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 à l'informatique II (ITI 1521)
Examen final

Instructeur: Marcel Turcotte

Avril 2008, durée: 3 heures

Identification

Nom : _____ Prénom : _____

Numéro d'étudiant : _____ Signature : _____

Consignes

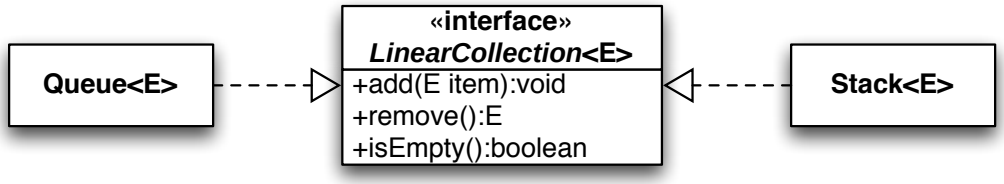
1. Livres fermés ;
2. Sans calculatrice ou toute autre forme d'aide ;
3. Commentez vos réponses afin d'obtenir des points partiels ;
4. Écrivez lisiblement, votre note en dépend ;
5. Répondez sur ce questionnaire, utilisez le verso des pages si nécessaire, il y a aussi deux pages blanches à la fin du questionnaire, mais vous ne pouvez remettre aucune page additionnelle ;
6. **Ne retirez pas l'agrafe**.

Barème

Question	Points	Cote
1	10	
2	15	
3	15	
4	10	
5	5	
6	10	
7	15	
8	10	
Total	**90**	

Question 1 (10 points)

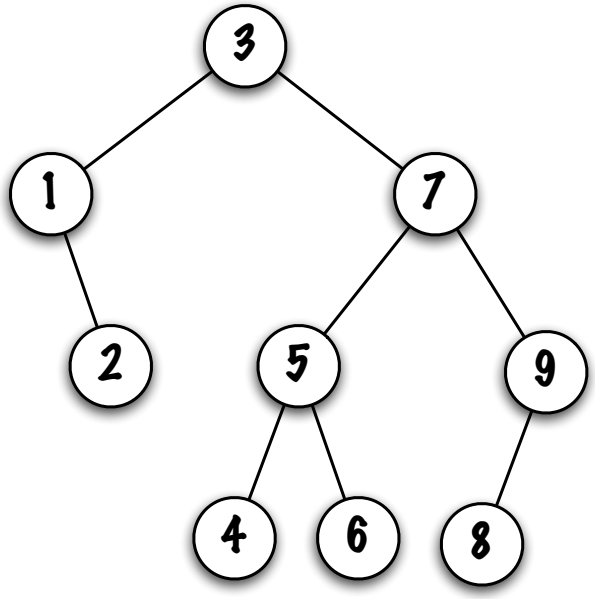
Ici, les classes **Queue** et **Stack** implémentent toutes les deux l'interface **LinearCollection**.



Queue, comme toutes autres implémentations d'une file, est telle que sa méthode **add** ajoute l'**item** à l'arrière de la file, la méthode **remove** retire et retourne l'élément avant, et la méthode **isEmpty** retourne **true** si cette file ne contient aucun élément.

Stack, comme toutes autres implémentations d'une pile, est telle que sa méthode **add** ajoute l'**item** sur le dessus de la pile, la méthode **remove** retire et retourne l'élément du dessus, et la méthode **isEmpty** retourne **true** si cette pile ne contient aucun élément.

Les énoncés ci-dessous ont servi pour la création d'un arbre binaire de recherche et l'ajout d'éléments ; l'arbre ainsi créé est représenté à droite.




```
BinarySearchTree<Integer> t;
t = new BinarySearchTree<Integer>();

t.add( 3 );
t.add( 1 );
t.add( 7 );
t.add( 9 );
t.add( 5 );
t.add( 4 );
t.add( 6 );
t.add( 2 );
t.add( 8 );
```

A. Encerclez la réponse qui correspond au résultat de l'appel qui suit :

```
t.traverse( new Queue< Node<Integer> >() );
```

