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

Avril 2017, 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 hypothèses et commentaires afin d'obtenir des points partiels.
5. Écrivez lisiblement, puisque votre note en dépend.
6. Ne retirez pas l'agrafe du questionnaire d'examen.
7. Attendez l'annonce de début de l'examen.

Barème

Question	Maximum	Résultat
1	15	
2	45	
3	20	
4	10	
Total	**90**	

Question 1 [15 points]

A. Considérant la déclaration de classe ci-dessous :

```
public class A extends B implements C {

}
```

Lesquels des énoncés suivants sont **valides** et lesquels sont **invalides** ?

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

Valide ou **invalide**

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

Valide ou **invalide**

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

Valide ou **invalide**

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

Valid ou **invalide**

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

Valid ou **invalide**

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

Valid ou **invalide**

B. Une exception est déclarée («*checked*»), si l'exception est lancée dans un bloc **try/catch**.

Vrai ou **faux**

C. Une fois l'exception prise en charge par une clause **catch**, l'application reprend l'exécution à partir du point où l'exception a été levée (lancée).

Vrai ou **faux**

D. Construire un arbre binaire de recherche (objet de la classe **BinarySearchTree**) en ajoutant les éléments en **ordre croissant** produira un arbre **balancé**.

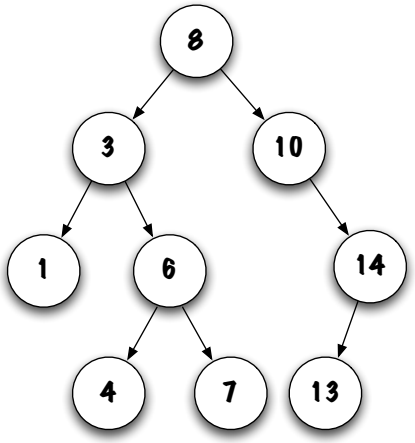
Vrai ou **faux**.

E. La profondeur d'un arbre binaire balancé de taille n est $\lfloor \log_2 n \rfloor$.

Vrai ou **faux**

F. Le parcours en **ordre postfixe** de l'arbre ci-dessous visite les éléments dans l'ordre suivant : 1, 4, 7, 6, 3, 13, 14, 10 et 8.

Vrai ou **faux**



G. On appelle **deque** («*double-ended queue*» et on prononce *deck*) le type abstrait de données semblable à la file, mais permettant l'ajout et le retrait d'éléments aux deux extrémités (avant et arrière). Tout comme les piles et les files, on peut implémenter ce type abstrait de données à l'aide d'un tableau, d'un tableau circulaire, de noeuds simplement ou doublement chaînés, etc. Pour cette question, nous avons les implémentations suivantes :

- **ArrayDeque** : utilise un simple tableau, possède une variable d'instance pour désigner le tableau et une variable pour sauvegarder le nombre d'éléments.
- **CircularArrayDeque** : utilise un tableau circulaire, possède une variable d'instance pour désigner le tableau, ainsi que deux variables d'instance de type entier pour sauvegarder les positions avant et arrière de la file.
- **LinkedDeque** : utile des noeuds simplement chaînés et une variable d'instance afin de désigner le premier noeud.
- **DoublyLinkedDeque** : utilise des noeuds doublement chaînés, une variable d'instance désigne le premier noeud, et une autre variable d'instance désigne le dernier noeud.

Pour chacune des quatre méthodes ci-dessous et pour chaque implémentation, indiquez si la méthode est **rapide** (son temps d'exécution est indépendant du nombre d'éléments qui s'y trouve) ou encore **lente** (le temps d'exécution est proportionnel au nombre d'éléments sauvegardés).

Notez que les implémentations à base de tableaux utilisent toutes les deux la technique du tableau dynamique. Elles peuvent donc sauvegarder un nombre arbitrairement grand d'éléments. Cependant, la taille physique ne diminue pas automatiquement.

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

Question 2 [45 points]

À titre de rappel, voici la déclaration de l'interface **Stack** et celle de l'interface **Queue**.

```
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();
}
```

Pour répondre à cette question, supposez l'existence de la classe **StackImplementation** qui réalise l'interface **Stack** ainsi que l'existence de la classe **QueueImplementation** qui réalise l'interface **Queue**. Vous pouvez utiliser toutes les instances de ces deux classes que vous jugez nécessaire, mais, aucunes autres structures de données, notamment, aucun tableau.

L'objectif est de concevoir une classe nommée **StacksAndQueues** afin d'y implémenter des **méthodes de classe** pour des opérations de base sur des piles et des files. Pour simplifier les choses, nous supposons que la classe **StacksAndQueues** manipule exclusivement des piles et des files de chaînes de caractères (objets de la classe **String**).

