Computing Science 115

Final Examination

April 19, 2000

Section: B3- Szafron

Last Name: ______________________

First Name: ______________________

Student #: ______________________

E

04719
CMPUT 115 (B3)
SZAFRON
APR 00 FINAL
PAGES: 12

Instructions:

The time for this test is 3 hours. No references or calculators are allowed. Place all answers in this booklet and do not hand in any other work. The mark total for this exam is 100.

#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	#12	TOTAL
/10	/8	/10	/8	/5	/10	/10	/4	/8	/5	/12	/10	/100

#1 [10 marks] Consider five sorts: BubbleSort, SelectionSort, InsertionSort, MergeSort, and QuickSort(standard version with left element as pivot). For each of the following situations, pick **ONE** of **these** five sorts that you think would be best for the situation, assuming that you want the fastest sort possible that will work. Then say why it is best in 20 words or less. Do not simply say that it is the best because it is the fastest. Indicate why it is the fastest. Assume that you have as much memory as necessary to use any of the sorts, except in those situations where memory availability is specifically mentioned.

a) Start with a Vector of Strings that has 1,000,000 elements and is already sorted in ascending order. Add 5 more Strings to the end of the Vector. Now sort the elements in ascending order.

Which Sort ? ____________________

Why ?____

____

b) Start with one Vector of Strings whose 500,000 elements are already sorted in ascending order and a second Vector of Strings whose 500,000 elements are also sorted in ascending order. Start with a third Vector that is empty. Add all elements from the first vector to the third Vector in order. Then append all the elements from the second Vector to the third vector in order. Now sort the elements of the third vector in ascending order.

Which Sort ? ____________________

Why ?____

____

c) Start with one Vector of Strings whose 1,00,000 elements are already sorted in **descending** order. Now sort the elements of the vector in **ascending** order.

Which Sort ? ____________________

Why ?____

____

d) Start with one Vector of Strings whose 1,00,000 elements are in an unknown order. In fact, there is a (1/3) chance they are in random order, a (1/3) chance they are in ascending order and a (1/3) chance they are in descending order). Now sort the elements of the vector in **ascending** order.
Which Sort ? ____________________

Why ?____

____

e) Start with one Vector of Persons whose 1,00,000 elements are in **random** order. Assume that you have enough memory to store this vector, plus enough memory for as many stack frames as any of the sorts would require. However, assume that you do not have enough memory to store more than 1,000 extra Person elements. Now sort the elements of the vector in **ascending** order.

Which Sort ? ____________________

Why ?____

____

#2 [8 marks] Consider two implementations of the List Interface: SinglyLinkedList, and CircularList (the standard textbook implementation that keeps a reference to the last node of the list) and four methods: addToHead(Object), addToTail(Object), removeFromHead(Object) and removeFromTail(Object). Beside each of the following combinations, circle **efficient** if the implementation of the method in the class is O(C) or **inefficient** if the implementation of the method in the class is O(n).

Since guessing can produce 4 correct answers without any knowledge, the marks for this question are as follows: 0, 1, 2 or 3 correct - 0 marks, 4 correct - 1 mark, 5 correct - 2 marks, 6 correct - 4 marks, 7 correct - 6 marks, 8 correct - 8 marks. If you don't know the answer to one of the parts, you might as well guess, there is no penalty for guessing.

a)	SinglyLinkedList - addToHead(Object)	efficient	inefficient
b)	SinglyLinkedList - removeFromHead(Object)	efficient	inefficient
c)	SinglyLinkedList - removeFromHead(Object)	efficient	inefficient
d)	SinglyLinkedList - removeFromTail(Object	efficient	inefficient
e)	CircularLinkedList - addToHead(Object)	efficient	inefficient
f)	CircularLinkedList - removeFromHead(Object)	efficient	inefficient
g)	CircularLinkedList - removeFromHead(Object)	efficient	inefficient
h)	CircularLinkedList - removeFromTail(Object)	efficient	inefficient

