

# Computing Science 115

## Final Examination

April 17, 2000

2:00PM EDUCATION GYM

Section: B2- Sorenson

Last Name: \_\_\_\_\_

First Name: \_\_\_\_\_

Student #: \_\_\_\_\_

**E**  
04718  
CMPUT 115 (B2)  
SORENSEN  
APR 00 FINAL  
PAGES: 11

### Instructions:

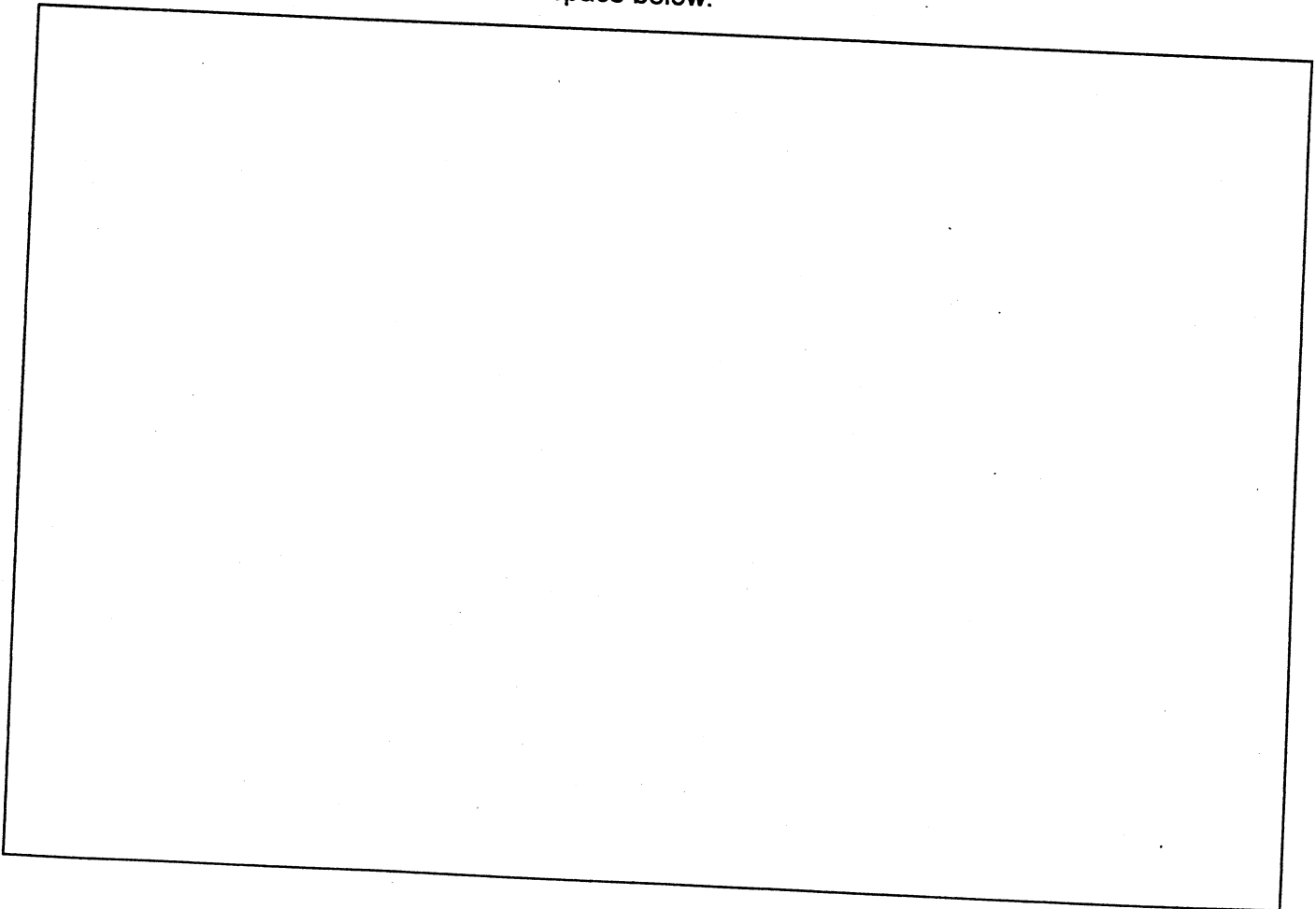
The time for this test is 3 hours. No references or calculators are allowed. Place all answers in this booklet and do not hand in any other work. The mark total for this exam is 100.

