Name _____ Section _____

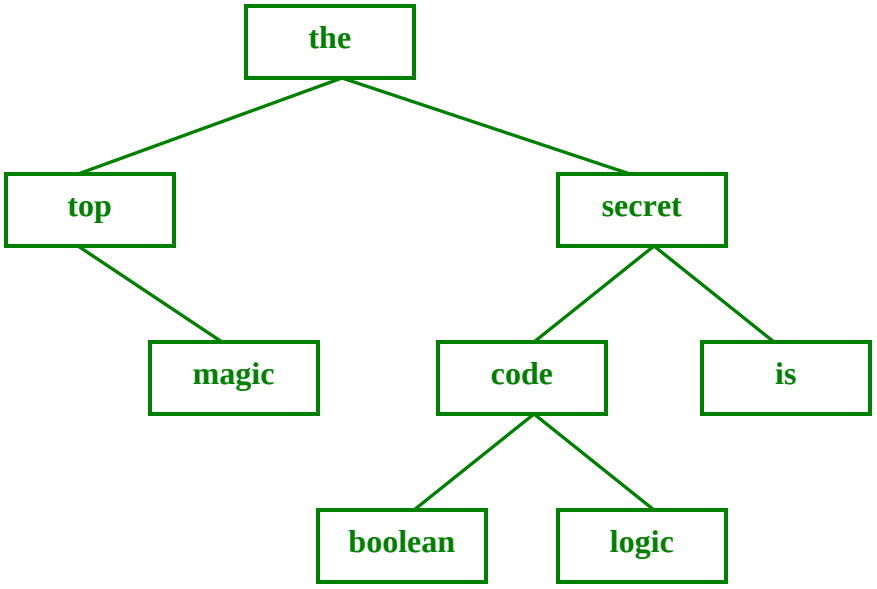
Question 1. (12 points) Binary trees. A newspaper reporter is trying to solve the mystery of the DeMorgan Code and has run into a clue that could benefit from your computer science expertise. The clue is that a particular set of words, when arranged in a binary tree, says something useful about the mystery. Unfortunately, the tree is not available, but the reporter has two lists of the words from the tree, one list showing an inorder traversal of the tree, and the other a postorder traversal.

Your job is to draw a picture of the one binary tree that contains these words in its nodes and whose traversals yield the words in the following orders:

Inorder: `top magic the boolean code logic secret is`

Postorder: `magic top boolean logic code is secret the`

Every node, both leaves and interior (non-leaf) nodes, contains one word. Draw a picture of the tree below:



Notes: While at first it may seem like this requires random tinkering to solve, there is a more systematic way to approach it. The postorder traversal uniquely identifies the root of the tree – it's the last thing in the list. Using this information, the inorder traversal can be used to identify all nodes to the left and the right of the root. Repeat for each subtree.

Question 2. (15 points) Linked lists. Suppose that we have a list class containing a ***sorted*** list of strings that uses a linked list to hold the data. The individual links are represented by instances of the following class:

```
/** links in a list of strings */
public class StringNode {
   public String s;         // string referred to by this link
   public StringNode next;  // next link in list; null if none
}
```

The sorted string list itself has a single instance variable, which is a reference to the first link in the list, or null if the list is empty.

```
/** a sorted list of strings */
public class SortedStringList {
   private StringNode front;    // first link or null if empty
}
```

Your job is to complete the definition of method `removeDups` in `SortedStringList` so that it removes all but one occurrence of any string that appears more than once in the (sorted) list. For example, if the list initially contains the strings `apple apple banana cherry cherry potato potato potato tomato`, then, after `removeDups` is executed, the list should contain `apple banana cherry potato tomato`. For full credit, your answer **must** run in linear ($O(n)$) time, **may not** define any new instance variables or other methods, and **may not** use any other collection data structures like lists, queues, or stacks.

```
/** remove duplicate occurrences of words from this list */
public void removeDups() {
   if (front == null) {
      return;
   }
   StringNode p = front;
   while (p.next != null) {
      if (p.s.equals(p.next.s)) {
         p.next = p.next.next;
      } else {
         p = p.next;
      }
   }

}
```

Question 3. (15 points) It is often useful when debugging or monitoring a program to have it print messages as it runs. For this question we want to create an extended version of the queue class used in assignment 4 so that it prints messages on `System.out` whenever an item is added to or removed from a queue. The messages should show the item added or removed and the resulting queue size.

The queue is specified by the following interface. (The difference from assignment 4 is that this is a simple queue of `String` values, not a generic class that has a parameter `<E>` for the type of the items in the queue.)

```
public interface StringQueue {
   // post: given value inserted at the end of the queue
   public void enqueue(String value);

   // pre : !isEmpty()
   // post: remove and return the value at the front
   //       of the queue
   public String dequeue();

   // post: return true if the queue is empty, false otherwise
   public boolean isEmpty();

   // post: return the current number of elements in the queue
   public int size();
}
```

Your job is to write a new class `LoggingStringQueue` that extends the class `LinkedStringQueue` that implements this interface. The `enqueue` and `dequeue` methods in the extended class should print messages to `System.out` as in the following example. If we execute the following operations:

```
LoggingStringQueue q = new LoggingStringQueue();
q.enqueue("cheese");
q.enqueue("grapes");
String ignore = q.dequeue();
q.enqueue("chocolate");
```

then the following messages should be printed on `System.out`:

```
cheese enqueued, size is 1
grapes enqueued, size is 2
cheese dequeued, size is 1
chocolate enqueued, size is 2
```

For full credit, your `LoggingStringQueue` class **must use** the appropriate operations of `LinkedStringQueue` to actually manipulate the queue. Write your answer on the next page. You may remove this page for reference.

