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 2007, 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 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	20	
2	10	
3	15	
4	15	
5	10	
6	10	
7	10	
8	10	
Total	**100**	

Question 1: Applications of queues (20 marks)

The context for this question is a finance software package called **JStock**. Corporations are raising funds by selling **shares** (equal portions of their capital). Collectively, the shares of a given corporation is called its **stock**. The shares prices vary, here on a daily basis. A shareholder makes a **capital gain** when selling shares if the selling price is higher than the price at which the shares were bought; or, suffers a **capital loss** if the selling price is lower than the price at which they were bought.

When selling shares, the calculation of the **capital gain** is easy if all the shares were purchased at the same price (e.g. a single transaction). However, the computation is more complex when selling shares acquired through several transactions. In that case, standard accounting principles dictate that the oldest shares must be sold first.

For example, a shareholder purchases 100 shares at \$20 each in a first transaction, then purchases 20 shares at \$24 each in a second transaction, then purchases 200 shares at \$36 each in a third transaction, and then sells 150 shares at \$30 each. In that case, the capital gain is $100 \times (30 - 20) + 20 \times (30 - 24) + 30 \times (30 - 36) = 940$ dollars.

JStock is a software package to help shareholders manage their portfolio. For simplicity, **JStock** holds shares of single stock (the shares of single corporation). However, the shares are generally acquired through several transactions.

All the shares that were purchased in a given transaction are kept in a **Transaction** object. The number of shares and the price for each one is specified when creating a new **Transaction** object. The class declares the methods **getShares** and **getSharePrice** that returns for a given transaction the number of shares and the price of each share, respectively. The class also implements a method to decrease the number of shares of a given transaction, the method is called **sell**, its parameter is the number of shares sold.

Complete the implementation of the class **JStock** on pages 4 and 5. **JStock** uses a queue to store the transactions of a given stock.

A. Implement the method **void buy(int num, int sharePrice)**. It adds a new transaction at the rear of the queue. The values of the parameters are used to create a new transaction;

B. Implement the method **int sell(int num, int sharePrice)**. It updates the queue of transactions so as to reduce the total number of shares by **num**, and returns the resulting capital gain or capital loss (a negative gain);

C. Implement the method **int getValue()** that returns the total value of the portfolio. This is the sum of the value of all the transactions. The value of a transaction is simply the product of the number of shares by the share price.

Note: when using a queue, you can only use the methods that are defined in its public interface.

For this question, there is a class called **LinkedQueue** that implements the interface **Queue** below.

```
public interface Queue<E> {

    /**
     * Returns true if this queue has no elements.
     *
     * @return true if this queue has no elements.
     */

    public abstract boolean isEmpty();

    /**
     * Returns a reference to the front element; does not change
     * the state of this queue.
     *
     * @return The front element of this queue without removing it.
     */

    public abstract E peek() throws EmptyQueueException;

    /**
     * Add an element at the rear of this queue.
     *
     * @throws FullQueueException if this queue is full.
     */

    public abstract void enqueue( E o ) throws QueueOverflowException;

    /**
     * Remove and returns the front element of this queue.
     *
     * @return the front element of this queue.
     * @throws EmptyQueueException if this queue contains no elements.
     */

    public abstract E dequeue() throws EmptyQueueException;
}
```

Here is the declaration of the class **Transaction**.

```
public class Transaction {

    private int shares;
    private int sharePrice;

    public Transaction( int shares, int sharePrice ) {
        this.shares = shares;
        this.sharePrice = sharePrice;
    }
    public int getShares() {
        return shares;
    }
    public void sell( int num ) {
        if ( num < 0 || num > shares ) {
            throw new IllegalArgumentException( Integer.toString( num ) );
        }
        shares = shares - num;
    }
    public int getSharePrice() {
        return sharePrice;
    }
}
```

Here is the declaration of the class **JStock**.

```
public class JStock {

    private Queue<Transaction> myShares;

    public JStock() {
        myShares = new LinkedQueue<Transaction>();
    }

    public void buy( int num, int sharePrice ) {









    } // End of buy
```

```
    public int sell( int num, int sharePrice ) {

    } // End of sell

    public int getValue() {

    } // End of getValue
} // End of JStock
```

Question 2: CircularQueue (10 marks)

Show the result that will be displayed on the screen for each of the following two calls to the method **dump** in the code fragment below. The class **CircularQueue** can be found on pages 7 and 8.

```
CircularQueue<Integer> q;
q = new CircularQueue<Integer>( 4 );

int i = 0;
while ( ! q.isFull() ) {
    i = i + 1;
    q.enqueue( new Integer( i ) );
}

if ( ! q.isEmpty() ) {
    q.dequeue();
}
if ( ! q.isEmpty() ) {
    q.dequeue();
}

while ( ! q.isFull() ) {
    i = i + 1;
    q.enqueue( new Integer( i ) );
}
q.dump();

while ( ! q.isEmpty() ) {
    q.dequeue();
}
q.dump();
```