#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 manage a collection of 50,000 elements that correspond to the names of places (i.e., cities and towns) in the North America. This collection contains the following attributes about places: name, population, year of population count, country, province/state, latitude/longitude, and three largest industries. An example element might be <"Edmonton", 825000, 1999, "Canada", "Alberta", 53.5/113.5, "oil", "govt", "high tech">. How would you organize this collection assuming:

- it is initially unordered
- the collection grows at a rate of 250 elements per month.
- the information about a place changes on average once every three days.
- places are rarely removed from the collection.
- your program is to produce in response to the input of a place name, a report that lists all the information about that place. This report is produced on demand an average 500 times daily.
- your program is to produce a monthly report called the *almanac* that prints in alphabetical order by place name, the information about each place for all 50,000+ places in our collection.

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 a selection sort?

When would you use a quicksort as opposed to mergesort?

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

a)  $O(n)$  method chosen is: \_\_\_\_\_

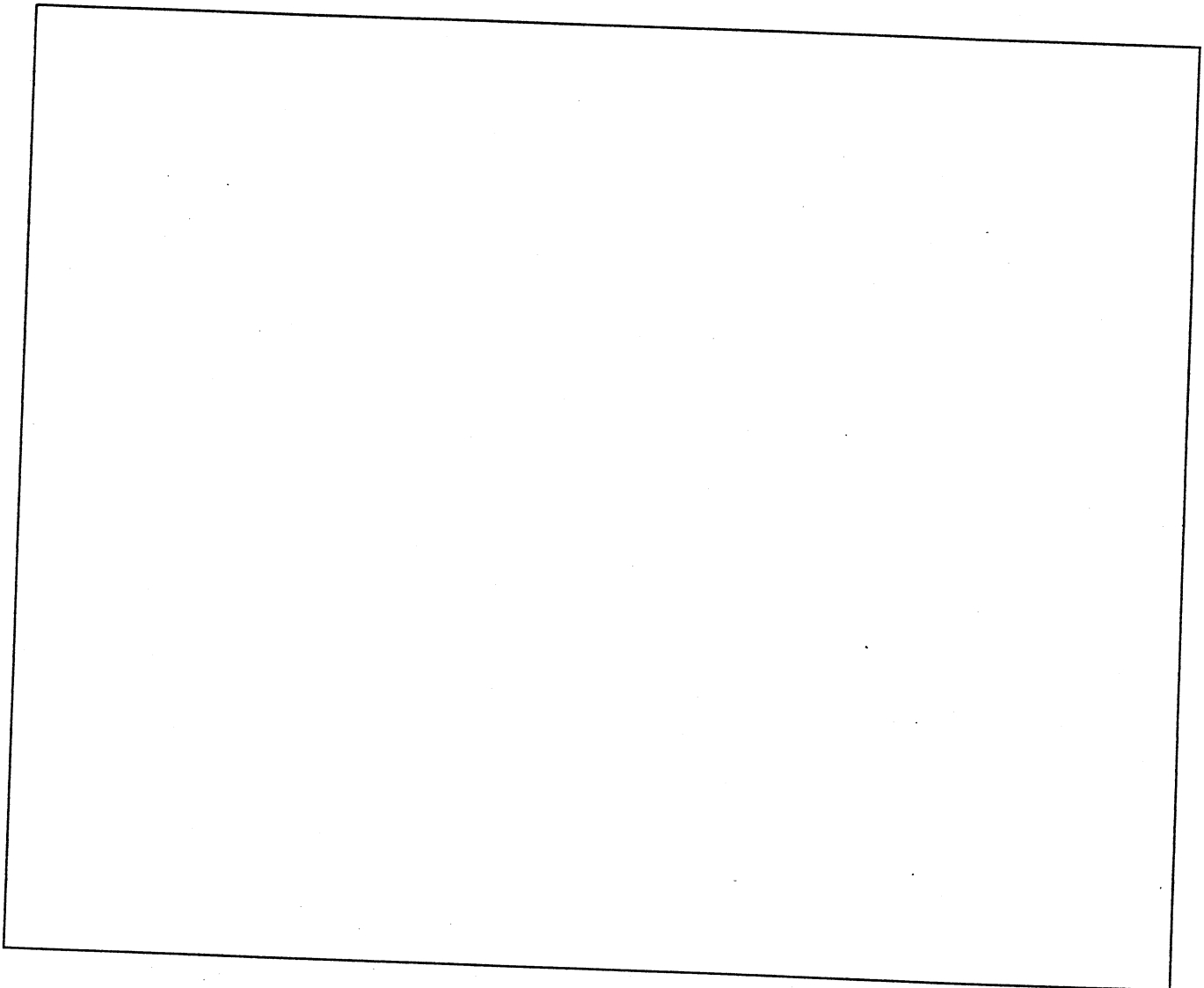
b) Show proof by induction that it is  $O(n)$  in worst case.

