CSE 143 Sample Final Exam #4
(Section Handout #20)

1. **Inheritance and Polymorphism**. Consider the following classes (`System.out.println` has been abbreviated as `S.o.pln`):

```
public class Eye extends Mouth {
    public void method1() {
        S.o.pln("Eye 1");
        super.method1();
    }
}

public class Mouth {
    public void method1() {
        S.o.pln("Mouth 1");
    }

    public void method2() {
        S.o.pln("Mouth 2");
        method1();
    }
}

public class Nose extends Eye {
    public void method1() {
        S.o.pln("Nose 1");
    }

    public void method3() {
        S.o.pln("Nose 3");
    }
}

public class Ear extends Eye {
    public void method2() {
        S.o.pln("Ear 2");
    }

    public void method3() {
        S.o.pln("Ear 3");
    }
}
```

The following variables are defined:

```
Mouth var1 = new Nose();
Ear var2 = new Ear();
Mouth var3 = new Eye();
Object var4 = new Mouth();
Eye var5 = new Nose();
Mouth var6 = new Ear();
```

In the table below, indicate in the right-hand column the output produced by the statement in the left-hand column. If the statement produces more than one line of output, indicate the line breaks with slashes as in "a / b / c" to indicate three lines of output with "a" followed by "b" followed by "c". If the statement causes an error, fill in the right-hand column with the phrase "error" to indicate this.

Statement	**Output**
`var1.method1();`	_____
`var2.method1();`	_____
`var3.method1();`	_____
`var1.method2();`	_____
`var2.method2();`	_____
`var3.method2();`	_____
`var4.method2();`	_____
`var5.method2();`	_____
`var6.method2();`	_____
`var1.method3();`	_____
`var2.method3();`	_____
`var3.method3();`	_____
`((Nose) var5).method3();`	_____
`((Eye) var1).method1();`	_____
`((Eye) var4).method1();`	_____
`((Nose) var1).method3();`	_____
`((Mouth) var4).method1();`	_____
`((Ear) var5).method3();`	_____
`((Eye) var6).method3();`	_____
`((Mouth) var4).method2();`	_____

2. **Inheritance and Comparable**. You have been asked to extend the `ArrayIntList` class that we have been studying. Recall that it includes the following public constructors and methods:

`public ArrayIntList()`	constructs an empty list with default capacity
`public ArrayIntList(int capacity)`	constructs an empty list with the given capacity
`public void add(int value)`	adds given value to end of list
`public void add(int index, int value)`	adds given value at given index
`public boolean contains(int value)`	returns whether value occurs anywhere in the list
`public int get(int index)`	returns value at given index
`public int indexOf(int value)`	index of first occurrence of value; -1 if not found
`public boolean isEmpty()`	returns whether list contains no elements
`public void remove(int index)`	removes the value at the given index
`public int set(int index, int value)`	sets element at given index to store given value
`public int size()`	returns number of elements in list
`public String toString()`	returns comma-separated string version of list

You are to **define a new class called `HistoryList` that extends this class through inheritance**. It should behave like an `ArrayIntList` except that it should keep track of the history of modifications to the list. This history is stored as a sequence of `String` values indicating the list's state at each point. For example, if the operations below at left are performed, then the `HistoryList` object should keep track of the sequence of `String`s shown below at right:

Operations	History Entries
`HistoryList list = new HistoryList();`	`"[]"`
`list.add(18);`	`"[18]"`
`list.add(27);`	`"[18, 27]"`
`list.add(0, 45);`	`"[45, 18, 27]"`
`list.remove(1);`	`"[45, 27]"`
`list.set(0, -15);`	`"[-15, 27]"`
`list.add(9);`	`"[-15, 27, 9]"`

When a `HistoryList` is constructed, the first `String` in the list above should be added to its history. Each time any of methods `add`, `remove`, or `set` are called, a new entry is added to the history showing the state of the list after the call. You must exactly reproduce the format shown above.