#3 [10 marks] Consider the four search scenarios listed below (a, b, c, and d). For each of the scenarios, circle the word **binary** if it is possible to do a binary search **for that scenario** or circle **sequential** if a binary search is not possible. Then, use a binary search if possible and a sequential search if a binary one is not possible. Count the number of element comparisons that will be performed during the search and write this **number** in the space provided.

a) Searching for a String in an unsorted Vector of 1,024 Strings and not finding it.

binary sequential Number of element comparisons ______________

b) Searching for a String in a sorted Vector of 1,024 Strings and finding it at the last location **searched.**

binary sequential Number of element comparisons ______________

c) Searching for a String in an unsorted SinglyLinkedList of 1,024 Strings and not finding it.

binary sequential Number of element comparisons ______________

d) Searching for a String located at position 512 of a sorted SinglyLinkedList of 1,024 Strings.

binary sequential Number of element comparisons ______________

e) Searching for an Integer in a sorted SinglyLinkedList of 1,024 Integers and not finding it. Assume the List contains the Integers: 1, 3, 5, ... and assume you are looking for the Integer 512.

binary sequential Number of element comparisons ______________

#4 [8 marks] Consider the following program. What is the output? Ignore any syntax errors that may be in the program.

```
import structure.*;

public class StacksAndQueues {
/*
  Program description.
*/
  public static void main(String args[ ]) {
  /* Program statements go here. */

     String[ ] myStrings = { "wilma", "fred", "barney", "pebbles"};
     Stack stack1;
     Stack stack2;
     Queue myQueue;
     int index;
     Object element;

     stack1 = new StackList();
     stack2 = new StackList();
     myQueue = new QueueVector();
     for (index = 0; index < myStrings.length; index++)
        stack1.add(myStrings[ index] );
     System.out.println(stack1);
     for (index = 0; index < myStrings.length / 2; index++) {
        element = stack1.remove();
        myQueue.add(element);
        stack2.add(element);
     }
     System.out.println("Stack 1: " + stack1);
     while (!stack1.isEmpty()) {
        element = stack1.remove();
        myQueue.add(element);
        element = myQueue.remove();
        stack2.push(element);
     }
     System.out.println("Stack 2: " + stack2);
     System.out.println("Queue: " + myQueue);
     while (!myQueue.isEmpty()) {
        element = myQueue.remove();
        stack2.add(element);
     }
     System.out.println("Stack 2: " + stack2);
  }
}
```

OUTPUT

#5 [5 marks] Suppose you have a hash table with room for seven entries (indexed 0 through 6). This table uses open addressing with the hash function that maps each String to its length modulo 7. Rehashing is accomplished using linear-probing with a jump of 1. Draw the table after each of Strings: "fred", "pebbles", "barney", "dino" and "betty", have been added consecutively.

"fred"

0	
1	
2	
3	
4	
5	
6	

"fred", "pebbles"

0	
1	
2	
3	
4	
5	
6	

"fred", "pebbles" "barney"

0	
1	
2	
3	
4	
5	
6	

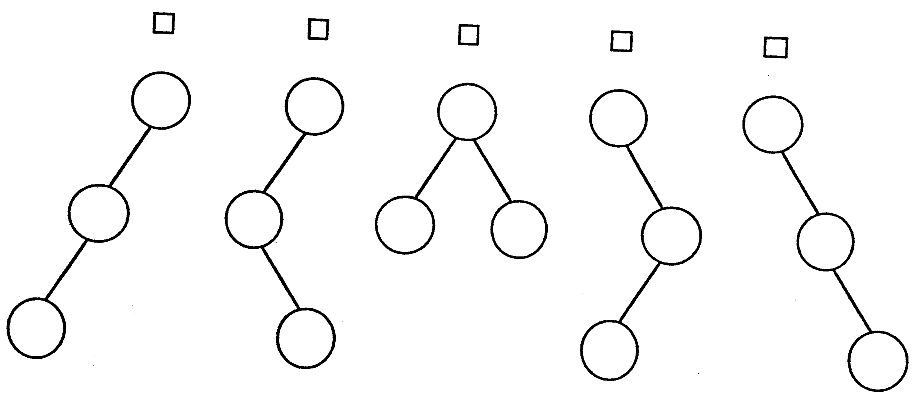
"fred", "pebbles" "barney", "dino"

