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 1221)
Final Examination

Instructor: Marcel Turcotte

April 2006, duration: 3 hours

Identification

Last name: ______________________ First 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 marks;
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.
   You may **not** hand in additional pages;

Marking scheme

Question	Maximum	Result
1	15	
2	15	
3	10	
4	15	
5	15	
6	5	
7	15	
8	10	
Total	**100**	

Question 1: isPalindrome (15 marks)

Complete the implementation of the static method **boolean isPalindrome(CharReader r)**. Let's define a **palindrome** as a word or a phrase that reads the same forward and backward if the punctuation symbols and spaces are ignored. Examples of palindromes include:

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

Follow all the directives.

- **boolean isPalindrome(CharReader r)**; returns **true** if the whole word or phrase specified by the reader is a palindrome according to the above definition, and **false** otherwise;
- The parameter of the method is a **CharReader**. A **CharReader** has two instance methods.
  - **boolean hasMoreChars()**; returns **true** if the reader has more characters to return, that is if a call to **char nextChar()** would succeed, and **false** otherwise;
  - **char nextChar()**; returns the next character of the input.
- You can only use instances of a **Stack** and/or a **Queue** as temporary storage (in particular, you cannot use arrays or strings);
- The class **StackImpl** implements the interface **Stack**. For this question, a **Stack** stores characters.

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

- The class **QueueImpl** implements the interface **Queue**. For this question, a **Queue** stores characters.

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

- **StackImpl** and **QueueImpl** can store an arbitrarily large number of characters;
- **Character.isLetter(c)** can be used to determine if the character **c** is a letter.

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

    boolean answer = true;






    while ( reader.hasMoreChars() ) {

        char c = reader.nextChar();
























    return answer;
}
```

Question 2: CircularStack (15 marks)

Complete the implementation of the class **CircularStack**. The context for this question is an application that is required to support a fixed number of **undo** operations. You can imagine a text editor that allows to add, delete or replace characters. For every operation that is performed (add, delete or replace) an object is pushed onto a stack. Whenever the application is required to undo an operation, it retrieves an element from the stack. However, since the stack has a fixed capacity, the maximum number of operations that can be undone is equal to the size of stack. Follow all the directives.

- Because of memory constraints, only a fixed number of undo operations are allowed;
- Whenever the stack is full, the method **push** discards the oldest (bottom) element to make room for the new element to be inserted;
- However, the method **push** should not move the elements that are currently stored in the stack. Instead, it overwrites the oldest (bottom) element. Notice the similarity with the circular array implementation of the **Queue** seen in class;
- **void push(Object o)**; pushes an element onto the top of **this** stack, **null** is a valid value;
- **Object pop()**; removes and returns the top element of the stack. If the stack is empty, the method must throw an exception of 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;
    }
```

Complete the implementation of the methods **push** and **pop** on the next page.

```
    public void push( Object item ) {















    }

    public Object pop() {
















    }

} // End of CircularStack
```

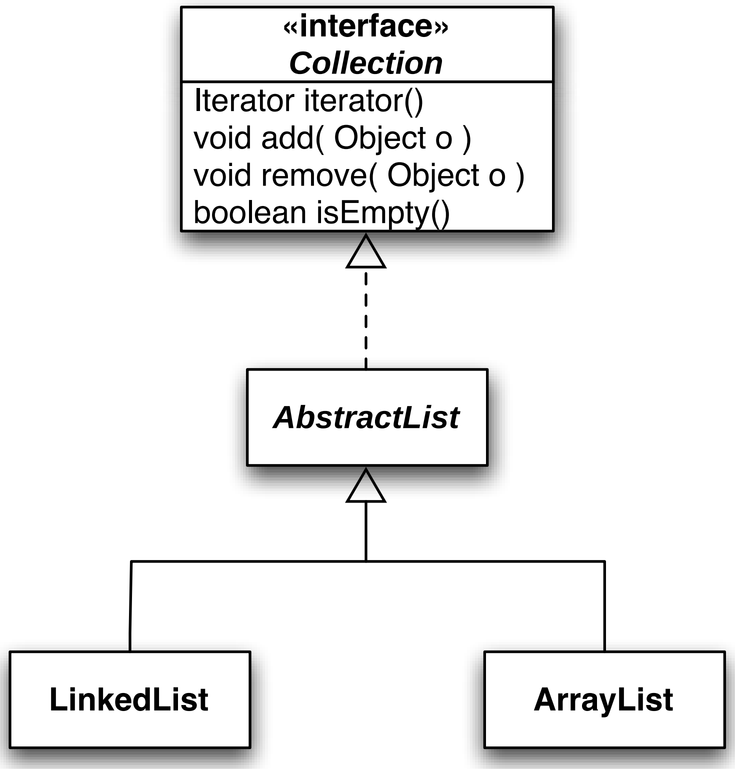
Question 3: ArrayListIterator (10 marks)

In the class **ArrayList** below, complete the implementation of the iterator. For this question, the declaration of the interface **Iterator** is as follows.

```
public interface Iterator {

    // Returns true if the iteration has more elements.

    public abstract boolean hasNext();

    // Returns the next element in the interation. Throws
    // NoSuchElementException if the iteration has no next element.

    public abstract Object next();
}

import java.util.NoSuchElementException;

public class ArrayList {

    // Instance variables
    private Object[] elems;
    private int size = 0;

    // Constructor
    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() { // implement hasNext()
            boolean answer;






            return answer;
        }

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











            return answer;
        }
    } // end of ArrayListIterator

} // end of ArrayList
```

Question 4: equals (15 marks)



In the abstract class **AbstractList** found on the next page, override the method **boolean equals(Object other)**. Follow all the directives for writing the method.

