Université d'Ottawa
Faculté de génie

École d'ingénierie et de
technologie de l'information

uOttawa

L'Université canadienne
Canada's university

University of Ottawa
Faculty of Engineering

School of Information
Technology and Engineering

# Introduction informatique II (ITI 1621)
# Examen final

Instructeur: Marcel Turcotte

Avril 2006, durée: 3 heures

## Identification

Nom : ______________________ Prénom : ______________________

Numéro d'étudiant : __________ Signature : ______________________

## Consignes

1. Livres fermés ;
2. Sans calculatrice ou toute autre forme d'aide ;
3. Répondez sur ce questionnaire, utilisez le verso des pages si nécessaire, mais vous ne pouvez remettre aucune page additionnelle ;
4. Écrivez lisiblement, votre note en dépend ;
5. Commentez vos réponses ;
6. Ne retirez pas l'agrafe.

## Barème

| Question | Maximum | Résultat |
|---:|---:|---:|
| 1 | 15 | |
| 2 | 15 | |
| 3 | 10 | |
| 4 | 15 | |
| 5 | 15 | |
| 6 | 5 | |
| 7 | 15 | |
| 8 | 10 | |
| **Total** | **100** | |

# Question 1 : isPalindrome (15 points)

Complétez l'implémentation de la méthode statique **boolean isPalindrome( CharReader reader )**. Définissons un **palindrome** comme étant un mot ou une phrase qui peut être lu indifféremment de gauche à droite ou de droite à gauche lorsqu'on ignore les ponctuations, accents et espaces. Voici des exemples de palindromes :

- i prefer pi
- never odd or even
- was it a cat i saw

Vous devez vous conformer aux consignes qui suivent.

- **boolean isPalindrome( CharReader reader )** ; retourne **true** si tout le mot ou toute la phrase spécifié par l'objet **reader** est un palindrome selon la définition ci-haut, et **false** sinon ;
- Le paramètre de la méthode est un objet **CharReader**. Un tel objet possède deux méthodes d'instance.
  - **boolean hasMoreChars()** ; retourne **true** s'il y a des caractères supplémentaires à retourner, c'est-à-dire si le prochain appel à **nextChar()** pourra être complété avec succès, et **false** sinon ;
  - **char nextChar()** ; retourne le prochain caractère de l'entrée ;
- Afin de sauvegarder des résultats temporaires, vous devez utiliser des piles et/ou des files (en particulier, vous ne devez pas créer des tableaux ou des chaînes de caractères) ;
- Il existe une classe **StackImpl** qui réalise l'interface **Stack**. Pour cette question, les piles servent à sauvegarder des caractères.

```
public interface Stack {
    public abstract boolean isEmpty();
    public abstract char peek();
    public abstract char pop();
    public abstract void push( char element );
}
```

- La classe **QueueImpl** réalise l'interface **Queue**. Pour cette question, les files servent à sauvegarder des caractères.

```
public interface Queue {
    public abstract boolean isEmpty();
    public abstract char dequeue();
    public abstract void enqueue( char element );
}
```

- **StackImpl** et **QueueImpl** peuvent contenir un nombre illimité de caractères.

```
public static boolean isPalindrome( CharReader reader ) {

    boolean answer = true;







    while ( reader.hasMoreChars() ) {

        char c = reader.nextChar();





















    return answer;
}
```

# Question 2 : CircularStack (15 points)

Complétez l'implémentation de la classe **CircularStack**. Cette classe serait utile afin de développer une application pour laquelle on pourrait annuler l'effet d'un nombre fixe d'opérations. Par exemple, imaginez un éditeur de texte permettant l'ajout, l'effacement ou encore la substitution de caractères. Pour chaque opération effectuée (ajout, effacement, substitution), l'application empile un objet sur une pile. Lorsque l'application doit annuler une opération, elle obtient l'information nécessaire en retirant un élément de la pile. Cependant, puisque la pile a une taille fixe, le nombre maximal d'opérations pouvant être annulées est égal à la taille de la pile. Vous devez vous conformer aux consignes qui suivent.

