UNIVERSITY OF WATERLOO

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

E&CE 250 – ALGORITHMS AND DATA STRUCTURES

Midterm Examination Douglas Wilhelm Harder 1.5 hrs, 2005/02/17
11 pages

Name (last, first):									Student ID:			
1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.

Do all of the questions. The exam is out of 70. Closed book. No aids. The number in brackets denotes the weight of the problem. If information appears to be missing from a problem, make a reasonable assumption, state it, and proceed. If the space to answer a question is insufficient, use the reverse side of the previous page.

The only question which you may ask is "May I go to the washroom?" No other questions will be answered by either the teaching assistants or the instructor.

You may immediately proceed by detaching the last page of this exam. Otherwise, wait until you are given notice to open this booklet.

You may abbreviate any class name by using the capital letters only. For example, you may write **`QALL`** instead of **`QueueAsLinkedLlist`**, or **`E`** instead of **`Enumeration`**. Similarly, you may abbreviate method names by using the initial lower case letter and all other upper case letters.

For example, you may write

```
E e = list.gE();
```

instead of

```
Enumeration e = list.getEnumeration();
```

Your variable names must still be distinct from the abbreviated class/method names. You may not abbreviate any instance variable or parameter names which have been given in this midterm.

Guidelines for the number of lines of code (lines with at least one alphanumeric character in them, assuming normal formatting rules) for each solution are given. These are approximations only, and longer or shorter solutions may exist.

If you need an exception and no guidance is given as to which exception you should use, use a **`RuntimeException`** with an appropriate string.

Asymptotic Analysis

1. [9] Perform an analysis of the algorithm given in Figure 1 using the detailed model of a computer by filling in the empty lines in Table 1. You may represent **`array.length`** by *n*. Do not add up your totals.

```
 1: public static int[] partialSums ( int[] array ) {
 2:    int[] sum = new int[array.length];
 3:
 4:    if ( array.length % 2 == 1 )
 5:        throw new RuntimeException(); // analysis done for you
 6:
 7:    for ( int i = 0; i < array.length; i++ ) {
 8:          for ( int j = i; j < array.length; j++ )
 9:                sum[i] += array[j];
10:    }
11:
12:    return sum;
13: }
```

Figure 1. Source code for partial sums

Line	*n* is odd	*n* is even
2	$2\tau_{\text{fetch}} + 2\tau_{\text{store}} + \tau_{\text{call}} + T\langle\texttt{int[]}\rangle$	$2\tau_{\text{fetch}} + 2\tau_{\text{store}} + \tau_{\text{call}} + T\langle\texttt{int[]}\rangle$
4		
5	$\tau_{\text{new}} + \tau_{\text{call}} + T\langle\texttt{RE}\rangle + \tau_{\text{fetch}} + \tau_{\text{return}}$	N/A
7a		
7b		
7c		
8a		
8b		
8c		
9		
12		

Table 1. Detailed analysis of algorithm in Figure 1.

For your reference: τ_{fetch}, τ_{store}, τ_{+}, τ_{-}, τ_{*}, $\tau_{/}$, $\tau_{\%}$, $\tau_{<}$, $\tau_{[]}$, τ_{return}, τ_{new} and $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.

2. [3] Solve the recurrence relation given in Figure 2 for $T(n)$. You should assume that $n = 2^k$ where k is a nonnegative integer (i.e., 0, 1, 2, ...).

$$T(n) = \begin{cases} 1 & n = 1 \\ 2T\left(\frac{n}{2}\right) + n & n > 1 \end{cases}$$

Figure 2. The recurrence relation for the quick-sort algorithm.

3. [2] Suppose a program has an initialization routine which reads the header of n files and stores this header information in an array of size n. How would you respond to the comment that this initialization routine runs in O(ln(n)) time? (One or two sentences.)

4. [2] Show, using the limit definition of big-O, that ln(n) = O(n).

