(a) Precisely and concisely explain what a data structure is and why we will spend an entire quarter studying data structures.

(b) Implement the ArrayList.add(E element) method assuming that the ArrayList has one attribute: Object[] data and that the value of data is never `null` when entering the add() method.

(a) Describe any differences between the approach used to implement the ArrayList in lecture verses the ArrayList implementation shown on the tutorial page.

(b) On the http://msoe.us/taylor/tutorial/cs2852/generics page, it states:
"*The main purpose of generics is to generate compiler errors... this is a good thing.*"

Recall the LinkedList class that we have been developing in lecture had one attribute: head. Implement the following method that will add an element to the front of the list:

```
public boolean addToFront(E element) {
```

}

Implement a Queue<E> class with the same three methods as the homework problem using an ArrayList<E> to store the items in the queue. The queue should not have a fixed size and should not allow `null` elements.

```
public class Queue<E> {
```

Consider a class called Image that has the following methods:

- int getHeight() – Returns the height of the image.
- int getWidth() – Returns the width of the image.
- boolean isFilled(int x, int y) – Returns true if the pixel at the specified location is filled.
- void setFilled(int x, int y) – Fills the pixel at the specified location.
- int areaFill(int x, int y) – Recursively fills the specified pixel and the area surrounding it up to bordering pixels at are filled or the edge of the image and returns the total number of pixels filled.

An IllegalArgumentException is thrown if an invalid pixel location is passed to any of the last three methods. Implement the areaFill method.

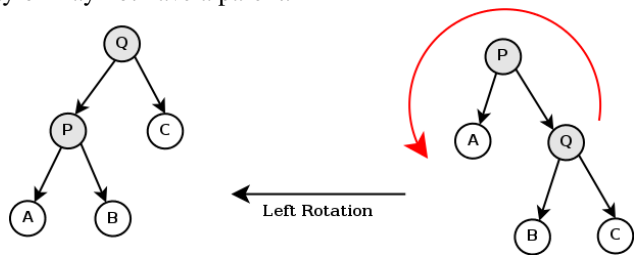
Recall that part of HashTable<E> class we developed in lecture looked something like this:

```
public class HashTable<E> {

  private int size;
  private Object[] table;

  public HashTable() {
    this(511);
  }

  public HashTable(int capacity) {
    table = new Object[capacity];
  }

  // ...

  public boolean contains(E target) {
    boolean found = false;
    if(target!=null) {
      int code = Math.abs(target.hashCode()%table.length);
      found = target.equals(table[code]);
    }
    return found;
  }
}
```

Rewrite the one-argument constructor and contains() method to handle collisions. Assume the following modification is made to the table attribute:

```
  private Collection<E>[] table;
```

Consider the following graphic that shows a similar rotation to the one discussed in lecture. Complete the leftRotate method shown below. The method is passed a reference to **P** and must return a reference to **Q**. The tree on the right shows the tree before leftRotate is called and the tree on the left shows the tree after the call. Keep in mind that **P** may or may not have a parent.




```
public Node leftRotate(Node p) {
  Node q = p.right;
  if(q!=null) {
    Node a = p.left;
    Node b = q.left;
    Node c = q.right;
    Node parent = p.parent;

  }
  return q;
}
```