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 to Computer Science II (ITI 1121)
Final Examination

Instructor: Marcel Turcotte

April 2008, duration: 3 hours

Identification

Student name: ______________________________

Student number: ______________ Signature: ______________________

Instructions

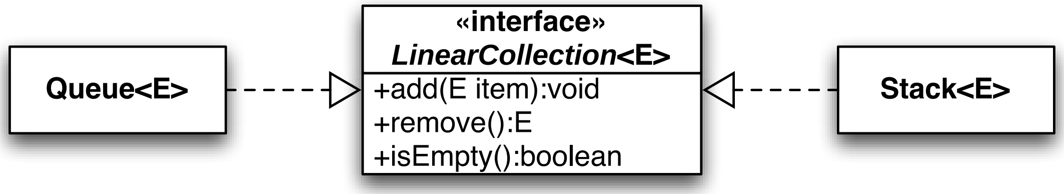
1. This is a closed book examination;
2. No calculators or other aids are permitted;
3. Write comments and assumptions to get partial points;
4. Beware, poor hand writing can affect grades;
5. **Do not remove the staple holding the examination pages together**;
6. Write your answers in the space provided. Use the backs of pages if necessary. There are two blank pages at the end. You may **not** hand in additional pages.

Marking scheme

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

Question 1 (10 points)

For this question, the classes **Queue** and **Stack** both implement the interface **LinearCollection**.



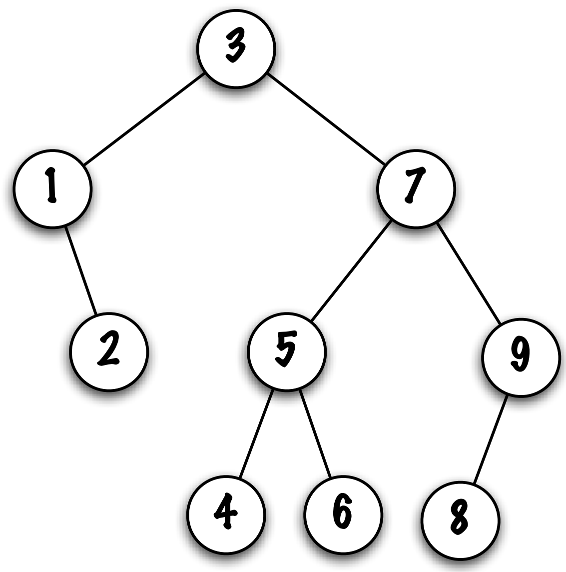
Queue, as all the other implementations of a queue, is such that the method **add** enqueues the **item** at the rear of the queue, the method **remove** dequeues the front element, and the method **isEmpty** returns **true** if this queue contains no elements.

Stack, as all the other implementations of a stack, is such that the method **add** pushes the **item** onto the top of the stack, the method **remove** pops (removes and returns) the top element, and the method **isEmpty** returns **true** if this stack contains no elements.

A **BinarySearchTree** was created by adding elements in the order that follows; the resulting tree is shown to the right.

```
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. Circle the answer that corresponds to the following method call:

```
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. Circle the answer that corresponds to the following method call:

```
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

The source code for the method **traverse** can be found on the next page.

```
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)

The abstract data type **Deque** — pronounced "deck" — combines features of both a queue and a stack. In particular, a **Deque** ("Double-Ended QUEue") allows for

- efficient insertions at the front or rear;
- efficient deletions at the front or the rear.

Below, you will find a complete implementation of the class **Deque** that uses a circular array to store its elements, and has an instance variable, **size**, to keep track of the number of elements. Here are the descriptions of the four main methods of this class.

boolean offerFirst(E item): adds an item at the **front** of this **Deque**, returns **true** if the item was successfully added;

boolean offerLast(E item): adds an item at the **rear** of this **Deque**, returns **true** if the item was sucessfully added;

E pollFirst(): removes and returns the **front** item of this **Deque**, returns **null** if this **Deque** was empty;

E pollLast(): removes and returns the **rear** item of this **Deque**, returns **null** if this **Deque** was empty.

```
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;
    }

    // continues on the next page...
```

```
// ... continues

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;
}

// continues on the next page...
```

```
    // ... continues

    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 ) {
        // ...
    }
} // End of Deque
```

