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. Of the given four data structures, for each of the five properties, rank the four data structures with respect to. For the hash table, you must assume that the load factor is one, that is, $m = n$.

	Inserting	**Finding Least Entry**	**Finding Greatest Entry**	**Finding Next Entry**	**Checking Membership**
Sorted List	O(n)	O(1)	O(1)	O(1)	O(log(n))
Hash Table	O(1)	O(n)	O(n)	O(n)	O(1)
Binary Search Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))
Priority Queue (minimum)	O(log(n))	O(1)	O(n)	O(n)	O(n)

2. For the given tree, print out the keys as if the tree is traversed using an:
   a. Pre-order traversal,
   b. Post-order traversal, and
   c. Where applicable, an in-order traversal.

ONE GENERAL TREE
ONE BINARY TREE TREE
ONE 3-WAY TREE

Write the following expression for I(n) in terms of I(n) and I($n - 1$).

$$\mathrm{I}(n+1) = \begin{cases} 1 & n \le 0 \\ \sum_{i=0}^{n} i\,\mathrm{I}(i) & n \ge 1 \end{cases}$$

Create a min-heap by placing the following seven numbers into the heap in the given order:

5 16 8 15 2 23 4

Given the following min-heap, indicate the state of the heap after each of three calls to the method **`dequeueMin`**.

null	1	3	5	4	7					

When inserting an element into a min heap, it may be necessary to

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.)

Given the general formula

Given divide and conquer $T(n) = a\,T(n/b) + O(n^k)$ and the resulting formula

$$T(n) = \begin{cases} O(n^{\log_b a}) & a > b^k \\ O(n^k \log_b n) & a = b^k \\ O(n^k) & a < b^k \end{cases}$$

explain how quick sort is $O(n \log_2 n)$.

Recalling that an N-ary tree with $n \geq 0$ internal nodes contains $(N - 1)n + 1$ external nodes. Prove, using induction, that an N-ary tree of height h has a maximum of N^h leaf nodes.

What is the total internal path length of the following three trees:

What are upper and lower bounds on the total internal path length of a binary tree with $2^{h+1} - 1$ internal nodes where $h \geq 1$? Expand each answer so that it is a sum of products. Justify your answer briefly.

You may need the formulae $\sum_{i=0}^{n} i2^i = (n-1)2^{n+1} + 2$ and $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$.

A perfect binary tree is an AVL tree which is perfectly balanced at each node. Draw an example of an AVL tree with three and seven internal nodes which is the least balanced. That is, as many nodes as possible are unbalanced. There are sixteen possible solutions for a tree with seven nodes to this question; you only need to find one. Do not prove that your solution is the least balanced solution.

Consider a scatter table which places an integer into the bin corresponding to the last digit. Assume the table is partially filled as follows where -1 is used as a flag to indicate the end of a chain.

Bin	0	1	2	3	4	5	6	7	8	9
Datum		121	31	42	32		76	36	77	
Next	-1	2	3	4	-1	-1	7	8	-1	-1

Insert the numbers 33, 55, and 88 into this chained scatter table.

Bin	0	1	2	3	4	5	6	7	8	9
Datum										
Next										

For chained scatter tables, you could use a special flag to indicate that a chain terminates. For example, the *next* field could be occupied by -1. Why is this not a good idea if the chained scatter table has $2^8 = 256$ entries (i.e., 0 to 255)?

Consider a chained scatter table with 10 slots. For a given integer, let the last digit determine which slot the number appears in. Given the following chained scatter table, indicate the successive states when 25, 147, 100, and 37 are withdrawn, in that order.