Question 3. (cont.) The class you are to extend is the following:

```
class LinkedStringQueue implements StringQueue {
   // implementation omitted
   ...
}
```

Below, write your definition of class `LoggingStringQueue` that extends `LinkedStringQueue` as described on the previous page.

```
public class LoggingStringQueue extends LinkedStringQueue {

   public void enqueue(String value) {
      super.enqueue(value);
      System.out.println(value+" enqueued, size is "+size());
   }

   public String dequeue() {
      String result = super.dequeue();
      System.out.println(result+" dequeued, size is "+size());
      return result;
   }
}
```

Note: There is no need to define an explicit constructor, since a null (zero-argument) constructor that calls `super()` is assumed implicitly, which is what is needed.

Question 4. (15 points) Gardening. Early summer is a time for trees to grow. For this problem, we want to "grow" a binary tree by changing every leaf node into an interior (branching) node that has two new leaf nodes as children. Each node has an instance variable that has type `Color`. In an existing leaf node, the value of this instance variable can be either `Color.GREEN` or `Color.BROWN`; in an interior node, its value is always `Color.BROWN`. In a new leaf node, the color is always `Color.GREEN`. For example, here is a tree before and after a "grow" operation has been performed.

In other words, each leaf node becomes an interior node whose color is changed to `Color.BROWN` (if it wasn't already that color) and has as its children two new leaf nodes containing `Color.GREEN`.

A node is represented by instances of the following class:

```
public class TreeNode {
   public Color color;        // color of this node
   public TreeNode left;      // left child; null if none
   public TreeNode right;     // right child; null if none

   // constructor
   public TreeNode(Color c, TreeNode left, TreeNode right) ...
}
```

Complete the definition of procedure `grow`, on the next page, so it grows a tree as described above. For **full credit** you must use recursion to traverse the tree, and you may not define any global (instance) variables, or structured variables like lists, queues, stacks, or so forth. You may define additional helper (auxiliary) methods if they are useful.

(Note: It doesn't matter exactly how type `Color` is represented or what it is. All you need to know is that `Color.GREEN` and `Color.BROWN` are constants of type `Color` and can be stored in variables that have that type. If you need to, you can use `==` and `!=` to compare values of type `Color`.)

Question 4. (cont.) Write your solution below.

```
/** Grow the tree starting at node t by changing each leaf
 *  node so its color is brown and so it has two new child
 *  leaf nodes that are colored green */
public void grow(TreeNode t) {
   if (t == null) {
      return;
   }
   if {t.left == null && t.right == null) {
      t.color = Color.BROWN;
      t.left  = new TreeNode(Color.GREEN, null, null);
      t.right = new TreeNode(Color.GREEN, null, null);
   } else {
      grow(t.left);
      grow(t.right);
   }

}
```

Question 5. (15 points) Fractions. In this problem we want to implement part of a class used to represent fractions (just like in grade school – numbers like 1/2, 5/16, -7/32, etc.). A fraction is represented by two integer instance variables for the numerator and denominator. The denominator will always be positive, i.e., the representation of -(5/3) has a numerator of -5 and a denominator of 3, not 5 and -3. However fractions are not necessarily stored in lowest terms. All of the numbers 1/2, 3/6, 1024/2048, and so forth are possible representations for 1/2 and all are considered to be equal.

For this problem, implement a proper `compareTo` method for class `Fraction` that returns a negative, zero, or positive integer value depending on the result of the comparison between the two fraction values. Write your solution below.

```
public class Fraction implements Comparable<Fraction> {
   // instance variables
   private int num;      // fraction numerator
   private int denom;    // fraction denominator, always
                         //    positive (i.e., >0)

   // write your compareTo method for class Fraction here

   public int compareTo(Fraction other) {
      return this.num*other.denom - other.num*this.denom;
   }

}
```

Question 6. (8 points) Complexity. Answer **only one of the following two** questions. Either leave the other question blank, or cross out the other question so it is clear which one you want us to grade. If you answer both, we will grade the first one and ignore the second. Whichever one you pick, keep your answer brief and to the point, but be sure to include enough detail so it is clear that you understand the technical issues involved, as opposed to giving a general, fuzzy answer that is not specific.

Version 1. Under good conditions, the lookup, insert, and delete operations in a hash table take $O(1)$ time. However, the time needed can be significantly worse, up to $O(n)$. Explain why the worst-case time can be as bad as $O(n)$ and what is needed to ensure that the operations only require the expected time of $O(1)$.