For each of the following (5) blocks of code, give the result of the call to the method d.dump(). Write your answers in the boxes.

```
// Block 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();
```

```
// Block 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();
```

```
// Block 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();
```

```
// Block 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();
```

```
// Block 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)

Complete the implementation of the instance method **void insertAfter(E obj, LinkedList<E> other)**. The method inserts the content of **other** after the leftmost occurrence of **obj** in this list, the elements are removed from **other**.

An exception, **IllegalArgumentException**, is thrown if **obj** is **null**, or the parameter **obj** is not found in this list. The implementation of **LinkedList** has the following characteristics.

- An instance always starts off with a dummy node, which serves as a marker for the start of the list. The dummy node is never used to store data. The empty list consists of the dummy node only;
- In the implementation for this question, the nodes of the list are doubly linked;
- In this implementation, the list is circular, i.e. the reference **next** of the last node of the list is pointing at the dummy node, the reference **previous** of the dummy node is pointing at the last element of the list. In the empty list, the dummy node is the first and last node of the list, its references **previous** and **next** are pointing at the node itself;
- Since the last node is easily accessed, because it is always the previous node of the dummy node, the header of the list does not have (need) a tail pointer.

Example: **xs** contains [**a,b,c,f**], **ys** contains [**d,e**], after the call: **xs.insertAfter("c", ys)**, **xs** contains [**a,b,c,d,e,f**], and **ys** is empty.

Write your answer in the class **LinkedList** on the next page. **You cannot use the methods of the class LinkedList. In particular, you cannot use the methods add() or remove().**

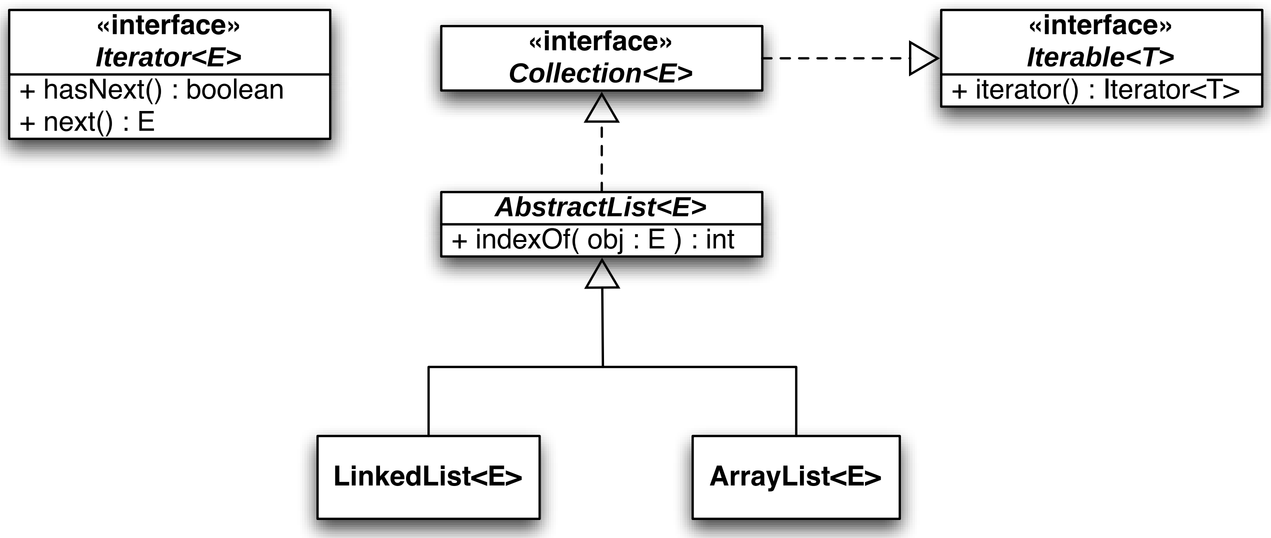
Hint: draw detailed memory diagrams.

```
public class LinkedList<E> {
    private static class Node<T> { // implementation of the doubly linked nodes
        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 ) {
```

```
    } // End of insertAfter
} // End of LinkedList
```

Question 4 (10 points)



A. In the abstract class **AbstractList**, implement the instance method **int indexOf(E obj)**. The known information about the class **AbstractList** is summarized in the above UML diagram.

The method **int indexOf(E obj)** returns the index of the leftmost occurrence of the specified element in this list, or -1 if this list does not contain the element.

B. Make **all** the necessary changes so that **Iterator** and **LinkedList** declare and implement the instance method **int nextIndex()**.

The method **int nextIndex()** returns the index of the element that would be returned by a subsequent call to **next**, or the size of the list if the iterator is at end of list.

You may add new instance variables to the class **LinkedListIterator** if you wish so, but not to the class **LinkedList**.

The class **LinkedList** implements a singly linked list. There are **no** dummy nodes.

A. You don't need to write the class declaration, write only the implementation of the method **indexOf**. Write your answer in the space below.

B. Implement the method **int nextIndex()**, additional space is available on the next page.

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

