(a) Consider each grouping of code below. Identify any errors in the code and explain what is wrong.

```
ArrayList<int> numbers = new ArrayList<int>();
```

```
List<String> words = new ArrayList<String>();
```

```
Integer[] integers = new Integer[3];
integers[0].toString();
```

(b) Describe any differences between the java.util.ArrayList and the wk1.ArrayList. List any advantages and disadvantages that result from the differences.

(a) Give the big-oh notation for the time complexity of the following algorithm where n is the size of the ArrayList passed to the method. Justify your answer.

```
public static void doStuff(ArrayList<Integer> numbers) {
  for(int i=0; i<numbers.size(); ++i) {
    numbers.set(i, numbers.get(i)*2);
  }
}
```

(b) Give the big-oh notation for the time complexity of the following algorithm where n is the size of the ArrayList passed to the method. Justify your answer.

```
public static int doStuff(ArrayList<Integer> numbers) {
  int middle = numbers.size()/2;
  return numbers.get(middle)*4;
}
```

Suppose that the LinkedList class that we have been developing in lecture was modified so that it only had one attribute: tail that pointed to the last node in the list. Further, suppose that the Node class was modified so that the next attribute was replaced with a prev attribute that points to the previous node in the list instead of the next node in the list. Implement the contains method for the list.

Consider the following partial implementation of a CircularQueue class. Identify all (if any) errors in the code. For each error identified, explain why it is an error.

```
public class CircularQueue<E> {

  private static final int DEFAULT_CAPACITY = 9;
  private E[] data;
  private boolean isEmpty;
  private int front;
  private int back;

  public CircularQueue() {
    this(DEFAULT_CAPACITY);
  }

  public CircularQueue(int capacity) {
    data = (E[])new Object[capacity];
    isEmpty = true;
    front = capacity-1;
    back = capacity-1;
  }

  public boolean offer(E element) {
    boolean added = false;
    if((isEmpty || front!=back) && element==null) {
      added = true;
      isEmpty = false;
      data[back++] = element;
    }
    return isEmpty;
  }
// ...









}
```

Implement the recursive add method called in the non-recursive add method below:

```
public class LinkedList<E> implements List<E> {
  private class Node {
    E value;
    Node next;

    private Node(E value) {
      this.value = value;
    }
  }

  private Node head = null;

  public boolean add(E value) {
    if(head==null) {
      head = new Node(value);
    } else {
      add(head, value);
    }
    return true;
  }
```

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;
```

(a) Recall the add method for the binary search tree that we developed in lecture a couple weeks ago that does not do any rebalancing. Draw what the tree would look like after the following items are added: **D, B, C, A, E, G, F**.

(b) In addition, recall that with the AVL tree we placed a number next to each node indicating whether the subtree was balanced (**0**) or unbalanced (positive or negative integer). For the tree you have drawn, write the AVL balance number next to each node in the tree.

(c) Finally, indicate the rotation(s) that must be performed in order to produce a tree that meets the rules of AVL trees.