A. 1 2 3 4 5 6 7 8 9
B. 2 1 4 6 5 8 9 7 3
C. 2 3 4 5 5 8 9 7 3
D. 3 1 7 2 5 9 4 6 8
E. 3 1 2 7 5 4 6 9 8
F. 3 7 1 9 5 2 8 6 4
G. 3 7 9 8 5 6 4 1 2
H. 8 6 4 9 5 2 7 1 3
I. 9 8 7 6 5 4 3 2 1

B. Encerclez la réponse qui correspond au résultat de l'appel qui suit :

```
t.traverse( new Stack< Node<Integer> >() );
```

A. 1 2 3 4 5 6 7 8 9
B. 2 1 4 6 5 8 9 7 3
C. 2 3 4 5 5 8 9 7 3
D. 3 1 7 2 5 9 4 6 8
E. 3 1 2 7 5 4 6 9 8
F. 3 7 1 9 5 2 8 6 4
G. 3 7 9 8 5 6 4 1 2
H. 8 6 4 9 5 2 7 1 3
I. 9 8 7 6 5 4 3 2 1

Le code source de la méthode **traverse** se trouve à la page qui suit.

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

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

    private Node<E> root = null;

    public void traverse( LinearCollection< Node<E> > store ) {

        if ( root != null ) {

            store.add( root );

            while ( ! store.isEmpty() ) {

                Node<E> current = store.remove();

                System.out.print( " " + current.value );

                if ( current.left != null ) {
                    store.add( current.left );
                }

                if ( current.right != null ) {
                    store.add( current.right );
                }

            }

            System.out.println();
        }

    }

    public boolean add( E obj ) { ... }

}
```

Question 2 (15 points)

Le type abstrait de données (TAD) **Deque** ("Double-Ended QUEue") combine à la fois les caractéristiques d'une file et d'une pile. En particulier, le TAD **Deque** — prononcé «deck» — permet

- des insertions efficaces à l'avant et à l'arrière de la structure de données ;
- des retraits efficaces à l'avant et à l'arrière de la structure de données.

Vous trouverez ci-dessous une implémentation complète de la classe **Deque** qui utilise un tableau circulaire pour la sauvegarde des éléments, ainsi qu'une variable d'instance, **size**, qui représente la taille logique. Voici une description des quatre méthodes d'accès de cette classe.

boolean offerFirst(E item) : ajoute un **item** à l'**avant** de ce **Deque**, retourne **true** si l'ajout est un succès ;

boolean offerLast(E item) : ajoute un **item** à l'**arrière** de ce **Deque**, retourne **true** si l'ajout est un succès ;

E pollFirst() : retire et retourne l'**item avant** de ce **Deque**, retourne **null** si ce **Deque** était vide ;

E pollLast() : retire et retourne l'**item arrière** de ce **Deque**, retourne **null** si ce **Deque** était vide.

```
public class Deque<E> {

    private E[] elems;
    private int size = 0, front, rear, capacity;

    public Deque( int capacity ) {
        this.capacity = capacity;
        elems = (E[]) new Object[ capacity ];
        front = 0;
        rear  = capacity-1;
    }

    public int size() {
        return size;
    }

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

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

    // se poursuit à la page qui suit...
```

```
// ... suite

public boolean offerFirst( E obj ) {
    boolean added = false;
    if ( size < capacity ) {
        front = ( front + ( capacity - 1 ) ) % capacity;
        elems[ front ] = obj;
        added = true;
        size++;
    }
    return added;
}

public boolean offerLast( E obj ) {
    boolean added = false;
    if ( size < capacity ) {
        rear = (rear + 1) % capacity;
        elems[ rear ] = obj;
        added = true;
        size++;
    }
    return added;
}

public E pollFirst() {
    E obj = null;
    if ( size > 0 ) {
        obj = elems[ front ];
        elems[ front ] = null;
        front = (front + 1) % capacity;
        size--;
    }
    return obj;
}

public E pollLast() {
    E obj = null;
    if ( size > 0 ) {
        obj = elems[ rear ];
        elems[ rear ] = null;
        rear = ( rear + ( capacity - 1 ) ) % capacity;
        size--;
    }
    return obj;
}

