer un arbre binaire de recherche équilibré (balancé) à partir d'un arbre binaire de recherche existant et possiblement déséquilibré, en utilisant des énoncés similaires à ceux qui sont présentés ci-dessous :

```
    public BinarySearchTree<E> createBalanced(BinarySearchTree<E> unbalanced) {

        List<E> list;

        list  = ...  // Initialisation

        unbalanced.buildSortedList(list);

        BinarySearchTree<E> balanced;

        balanced = new BinarySearchTree<E>(list);

        return balanced;
    }
```

Il faut donc que ce code soit aussi efficace que possible [1], ce qui signifie que notre méthode **buildSorted-List** et notre nouveau constructeur pour l'arbre binaire de recherche doivent être efficaces.

Pour l'implémentation de notre liste, nous avons deux choix : notre premier choix est une liste circulaire doublement chaînée avec un noeud factice. Cette implémentation a un seul constructeur, sans paramètre :

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

        public DoublyLinkedListDummyNode() {
            // Code source
        }

        // Encore du code source
}
```

Notre deuxième choix est une implémentation de liste à l'aide d'un tableau dynamique et circulaire similaire à celle de la Question 3. Dans cette implémentation, la taille du tableau est doublée lorsqu'un élément est ajouté à un tableau rempli. Cette implémentation a deux constructeurs : un constructeur sans paramètre, et un autre qui reçoit une valeur de type **int** en paramètre d'entrée. Avec le premier constructeur, une valeur par défaut est utilisée pour la taille initiale du tableau. Avec le second constructeur, la valeur du paramètre d'entrée est utilisé comme taille initiale du tableau :

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

        public CircularDynamicArrayList() {
            // Utilise une valeur par défaut pour la taille initiale du tableau
        }

        public CircularDynamicArrayList(int initialSize) {
            // Utilise initialSize pour la taille initiale du tableau
        }

        // Encore du code
}
```

[1] Spécifiquement, il doit être linéaire en fonction de la taille de l'arbre binaire de recherche déséquilibré.

Vous devez décider quelle implémentation utiliser dans la méthode **createBalanced** afin d'avoir une implémentation rapide. Ajoutez votre initialisation de la variable **list** dans le code ci-dessous :

```
public BinarySearchTree<E> createBalanced(BinarySearchTree<E> unbalanced) {

    List<E> list;

    list = ____________________________________________________________

    unbalanced.buildSortedList(list);

    BinarySearchTree<E> balanced;
    balanced = new BinarySearchTree<E>(list);

    return balanced;

}
```