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 CircularList (the standard textbook implementation that keeps a reference to the last node of the list). Four important methods we discussed for CircularList are: `addToHead(Object)`, `addToTail(Object)`, `removeFromHead(Object)` and `removeFromTail(Object)`. Suppose you were implementing a queue, which two methods would you need? Provide an implementation for these two methods for CircularList.

a) Method chosen are: \_\_\_\_\_ and \_\_\_\_\_

b) Provide implementations in space below.



**#5 [8 marks ] Principle Questions:**

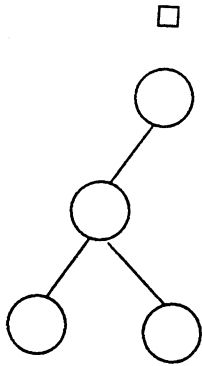
Many principles are discussed in the textbook and four such principles are listed below. Choose one from a) or b) and one from c) or d) and, for each case, explain in at most two sentences what the principle means and provide an brief example of how/when the principle applies.

- a) *Free the future: reuse code.*
- b) *Recursive structures must make "progress" towards a "base case".*
- c) *Declare parameters of overriding methods with the most general types possible.*
- d) *Provide a method for hashing the objects you implement.*

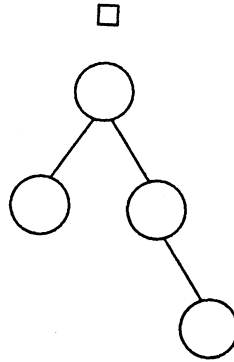
Explanation for \_\_\_\_ [which one a) or b)?].

Explanation for \_\_\_\_ [which one c) or d)?]

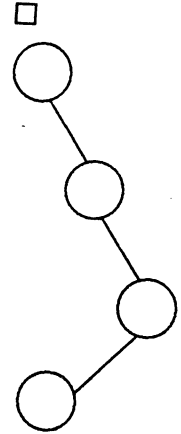
#6 [8 marks] Consider each of the following three different binary trees:



i) preorder



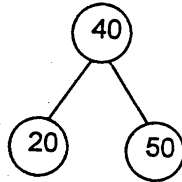
ii) postorder



iii) inorder

- a) For tree i), fill in the nodes with 1, 2, 3 and 4 in such a way that a preorder traversal will have the order: 1, 2, 3, 4. For tree ii), fill in the nodes with 1, 2, 3 and 4 in such a way that a postorder traversal will have the order: 1, 2, 3, 4. For tree iii), fill in the nodes with 1, 2, 3 and 4 in such a way that a inorder traversal will have the order: 1, 2, 3, 4.
- b) Put an x in the square above each binary tree from a) that is also a binary search tree.