// se poursuit à la page qui suit...
```

```
    // ... suite

    public void dump() {

        System.out.println( "front = " + front );
        System.out.println( "rear = " + rear );

        for ( int i=0; i<elems.length; i++ ) {
            System.out.print( "  elems[" + i + "] = " );
            if ( elems[ i ] == null ) {
                System.out.println( "null" );
            } else {
                System.out.println( elems[ i ] );
            }
        }

        System.out.println();
    }

    public static void main( String[] args ) {
        // ...
    }
} // Fin de Deque
```

Pour chacun des (5) blocs de code qui suivent, donnez le résultat de l'appel à la méthode d.dump(). Inscrivez vos réponses dans les boîtes.

```
// Bloc 1

int n = 6; int num=1;

Deque<String> d = new Deque<String>( n );

for ( int i=0; i<4; i++ ) {
    d.offerLast( "item-" + num++ );
}

for ( int i=0; i<2; i++ ) {
    d.pollFirst();
}

for ( int i=0; i<2; i++ ) {
    d.offerLast( "item-" + num++ );
}

System.out.println( d );
d.dump();
```

```
// Bloc 2

int n = 4; int num = 1;

d = new Deque<String>( n );

for ( int i=0; i<4; i++ ) {
    d.offerLast( "item-" + num++ );
}

for ( int i=0; i<3; i++ ) {
    d.pollFirst();
}

for ( int i=0; i<2; i++ ) {
    d.offerLast( "item-" + num++ );
}

System.out.println( d );
d.dump();
```

```
// Bloc 3

int n = 4; int num = 1;

d = new Deque<String>( n );

for ( int i=0; i<4; i++ ) {
    d.offerLast( "item-" + num++ );
}

for ( int i=0; i<3; i++ ) {
    d.pollLast();
}

for ( int i=0; i<2; i++ ) {
    d.offerFirst( "item-" + num++ );
}

System.out.println( d );
d.dump();
```

```
// Bloc 4

int n = 4; int num = 1;

d = new Deque<String>( n );

for ( int i=0; i<4; i++ ) {
    d.offerLast( "item-" + num++ );
}

for ( int i=0; i<3; i++ ) {
    d.pollFirst();
}

for ( int i=0; i<2; i++ ) {
    d.offerLast( "item-" + num++ );
}

for ( int i=0; i<2; i++ ) {
    d.pollFirst();
}

System.out.println( d );
d.dump();
```

```
// Bloc 5

int n = 4; int num = 1;

d = new Deque<String>( n );

