UNIVERSITY OF WATERLOO

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

ECE 250 – ALGORITHMS AND DATA STRUCTURES

Midterm Examination | Instructor: R.E.Seviora | 1.5 hrs, Oct 29, 2001

SOLUTIONS (unchecked)						Student ID:		
1. A	1. B	2. A	2. B	3. A	3. B	4. A	4. B	Total:

Do all problems. The number in brackets denotes the weight of the problem (out of 100). 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 last (overflow) page. Closed book. No calculators allowed.

PROBLEM 1 [25]

A. Detailed and Simplified Model

1 In order to predict the running times of programs, we must know something about the computer on which the program will execute. Two models of the computer (in the Java VM context) were considered in this course, the detailed one and the simplified one. In the table below, state the timing parameters of the detailed model. For brevity, give the answers in the manner shown for fetch and store.

1. fetch, store:	τ_{fetch}, τ_{store}
2. *elementary arithmetic operations (add, subtract, multiply, divide, comparison)*	τ_{+}, τ_{-}, τ_{x}, $\tau_{\div}$, $\tau_{<}$,
3. *method call and return*	τ_{call}, τ_{return}
4. *pass an argument*	τ_{store}
5. *address calculation for indexing operation*	$\tau_{[]}$
6. *create an object*	τ_{new}

2 Determine the running time predicted by the detailed and the simplified computer model for the following program fragment (where n > 1):

```
1 for (int i=n-1; i!=0; i/=2)
2   ++k;
```

statement	time
1a	$\tau_{\text{fetch}} + \tau_{\text{store}}$
1b	$(2\tau_{\text{fetch}} + \tau_{<}) \times (\lceil \log_2 n \rceil + 1)$
1c	$(2\tau_{\text{fetch}} + \tau_{\div} + \tau_{\text{store}}) \times \lceil \log_2 n \rceil$
2	$(2\tau_{\text{fetch}} + \tau_{+} + \tau_{\text{store}}) \times \lceil \log_2 n \rceil$
TOTAL	$(6\tau_{\text{fetch}} + \tau_{+} + \tau_{\div} + \tau_{<} + 2\tau_{\text{store}}) \times \lceil \log_2 n \rceil + (3\tau_{\text{fetch}} + \tau_{<} + \tau_{\text{store}})$

statement	time
1a	2
1b	$3(\lceil \log_2 n \rceil + 1)$
1c	$4\lceil \log_2 n \rceil$
2	$4\lceil \log_2 n \rceil$
TOTAL	$11\lceil \log_2 n \rceil + 5$

B. Big Oh and Theta

Order the following functions by the big-Oh relationship. Group together (e.g. by circling) those functions that are Θ (theta) of one another.

$$|6n \log n| \quad 2^{2^n} \quad 3n^{0.5} \quad 4^n \quad 2^{100} \quad \lceil \sqrt{n} \rceil \quad 5n \quad n^{0.01} \quad \lfloor n^2 \log n \rfloor \quad 4^{\log_2 n}$$

$$2^{100} \quad n^{0.01} \quad \boxed{\lceil \sqrt{n} \rceil \quad 3n^{0.5}} \quad 5n \quad |6n \log n| \quad 4^{\log_2 n} \quad \lfloor n^2 \log n \rfloor \quad 4^n \quad 2^{2^n}$$

PROBLEM 2 [25]

A. Asymptotic Analysis (Project 2)

Consider a typical implementation of the `PolynomialAsArray` class, with two non-static fields, the array of coefficients `double coef[]`, and the degree of the polynomial `int n`. The `toString()` method returns a string representation of the polynomial object. The program below shows a simplified version of the `toString()` method. (The actual algorithm is somewhat more complex – the body of the loop must skip zero terms, deal with the signs of coefficients etc.)

```
  public String toString() {
1   String outString = new String ("");
2   for (int i=0; i <= n; i++)
3    outString = outString + coef[i] + "X^" + i;
4   return outString;        }
```

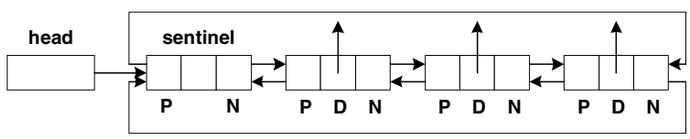
Carry out the big Oh analysis of the worst case running time for the above `toString()` method. The degree of the polynomial, n, represents the input size.

In each iteration of the loop, a new string is created which contains the concatenation of the previous outString and the printing representation of the i-th term. The length of the latter has an upper bound k, which is independent of n. The running time of string creation is proportional to the length of the string.

Statement	*Running Time*	*big Oh*
1	$O(1)$	$O(1)$
2a	$O(1)$	$O(1)$
2b	$(n+2)\times O(1)$	$O(n)$
2c	$(n+1)\times O(1)$	$O(n)$
3	$\sum_0^n (k \times O(1)) = k(n^2+n)/2$	$O(n^2)$
4	$O(1)$	$O(1)$
TOTAL		$O(n^2)$

B. Doubly Linked List

Write the `prepend` method for a `DoublyLinkedList` class which uses sentinel (see the figure below). Note that each `Element` contains two references, `next` and `previous`. You may assume that the constructor of `DoublyLinkedList` creates the sentinel, makes the `head` refer to the sentinel and sets the `next` and `previous` fields of the sentinel appropriately.