- Parce que la mémoire est limitée, un nombre fixe d'opérations peuvent être annulées ;
- Lorsque la pile est pleine, la méthode **push** écarte l'élément le plus ancien (celui du fond) afin de libérer de l'espace pour le nouvel élément ;
- Cependant, la méthode **push** ne doit pas déplacer les éléments qui s'y trouvent. La méthode va plutôt écraser l'élément le plus ancien (celui du fond). Vous remarquerez la similarité avec l'implémentation de la file à l'aide d'un tableau circulaire vue en classe ;
- **void push( Object o )** ; ajoute un élément sur le dessus de cette pile, la valeur **null** est valide ;
- **Object pop()** ; retire l'élément du dessus. Si la pile était vide au moment de l'appel, la méthode lancera un exception de type **EmptyStackException**.

```
import java.util.EmptyStackException;

public class CircularStack {

    private Object[] stack;
    private int top = 0;
    private int size = 0;

    public CircularStack( int capacity ) {
        if ( capacity < 0 ) {
            throw new IllegalArgumentException( "negative number" );
        }
        stack = new Object[ capacity ];
    }

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

Complétez l'implémentation des méthodes **push** et **pop** sur la page suivante.

```
    public void push( Object item ) {

    }

    public Object pop() {

    }

} // End of CircularStack
```

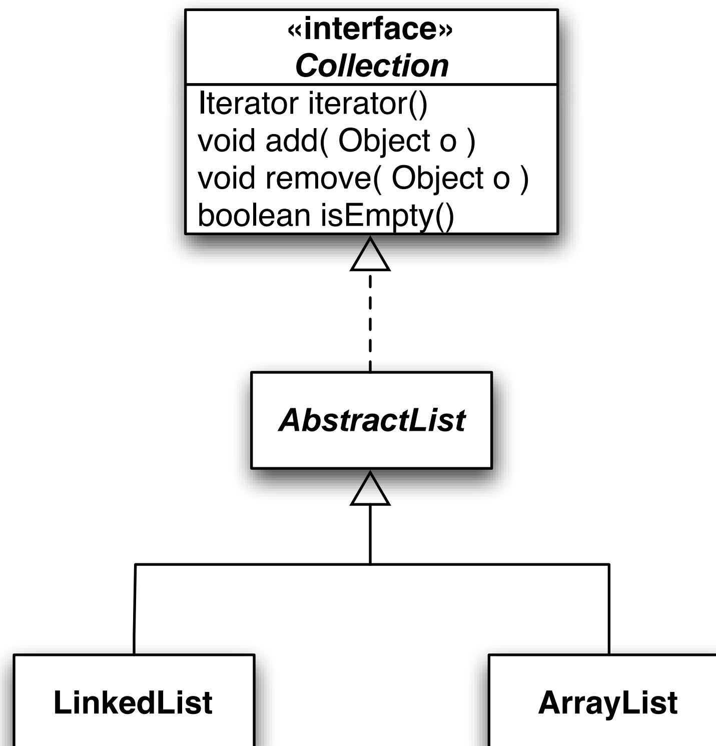
# Question 3 : ArrayListIterator (10 points)

Pour la classe **ArrayList** ci-bas, complétez l'implémentation de l'itérateur. Pour cette question, l'interface **Iterator** est définie comme suit.

```
public interface Iterator {

    // Retourne true si l'itération n'est pas terminée.

    public abstract boolean hasNext();

    // Retourne le prochain élément de l'itération. Lance
    // NoSuchElementException si l'itération n'a pas d'élément à retourner.

    public abstract Object next();
}

import java.util.NoSuchElementException;

public class ArrayList {

    // Variables d'instance
    private Object[] elems;
    private int size = 0;