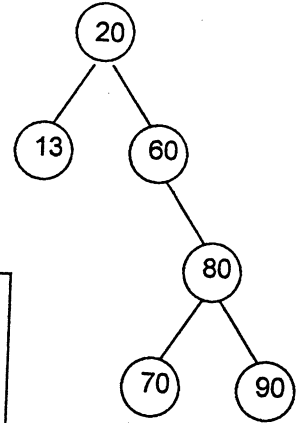
#7 [5 marks] Here is a binary search tree after the Integers 40, 20 and 50 have been added to an empty binary search tree in that order.



- a) Draw this binary search tree after 60, 30, and 40 have been added in the given order.

- b) Draw this binary search tree after 60 and then 20 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 MNP.

```
public static int mnp (BinaryTreeNode n)
```

```
// post: returns the length of the minimum path between the root node and
```

```
// a leaf of the tree
```

- b) What is the order (in O notation) of your `mnp` method? \_\_\_\_\_

- c) Discuss how you would implement a method called `rebalance` that would be invoked whenever `insert` method is invoked for a Binary 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 inorder traversal iterator.

```
protected BinaryTreeNode root; // root of s subtree to be traversed
protected Stack todo; // stack of unvisited ancestors of current
```

```
public void reset() {
```

```
    //post: resets the iterator to traverse again.
```

```
    BinaryTreeNode current;
```

```
    this.todo.clear();
```

```
    _____
    while (current != null){
```

```
        _____
        _____
    }
```

```
public Object nextElement() {
```

```
    //pre: hasMoreElements()
```

```
    //post: returns the next element, increments the iterator.
```

```
    BinaryTreeNode old;
```

```
    BinaryTreeNode current;
```

```
    Object result;
```

```
    _____
    _____
    while (current != null) {
```

```
        _____
        _____
    }
```

```
    _____;
}
```



#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 "Alan" 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 ("Yates", poet)
2. put ("Shakespeare", playwright)
3. put ("Dunn", financier)
4. remove ("Shakespeare")
5. put ("Wordswirthe", unknown)

0	
1	
2	
3	
4	"Alan"
5	
6	

After operation 1

0	
1	
2	
3	
4	"Alan"
5	
6	

After operation 2

0	
1	
2	
3	
4	"Alan"
5	
6	

After operation 3

0	
1	
2	
3	
4	"Alan"
5	
6	

After operation 4

0	
1	
2	
3	
4	"Alan"
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 arbitrary key in the hash table after operation 5 above has been completed?