5. [6] Order the following six expressions in n such that $f_i(n) = O(f_{i+1}(n))$ where $f_i(n)$ is the expression in the i*th* box.

2^n $n \ln(n)$ $5n$ n^2 $3 \ln(n)$ $2n^2 + 3$

1.	2.	3.	4.	5.	6.

Arrays and Linked List

6. [1] Suppose a container is being implemented using an array. Why is it necessary to set all the entries to **`null`** when calling the **`purge`** method? Answer with two words:

_____ _____

7. [8] Figure 3 shows an implementation of the **`Element`** class for a singly-linked list. In this case, **`insertBefore`** is an O(*n*) operation. Suppose we implement a doubly-linked list, where each element points to both the next and previous elements. In this case, both **`insertBefore`** and **`insertAfter`** will run in O(1) time. Fill in the implementation on the next page. Hint: Draw a picture in the space below.

```
public class Element {
    Object datum;
    Element next;

    public Element( datum, next ) {
        this.datum = datum;
        this.next = next;
    }

    public Object  getDatum() { return datum; }
    public Element getNext()  { return next;  }

    public void insertAfter( Object obj ) {
        next = new Element( obj, next );

        if ( tail = this )
            tail = next;
    }

    public void insertBefore( Object obj ) {
        if ( head = this ) {
            head = new Element( obj, this );
        } else {
            Element ptr;
            for ( ptr = head; ptr.next != this; ptr++ )
                ; // does nothing
            ptr.next = new Element( obj, this );
        }
    }
}
```

Figure 3. Element class from a singly-linked list class.

```
public class Element {
    Object datum;
    Element previous, next;

    public Element( datum, previous, next ) {
        this.datum = datum;
        this.previous = previous;
        this.next = next;
    }

    public Object  getDatum()    { return datum;    }
    public Element getPrevious() { return previous; }
    public Element getNext()     { return next;     }

    public void insertAfter( Object obj ) {
        Element tmp = new Element( obj,               ,              );









    }
    public void insertBefore( Object obj ) {
        Element tmp = new Element( obj,               ,              );









    }
} // Approximately 8 additional lines of code & fill in blanks
```

Abstract Data Types

8. [4] An abstract data type describes an idea. That idea can be implemented in many different ways. In class, we have seen how various abstract data type may be implemented using arrays and linked lists. In some cases, though, some of the weaknesses of arrays and linked lists may not apply.

Normally, withdrawing an object from a linked list is an O(*n*) operation, where *n* is the number of objects in the linked list. Give one condition where withdrawing an object from a linked list is an O(1) operation.

Normally, given a cursor pointing into an array-based implementation, inserting after the cursor is an O(*n*) operation. Give one condition on the position of the cursor for which inserting after the cursor an O(1) operation.

Containers and Enumerations

9. Recall that inserting an element into an array is an O(*n*) operation. It is very common in industry to therefore temporarily store newly inserted data in a separate container which allows fast insertions. This temporarily stored data is periodically merged into the array.

The container **`AList`** (below) does exactly this: The data is stored in the array, but the **`insert`** method first stores the data in a queue. The new data is merged into the array after every ten insertions. The queue is empty after **`merge`** is called.

Of course, an enumeration must still iterate through all the elements of the container, regardless of whether they are in the array or in the queue. Fill in the implementation of the private **`AnEnumeration`** class. Remember that the **`QueueAsLinkedList`** has its own **`getEnumeration`** and **`getCount`** methods.