These `String`s that are part of the history will be accessed by clients using the following new methods:

`public int historySize()`	returns number of `String`s in history
`public String getHistory(int index)`	returns given history `String` (0=first, 1=second, etc.)

In the example above, after executing the sample code, the history size will be 6 and the six different history Strings can be accessed by calls on `getHistory` passing indexes between 0 and 5. For example:

```
for (int i = 0; i < list.historySize(); i++) {
     System.out.println(list.getHistory(i));
}
```

You must also **make `HistoryList` objects comparable to each other using the `Comparable` interface**. A `HistoryList` with more strings in its history is considered to be "greater than" one with fewer history entries. If two `HistoryList`s have the same number of history entries, the one that contains more elements (the one with the greater size) is considered to be greater. If the two lists have the same number of history entries and the same size, they are considered to be "equal" for this problem.

3. **Linked List Programming**. Write a method `removeDuplicates` that could be added to the `LinkedIntList` class from lecture and section. The method should remove any duplicates from the linked list of integers. The resulting list should have the values in the same relative order as their first occurrence in the original list. In other words, a value *i* should appear before a value *j* in the final list if and only if the first occurrence of *i* appeared before the first occurrence of *j* in the original list. For example, if a variable called `list` stores the following list:

```
[14, 8, 14, 12, 1, 14, 11, 8, 8, 10, 4, 9, 1, 2, 5, 2, 4, 12, 12]
```

After the call of `list.removeDuplicates();`, the list should store these values:

```
[14, 8, 12, 1, 11, 10, 4, 9, 2, 5]
```

Recall the `LinkedIntList` and node classes:

```
public class LinkedIntList {
    private ListNode front;

    methods
}

public class ListNode {
    public int data;       // data stored in this node
    public ListNode next;  // link to next node in the list
    ...
}
```

You may not call any methods of your linked list class to solve this problem, you may not construct any new nodes, and you may not use any auxiliary data structure to solve this problem (such as an array, `ArrayList`, `Queue`, `String`, etc). You also may not change any `data` fields of the nodes. You must solve this problem by rearranging the links of the list.

4. **Searching and Sorting**.

(a) Suppose we are performing a **binary search** on a sorted array called `numbers` initialized as follows:

```
// index          0   1   2   3   4   5   6   7   8   9  10  11  12
int[] numbers = {-5, -1,  3,  5,  7, 10, 18, 29, 37, 42, 58, 63, 94};

// search for the value 33
int index = binarySearch(numbers, 33);
```

Write the indexes of the elements that would be examined by the binary search (the `mid` values in our algorithm's code) and write the value that would be returned from the search. Assume that we are using the binary search algorithm shown in lecture and section.

- Indexes examined: _____
- Value Returned: _____

(b) Write the state of the elements of the array below after each of the first 3 passes of the outermost loop of the selection sort algorithm.

```
int[] numbers = {15, 56, 24,  5, 39, -4, 27, 10};
selectionSort(numbers);
```

(c) Trace the complete execution of the merge sort algorithm when called on the array below, similarly to the example trace of merge sort shown in the lecture slides. Show the sub-arrays that are created by the algorithm and show the merging of sub-arrays into larger sorted arrays.

```
int[] numbers = {15, 56, 24,  5, 39, -4, 27, 10};
mergeSort(numbers);
```

5. **Binary Search Trees**.

(a) Write the binary search tree that would result if these elements were added to an empty tree in this order:

- Leia, Boba, Darth, R2D2, Han, Luke, Chewy, Jabba

(b) Write the elements of your tree above in the order they would be visited by each kind of traversal:

- Pre-order: _____
- In-order: _____
- Post-order: _____

6. **Binary Tree Programming**. Write a method `equals` that could be added to the `IntTree` class from lecture and section. The method accepts another binary tree of integers as a parameter and compares the two trees to see if they are equal to each other. For example, if variables of type `IntTree` called `t1` and `t2` have been initialized, then `t1.equals(t2)` will return `true` if the trees are equal and `false` otherwise.