- Compares **other** with **this** list for equality;
- Returns **true** if and only if **other** is also an **AbstractList** (more precisely, the object designated by **other** is an instance of a subclass of **AbstractList**), both lists have the same size, and all the corresponding pairs of elements in the two lists are equal. Otherwise, the method returns **false**;
- The value **null** is a valid element;
- **AbstractList** implements the interface **Collection**;
- **LinkedList** and **ArrayList** are two examples of subclasses of **AbstractList** but there could be more;
- Use iterators to implement the method.

The declarations of the interfaces **Collection** and **Iterator** can be found on page 10.

```
public class AbstractList implements Collection {

    public boolean equals( Object other ) {




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

```
public interface Collection {

    /* Returns an iterator over the elements in this collection.
     */

    public abstract Iterator iterator();

    /* Add the item to the Collection and return true if the
     * collection changed as a result of this call.
     */

    public abstract boolean add( Object item );

    /* Removes a single instance of the specified element from this
     * collection, if it is present.  Returns true if this collection
     * changed as a result of the call.
     */

    public abstract boolean remove( Object item );

    /* Returns true if this collection contains no elements.
     */

    public abstract boolean isEmpty();
}

public interface Iterator {

    /* Returns true if the iteration has more elements.
     */

    public abstract boolean hasNext();

    /* Returns the next element in the interation. Throws
     * NoSuchElementException if the iteration has no next element.
     */

    public abstract Object next();
}
```

Question 5: splitAt (15 marks)

Complete the implementation of the instance method **LinkedList splitAt(int n)**. The method **splitAt** splits this **LinkedList** in two parts. The first **n** elements remain part of this list while the rest is returned in a new **LinkedList**. In particular,

- After the call **t = l.splitAt(0)**, **l** is empty and **t** contains all the elements that were initially present in **l**;
- After the call **t = l.splitAt(1)**, **l** contains one element and **t** contains all the elements that were initially present in **l** except one;
- After the call **t = l.splitAt(i)**, **l** contains **i** elements and **t** contains **size-i** elements, where **size** is the length of **l** before the call;
- After the call **t = l.splitAt(l.size())**, **l** is unchanged and **t** designates an empty **LinkedList**;
- An exception, **IllegalArgumentException**, is thrown if the parameter **n** is larger than the size of the list.

The implementation of the **LinkedList** has the same characteristics as the one of the assignment 4.

- This implementation 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, it is always the previous node of the dummy node, the header of the list does not need (have) a tail pointer.

Write your answer in the class **LinkedList** on the next page. No method calls are allowed.

Hint: draw the memory diagram for the special and general cases.

```
public class LinkedList {
    private static class Elem { // Implementation of the doubly linked nodes
        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 ( ______________________________ ) { // complete











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

Question 6: foo (5 marks)

The **recursive** method **SinglyLinkedList foo()** was applied to a list containing the following integers (objects of the class **Integer**): "[1,2,3,4,5,6,7,8,9]". Which of the following lists represents the result of the execution of the method **SinglyLinkedList foo()**? Circle the right answer.

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;
    }
}
```

The implementation of the class **SinglyLinkedList** can be found on the next page.

```
public class SinglyLinkedList {

    // Objects of the static nested class Node are used to create
    // the structure of the linked list.

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

    // The first Node of the linked list.

    private Node first;

    // Adds an element at the start of the list.

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

    // Override the method 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 marks)

Complete the implementation of the method **LinkedList zip(Operator op, LinkedList l1, LinkedList l2)** on the next page.

- Returns a new **LinkedList** that is of the same length as the two input lists and such that the values of this list are the result of applying the operator **op** to the elements at the respective position within each list;
- The interface **Operator** is defined as follows:

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

- Both arguments must be of the same length, otherwise an **IllegalArgumentException** is thrown;
- Both **LinkedList** arguments remain unchanged by a call to **zip**;
- The method **zip** is implemented outside of the class **LinkedList**. Here are the public methods that you can use to implement **zip**:
  - **LinkedList()**; constructor;
  - **void addFirst(Object item)**; adds **item** at the start of **this** list;
  - **void addLast(Object item)**; adds **item** at the end of **this** list;
  - **void deleteFirst()**; deletes the first element of **this** list;
  - **boolean isEmpty()**; returns **true** if and only if **this** list is empty;
  - **Object head()**; returns a reference to the object stored in the first node of **this** list;
  - **LinkedList split()**; returns the tail of **this** list, **this** list now contains a single element;
  - **void join(LinkedList other)**; appends **other** at the end of **this** list, **other** is now empty.
- Given two lists of integers (objects of the class **Integer**) **l1** and **l2**:

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

  The execution of **l3 = zip(new Plus(), l1, l2)** produces a list where each element is the sum of the elements at the respective position within each list; **l1** and **l2** remain unchanged:

```
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 marks)

For the class **BinarySearchTree**, implement the instance method **int getLeavesCount()**. It returns an integer equal to the number of leaves in this binary tree.

```
public class BinarySearchTree {

    // Objects of the static nested class Node are used to create
    // the structure of the binary tree.

    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
```

(blank space)