for ( int i=0; i<6; i++ ) {
    d.offerLast( "item-" + num++ );
}

for ( int i=0; i<3; i++ ) {
    d.pollLast();
}

for ( int i=0; i<2; i++ ) {
    d.offerFirst( "item-" + num++ );
}

for ( int i=0; i<2; i++ ) {
    d.pollLast();
}

System.out.println( d );
d.dump();
```

Question 3 (15 points)

Complétez l'implémentation de la méthode d'instance **void insertAfter(E obj, LinkedList<E> other)**. Cette méthode insère le contenu d'**other** après l'occurrence la plus à gauche d'**obj** dans cette liste, les éléments sont retirés d'**other**.

Une exception de type **IllegalArgumentException** sera lancée si la valeur d'**obj** est **null**, ou si l'objet est absent de cette liste. Voici les caractéristiques de cette implémentation.

- L'instance débute toujours par un noeud factice. Ce dernier marque le début de la liste. On n'y sauvegarde jamais d'élément. Une liste vide ne comprend que le noeud factice ;
- Les noeuds de la liste sont doublement chaînés ;
- La liste est circulaire, c'est-à-dire que la référence **next** du dernier noeud désigne le noeud factice, la référence **previous** du noeud factice désigne le dernier noeud de la liste. Si la liste est vide, le noeud factice sera à la fois le premier et le dernier noeud de la liste, ses références **previous** et **next** pointeront vers ce noeud unique ;
- Puisqu'on accède facilement au dernier élément de la liste, en effet, c'est le noeud qui précède le noeud factice, l'en-tête de la liste ne possède pas de pointeur arrière.

Exemple : le contenu de **xs** est [**a,b,c,f**], le contenu de **ys** est [**d,e**], à la suite de l'appel **xs.insertAfter("c", ys)**, le contenu de **xs** sera [**a,b,c,d,e,f**], et **ys** sera vide.

Inscrivez votre réponse dans la classe **LinkedList** qui se trouve à la page qui suit. **Vous ne devez pas utiliser les méthodes de la classe LinkedList pour la conception de cette méthode. En particulier, vous ne devez pas utiliser les méthodes add() et remove().**

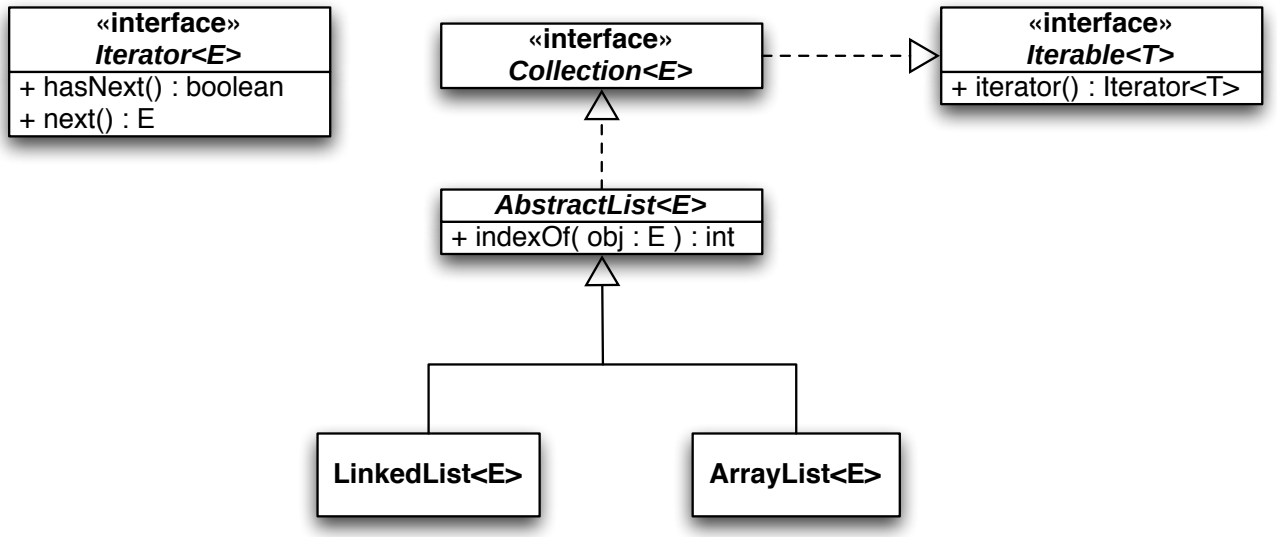
Suggestion : dessinez des diagrammes de mémoire détaillés.

```
public class LinkedList<E> {
    private static class Node<T> { // noeuds doublement chaînés
        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 int size;
    public LinkedList() {
        head = new Node<E>( null, null, null );
        head.next = head.previous = head;
        size = 0;
    }