Part of the class has been implemented for you. Fill in the rest so that all elements in both the array and the queue are iterated through. You may assume that the container does not change during the life of the enumeration, so do not include any error checking.

```
public class AList extends AbstractContainer {
    protected Object[] array;
    protected Queue tmpQ;

    public AList( int n ) {
        array = new Object[n];
        count = 0;
        tmpQ = new QueueAsLinkedList();
    }

    public void insert( Object obj ) {
        if ( count == array.length ) throw new ContainerFullException();
        tmpQ.enqueue( obj );  // place in queue and increment count
        count++;
        if ( tmpQ.getCount() == 10 ) merge(); // merge the data with array
    }

    public Enumeration getEnumeration() { return new AnEnumeration(); }

    private class AnEnumeration implements Enumeration {
        private int tmpCount;  // add other instance variables here




        public AnEnumeation() {
            tmpCount = 0;



        }

        public boolean hasMoreElements() { return tmpCount != count; }

        public Object nextElement() {
            if ( tmpCount == count )
                throw new NoMoreElementsException();









        }
    }
} // Approximately 9 - 11 additional lines of code
```

Stacks and Queues

10. [3] Consider the following rules for a dynamic stack-as-array class called **`DynamicStackAsArray`**.

1. The stack is initialized to contain an array of size one.
2. If the **`push`** method is ever called on a stack which is full, the array size is doubled before the new object is placed into the stack.
3. If during a call to the **`pop`** method, if the array is reduced to being only half full, the size of the array is halved.

What is the problem with this design? For example, you may wish to consider the code fragment given in Figure 4.

```
Stack s = new DynamicStack();
for ( int i = 0; i < 8; i++ )
     s.push( new Rational( 1, i ) );
for ( int i = 0; i < 10; i++ ) {
     s.push( new Rational( 1, i ) );
     System.out.println( s.pop() );  // push, pop, push, pop, ...
}
```

Figure 4. Example of potentially wasteful code.

Suggest a modification to the **`pop`** method which would reduce the severity of this problem. Full marks are awarded to solutions which are in the same *spirit* of the original design.

11. [12] In class we discussed the idea of a queue. A priority queue of order N (where N is a positive integer, i.e., 1, 2, 3, ...) is a queue were each object is associated with a priority, where 0 is the lowest propriety and $N - 1$ is the highest priority. If two objects, **`A`** and **`B`**, of the same priority are placed into a priority queue in that order, then **`A`** will be dequeued first. If two objects, **`A`** and **`B`**, are placed into a priority queue such that **`A`** has a higher priority than **`B`**, then, regardless of the order **`A`** and **`B`** were placed into the priority queue, **`A`** will be dequeued first.

The interface for a priority queue is given in Figure 5. Note that because it extends the **`Queue`** interface, it contains all the methods of that interface.

```
public interface PriorityQueue extends Queue {
     void enqueue( Object obj, int n );
}
```

Figure 5. Interface for a priority queue.

Using the **`QueueAsLinkedList`** class, design a priority queue which implements this interface by doing the following: Create an array of N **`QueueAsLinkedList`**s. An object of priority i is placed into the ith queue. The **`Dequeue`** method finds the first nonempty queue with highest priority and dequeues an object off of that queue.

The one-argument **`enqueue`** method is already implemented for you – it simply calls the two argument **`enqueue`** method with the lowest priority, namely 0. Do not implement the **`getHead`** method.

You may need the **`ContainterFullException`** or the **`ContainerEmptyException`**. For other exceptions, use a **`RuntimeException`** with an appropriate string.

```
public class PriorityQueueAsLinkedList
    implements PriorityQueue
    extends AbstractContainer
{
    // put instance variable(s) here




    public PriorityQueueAsLinkedList( int n ) {









    }

    // this method simply calls the next method with priority 0
    public void enqueue( Object obj ) {
        enqueue( obj, 0 );
    }

    public void enqueue( Object obj, int n ) {









    }

    public Object dequeue() {









    }
    public Object getHead() {  /* DO NOT IMPLEMENT */ }
} // Approximately 15 lines of code total
```