0	
1	
2	
3	
4	
5	
6	

"fred", "pebbles" "barney", "dino", "betty"

0	
1	
2	
3	
4	
5	
6	

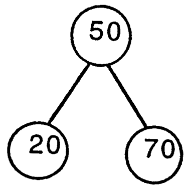
#6 [10 marks] Consider each of the five different three node binary trees:



a) For each tree, fill in the nodes with 1, 2 and 3 in such a way that an inorder traversal will have the order: 1, 2, 3.

b) Put an X in the square above each binary tree from a) that is also a binary search tree.

#7 [5 marks] Here is a binary search tree after the Integers 50, 20 and 70 have been added to an empty binary search tree in that order.



a) Draw this binary search tree after 40 has been added.

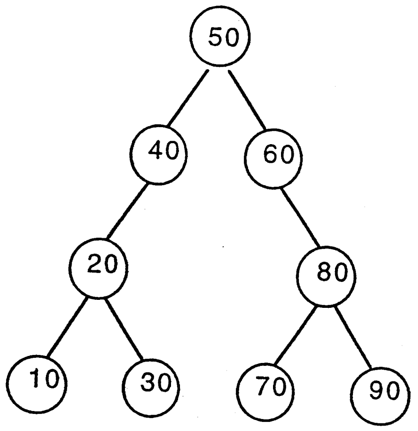
b) Draw this binary search tree after 60 has been added to the tree from part a).

c) Draw this binary search tree after 65 has been added to the tree from part b).

d) Draw this binary search tree after 20 has been added to the tree from part c).

e) Draw this binary search tree after 50 has been added to the tree from part d).

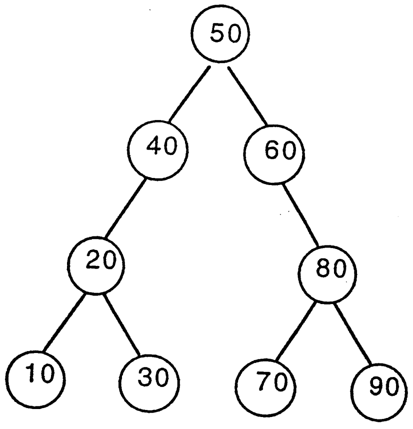
#8 [4 marks] Here is a binary search tree.



a) Draw this binary search tree after the node containing 50 has been removed.

b) Draw the binary search tree after the node containing 40 has been removed from the tree in part a).

#9 [8 marks] Consider this binary tree:



a) List the elements in the order of a postorder traversal.

b) List the elements in the order of apreorder traversal

c) There is a standard way of representing a binary tree in an array. Fill in the elements from the binary tree into the array using this standard representation. Put an N in any array location that should contain null.

0	1	2	3	4	5	6	7	8	9	10	11	12	13

#10 [5 marks] Consider the Interface Predicate. Fill in the code for the class EvenPredicate that implements it. Recall that there is an operator called (instanceof) that can be used to check if an object is an instance of some class.

```
 public interface Predicate {

 /*
   Each class that implements this interface
   represents a predicate. A predicate is a one
   argument function that returns a boolean.
 */


  public boolean select(Object anObject);
  /*
     post: Return true if the given object satisfies
     this predicate.
  */

}

public class EvenPredicate implements Predicate {
/*
  This class represents the predicate that returns
  true if an object is an even Integer and false
  otherwise.
*/


 public boolean select(Object anObject) {
 /*
    post: Return true if the given object is an
    even Integer and false otherwise.
 */




 }
}
```