```
import java.util.NoSuchElementException;

public class LinkedList<E> extends AbstractList<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> first = null;

    private class LinkedListIterator implements Iterator<E> {

        private Node<E> current = null;

        public boolean hasNext() {
            return ( ( ( current == null ) && ( first != null ) ) ||
                     ( ( current != null ) && ( current.next != null ) ) );
        }

        public E next() {

            if ( current == null ) {
                current = first;
            } else {
                current = current.next;
            }

            if ( current == null ) {
                throw new NoSuchElementException();
            }

            return current.value;
        }

    } // End of LinkedListIterator

    public Iterator<E> iterator() {
        return new LinkedListIterator();
    }

    // The other methods of the class LinkedList would be here

} // End of LinkedList
```

(**Question 4 continues**)

Question 5 (5 points)

Which of these statements characterizes the execution of the main method of the following program.

A. Prints []
B. Prints [a,c,e]
C. Prints [e,c,a]
D. Prints [a,b,c,d,e]
E. Prints [e,d,c,b,a]
F. Prints [a,a,a,a,a]
G. Prints [e,e,e,e,e]
H. Prints [a,a,b,b,c,c,d,d,e,e]
I. Prints [e,e,d,d,c,c,b,b,a,a]
J. Causes a stack overflow exception
K. None of the above

```
public class LinkedList< E > {
    private static class Node<T> { // singly linked nodes
        private T value;
        private Node<T> next;
        private Node( T value, Node<T> next ) {
            this.value = value;
            this.next  = next;
        }
    }

    private Node<E> first; // instance variable

    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 );
    }

    // continues on the next page
```

```
    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)

Complete the implementation of the method **removeAll(Sequence<E> l, E obj)**. It removes all the occurrences of **obj** in **l**. Its implementation **must be recursive**. The class **Sequence** is a linked list with additional methods to write efficient recursive list processing methods outside of the implementation of the list. Here are the characteristics of the class **Sequence**. The methods of the class **Sequence** include.

- **boolean isEmpty()**; returns **true** if and only if **this** list is empty;
- **E head()**; returns a reference to the object stored in the first node of **this** list;
- **Sequence<E> split()**; returns the tail of **this** sequence, **this** sequence now contains a single element. It throws **IllegalStateException** if the **Sequence** was empty when the call was made;
- **void join(Sequence<E> other)**; appends **other** at the end of **this** sequence, **other** is now empty.

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











    // End of removeAll
// End of Q6
```

Question 7 (15 points)

For the partial implementation of the class **BinarySearchTree** below, implement the methods **isLeaf** and **getHeight**.

A. **boolean isLeaf(E obj)**: this instance method returns **true** if the node containing **obj** is a leaf, and **false** otherwise. The method throws **IllegalArgumentException** if **obj** is **null**. The method throws **NoSuchElementException** if **obj** is not found in this tree. (8 points)

B. **int getHeight()**: this instance method returns the height of this tree. Do not add any new instance variable. (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 continues)

Question 8 (10 points)

A. Which of these statements best characterizes the following block of code. (3 points)

(a) Prints “c = 0”;

(b) Prints “c = Infinity”;

(c) Prints “** caught ArithmeticException **”, “c = 3”;

(d) Prints “** caught Exception **”, “c = 2”;

(e) Does not compile: “exception java.lang.ArithmeticException has already been caught”;

(f) Produces a run time error, with a stack trace.

```
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. Create a new checked exception type called **MyException**. (3 points)

C. Modify the following program so that it will compile. (4 points)

```
import java.io.*;

public class Q8 {

    public static String cat( String fileName ) {

        FileInputStream fin = new FileInputStream( fileName );

        BufferedReader input = new BufferedReader( new InputStreamReader( fin ) );

        StringBuffer buffer = new StringBuffer();

        String line = null;

        while ( ( line = input.readLine() ) != null ) {

            line = line.replaceAll( "\\s+", " " );

            buffer.append( line );

        }

        fin.close();

        return buffer.toString();

    } // End of cat

    public static void main( String[] args ) {

        System.out.println( cat( args[ 0 ] ) );

    }
}
```

where

- **FileInputStream(name)** throws **FileNotFoundException** (a checked exception) if the file does not exist, is a directory rather than a regular file, or for some other reason cannot be opened for reading;
- **readLine()** throws **IOException** (a checked exception) if an I/O error occurs;
- **line.replaceAll(expression, replacement)**, here, this replaces consecutive white space characters by a single white space. It throws **PatternSyntaxException** (an unchecked exception) if the syntax of the first parameter (a regular expression) is not valid;
- **close()** throws **IOException** (a checked exception) if an I/O error occurs.

(blank page)

(blank page)