Computing Science 115

Final Examination

April 27, 2000

9:00AM PAVILLION

Section: B1- Sorenson

Last Name: ______________________

First Name: ______________________

Student #: ______________________

E

04717
CMPUT 115 (B1)
SORENSON
APR 00 FINAL
PAGES: 11

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		TOTAL
/10	/6	/10	/10	/8	/8	/5	/15	/10	/8	/10		/100

#1 [10 marks] Assume you must develop a system to allow a dispatcher to manage a collection of clients for a taxi cab company. Assume that the size of the collection is very dynamic ranging in size from 200 to as many as 4,000 calls in peak periods (e.g., New Years' Eve). When a customer calls for a taxi, his/her phone number is recorded along with his/her name, address of the pick up, time of call, and time when pick up is required. This information is stored as an element in the collection. When a pick up is made, the driver notifies the dispatcher, who removes the client's information from the collection. How would you organize this collection assuming:

- your program is to produce in response to the input of a phone number, a screen that lists the phone number and associated client information as well as the phone number and client information for the six closest numbers that are in the collection. For example if the dispatcher types 492-1071, a list such as the following would appear on the screen, with the target number and associated client information presented in the middle and in bold type.

```
492-1001 C. Nichols, 213 Elm Street, 08:12/09:00/
492-1043 D. William, ....
492-1055 E. Wladek, ....
492-1071 C. Jones, 2441 115 Street, 08:34/09:15/
492-1121 T. Todd, .....
492-1334 A. Allen, .....
492-2109 J. Black, ....
```

Specifically, you are to state what classes and methods you would use to organize and maintain this collection. Your design should recognize the size and growth of the collection and satisfy the reporting requirements as outlined above. You must briefly justify your choice of collection organization and associated methods in no more than the space below.

#2 [6 marks] Assume you must manage a collection of information.

When would you use an insertion sort?

When would you use a mergesort as opposed to quicksort?

#3 [10 marks] Pick a method for a List as described in the text which has a worst case performance of O(n), where n is the number of elements in the List. Prove by induction that this method is indeed O(n) in its worst case.

a) O(n) method chosen is: ______________________________

b) Show proof by induction that it is O(n) in worst case.

c) What is the order of the best case for your chosen method? _____

#4 [10 marks] Consider the implementation of a List Interface using a Doubly Linked representation. Four important methods we discussed for List are: `addToHead(Object)`, `addToTail(Object)`, `removeFromHead( )` and `removeFromTail( )`. Suppose you were implementing a stack, which two methods would you need? Provide an implementation for these two methods for Doubly Linked List.

a) Method chosen are: ____________________ and ____________________

b) Provide implementations in space below.

#5 [8 marks] Principle Questions:

Many principles are discussed in the textbook and four such principles are listed below. Choose one from a) or b) and one from c) or d) and, for each case, explain in at most two sentences what the principle means and provide an brief example of how/when the principle applies.

a) *Recursive structures must make "progress" towards a "base case".*

b) *Every public method of an object should leave the object in a consistent state.*

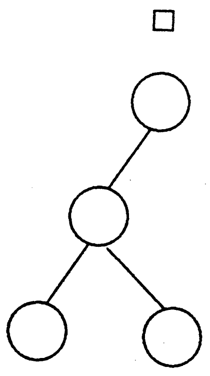
c) *Never modify a data structure while an associated* ***Enumeration*** *is live.*

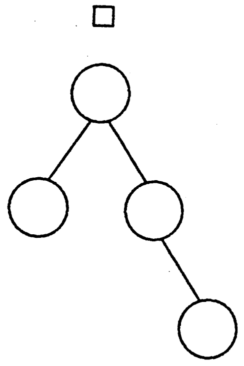
d) *Don't let opposing references show through the interface.*

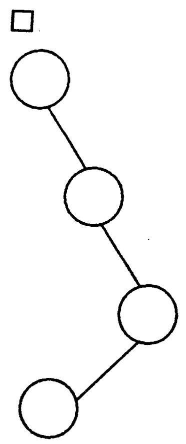
Explanation for _____ [which one a) or b)?].

Explanation for _____ [which one c) or d)?]

#6 [8 marks] Consider each of the following three different binary trees:


i) *inorder*

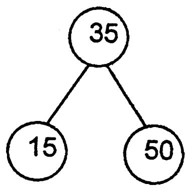

ii) *preorder*


iii) *postorder*

a) For tree i), fill in the nodes with values 6, 4, 2 and 8 in such a way that a inorder traversal will have the order: 2, 4, 6, 8. For tree ii), fill in the nodes with 6, 4, 2 and 8 in such a way that a preorder traversal will have the order: 2, 4, 6, 8. For tree iii), fill in the nodes with 2, 4, 6 and 8 in such a way that a postorder traversal will have the order: 2, 4, 6, 8.

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

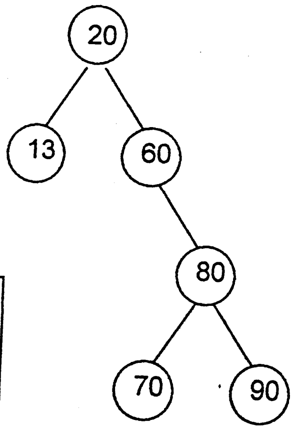
#7 [5 marks] Here is a binary search tree after the Integers 35, 15 and 50 have been added to an empty binary search tree in that order.



a) Draw this binary search tree after 60, 30, and 50 have been added in the given order.

b) Draw this binary search tree after 60 and then 35 have been removed from the tree you drew in answering part 7a).