    // Constructeur
    public ArrayList( int capacity ) {
        if ( capacity < 0 ) {
            throw new IllegalArgumentException();
        }
        elems = new Object[ capacity ];
    }
    public boolean isEmpty() {
        return size == 0;
    }
    public void addLast( Object element ) {
        if ( size == elems.length ) {
            increaseSize();
        }
        elems[ size ] = element;
        size++;
    }
    private void increaseSize() {
        Object[] newElems;
        newElems = new Object[ 2 * elems.length ];
        System.arraycopy( elems, 0, newElems, 0, elems.length );
        elems = newElems;
    }
```

```
    public Object remove( int index ) {
        if ( index < 0 || index > (size - 1) ) {
            throw new IndexOutOfBoundsException( "Index: "+index );
        }
        Object savedElem = elems[ index ];
        System.arraycopy( elems, index+1, elems, index, size - index - 1 );
        size--;
        elems[ size ] = null;
        return savedElem;
    }

    public Iterator iterator() {
        return ______________________________;
    }

    private _______________ class ArrayListIterator implements Iterator {

        private _______________ current = _______________;

        public boolean hasNext() { // complétez hasNext()
            boolean answer;






            return answer;
        }

        public Object next() { // complétez next()
            Object answer;
















            return answer;
        }
    } // end of ArrayListIterator

} // end of ArrayList
```

# Question 4 : equals (15 points)




À l'intérieur de la classe abstraite **AbstractList** se trouvant à la page qui suit, vous devez redéfinir la méthode **boolean equals( Object other )**. Vous devez vous conformer aux consignes qui suivent.

- La méthode détermine l'égalité de cette liste et **other** ;
- Elle retourne **true** si et seulement si **other** désigne aussi une instance de la classe **AbstractList** (plus précisément, si **other** désigne une instance d'une sous-classe de **AbstractList**), les deux listes ont la même taille, et toutes les paires correspondantes d'éléments des deux listes sont égales. Sinon, la méthode retourne **false** ;
- La valeur **null** est un élément valide ;
- **AbstractList** réalise l'interface **Collection** ;
- **LinkedList** et **ArrayList** sont deux exemples de sous-classes d'**AbstractList**, mais elles ne sont pas les seules ;
- Utilisez des itérateurs pour l'implémentation de la méthode.

Les déclarations des interfaces **Collection** et **Iterator** se trouvent à la page 10.

```
public class AbstractList implements Collection {

    public boolean equals( Object other ) {

    } // End of equals
} // End of AbstractList
```

```
public interface Collection {

    /* Retourne un itérateur pour les éléments de cette collection.
     */

    public abstract Iterator iterator();

    /* Ajoute l'élément à la collection et retourne true si la
     * collection a changé suite à cette opération.
     */

    public abstract boolean add( Object item );

    /* Retire une seule instance de l'objet spécifié, s'il est présent.
     * Retourne true si a la collection a changé suite à cet appel.
     */

    public abstract boolean remove( Object item );

    /* Retourne true si la collection est vide.
     */

    public abstract boolean isEmpty();
}

public interface Iterator {

    /* Retourne true si l'itération n'est pas terminée.
     */

    public abstract boolean hasNext();

    /* Retourne le prochain élément de l'itération. Lance
     * NoSuchElementException s'il n'y a pas de prochain élément.
     */