Operations in a hash table can be slow if there are lots of collisions, where lots of key values wind up in the same bucket (hash table entry). In the worst case, where the number of items in a bucket is $O(n)$, the operations will also be $O(n)$.

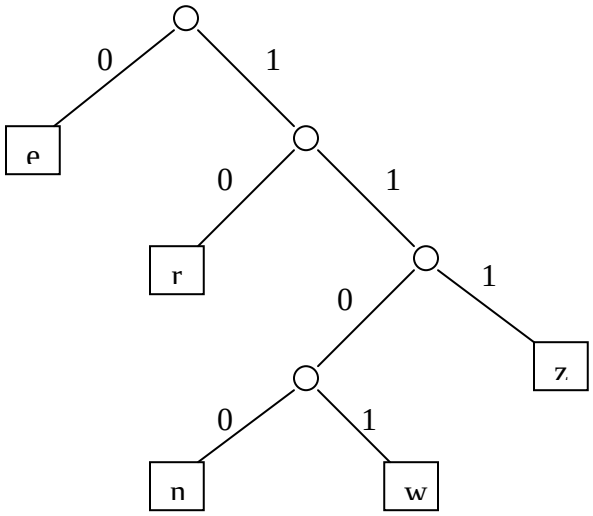
For the operations to be fast, we obviously need to be able to compute the hash function in $O(1)$ time, but we assume this is true. Beyond that that, the following two things need to be true:

- **The hash function should do a good job of distributing the keys uniformly among the available buckets.**
- **The number of buckets needs to be relatively large in proportion to the number of items being stored in the table. (More technically, the load factor = #items / #buckets, needs to be small.)**

Version 2. Under good conditions, quicksort requires $O(n \log n)$ time to sort an array containing n items. However, the time needed can be significantly worse, up to $O(n^2)$. Explain why the worst-case time can be as bad as $O(n^2)$ and what is needed to ensure that the sort only requires the expected time of $O(n \log n)$.

Quicksort is fast if the partition function picks good pivot values that result in two partitions each of which contain roughly half of the values in the original section of the array. In that case, the time needed by quicksort is $O(n \log n)$. If the partition function tends to pick either the largest or smallest value for the pivot, then each successive partition will result in one section of the array that is almost the same size as the previous section, and the total time needed will be roughly $O(n^2)$.

Question 7. (20 points) Magic secret decoder. In this problem we want to write a method that decodes characters that are encoded as a sequence of "0"s and "1"s. As in the Huffman code assignment, the main data structure is a tree whose leaves contain individual characters. The codes for the characters are described by the arrangement of nodes in the tree and the paths from the root to the leaf nodes. For example, one possible tree is the following:



In this tree, the code for 'e' is "0", the code for 'r' is "10", and so forth.

We will represent nodes of this tree as follows.

```
public class CodeNode {
   char ch;              // char value if leaf; value is not
                         //  specified or defined if not a leaf
   CodeNode left;        // left (0) subtree, or null if none
   CodeNode right;       // right (1) subtree, or null if none
}
```

Your job is to implement method decode on the next page. The input is a string of "0"s and "1"s and the root of the tree. The result of decode should be the integer code of the character if the input string defines a character in the tree. However, if the input does not give the code of a character in the tree, the result should be -1. For example (using the above tree):

Input string	Result of decode
"0"	101 (ASCII code for 'e')
"1101"	119 (ASCII code for 'w')
"101"	-1 (no such character)
"110"	-1 (incomplete code)

You may assume that the input string consists only of "0"s and "1"s, but you may not assume that it is the code for any specific character in the tree. You may also assume that the initial string to be decoded has at least one character in it ("0" or "1").

Question 7. (cont.) Complete the definition of method `decode` below. You may use either iteration or recursion to solve the problem, and you may define additional helper methods if you find them useful. For full credit, you **may not** define any instance (global) variables, or any variables holding collections of data like lists, queues, stacks, trees, and so forth.

Here are two solutions, one recursive and one iterative.

```
/* Return the integer value of the ASCII character described
 * by the code c, given the code tree starting at node t.  If
 * c does not describe a complete code in tree t, return -1.
 */
public int decode(String c, CodeNode t) {
   if (t == null) {
      return -1;
   }
   if (c.equals("")) {
      if (t.left == null && t.right == null) {
         return (int)t.ch;    // (int) cast is optional
      } else {
         return -1;
      }
   }
   if (c.charAt(0) == '0') {
      return decode(c.substring(1), t.left);
   } else { // c.charAt(0) is '1'
      return decode(c.substring(1), t.right);
   }
}

public int decode(String c, CodeNode t) {
   CodeNode p = t;
   int k = 0;
   while (p != null && k < c.length()) {
      if (c.charAt(k) == '0') {
         p = p.left;
      } else {  // c.charAt(k) == '1' assumed
         p = p.right;
      }
      k++;
   }
   // loop postcondition: p == null || k == c.length()
   if (p != null && p.left == null && p.right == null) {
      return (int)p.ch;
   } else {
      return -1;
   }
}
```