First call:	**Second call:**

```
public class CircularQueue<E> implements Queue<E> {

    public static final int DEFAULT_CAPACITY = 100;
    private final int MAX_QUEUE_SIZE;
    private E[] elems;
    private int front, rear;

    public CircularQueue() {
        this( DEFAULT_CAPACITY );
    }
    public CircularQueue( int capacity ) {
        if ( capacity < 0 ) {
            throw new IllegalArgumentException( Integer.toString( capacity ) );
        }
        MAX_QUEUE_SIZE = capacity;
        elems = (E []) new Object[ MAX_QUEUE_SIZE ];
        front = 0;
        rear = -1; // Represents the empty queue
    }

    public boolean isEmpty() {
        return ( rear == -1 );
    }
    public boolean isFull() {
        return ( ! isEmpty() ) && nextIndex( rear ) == front;
    }
    private int nextIndex(int index) {
        return ( index+1 ) % MAX_QUEUE_SIZE;
    }

    public void dump() {

        System.out.println( "MAX_QUEUE_SIZE = " + MAX_QUEUE_SIZE );
        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 void enqueue( E o ) {

        if ( o == null ) {
            throw new IllegalArgumentException( "null" );
        }

        if ( isFull() ) {
            throw new QueueOverflowException();
        }

        rear = nextIndex( rear );

        elems[ rear ] = o;
    }

    public E dequeue() {

        if ( isEmpty() ) {
            throw new EmptyQueueException();
        }

        E result = elems[ front ];
        elems[ front ] = null; // ``scrubbing''

        if ( front == rear ) { // Following this call to dequeue
            front = 0;         // the queue will be empty
            rear = -1;
        } else {
            front = nextIndex( front );
        }

        return result;
    }

} // End of CircularQueue
```

Question 3: splitAfter (15 marks)

Complete the implementation of the instance method **LinkedList<E> splitAfter(E obj)**. The method **splitAfter** splits this **LinkedList** in two parts. The original list retains all the elements up to and including the left most occurrence of **obj** while the remaining elements are returned in a new **LinkedList**. An exception, **IllegalArgumentException**, is thrown if the parameter **obj** is not found in this list.

The implementation of **LinkedList** has the same characteristics as the one of the assignment 5.

- 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, because 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; except for calls to the constructors.

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

```
public class LinkedList<E> {
    private static class Node<E> { // Implementation of the doubly linked nodes
        private E value;
        private Node<E> previous;
        private Node<E> next;
        private Node( E value, Node<E> previous, Node<E> 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 LinkedList<E> splitAfter( E obj ) {
```

```
    } // End of splitAfter
} // End of LinkedList
```

Question 4: Iterator (15 marks)

The two parts of this question have to do with the class **LinkedList** (pages 13–14) and its **Iterator** (page 15).

A. For the inner class **LinkedListIterator** on the next page, implement the method **Iterator<E> copy()**. It returns a copy of this iterator;

B. For the class **Test** below, complete the implementation of the method **compress**. It transforms the input list so as to preserve a single copy of consecutive elements that are equals. Let **l** be a list that contains the following elements: "a,a,a,a,b,b,b,b,c,d,d,a". Following a call to the method **compress**, the content of the list **l** will be: "a,b,c,d,a".

- Iterators **must** be used to traverse the list;
- The **only** method of the class **LinkedList** that you can use is **Iterator<E> iterator()**;
- You can use all the methods of the **Iterator**.

```
public class Test {

    public static <E> void compress( LinkedList<E> xs ) {

    } // End of compress
} // End of Test
```

```
import java.util.ConcurrentModificationException;

public class LinkedList<E> {

    private class LinkedListIterator implements Iterator<E> {

        private Node<E> current;
        private int expectedModCount;

        private LinkedListIterator() {
            expectedModCount = modCount;
            current = head;
        }

        public E next() { ... }
        public boolean hasNext() { ... }
        public void remove() { ... }

        public Iterator<E> copy() {









       }

        private void checkConcurrentModification() {
            if ( expectedModCount != modCount ) {
                throw new ConcurrentModificationException();
            }
        }
    } // End of LinkedListIterator
```

```
    private static class Node<E> { // Doubly linked nodes
        private E value;
        private Node<E> previous;
        private Node<E> next;
        private Node( E value, Node<E> previous, Node<E> next ) {
            this.value = value;
            this.previous = previous;
            this.next = next;
        }
    }

    private Node<E> head; // Designates the dummy node
    private int modCount; // Implements the fail-fast technique

    // Constructor

    public LinkedList() {
        head = new Node<E>( null, null, null ); // Dummy node
        head.next = head.previous = head;
        modCount = 0;
    }

    // Returns an iterator for this list

    public Iterator<E> iterator() { ... }

    // All the other methods of LinkedList would be here but cannot be used

} // End of LinkedList
```

```
public interface Iterator<E> {

    /**
     * Returns true if the iteration has more elements. (In other
     * words, returns true if next would return an element rather than
     * throwing an exception.)
     *
     * @return true if the iterator has more elements.
     */

    public abstract boolean hasNext();

    /**
     * Returns the next element in the interation.
     *
     * @return the next element in the iteration.
     * @exception NoSuchElementException iteration has no more elements.
     */

    public abstract E next();

    /**
     * Removes from the list the last element that was returned by
     * next.  This call can only be made once per call to next.
     *
     * @exception IllegalStateException if next has not been called, or
     *            the number of element removed exceeds the number of time
     *            next was called.
     */

    public void remove();

    /**
     * Returns a copy of this iterator.
     *
     * @return a copy of this iterator.
     */

    public Iterator<E> copy();

}
```