Bin	0	1	2	3	4	5	6	7	8	9
Datum	100	21	38	27	24	34	25	37	147	28
Next	1	2	3	3	5	6	6	8	9	0

Bin	0	1	2	3	4	5	6	7	8	9
Datum										
Next										

Bin	0	1	2	3	4	5	6	7	8	9
Datum										
Next										

Bin	0	1	2	3	4	5	6	7	8	9
Datum										
Next										

Bin	0	1	2	3	4	5	6	7	8	9
Datum										
Next										

The homework asks you to implement a hash table which doubles the number of bins in the hash table if the load factor is $\lambda = 1$ after inserting and to halve the number of bins in the hash table if the load factor is $\lambda = 0.5$. Why is this implementation sub-optimal?

For the given sequence of keys, determine the binary search tree obtained when the keys are inserted one-by-one in the order given into an initially empty binary search tree:

7 2 4 6 1 8 3 5

Using these same keys, determine the AVL tree obtained when the keys are inserted one-by-one in the order given into an initially empty AVL tree.

Find upper and lower bounds on the internal path lengths of trees with 1, 3 and 7 nodes. Generalize this for a tree with $2^n - 1$ nodes.

To delete a non-leaf node from a binary search tree, we swap it either with the smallest key its right subtree or with the largest key in its left subtree and then recursively delete it from the subtree. In a tree of *n* nodes, what is the minimum and maximum number of swaps needed to delete a key?

Are either of these optimal cases realizable with an AVL tree with *n* nodes?

Asymptotically speaking, what is the average number of swaps required for an AVL tree when deleting the root node?

Using the complete binary tree as shown in class, enter the following numbers into a min-heap.

6 4 1 7 2 5 3 8

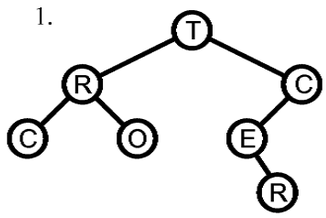
0	1	2	3	4	5	6	7	8	9	10	11

It is not necessary that a min-heap be implemented as a binary tree. For example, a general tree is a min-heap if the keys of each of the subtrees are greater than the given key and each sub tree is a heap. Implement a method which returns **`true`** if the given general tree is a heap and **`false`** otherwise.

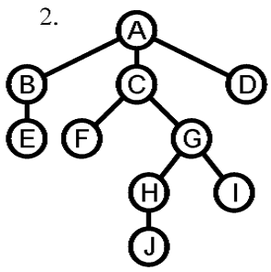
```
public class GeneralTree extends AbstractTree {
        protected LinkedList[] list;

        public boolean isHeap() {




        }
}
```

For each of the following trees (one binary, one general, and one 3-way tree), do a pre-order traversal, an in-order traversal, and a post-order traversal. Do not do traversals where specified not to with a dash.

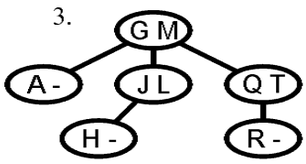
1.



2.



3.



	1	2	3
Pre-Order			–
In-Order		–	
Post-Order			–

Give examples of two different conditions under which you would want to use a sorted list instead of a binary search tree to store sorted data which needs to be searched. Your examples should contain quantitative values (either exact or asymptotic).

1. When you need to quickly find the next element

2. When space is limited

What are the memory requirements for a priority queue implemented as follows:

1. The entries are stored in a complete binary tree,
2. An array, as described in class, is used to store the entries in the queue,
3. There are currently 32 entries in the priority queue,
4. The initial length of the array is 2 and each time a new element is added to the priority queue, the length of the array is doubled.

For which of these could you use the address of the object as a reasonable hash value, and for which would you have to use either the multiplicative method to create a better hash value?

[4] Suppose you wanted to use the mid-square method to get one decimal digit (0-9) hash values of *randomly* chosen two decimal digit numbers in the range 40 through 99. Rank, in order of usefulness, which digit (1^{st}, 2^{nd}, 3^{rd}, and 4^{th}) of the square you would prefer to use for the hash value.