    public void insertAfter( E obj, LinkedList<E> other ) {
```

```
    } // Fin de insertAfter
} // Fin de LinkedList
```

Question 4 (10 points)



A. Dans la classe abstraite **AbstractList**, vous devez implémenter la méthode d'instance **int indexOf(E obj)**. Le diagramme UML ci-haut résume les informations nécessaires pour répondre à cette question.

La méthode **int indexOf(E obj)** retourne l'index de l'occurrence la plus à gauche de l'objet spécifié, ou -1 si l'objet est absent de la liste.

B. Effectuez tous les changements nécessaires afin qu'**Iterator** et **LinkedList** déclarent et implémentent la méthode d'instance **int nextIndex()**.

La méthode **int nextIndex()** retourne l'index de l'élément qui serait retourné par un appel à la méthode **next**, ou la taille de la liste si l'itérateur est positionné en fin de liste.

Si vous le désirez, vous pouvez ajouter des variables d'instance à la classe **LinkedListIterator**, mais pas à la classe **LinkedList**.

Les noeuds de la liste (**LinkedList**) sont simplement chaînés. Il n'y a pas de noeud factice.

A. Vous n'avez pas à donner la déclaration de la classe, vous n'avez qu'à implémenter la méthode **indexOf** dans l'espace ci-dessous.

B. Implémentez la méthode **int nextIndex()**, utilisez la page qui suit au besoin.

```
1  public interface Iterator<T> {
2      public abstract boolean hasNext();
3      public abstract T next();
4  }
```

```
1  import java.util.NoSuchElementException;
2
3  public class LinkedList<E> extends AbstractList<E> {
4
5      private static class Node<T> {
6
7          private T value;
8          private Node<T> next;
9
10         private Node( T value, Node<T> next ) {
11             this.value = value;
12             this.next = next;
13         }
14     }
15
16     private Node<E> first = null;
17
18     private class LinkedListIterator implements Iterator<E> {
19
20         private Node<E> current = null;
21
22         public boolean hasNext() {
23             return ( ( ( current == null ) && ( first != null ) ) ||
24                      ( ( current != null ) && ( current.next != null ) ) );
25         }
26
27         public E next() {
28
29             if ( current == null ) {
30                 current = first;
31             } else {
32                 current = current.next;
33             }
34
35             if ( current == null ) {
36                 throw new NoSuchElementException();
37             }
38
39             return current.value;
40         }
41
42     } // Fin de LinkedListIterator
43
44     public Iterator<E> iterator() {
45         return new LinkedListIterator();
46     }
47
48     // Les autres méthodes de la classe LinkedList seraient ici
49
50 } // Fin de LinkedList
```

(**Question 4 suite**)

Question 5 (5 points)

Quel énoncé caractérise l'exécution de la méthode principale du programme qui suit ? Encerclez votre choix.

A. Affiche []
B. Affiche [a,c,e]
C. Affiche [e,c,a]
D. Affiche [a,b,c,d,e]
E. Affiche [e,d,c,b,a]
F. Affiche [a,a,a,a,a]
G. Affiche [e,e,e,e,e]
H. Affiche [a,a,b,b,c,c,d,d,e,e]
I. Affiche [e,e,d,d,c,c,b,b,a,a]
J. Il y aura un débordement de pile
K. Aucune des réponses ci-dessus

```
public class LinkedList< E > {
    private static class Node<T> { // noeuds simplement chaînés
        private T value;
        private Node<T> next;
        private Node( T value, Node<T> next ) {
            this.value = value;
            this.next  = next;
        }
    }

    private Node<E> first; // variable d'instance

    public void addFirst( E item ) {
        first = new Node<E>( item, first );
    }

    public static void main( String[] args ) {

        LinkedList<String> xs = new LinkedList<String>();

        xs.addFirst( "e" );
        xs.addFirst( "d" );
        xs.addFirst( "c" );
        xs.addFirst( "b" );
        xs.addFirst( "a" );

        xs.f();

        System.out.println( xs );
    }

    // se poursuit à la page qui suit...
```

```
    public void f() {
        f( true, first );
    }

    private void f( boolean predicate, Node<E> current ) {

        if ( current == null ) {
           return;
        }

        if ( predicate ) {
            current.next = new Node<E>( current.value, current.next );
        }

        f( ! predicate, current.next );

        return;
    }

    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 6 (10 points)

Complétez l'implémentation de la méthode **removeAll(Sequence<E> l, E obj)**. Elle retire toutes les occurrences de **obj** de la liste **l**. Son implémentation **doit être récursive**. La classe **Sequence** est une liste chaînée qui possède un ensemble de méthodes pour la conception de méthodes récursives efficaces à l'extérieur de la classe. Voici les caractéristiques de cette classe.

