# CMPUT 115 Section B2
# Final Exam

April 20, 2001

**Instructions:**

- This is a closed book, no notes exam.
- You will be given 2 hours to complete the exam.
- Try to put all of your answers in the space provided.
- Be sure to write your student id number on each internal page.

| First Name: |
| --- |

| Last Name: |
| --- |

Student ID: _____

1. [1 Mark] Assume that **h1** and **h2** are **SinglyLinkedListElements**, and are each at the head of a nonempty list. Which of the following correctly appends the second list to the end of the first list.

a.
```
SinglyLinkedListElement tail = h1;
while ( tail != null ) tail = tail.next();
tail.setNext( h2 );
```

b.
```
SinglyLinkedListElement tail = h1;
while ( tail != null ) tail = tail.next();
tail = h2;
```

c.
```
SinglyLinkedListElement tail = h1;
while ( tail.next() != null ) tail = tail.next();
tail.setNext( h2 );
```

2. [2 Marks] Under what circumstances will a Hashtable's put method be a constant time operation?

3. [5 Marks] Paying attention to time efficiency, write a method that takes as an argument a Vector and returns the Object that appears most often in the Vector. The following is an example of how your method should work. There is space for your answer on the next page.

```
Vector v = new Vector();
v.addElement( "c" );  v.addElement( "b" );
v.addElement( "c" );  v.addElement( "x" );
v.addElement( "b" );  v.addElement( "a" );
// v now contains the String objects: c, b, c, x, b, a

// the following call should return c or b (it does not mater which)
Object obj = mostCommonElement( v );
```

```
public static Vector mostCommonELement( Vector v )
// pre:  v is not null and not empty
// post: returns the most common element in v
// hint: consider which data structure could be used to temporarily
//       (and efficiently) store the objects
{
```

4. [4 Marks] Implement the **depth** method of the **BinaryTreeNode** class.

```
public class BinaryTreeNode
{
    protected Object val; // value associated with node
    protected BinaryTreeNode parent; // parent of node
    protected BinaryTreeNode left; // left child of node
    protected BinaryTreeNode right; // right child of node


    public static int depth(BinaryTreeNode n)
    // post: returns the depth of a node in the tree
    {









    }
}
```

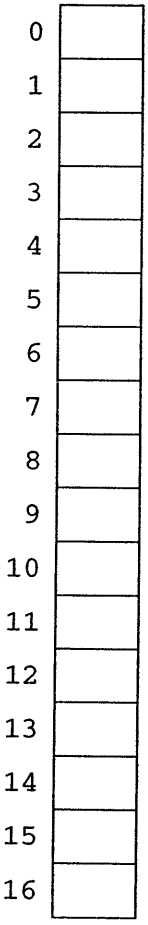
5. [3 Marks] Under what circumstances will a **BinarySearchTree's add** method be a $O(\log_2 n)$ operation?

6. [5 Marks] Recall that:

- `(new Integer(1)).hashCode()` returns `1`,
- `(new Integer(2)).hashCode()` returns `2`, and in general
- `(new Integer(i)).hashCode()` returns `i`.

Use the diagram below to describe the contents of a ChainedHashtable after the following code fragment has been executed. The source code for the put method is included at the end of this exam booklet.

```
ChainedHashtable ht = new ChainedHashtable(17);
ht.put( new Integer(9), "B" );
ht.put( new Integer(1), "B" );
ht.put( new Integer(10), "B" );
ht.put( new Integer(26), "B" );
ht.put( new Integer(27), "B" );
ht.put( new Integer(37), "B" );
ht.put( new Integer(10), "C" );
```