#8 [15 marks] Consider a binary search tree (BST) such as the following. BSTs provide good search performance O(log n) provided that they are relatively balanced. One rough measure of balance is to compare the difference between the minimum path (MNP) from root to a leaf and the maximum path (MXP) from the root to a leaf. In the following tree the MXP – MNP = 3 -1 = 2. One strategy to assist in maintaining good search performance in a BST is to totally rebalance a tree if MXP – MNP > 2. Please answer the following questions related to this rebalancing strategy.



a) Complete the following method for calculating MXP.

```
public static int mxp (BinaryTreeNode n)
// post: returns the length of the longest path between the root node and
// a leaf of the tree
```

b) What is the order (in O notation) of your `mxp` method? ________

c) Discuss how you would implement a method called `rebalance` that would be invoked whenever `insert` method is invoked for a Binary Search Tree and the condition MXP – MNP > 2 holds. Note: you should not provide Java code for the method; simply describe the steps that should be carried out in `rebalance` in order that the resulting tree is guaranteed to have the property that MXP – MNP is either 0 or 1.

#9 [10 marks] The set of iterators for tree traversals have four methods excluding the constructor. Two of the methods are the same for all tree traversal iterators. What are they called? ____________ , ____________. The following two methods: `reset` and `nextElement` are unique to each iterator. Complete the code that is needed to implement these methods for an preorder traversal iterator.

```
protected BinaryTreeNode root;   // root of s subtree to be traversed
protected Stack todo;   // stack of unvisited ancestors of current
public void reset()  {

    ________________________________________;

     if (root != null)

        ________________________________________;
}

public Object nextElement() {
   //pre: hasMoreElements()
   //post: returns the next element, increments the iterator.
   BinaryTreeNode old;
   BinaryTreeNode current;
   Object result;

   ________________________________________;

   ________________________________________;

   if (old.right != null)

      ________________________________________;

   if ________________________________________

      ________________________________________;

  ________________________________________;
}
```

#10 [8 marks] Hash Tables:

a) Suppose you have a hash table with room for seven entries (indexed 0 through 6). This table uses open addressing with linear probing and a hash function that maps each String to its length modulo 7. Assume the table initially contains one element with the key of "Ellis" and the table is rehashed if the load factor gets over 0.7. Deleted entries should be marked as "reserved". Show the contents of just the key objects for the hash table (do not show the value object part) after the following methods are performed:

1. put ("Gretzky", centre)
2. put ("Salo", goalie)
3. put ("Lowe", coach)
4. remove ("Salo")
5. put ("Ranford", goalie)

0	
1	
2	
3	
4	"Dowd"
5	
6	

After operation 1

0	
1	
2	
3	
4	"Dowd"
5	
6	

After operation 2

0	
1	
2	
3	
4	"Dowd"
5	
6	

After operation 3

0	
1	
2	
3	
4	"Dowd"
5	
6	

After operation 4

0	
1	
2	
3	
4	"Dowd"
5	
6	

After operation 5

b) How many more entries must be added after operation 5 before we must rehash the table? ________

c) What is the average number of operations that must be performed to find an existing key in the hash table after operation 4 above has been completed?

d) What's wrong with the hash function we have chosen in this example?

#11 [10 marks] Consider adding a **MultiSortedVector** class to the project. When a MulitSortedVector is created, it is given an Array of aspect names, just like a MultiKeyedCollection. The code for the constructor is shown below. There is a Vector to keep track of the insertion order. The other instance variables for MultiSortedVector are similar to the instance variables for MultiKeyedVector. In MultiKeyedVector there was a Vector of aspect Strings and a parallel Vector of Arrays. In this case, we have a Vector of aspect Strings and a parallel Vector of Vectors. Each of these Vectors is a sorted Vector for the corresponding String aspect. Each time a new element is added to the MultiSortedVector, a reference to the new element is added to the correct location in each Vector, based on its sort order for each aspect. In addition, the new element is added to the end of the insertion Vector. The code for the **`add(MultiComparable)`** method is given, but it relies on the method **`findSortedLocation(Vector, String, MultiComparable)`**. Fill in the code for this method.

```
 public class MultiSortedVector {
 // Instance Variables
  protected Vector insertionVector;
  protected Vector aspects;
  protected Vector sortedVectors;

 // Constructor

 public MultiSortedVector(String[] aspects) {
 /*
  post: Initialize the MultiSortedVector to have zero elements.
 */
  int index;

  this.insertionVector = new Vector();
  this.aspects = new Vector();
  this.sortedVectors = new Vector();
  for (index = 0; index < aspects.length; index++) {
     this.aspects.addElement(aspects[index]);
     this.sortedVectors.addElement(new Vector());
  }
 }

public void add(MultiComparable element) {
/*
 post: The given element is added at the end of the insertion order Vector and at the
 appropriate locations for each of the sort order Vectors.
*/
 Vector vector; string aspect;
 int index;
 int location;
 Vector sortedVector;

 for (index = 0; index < this.aspects.size(); index++) {
    aspect = (String) this.aspects.elementAt(index);
    vector = (Vector) this.sortedVectors.elementAt(index);
    location = this.findSortedLocation(vector, aspect, element);
    vector.insertElementAt(element, location);
 };
 this.insertionVector.addElement(element);
}
```

```
protected int findSortedLocation(Vector vector, String aspect,
MultiComparable element) {
/*
 post: Return the index of the given Vector where the given element should be
 inserted, based on the given aspect.
*/
```

Have a Great Summer!