   Two trees are considered equal if they have exactly the same structure and store the same values. Each node in one tree must have a corresponding node in the other tree in the same location relative to the root and storing the same value. Two empty trees are considered equal to each other.

   You may define private helper methods to solve this problem, but otherwise you may not call any other methods of the class nor create any data structures such as arrays, lists, etc. Your method should not change the structure or contents of either of the two trees being compared.

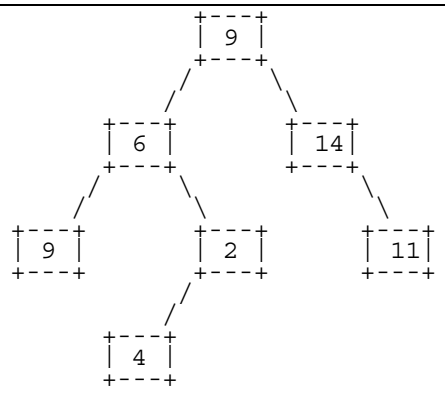
   Recall the `IntTree` and `IntTreeNode` classes as shown in lecture and section:

```
public class IntTreeNode {
    public int data;          // data stored in this node
    public IntTreeNode left;  // reference to left subtree
    public IntTreeNode right; // reference to right subtree

    public IntTreeNode(int data) { ... }
    public IntTreeNode(int data, IntTreeNode left, IntTreeNode right) {...}
}

public class IntTree {
    private IntTreeNode overallRoot;

    methods
}
```

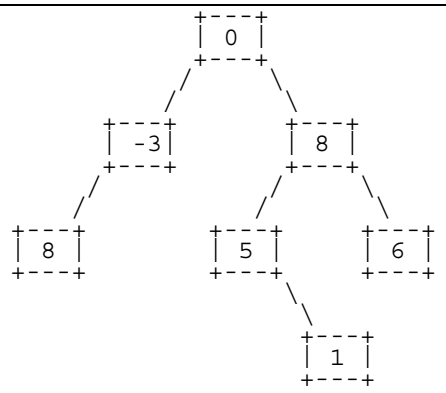
7. **Binary Tree Programming**. Write a method `combineWith` that could be added to the `IntTree` class from lecture and section. The method accepts another binary tree of integers as a parameter and combines the two trees into a new third tree which is returned. The new tree's structure should be a union of the structures of the two original trees; it should have a node in any location where there was a node in either of the original trees (or both). The nodes of the new tree should store an integer indicating which of the original trees had a node at that position (1 if just the first tree had the node, 2 if just the second tree had the node, 3 if both trees had the node).

   For example, suppose `IntTree` variables `t1` and `t2` have been initialized and store the following trees:

t1



t2

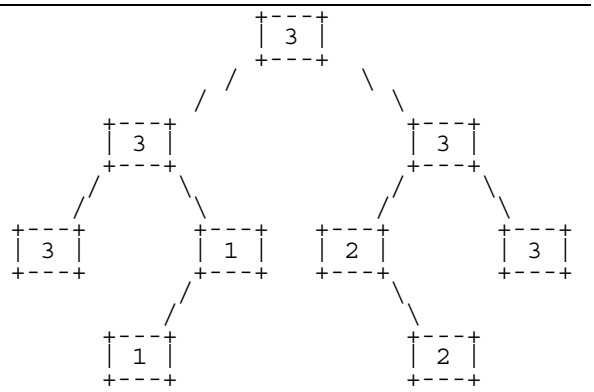


   Then the following call:

```
IntTree t3 = t1.combineWith(t2);
```

   will return a reference to the following tree:

t3



   You may define private helper methods to solve this problem, but otherwise you may not call any other methods of the class nor create any data structures such as arrays, lists, etc. Your method should not change the structure or contents of either of the two original trees being combined.