Question 5: take (10 marks)

In the class **SinglyLinkedList** below, write a **recursive** (instance) method that returns a **new** linked list consisting of the first **n** elements of this list. This instance must remain unchanged. The method **public LinkedList<E> take(int n)** must be implemented following the technique presented in class for implementing recursive methods inside the class, i.e. where a recursive method is made of a public part and a private recursive part, which we called the helper method. The public method initiates the first call to the recursive method.

```
public class SinglyLinkedList<E> {
    private static class Node<E> {
        private E value;
        private Node<E> next;
        private Node( E value, Node<E> next ) {
            this.value = value;
            this.next  = next;
        }
    }
    private Node<E> first; // Instance variable
    public void addFirst( E item ) { ... }
    public void addLast( E item ) { ... }

    public SinglyLinkedList<E> take( int n ) {









    } // End of take

    private                      takeRec(                          ) {









    } // End of takeRec
} // End of SinglyLinkedList
```

Question 6: findMax (10 marks)

Complete the implementation of the method **findMax**. It returns the largest value of the **Sequence**. Its implementation is recusive. The class **Sequence** is a linked list with additional methods to promote writing recursive list processing methods. Here are the characteristics of the class **Sequence**.

- The elements of the **Sequence** are **Comparable**;
- 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;
  - **void join(Sequence<E> other)**; appends **other** at the end of **this** sequence, **other** is now empty.

```
public class Q6 {
    public static < E extends Comparable<E> > E findMax( Sequence<E> xs ) {









    // End of findMax
// End of Q6
```

Question 7: getPathLength (10 marks)

Let the **path length** of a node be the number of links starting from the root that must be followed to reach that node. The path length of the root is 0. Implement the method **int getPathLength(E obj)** that returns the path length of the node where **obj** is found or **-1** is **obj** if not found in that tree.

```
public class BinarySearchTree<E extends Comparable<E> > {
    private static class Node<E> {
        private E value;
        private Node<E> left;
        private Node<E> right;
        private Node( E value ) {
            this.value = value;
            left = null;
            right = null;
        }
    }
    private Node<E> root = null;

    // Answer:

} // End of BinarySearchTree
```

Question 8: Exceptions (10 marks)

The method **readTransactions** reads an input file where each line is a transaction. A transaction consists of two integers representing the number of shares and the price for each share, respectively. A positive number of shares represents a purchase while a negative number represents selling shares.

A. Define a new type of exceptions named **TransactionFileFormatException**. This must be an unchecked exception. There should be two constructors: one has no parameters while the other has one, a message;

B. Make the necessary changes so that **readTransactions** throws an exception of type **TransactionFileFormatException** if the format of the input is not valid. Hint: the method **boolean hasNextInt()** of the **Scanner** returns true if the next token in this scanner's input can be interpreted as an int value;

C. The implementation of the method **readTransactions** is not valid because the strategy for handling the exceptions has not been defined. Make the necessary changes to make it valid (i.e. can be compiled). Make the changes directly in the source code, on the next page. In particular, here are the methods that are known to throw exceptions.

- **FileInputStream(String 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;
- **String readLine()** throws **IOException** (a checked exception) if an I/O error occurs;
- **int nextInt()** throws **InputMismatchException** (an unchecked exception) if the next token is not an int, or is out of range;
- **int nextInt()** throws **NoSuchElementException** (an unchecked exception) if input is exhausted;
- **close** throws **IOException** (a checked exception) if an I/O error occurs.

Answer to part A.

Answers to parts B and C.

```
public class JStock {

    // ...

    public static JStock readTransactions( String fileName ) {


        FileInputStream fin;
        BufferedReader input;
        Scanner scanner;
        String line;
        JStock js;

        fin = new FileInputStream( fileName );

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

        js = new JStock();

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

            scanner = new Scanner( line ); // Parses the line

            int a = scanner.nextInt(); // Returns the next int

            int b = scanner.nextInt(); // Returns the next int

            if ( a > 0 ) {
                js.buy( a, b );
            } else {
                js.sell( a, b );
            }

        }

        input.close();

        return js;

    } // End of readTransactions


} // End of JStock
```