    public abstract Object next();
}
```

# Question 5 : splitAt (15 points)

Complétez l'implémentation de la méthode d'instance **LinkedList splitAt( int n )**. La méthode **splitAt** brise la liste en deux parties. Les premiers **n** éléments demeurent dans la liste originale alors que le reste est retourné dans une nouvelle liste (objet **LinkedList**). En particulier,

- Suite à l'exécution de **t = l.splitAt( 0 )**, **l** est vide et **t** contient tous les éléments qui étaient initialement présents dans la liste **l** ;
- Suite à l'exécution de **t = l.splitAt( 1 )**, **l** ne contient qu'un seul élément alors que **t** contient tous les éléments qui étaient initialement présents dans la liste **l** sauf un ;
- Suite à l'exécution de **t = l.splitAt( i )**, **l** contient **i** éléments et **t** contient **size-i** éléments, alors que **size** est la longueur de **l** avant l'appel à **splitAt** ;
- Suite à l'exécution de **t = l.splitAt( l.size() )**, **l** est inchangée et **t** désigne une liste vide (de type **LinkedList**) ;
- L'exception **IllegalArgumentException** sera lancée si **n** est plus grand que la taille de la liste.

L'implémentation de **LinkedList** est la même que celle du devoir 4.

- Cette implémentation débute toujours par un noeud factice («dummy node») qui marque le début de la liste. Le noeud factice n'est jamais utilisé pour sauvegarder des données. La liste vide ne contient que le noeud factice ;
- Pour cette question, les noeuds de la liste sont doublement chaînés ;
- Ici, la liste est circulaire, c'est-à-dire que la référence **next** du dernier noeud désigne le noeud factice et que la référence **previous** du noeud factice désigne le dernier noeud de la liste. Dans le cas de la liste vide, les références **previous** et **next** du noeud factice pointent vers le noeud factice lui-même ;
- Puisque le dernier noeud est facile à accéder, c'est toujours le noeud qui précède le noeud factice, l'entête de la liste n'a qu'un pointeur **head**.

Aucun appel de méthode n'est permis. Complétez la méthode **splitAt** pour la classe **LinkedList** à la page qui suit.

**Suggestion :** dessinez les diagrammes de mémoire pour touts les cas possibles.

```
public class LinkedList {
    private static class Elem { // Implémentation des noeuds doublement chaînés
        private Object value;
        private Elem previous;
        private Elem next;
        private Elem( Object value, Elem previous, Elem next ) {
            this.value = value;
            this.previous = previous;
            this.next = next;
        }
    }
    private Elem head;
    private int size;
    public LinkedList() {
        head = new Elem( null, null, null );
        head.next = head.previous = head;
        size = 0;
    }

    public LinkedList splitAt( int n ) {
        if ( ______________________________ ) {
            throw new IllegalArgumentException();
        }
        _______________ answer = ______________________________;
        Elem p = ______________________________;

        for ( int i=0; i<_______________; i++ ) {
            p = p.next;
        }
        if ( ______________________________ ) {











            answer.size = ______________________________;
            size = ______________________________;
        }
        return answer;
    }
}
```

# Question 6 : foo (5 points)

Étant donné une liste d'entiers (objets de la classe **Integer**) contenant les valeurs suivantes : "[1,2,3,4,5,6,7,8,9]" (dans cet ordre), laquelle des listes qui suivent serait retournée par un appel à la méthode **récursive SinglyLinkedList foo()**. Encerclez la bonne réponse.

**A.** `[1,2,3,4,5,6,7,8,9]` ;

**B.** `[1,2]` ;

**C.** `[2,5,8,7,4,1]` ;

**D.** `[1,4,7,9,6,3]` ;

**E.** `[3,6,9,7,4,1]` ;

**F.** `[1,4,7,8,5,2]` ;

**G.** `[2,1]` ;

**H.** `[2,4,8,9,3,1]` ;

**I.** `[9,8,7,6,5,4,3,2,1]` ;

**J.** `[]`.

```
public SinglyLinkedList foo() {
    SinglyLinkedList answer;
    answer = new SinglyLinkedList();
    foo( first, 0, answer );
    return answer;
}

private static void foo( Node p, int index, SinglyLinkedList answer ) {
    if ( p == null ) {
        return;
    } else {
        if ( index % 3 == 0 ) {
            answer.addFirst( p.value );
        }
        foo( p.next, index+1, answer );
        if ( index % 3 == 1 ) {
            answer.addFirst( p.value );
        }
        return;
    }
}
```

L'implémentation de la classe **SinglyLinkedList** se trouve à la page suivante.