- **boolean isEmpty()** ; retourne **true** si et seulement si cette liste est vide ;
- **E head()** ; retourne une référence vers l'objet sauvegardé dans le premier noeud de cette liste ;
- **Sequence<E> split()** ; retourne le reste de la liste (nouvel objet **Sequence**), l'instance ne contient alors qu'un seul élément. La méthode lance l'exception **IllegalStateException** si la liste était vide au moment de l'appel ;
- **void join(Sequence<E> other)** ; ajoute les éléments de **other** à la suite des éléments de l'instance, **other** est vide à la suite de cet appel.

```
public class Q6 {
    public static <E> void removeAll( Sequence<E> l, E obj ) {









    // Fin de removeAll
// Fin de Q6
```

Question 7 (15 points)

Vous trouverez ci-dessous une implémentation partielle d'un arbre binaire de recherche (**BinarySearchTree**), vous devez y implémenter les méthodes **isLeaf** et **getHeight**.

A. **boolean isLeaf(E obj)** : cette méthode d'instance retourne **true** si le noeud qui contient **obj** est une feuille, et **false** sinon. La méthode lance l'exception **IllegalArgumentException** si la valeur de **obj** est **null**. La méthode lance l'exception **NoSuchElementException** si **obj** est absent de cet arbre. (8 points)

B. **int getHeight()** : cette méthode d'instance retourne la hauteur de l'abre. N'ajoutez aucune nouvelle variable d'instance. (7 points)

```
import java.util.NoSuchElementException;

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

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

    private Node<E> root = null;
```

(Question 7 suite)

Question 8 (10 points)

A. Quel énoncé décrit le mieux le bloc de code qui suit ? Encerclez votre choix. (3 points)

(a) Affiche "c = 0" ;

(b) Affiche "c = Infinity" ;

(c) Affiche "** caught ArithmeticException **", "c = 3" ;

(d) Affiche "** caught Exception **", "c = 2" ;

(e) Erreur de compilation : "exception java.lang.ArithmeticException has already been caught" ;

(f) Produira une erreur d'exécution, ainsi qu'une trace d'exécution.

```
int a = 1, b = 0, c = 0;
try {
    c = a/b;
} catch ( Exception e ) {
    System.err.println( "** caught Exception **" );
    c = 2;
} catch ( ArithmeticException ae ) {
    System.err.println( "** caught ArithmeticException **" );
    c = 3;
}
System.out.println( "c = " + c );
```

B. Créez un nouveau type d'exception validée («checked») nommé **MyException**. (3 points)

C. Modifiez le programme qui suit afin d'éliminer les erreurs de compilation. (4 points)

```
1  import java.io.*;
2
3  public class Q8 {
4
5      public static String cat( String fileName ) {
6
7          FileInputStream fin = new FileInputStream( fileName );
8
9          BufferedReader input = new BufferedReader( new InputStreamReader( fin ) );
10
11         StringBuffer buffer = new StringBuffer();
12
13         String line = null;
14
15         while ( ( line = input.readLine() ) != null ) {
16
17             line = line.replaceAll( "\\s+", " " );
18
19             buffer.append( line );
20
21         }
22
23         fin.close();
24
25         return buffer.toString();
26
27     } // End of cat
28
29     public static void main( String[] args ) {
30
31         System.out.println( cat( args[ 0 ] ) );
32
33     }
34 }
```

où

- **FileInputStream(name)** lance **FileNotFoundException** («checked exception») si le fichier n'existe pas, est un répertoire plutôt qu'un fichier régulier, ou pour toute autre erreur de lecture ;
- **readLine()** lance **IOException** («checked exception») si une erreur de lecture survient ;
- **line.replaceAll(expression, replacement)**, ici, cet appel remplace plusieurs occurrences successives du caractère blanc par un seul caractère blanc. La méthode lance l'exception **PatternSyntaxException** («unchecked exception») si la syntaxe du premier paramètre (une expression régulière) n'est pas valide ;
- **close()** lance **IOException** («checked exception») s'il y a une erreur lors de la fermeture du fichier.

(page blanche)

(page blanche)