- Sur les pages qui suivent, vous devez fournir l'implémentation des méthodes.
- Pour chaque méthode, assurez-vous de traiter les situations exceptionnelles.

Voici un exemple montrant l'utilisation de la classe **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);
```

L'exécution du programme ci-dessus affichera le résultat ci-dessous.

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

2.1 StacksAndQueues.reverseQueue(Queue<String> queue)

La méthode **StacksAndQueues.reverseQueue** est une méthode de classe qui reçoit en paramètre la référence d'une file de chaînes de caractères. À la suite d'un appel à la méthode, les éléments de la file désignée par **queue** sont maintenant dans l'ordre inverse.

Par exemple, si l'on passe en paramètre à la méthode **StacksAndQueues.reverseQueue** la file suivante :

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

À la suite de cet appel, le contenu de la file sera comme suit :

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

Complétez l'implémentation de la méthode **StacksAndQueues.reverseQueue** dans la boite ci-dessous :

```
public class StacksAndQueues {

    ____________________________ reverseQueue(Queue<String> queue) {

        if (queue_____________________________________) {

            ____________________________________________________________________

        }

        ____________________ tmp;

        tmp = ____________________________________

        while (_____________________________________) {

            tmp._____________________________________________________________

        }

        while (_____________________________________) {

            queue._____________________________________________________________

        }

    }
```

2.2 StacksAndQueues.reverseStack(Stack<String> stack)

La méthode **StacksAndQueues.reverseStack** est une méthode de classe qui reçoit en paramètre la référence d'une pile de chaînes de caractères. À la suite d'un appel à méthode, les éléments de la pile sont dans l'ordre inverse.

Par exemple, si l'on passe la pile suivante à la méthode **StacksAndQueues.reverseStack** :

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

à la suite de l'appel, la pile contiendra les éléments suivants :

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

Complétez l'implémentation de la méthode **StacksAndQueues.reverseStack** dans la boite ci-dessous :

```
_____ reverseStack(Stack<String> stack) {
    if (stack_____) {
        _____
    }
    _____ tmp;
    tmp = _____
    while(_____) {
        tmp._____
    }
    while(_____) {
        stack._____
    }
}
```

2.3 StacksAndQueues.removeAll(Queue<String> queue, String toRemove)

Cette méthode de classe **StacksAndQueues.removeAll** a deux paramètres : la référence d'une file de chaînes de caractères et la référence d'une chaîne de caractères. À la suite d'un appel à la méthode, toutes les occurrences de la chaîne de caractères ont été retirées de la file. S'il n'y avait aucune occurrence de cette chaîne de caractères dans la file, alors la file demeure inchangée.

Par exemple, si la file suivante est passée en paramètre à la méthode **StacksAndQueues.removeAll** :

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

et que le second paramètre est "a", à la suite de l'appel, la file doit contenir les éléments suivants :

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

Complétez l'implémentation de la méthode **StacksAndQueues.removeAll** dans la boite ci-dessous :

```
_____ removeAll(Queue<String> queue, String toRemove) {
    if (_____) {
        _____
    }
    _____ tmp;
    tmp = _____
    while (_____) {
        _____
        _____
        if (_____) {
            _____
        }
    }
    while(_____) {
        _____
    }
}
```

2.4 StacksAndQueues.removeAll(Stack<String> stack, String toRemove)

Cette (seconde) méthode **StacksAndQueues.removeAll** est une méthode de classe qui possède deux paramètres : la référence d'une pile de chaînes de caractères et la référence d'une chaîne de caractères. À la suite d'un appel à la méthode, toutes les occurrences de la chaîne de caractères ont été retirées de la pile. Si la pile ne contenait aucune instance de la chaîne de caractères, alors la pile demeure inchangée.

Par exemple, si la pile suivante est passée en paramètre à la méthode **StacksAndQueues.removeAll** :

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

et que le second paramètre est "a", à la suite de l'appel, la pile doit contenir les éléments suivants :

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

Complétez l'implémentation de la méthode **StacksAndQueues.removeAll** dans la boite ci-dessous :

```
_____ removeAll(Stack<String> stack, String toRemove) {
    if (_____) {
        _____
    }
    _____ tmp;
    tmp = _____
    while (_____) {
        _____
        _____
        if (_____) {
            _____
        }
    }
    while (_____) {
        _____
    }
}
```

2.5 StacksAndQueues.removeFirst(Queue<String> queue, String toRemove)

La méthode de classe **StacksAndQueues.removeFirst** a deux paramètres : la référence d'une file de chaînes de caractères et la référence d'une chaîne de caractères. À la suite d'un appel à cette méthode, **la première occurrence** de la chaîne de caractères est retirée de la file. S'il n'y avait aucune occurrence de la chaîne de caractères dans la file, alors la file demeure inchangée.

Par exemple, si la file suivante est passée en paramètre à la méthode **StacksAndQueues.removeFirst** :

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

et que le second paramètre est la chaîne de caractères "a", à la suite de l'appel, la file doit contenir les éléments suivants :

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

Si le second paramètre avait été "b", à la suite de l'appel, la file doit contenir les éléments suivants :

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

Complétez l'implémentation de la méthode **StacksAndQueues.removeFirst** dans la boite ci-dessous :