7. [8 Marks] On the following page there is a partial implementation of the **Hashtable** class from the structure package. A few lines of code are replaced with blanks (boxes actually). Each blank corresponds to one or two lines of code. Fill in each of the blanks.

```
public class Hashtable implements Dictionary {
        protected static Association reserved =
                    new Association("reserved",null);
        protected Association data[];
        protected int count;
        protected int capacity;
        protected final double loadFactor = 0.6;

        protected int locate(Object key) {
        // pre: key is non-null
        // post: returns ideal index of key in table
        //       (using linear probing)
            // compute an initial hash code
            int hash = Math.abs(                              );
            // keep track of first unused slot, in case we need it
            int firstReserved = -1;
            while (data[hash] != null) {
                if (data[hash] == reserved)
                                                              
                else // value located? if so return the index in table
                                                              

                                                              
            }
            // return first empty slot we encountered
            if (firstReserved == -1) return hash;
            else return firstReserved;
        }

        public Object remove(Object key) {
        // pre: key is non-null Object
        // post: removes key-value pair associated with key
            int hash = locate(key);
            Association a = data[hash];
            if (a == null || a == reserved)
            count--;
                                                              
                                                              
                                                              
        }
    }
```

8. [3 Marks] Draw the **BinarySearchTree** that would result if the following Integer Objects (and in the following order) were added: 8, 6, 9, 1, 2, 7, 1, 2, 2 and then 8.

9. [1 Mark] Given the tree built in question 8, what type of traversal will return the values in the order 1, 1, 2, 2, 2, 3, 6, 7, 8, 8 and 9.

   a. Preorder

   b. Inorder

   c. Postorder

   d. Levelorder

10. [2 Marks] What is the time-complexity of the following code? Express your answer as a function of **N**. Clearly show how you arrive at your answer.

```
Hashtable ht = new Hashtable(3*N);
for ( int i = 0; i < N; i++ )
  for ( int j = 0; j < N; j++ )
    ht.put( new Integer(i), new Integer(j) );
```

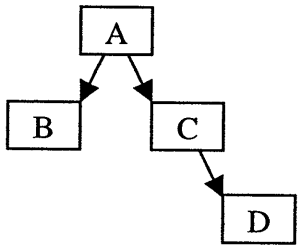
Student ID: _____

**Background information for question 11 and 12.**

The binary tree implementation that we studied in class is a linked structure. However a binary tree can, instead, be stored using an array of values. In this representation the root value of the tree is stored at position 0 in the array, its left child is stored at position 1 and its right child is stored at position 2. In general if a node is at position **i** then its children are at positions **2i+1** and **2i+2**. The following is a tree with 4 nodes stored in an array (in the diagram nulls are left blank).

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| A | B | C | | | | D |

Below is the same tree drawn as a linked structure. Notice that D is the right child of **C**, this is because **C** is at position 2 and **D** is at position 2*2+2.




11. [1 Marks] A **full** binary tree of height 4 is to be stored in an array as described above. What is the minimum length the array can be? Recall that the height of a tree is the length (i.e. the number of edges) of the longest path between the root node and a leaf. For example the tree in the diagram above has a height of 2.

    a. 30

    b. 31

    c. 32

    d. 33

12. [5 Marks] A **binary search tree** can be implemented using the array based binary tree described above. On the following page there is a partial implementation of such a binary search tree. Complete the **locate** method.

```
class ArrayBST implements OrderedStructure
// Array based implementation of a binary search tree.
{
      protected Comparable[] data;
      protected int count;

      public ArrayBST()
      // post: constructs an empty binary search tree.
      {
            data = new Comparable[100]; // some default size
            count = 0;
      }

      protected int locate( Comparable value )
      // pre:  value is non-null
      // post: returned: 1) position with the desired value (if present),
      //                 2) the position at which value should be added, or
      //                 3) -1 if there is no more room in data array.
      {
            int position = 0; // start search at the root
            // fill in the rest of this implementation ...









            if ( position < data.length ) return position;
            else return -1;
      }
}
```

**Code to help with question 6**

```
public class ChainedHashtable implements Dictionary
{
    protected List data[];
    protected int count;
    protected int capacity;

    public Object put(Object key, Object value)
    // pre: key is non-null object
    // post: key-value pair is added to hash table
    {
        List l = locate(key);
        Association newa = new Association(key,value);
        Association olda = (Association)l.remove(newa);
        l.addToHead(newa);
        if (olda != null)
        {
            return olda.value();
        }
        else
        {
            count++;
            return null;
        }
    }
}
```