```
public class SinglyLinkedList {

    // Des objets de la classe statique imbriquée Node servent à créer
    // la structure de cette liste chaînée.

    private static class Node {
        private Object value;
        private Node next;
        private Node( Object value, Node next ) {
            this.value = value;
            this.next = next;
        }
    }

    // Le premier noeud de cette liste chaînée.

    private Node first;

    // Ajout d'un élément au début de la liste.

    public void addFirst( Object item ) {
        first = new Node( item, first );
    }

    // Redéfinition de la méthode String toString().

    public String toString() {
        StringBuffer answer = new StringBuffer( "[" );
        Node p = first;
        while ( p != null ) {
            if ( p != first ) {
                answer.append( "," );
            }
            answer.append( p.value );
            p = p.next;
        }
        answer.append( "]" );
        return answer.toString();
    }
}
```

# Question 7 : zip (15 points)

Complétez l'implémentation de la méthode **LinkedList zip( Operator op, LinkedList l1, LinkedList l2 )** de la page suivante.

- Retourne un nouvel objet **LinkedList** ayant la même longueur que chacune des deux listes passées en paramètre. Un élément à la position $i$ dans cette liste est le résultat de l'application de l'opérateur **op** aux éléments se trouvant la position $i$ dans chacune des deux listes ;
- L'interface **Operator** est définie comme suit.

```
public interface Operator {
    public abstract Object apply( Object a, Object b );
}
```

- Lance l'exception **IllegalArgumentException** si les deux listes sont de longueurs différentes ;
- Les objets désignés par les paramètres de type **LinkedList** demeurent inchangés par l'appel à **zip** ;
- La méthode **zip** est implémentée à l'extérieur de la classe **LinkedList**, ainsi vous n'avez accès qu'aux méthodes publiques, dont voici la liste :
  - **LinkedList()** ; constructeur ;
  - **void addFirst( Object item )** ; ajout au début de la liste ;
  - **void addLast( Object item )** ; ajout à la fin de la liste ;
  - **void deleteFirst()** ; retrait du premier élément ;
  - **boolean isEmpty()** ; retourne **true** si et seulement si cette liste est vide ;
  - **Object head()** ; retourne une référence vers le premier objet de la liste ;
  - **LinkedList split()** ; retourne le reste de cette liste, cette liste ne contient alors qu'un seul élément ;
  - **void join( LinkedList other )** ; ajoute tous les éléments de la liste **other** à la suite des éléments de cette liste, **other** est alors vide ;
- Étant donné deux listes d'entiers (objets de la classe **Integer**) **l1** et **l2** :

```
l1 is [1,3,5,7,9]
l2 is [0,2,4,6,8]
```

  L'exécution de **l3 = zip( new Plus(), l1, l2 )** produira une liste dont les éléments sont la somme des éléments se trouvant à la même position dans chacune des listes d'entrée ; **l1** et **l2** demeurent inchangées.

```
l3 is [1,5,9,13,17]
```

```
public static LinkedList zip( Operator op, LinkedList l1, LinkedList l2 ) {

    LinkedList answer;

    if ( ______________________________ ) {
        throw new IllegalArgumentException( "first list is shorter" );
    }
    if ( ______________________________ ) {
        throw new IllegalArgumentException( "second list is shorter" );
    }

    if ( l1.isEmpty() && l2.isEmpty() ) {

        answer = new LinkedList();

    } else {

        LinkedList t1, t2;

        t1 = ______________________________;

        t2 = ______________________________;

        answer = zip( op, __________, __________ );

        Object current = ______________________________;

        answer.______________( current );

        ______________________________;

        ______________________________;
    }
    return answer;
}
```

# Question 8 : getLeavesCount (10 points)

Pour la classe **BinarySearchTree**, implémentez la méthode d'instance **int getLeavesCount()**. Elle retourne un entier égal au nombre de feuilles de cet arbre binaire.

```
public class BinarySearchTree {

    // Des objets la classe imbriquée statique Node sont utilisés afin
    // de créer la structure de cet arbre binaire.

    private static class Node {
        private Comparable value;
        private Node left;
        private Node right;
        private Node( Comparable value ) {
            this.value = value;
            left = null;
            right = null;
        }
    }

    private Node root = null;










} // End of BinarySearchTree
```

# (page blanche)