```
_____ removeFirst(Queue<String> queue, String toRemove) {
    if (_____) {
        _____
    }
    _____ tmp;
    tmp = _____
    _____
    while (_____) {
        _____
        if (_____) {
            _____
        } else {
            _____
        }
    }
    while (_____) {
        _____
    }
}
```

2.6 StacksAndQueues.removeFirst(Stack<String> stack, String toRemove)

La (seconde) méthode de classe **StacksAndQueues.removeFirst** a deux paramètres : la référence d'une pile de chaînes de caractères et la référence d'une chaîne de caractères. À la suite d'un appel à cette méthode, la **première occurrence** de la chaîne de caractère a été retirée de la pile. S'il n'y a aucune occurrence de la chaîne de caractères, alors la pile demeure inchangée.

Par exemple, si la pile suivante suivante est passée en paramètre à la méthode **StacksAndQueues.removeFirst** :

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

et que le second paramètre est "a", à la suite de l'appel, la pile doit contenir les éléments suivants :

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

Si le second paramètre était "b", à la suite de l'appel, la pile doit contenir les éléments suivants :

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

Complétez l'implémentation de la méthode **StacksAndQueues.removeFirst** dans la boite ci-dessous :

```
_____ removeFirst(Stack<String> stack, String toRemove) {
    if(_____) {
        _____
    }
    _____ tmp;
    tmp = _____
    _____
    while (_____) {
        _____
        if (_____) {
            _____
        } else {
            _____
        }
    }
    while (_____) {
        _____
    }
}
```

2.7 StacksAndQueues.removeLast(Queue<String> queue, String toRemove)

La méthode de classe **StacksAndQueues.removeLast** a deux paramètres : la référence d'une file de chaînes de caractères et la référence d'une chaîne de caractères. À la suite d'un appel à la méthode, la **dernière occurrence** de la chaîne de caractères a été retirée de la file. S'il n'y a aucune occurrence de la chaîne de caractères dans la file, alors la file demeure inchangée.

Par exemple, si la file suivante est passée en paramètre à la méthode **StacksAndQueues.removeLast** :

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

et que le second paramètre est "a", à la suite de l'appel, la file doit contenir les éléments suivants :

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

Complétez l'implémentation de la méthode **StacksAndQueues.removeLast** dans la boite ci-dessous :

```
    _____ removeLast(Queue<String> queue, String toRemove) {




}
```

2.8 StacksAndQueues.removeLast(Stack<String> stack, String toRemove)

Cette (seconde) méthode de classe **StacksAndQueues.removeLast** a deux paramètres : la référence d'une pile de chaînes de caractères et la référence d'une chaîne de caractères. À la suite d'un appel à la méthode, la **dernière occurrence** de la chaîne de caractères a été retirée de la pile. S'il n'y a aucune occurrence de la chaîne de caractères dans la pile, alors la pile demeure inchangée.

Par exemple, si la pile suivante est passée en paramètre à la méthode **StacksAndQueues.removeLast** :

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

et que le second paramètre est "a", à la suite de l'appel, la pile doit contenir les éléments suivants :

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

Complétez l'implémentation de la méthode **StacksAndQueues.removeLast** dans la boite ci-dessous :

```
    _____ removeLast(Stack<String> stack, String toRemove) {

    }
}
```

Question 3 [20 points]

Nous vous fournissons une implémentation opérationnelle (qui fonctionne) d'une liste doublement chaînée pour laquelle il y a une référence vers le noeud avant (**head** de type **Node**) et une référence vers le noeud arrière (**tail** de type **Node**). Les noeuds (objets de la classe **Node**) sont doublement chaînés. Voici les portions du code source qui sont pertinentes pour cette question.

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

    private static class Node<T> {

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

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

    private Node<E> head;
    private Node<E> tail;

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

    // ...
}
```

Pour cette question, vous devez implémenter un itérateur **à deux sens** («*two-way*») qui **boucle** («*looping*»).