For your reference, the 1^{st}, 2^{nd}, 3^{rd}, and 4^{th} digits for 40^2 through 99^2 are given:

```
1111122222222223333333334444444555555566666677777788888999999
6678901234567890123467890234679013467902457802357912468024680
0864321000000012346802469258260482727284062852964208643210000
0149656941014965694101496569410149656941014965694101496569410
```

For example, the first column is 1600, or 40^2.

Most Useful			Least Useful

A set is a data structure which has only one occurrence of each element. Thus, if insert receives an element which already exists in the set, then the new item should not be inserted into the set. Use whatever data structure you wish to implement a set so that **`insert`**, **`isMember`**, **`withdraw`** and **`find`** run in O(1) time. You may assume that the set will contain at most 1000 elements. You should not implement the data structure which you are using.

All of these methods, when properly implemented, are relatively short; in some cases, they are as short as one line.

```
public class Set
          implements SearchableContainer
          extends AbstractContainer
{
     // instance variables

     public Set() {




     }

     public boolean isMember( Comparable obj ) {




     }

     public insert( Comparable obj ) {




     }

     pubic void withdraw( Comparable obj ) {




     }

     public Comparable find( Comparable obj ) {




     }
}
```

[1] Consider a priority queue implemented as a binary search tree using the **`Tree`** data structure. Circle the traversals which you could use to convert the priority queue as a binary search tree to a priority queue as a complete tree implemented using an array in O(*n*) time?

Pre-Order Traversal In-Order Traversal Post-Order Traversal

[2] Why, recalling that each traversal runs in O(*n*) time?

The benefits of using an array with a complete binary tree to represent a priority queue is that **`enqueue`** and **`dequeueMin`** are O(log(*n*)) while **`getHead`** is O(1).

Suppose the data which is being temporarily stored in a priority queue is known to arrive approximately in order. For example, it may be known that, on average, a new datum will be two positions out of order relative to other entries which are being enqueued at the same time. For example, the data may be arriving as follows:

1 4 3 6 5 8 9 7 10 14 13 12 19 17 20 21

For each of **`enqueue`** or **`dequeueMin`**, explain if this will affect the run time of these methods.

`void enqueue( Comparable obj )`

Run time:

Why:

`Comparable dequeueMin()`

New run time:

Why:

Taking this into account, it is possible to create a priority queue where all three methods are O(1). It may help to look at the implementation of a **`QueueAsArray`**, the implementation of which is given on the next page. On the page thereafter, fill in the body for the given methods, making sure that, in the above scenario they run in O(1) time.

Use the **`Comparable`** interface as described on the last page and recall that the method call

```
( new Integer(3) ).compareTo( new Integer( 7 ) )
```

returns **`-1`**.

```
// Referece:  Implementation by Dr. Bruno Preiss

public class QueueAsArray implements Queue
          extends AbstractContainer
{
     protected Object[] array;
     protected int head;
     protected int tail;

     public QueueAsArray( int size ) {
          array = new Object[size];
          head = 0;
          tail = size – 1;
     }

     public Object getHead() {
          if ( count == 0 ) {
               throw new ContainerEmptyException();
          }

          return array[head];
     }

     public void enqueue( Object object ) {
          if ( count == array.length ) {
               throw new ContainerFullException();
          }

          tail = (tail + 1) % array.length;

          ++count;
          array[tail] = object;
     }

     public Object dequeue() {
          if ( count == 0 ) {
               throw new ContainerEmptyException();
          }

          Object result = array[head];
          array[head] = null;

          head = (head + 1) % array.length;

          --count;
          return result;
     }
}
```

```
public class PriorityQueueAsArray implements PriorityQueue
          extends AbstractContainer
{
     // instance variables



     public PriorityQueueAsArray( int size ) {




     }

     public Comparable getHead() {




     }

     public void enqueue( Comparable object ) {

```



```
     }

     public Comparable dequeueMin() {

```



```
     }
}
```

Interfaces and Classes

```
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();
}
```