Ordered Lists and Cursors

12. [8] Suppose you have an ordered list implemented as an array and a cursor into that list which stores an offset into the array. Implement the **`withdraw`** and **`insertAfter`** methods of the cursor. The value of **`offset`** should not change after these methods are called. You may need to use either the **`ContainerFullException`** or the **`ContainerEmptyException`**.

```
public class OrderedListAsArray implements AbstractContainer {
    protected Object [] array;

    public OrderedListAsArray( int n ) {
        array = new Comparable[n];
        count = 0;
    }


    public Cursor findPosition( Object obj ) {
        for ( int i = 0; i < count; i++ )
            if ( array[i].equals( obj ) )
                return new OLAACursor( i );

        throw new ObjectNotFoundException();
    }

    private class OLAACursor implements Cursor {
        protected int offset;

        public OLAACursor( int i ) {
            offset = i;
        }

        public Object getDatum() {
            if ( offset >= count )
                throw new IndexOutOfBoundsException();
            return array[offset];
        }

        public void withdraw() {
            if ( offset >= count )
                throw new IndexOutOfBoundsException();
```

```
        }

        public void insertAfter() {
            if ( offset >= count )
                throw new IndexOutOfBoundsException();
```

```
        }
    }
} // Approximately 10 lines of code
```

Project 1.

13. Implement the **`Comparable`** interface for the **`Rational`** class such that the following property holds: $\frac{a}{b} < \frac{c}{d}$ if and only if $a\,d < b\,c$.

For rationals **`p`** and **`q`**, your method should return:

```
p.compareTo( q ) = -1 if p < q
p.compareTo( q ) =  0 if p = q
p.compareTo( q ) =  1 if p > q
```

In this case, you must worry about overflow, but fortunately, if you use casting to **`long`**, this will be sufficient to overcome this situation.

```
public class Rational implements Comparable {
    protected int numerator;
    protected int denominator;

    // other stuff

    public int compareTo( Object obj ) {
        if ( obj instanceof Rational ) {
            Rational r = (Rational) obj;









        } else {
            throw new ClassCastException();
        }
    }
} // Approximately 6 lines of code.
```

```
public class LinkedList {
    protected Element head;
    protected Element tail;

    public LinkedList();
    public void purge();
    public Element getHead();
    public Element getTail();
    public boolean isEmpty();
    public Object getFirst();
    public Object getLast();
    public void prepend( Object obj );
    public void append( Object obj );
    public void assign( LinkedList list );
    public void extract( Object obj );

    public class Element {
        Object datum;   // visible in class LinkedList
        Element next;   // visible in class LinkedList

        public Object getDatum();
        public Element getNext();
        public void insertBefore( Object obj );
        public void insertAfter( Object obj );
    }
}

public interface Container {
        int getCount();
        boolean isEmpty();
        boolean isFull();
        void purge();
        Enumeration getEnumeration();
}

public abstract class AbstractContainer implements Container {
        int count = 0;
        public int getCount() { return count; }
        public int isEmpty() { return getCount() == 0; }
        public isFull() { return false; }
}

public interface Enumeration {
        boolean hasMoreElements();
        Object nextElement();
}

public interface Comparable {
        int compareTo( Comparable obj );
}
```

X.compareTo(Y) $\begin{cases} < 0 & X < Y \\ = 0 & X.\text{equals}(Y) \\ > 0 & X > Y \end{cases}$

```
public interface Stack extends Container {
        void push( Object obj );
        Object pop();
        Object getTop();
}

public interface Queue extends Container {
        void enqueue( Object obj );
        Object dequeue();
        Object getHead();
}

public interface SearchableContainer extends Container {
        boolean isMember( Comparable obj );
        void insert( Comparable obj );
        void withdraw( Comparable obj );
        Comparable find( Comparable obj );
}

public interface OrderedList {
        Comparable get( int i );
        Cursor findPosition( Comparable obj );
}

public interface Cursor {
        Comparable getDatum();
        void insertAfter( Comparable obj );
        void insertBefore( Comparable obj );
        void withdraw();
}
```