#11 [12 marks] Consider a filtering Iterator called SelectIterator that uses a base Iterator and an instance of some class of Predicate as defined in question #10. The instance of SelectIterator traverses all of the elements of the base Iterator for which the Predicate evaluates to true. Fill in the code for the SelectIterator class. Here is an example test program for it and the sample output.

```
 import structure.*;
 public class TestSI {
 // This is a test program for the SelectIterator class.
   private static final int[] array = {1, 2, 3, 4, 5, 6, 7};
   public static void main(String args[]) {
   // Create a base Iterator, a SelectIterator and try it.
      List list;
      int  index;
      Iterator base;
      Iterator select;
      Predicate predicate;
      list = new SinglyLinkedList();
      for (index = 0; index < array.length; index++)
         list.addToTail(new Integer(array[ index] ));
      base = list.elements();
      predicate = new EvenPredicate();
      select = new SelectIterator(base, predicate);
      while (select.hasMoreElements())
         System.out.println(select.nextElement());
   }
}
```

Output
2 4 6

```
import structure.*;
public class SelectIterator implements Iterator {
/*
  An instance of this class filters a base Iterator by selecting
  only those elements that satisfy a particular Predicate.
*/

// Instance variables
  protected Predicate predicate;
  protected Iterator iterator;
  protected Object next;

// Public methods
 public SelectIterator(Iterator base, Predicate predicate) {
 /*
    post: construct me to have the given base Iterator and use
    the given Predicate.
 */
    this.predicate = predicate;
    this.iterator = base;
    this.primeNext();
 }

 public void reset() {
 /*
    post: the iterator is reset to the beginning of the traversal.
 */
    ______________________________________________

    ______________________________________________
}
```

```
    public Object value() {
    /*
       pre: traversal has more elements
       post: returns the current value referenced by the iterator
    */

       ____________________________________________
    }

    public boolean hasMoreElements() {
    /*
       post: returns true iff the traversal is not complete
    */

       ____________________________________________
    }

    public Object nextElement() {
    /*
       pre: traversal has more elements
       post: returns the current value referenced by the iterator
       and increments the iterator
    */

       Object answer;

       Assert.pre(this.next != null, "Iterator has more elements.");
       ____________________________________________
       this.primeNext();

       ____________________________________________
    }

    protected void primeNext() {
    /*
       Traverse the base iterator until the next acceptable element
       is found and bind next to it. If no such element is found
       then bind next to null.
    */

       Object element;

       while ____________________________________________

          element = this.iterator.nextElement();

          if ____________________________________________

             ____________________________________________

             ____________________________________________
          }
          this.next = ____________________________________________
       }
    }
}
```

#12 [10 marks] Consider the MultiKeyedCollection interface from the project. We want to add another method to this interface:

```
public void selectionSort(String majorAspect, String minorAspect);
/*
 post: The sort order for the major aspect of the elements is updated using a
 selection sort. However, in the case of ties on the major aspect, the minor aspect
 should be used to break ties. If the elements don't recognize the major aspect, use
 the default aspect. If the elements don't recognize the minor aspect, use the
 default aspect. The insertion order and other sort orders are not changed.
*/
```

To complete the implementation of this method in the MultiKeyedVector class, all of the code has already been written for you except for the following method whose code you must complete. Assume that the elements you are sorting is contained in an Array of MultiComparables whose instance variable name is: sortedElements.

```
protected findMaximumElement(int size, String majorAspect, String minorAspect) {
/*
 pre: 0 < size <= size of element array
 post: The index of the largest element in the sortedElements array in the range
 0..size-1 is returned. The largest element is found by comparing elements using the
 majorAspect. However, if there is a tie, then the tied elements are compared using
 the minorAspect.
*/

}
```