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

**#11 [10 marks]** Consider adding a MultiSortedList class to the CMPUT 115 class project. When a MultiSortedList 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 DoublyLinkedList to keep track of the insertion order. The other instance variables for MultiSortedList are similar to the instance variables for MultiKeyedList. In MultiKeyedList 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 Array of SinglyLinkedListElements. Each of these SinglyLinkedListElements serves as the head of a sorted list for the corresponding String aspect. We cannot use an actual List since we need to add new elements in the middle to maintain the sort orders. Each time a new element is added to the MultiSortedList, a reference to the new element is added to the correct location in each List, based on its sort order for each aspect. In addition, the new element is added to the end of the insertion List. The code for the `add(MultiComparable)` method is given, but it relies on the method `placeInSortedList(int, String, MultiComparable)`. Fill in the code for this method in space provided on the next page.

```
public class MultiSortedList {
// Instance Variables
    protected DoublyLinkedList insertionList;
    protected Vector aspects;
    protected SinglyLinkedListElement[] sortedLists;

// Constructor

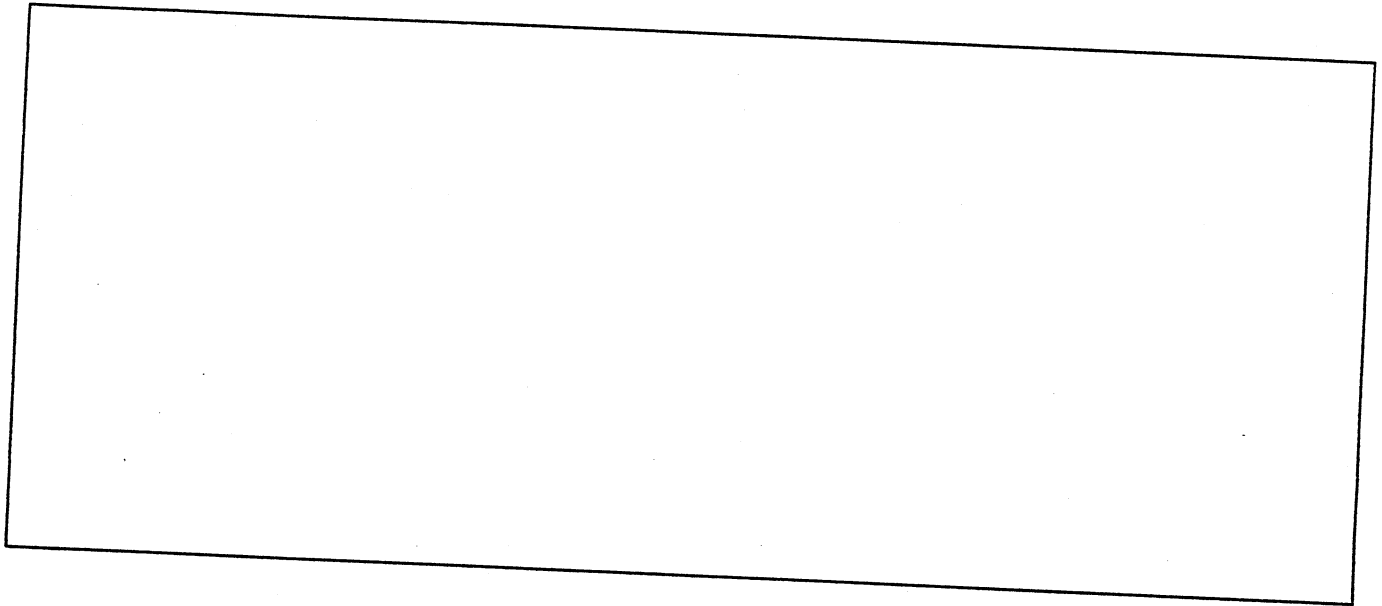
public MultiSortedList(String[] aspects) {
/*
    post: Initialize the MultiSortedList to have zero elements.
*/
    int index;

    this.insertionList = new DoublyLinkedList();
    this.aspects = new Vector();
    this.sortedLists = new SinglyLinkedListElement[aspects.length];
    for (index = 0; index < aspects.length; index++) {
        this.aspects.addElement(aspects[index]);
        this.sortedLists[index] = null;
    }
}

public void add(MultiComparable element) {
/*
    post: The given element is added at the end of the insertion order list and at the
    appropriate locations for each of the sort order lists.
*/
    int index;

    for (index = 0; index < this.aspects.size(); index++) {
        aspect = (String) this.aspects.elementAt(index);
        this.placeInSortedList (index, aspect, element);
    };
    this.insertionList.addToTail(element);
}
```

```
protected void placeInSortedList(int index, String aspect, MultiComparable
element) {
/*
  post: The given element is added at the appropriate location of the list with the
  given index using the given aspect.
*/
```



**Have a Great Summer!**