Note: If you have difficulty solving this problem, write an implementation of the `prepend()` method for a *singly* linked list with sentinel only for part mark.

```
public class DoublyLinkedList {
  protected Element head;
  public final class Element {
    Object datum;
    Element next;
    Element previous;
    //constructor
    Element (Object datum, Element previous, Element next) {
    //...

  public void prepend (Object o) {
    Element temp = new Element (o, head, head.next);
    head.next.previous = temp;
    head.next = temp;
   }
```

Note: *if the answer given is for the singly linked case only*

```
  public void prepend (Object o) {
    head.next = new Element (o, head.next);
  }
```

PROBLEM 3 [25]

A. Visitors

Consider a container class whose instances will contain objects of possibly differing types. Implement a `MaxLengthVisitor` whose `visit` method will find the maximum length of the printing representation (i.e. of the string returned by `toString()`) of the objects in the container. To determine the length of a string, you may use the `int length()` method of the class `java.lang.String`. Note that the visitor is only required to determine the maximum length; it is not required to keep reference to the object with the longest printing representation.

```
public class MaxLengthVisitor extends AbstractVisitor {

  //fields

   int maxL = 0;  //stores the max length value seen so far

  //methods
  public void visit (Object o)
  {
    int len = o.toString().length();
    if (len > maxL) maxL = len; }




  public boolean isDone()
  {
    return false; }



  public int getMaxLength()
  {
    return maxL; }
```

B. Enumerations and Visitors

Enumerations and visitors provide two ways to do the same thing – to visit, one-by-one, all the objets in a container. Give an implementation for the `accept` method of the `AbstractContainer` class that uses an enumeration.

```
public class SomeContainer extends AbstractContainer  {
  public void accept (Visitor v) {
    Enumeration e = getEnumeration();
    while (e.hasMoreElements() && ! v.isDone() ) {
      Object o = e.nextElement();
      v.visit (o);          }
  }
```

PROBLEM 4 [25]

A. Stacks

The array-based stack implementation discussed in the class uses a fixed length array. As a result, it is possible for the stack to become full.

1. Modify the algorithm for the `push(Object o)` method given below so that it doubles the length of the array when the array is full.

```
public class StackAsArray
 extends AbstractContainer
```

```
implements Stack {
  protected Object[] array;
  //other methods ...
  public void push (Object obj) {
      if (count == array.length)
       throw new ContainerFullException();
     array [count++] = obj;      }

public void push (Object obj) {
   int len = array.length;
   if (count == len) {
     Object temp = new Object [2*len];
     System.arraycopy(array, 0, tmp, 0, count); //alternative: loop w copy
     array = temp;  }
   array[count++] = obj;
}
```

2. Obtain a reasonably tight big Oh bound for the *average* time for such push operation (i.e. the average over *n* consecutive pushes). [Hint: Consider the case when, e.g., the initial array size is 2. The time consumed by successive pushes is shown below (^ denotes exponentiation, c and k are constants).]

n	0	1	2	3	4	5	6	7	8	9	...
time	c	c	k2^2	c	k2^3	c	c	c	k2^4	c	...

For illustration, see the graph on the overflow page. If $M=2^m$, the time for M consecutive pushes is

$$t_M = \sum_{i=0}^{M-1} k + \sum_{i=1}^{m} c2^{i+1} = kM + 4c\sum_{i=0}^{m-1} 2^i = kM + 4c(2^m - 1) = (k+4c)M - 4c.$$

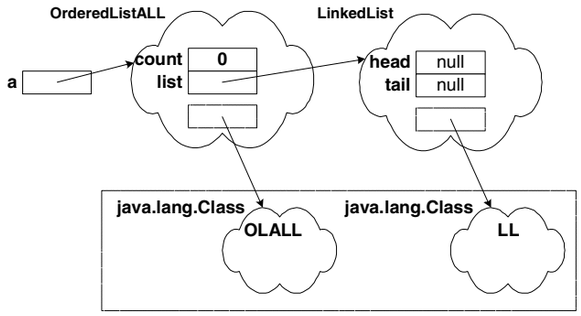
Hence, the average push time $t_{push} = \frac{1}{M}[(k+4c)M - 4c] \cong (k+4c) - \varepsilon = O(1)$

B. Ordered Lists

1 Consider the `OrderedListAsLinkedList` class discussed in the course.

```
public class OrderedListAsLinkedList
  extends AbstractSearchableContainer implements OrderedList  {
    protected LinkedList list;
    //constructor
    public OrderedListAsLinkedList () {
      list = new LinkedList(); }
    //...
}
```

Show graphically the object(s) created, their fields and their relationship after the execution of the statement: `OrderedListAsLinkedList a = new OrderedListAsLinkedList();`



2 Consider an implementation of the `OrderedList` interface that uses a doubly-linked list (i.e., each list element contains a reference to the immediately preceding and immediately following element on the list, if they exist). In the table below, give the big Oh running times of the `OrderedList` operations for such implementation.

method	*running time*	*method*	*running time*
`insert`	*O(1)*	`findPosition`	*O(n)*
`isMember`	*O(n)*	`Cursor.getDatum`	*O(1)*
`find`	*O(n)*	`Cursor.insertAfter`	*O(1)*
`withdraw`	*O(n)*	`Cursor.insertBefore`	*O(1)*
`get`	*O(n)*	`Cursor.withdraw`	*O(1)*

OVERFLOW SHEET [Please identify the question(s) being answered.]

Problem 4A2, figure illustrating the cumulative time for n pushes.