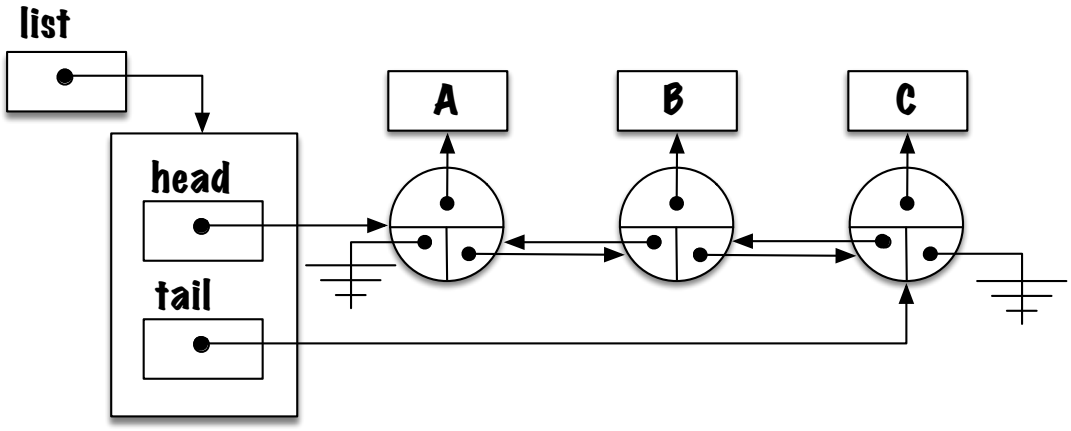
- Par itérateur **à deux sens**, on veut dire un itérateur qui se déplace soit vers l'avant, grâce à sa méthode **next**, ou vers l'arrière, grâce à sa méthode **prev**.
  - La méthode **next** déplace l'itérateur vers l'avant et retourne la prochaine valeur.
  - La méthode **prev** déplace l'itérateur vers l'arrière et retourne la prochaine valeur.
- Par **bouclage**, on entend un itérateur qui lorsqu'il a atteint la fin de la liste, dans un sens ou dans l'autre, continue à l'autre extrémité de la liste.

Voici la déclaration de l'interface **Iterator** :

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

La méthode **hasNext** (respectivement **hasPrev**) retourne **true** si et seulement si le prochain appel à la méthode **next** (respectivement **prev**) retournera un élément (de type **E**).

Supposons que la variable référence **list**, de type **List<String>**, désigne un objet de la classe **DoublyLinkedList<String>** dont le contenu est "**A**", "**B**", et "**C**".



L'exemple ci-dessous démontre le fonctionnement de l'itérateur : on y déclare une variable référence **i** de type **Iterator<String>**, on appelle la méthode d'instance **iterator** de l'objet désigné par la variable référence **list**, la valeur retournée (la référence de l'itérateur nouvellement créé) est sauvegardée dans la variable **i**. On déplace alors l'itérateur cinq (5) fois vers l'avant (ce dernier boucle vers l'avant au quatrième appel), l'itérateur se déplace ensuite trois (3) vers l'arrière (il boucle vers l'arrière au second appel) :

```
List<String> list;
list = new DoublyLinkedList<String>();
list.addLast("A"); list.addLast("B"); list.addLast("C");

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

System.out.println(i.next()); // affiche A
System.out.println(i.next()); // affiche B
System.out.println(i.next()); // affiche C
System.out.println(i.next()); // boucle vers l'avant et affiche A
System.out.println(i.next()); // affiche B
System.out.println(i.prev()); // affiche A
System.out.println(i.prev()); // boucle vers l'arrière et affiche C
System.out.println(i.prev()); // affiche B
```

Vous devez fournir une implémentation de la méthode **iterator()**, ainsi que tout le code source nécessaire pour que l'exemple ci-dessus fonctionne. On devrait pouvoir créer plusieurs itérateurs pour une même liste.

Écrivez votre réponse dans la boite qui suit, et celle de la prochaine page, si nécessaire. Ces énoncés seront ajoutés au contenu de la classe **DoublyLinkedList**. C'est-à-dire à l'intérieur de la classe.

```
public Iterator<E> iterator() {




}

// Le reste du code est inséré ici :
```

Question 4 [10 points]

Soit la classe **SinglyLinkedList** ci-dessous :

```
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;
    }

    // ...
```

Vous devez implémenter la méthode **and()** pour la classe **SinglyLinkedList**. La méthode **and()** est une méthode récursive qui retourne **true** si et seulement si **tous** les éléments de la liste ont la valeur booléenne **true**. Le comportement de la méthode n'est pas défini si la liste est vide.

L'exemple ci-dessous affiche d'abord la valeur **true** puis **false**.

```
SinglyLinkedList test;
test = new SinglyLinkedList();

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

Donnez une implémentation récursive de la méthode **and()** dans la boite sur la page suivante. **Veuillez noter** que deux points seront retranchés pour une implémentation qui visite inutilement trop de noeuds.