CSE 143, Winter 2009
Sample Final Exam #2

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

```
public class Bay extends Lake {
    public void method1() {
        S.o.pln("Bay 1");
        super.method2();
    }

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

public class Pond {
    public void method2() {
        S.o.pln("Pond 2");
    }
}

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

public class Lake extends Pond {
    public void method3() {
        S.o.pln("Lake 3");
        method2();
    }
}
```

The following variables are defined:

```
Lake var1 = new Ocean();
Pond var2 = new Pond();
Pond var3 = new Lake();
Object var4 = new Bay();
Lake var5 = new Bay();
Bay var6 = new Ocean();
```

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.method2();`	_____
`var2.method2();`	_____
`var3.method2();`	_____
`var4.method2();`	_____
`var5.method2();`	_____
`var6.method2();`	_____
`var1.method3();`	_____
`var2.method3();`	_____
`var3.method3();`	_____
`var4.method3();`	_____
`var5.method3();`	_____
`var6.method3();`	_____
`((Ocean) var5).method1();`	_____
`((Lake) var3).method3();`	_____
`((Lake) var4).method1();`	_____
`((Ocean) var1).method1();`	_____
`((Bay) var4).method1();`	_____
`((Lake) var2).method3();`	_____
`((Ocean) var5).method1();`	_____
`((Pond) var4).method2();`	_____

2. **Inheritance and Comparable Programming**. You have been asked to extend a pre-existing class `Date` that represents calendar dates such as March 19. The `Date` class includes these constructors and methods:

Constructor/Method	Description
`public Date(int month, int day)`	constructs a `Date` object with the given month/day
`public int getMonth()`	returns the month
`public int getDay()`	returns the day
`public void setMonth(int month)`	sets month to a new value
`public void setDay(int day)`	sets day to a new value
`public int daysInMonth(int month)`	returns number of days in the given month (examples: 4 → 30; 10 → 31; 2 → 28)
`public void nextDay()`	advances to next date, wrapping month if needed (3/19 → 3/20; 1/31 → 2/1; 12/31 → 1/1)
`public String toString()`	returns string version of date, such as `"03/19"`

You are to **define a new class called `CalendarDate` that extends this class through inheritance**. It should behave like a `Date` except that it should also keep track of the year. You should provide the same methods as the superclass, as well as the following new behavior:

Constructor/Method	Description
`public CalendarDate(int year, int month, int day)`	constructs a `CalendarDate` object with the given year/month/day
`public int getYear()`	returns the year
`public void setYear(int year)`	sets year to a new value

Some of the existing behaviors from `Date` should behave differently on `CalendarDate` objects:

- When a `CalendarDate` is printed with `toString`, it should be returned in a year/month/day format such as `"2009/03/19"`. Note that the year comes first. There should be a leading 0 if necessary if the month and/or day is less a single-digit number.
- When advancing a `CalendarDate` object to the next date using `nextDay`, if this is the last date of the year (December 31st), you should wrap the object to the next year. For example, the next day after December 31st, 2009 is January 1st, 2010.

You must also **make `CalendarDate` objects comparable to each other using the `Comparable` interface**. Calendar dates are compared by year, then by month, then by day. In other words, a `CalendarDate` object with a smaller year is considered to be "less than" one with a larger year. If two objects have the same year, the one with the lower month is considered "less." If they have the same year and month, the one with the lower day is considered "less." If the two objects have the same year, month, and day, they are considered to be "equal."

You may assume that all year/month/day values passed to your methods and constructors are valid. You should not worry about leap years for this problem; assume that February always has 28 days.

3. **Linked List Programming**. Write a method `removeAll` that could be added to the `LinkedIntList` class from lecture and section. The method should efficiently remove from a sorted list of integers all values appearing in a second sorted list of integers. For example, suppose `LinkedIntList` variables `list1` and `list2` refer to the following lists:

```
list1: [1, 3, 5, 7]
list2: [1, 2, 3, 4, 5]
```

If the call `list1.removeAll(list2);` is made, the lists should store the following values after the call:

```
list1:  [7]
list2:  [1, 2, 3, 4, 5]
```

Notice that all of the values from `list1` that appear in `list2` have been removed and `list2` is unchanged. If the call instead had been `list2.removeAll(list1);` , the lists would have these values afterwards:

```
list1:  [1, 3, 5, 7]
list2:  [2, 4]
```

Both lists are guaranteed to be in sorted (non-decreasing) order, although there might be duplicates in either list. Because the lists are sorted, you can solve this problem very efficiently with a single pass through the data. Your solution is required to run in O($M + N$) time where M and N are the lengths of the two lists.

Recall the `LinkedIntList` and `ListNode` classes seen in lecture and section:

```
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  13  14
int[] numbers = {-1,  0,  5, 11, 18, 29, 53, 57, 61, 64, 78, 82, 85, 89, 93};

// search for the value 86
int index = binarySearch(numbers, 86);
```

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 = {8, 5, -9, 14, 0, -1, -7, 3};
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 = {8, 5, -9, 14, 0, -1, -7, 3};
mergeSort(numbers);
```

5. **Binary Search Trees**.

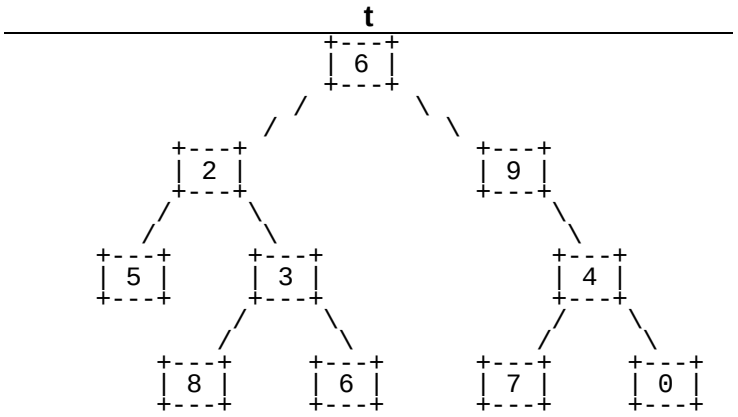
(a) Write the binary search tree that would result if these elements were added to an empty tree in this order:

- Lisa, Bart, Marge, Homer, Maggie, Flanders, Smithers, Milhouse

(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 `countMultiples` that could be added to the `IntTree` class from lecture and section. The method returns a count of the number of multiples of a particular value in the binary tree. Given a number *n*, a value *m* is considered a multiple of *n* if it can be expressed as (*k* * *n*) for some integer *k*. For example, suppose that an `IntTree` variable `t` stores a reference to the following tree:



The table below shows various calls and the values they should return:

Call	Value Returned	Reason
`t.countMultiples(2)`	6	six multiples of 2: 6, 2, 4, 8, 6, 0
`t.countMultiples(4)`	3	three multiples of 4: 4, 8, 0
`t.countMultiples(3)`	5	five multiples of 3: 6, 9, 3, 6, 0
`t.countMultiples(1)`	10	all ten numbers are multiples of 1

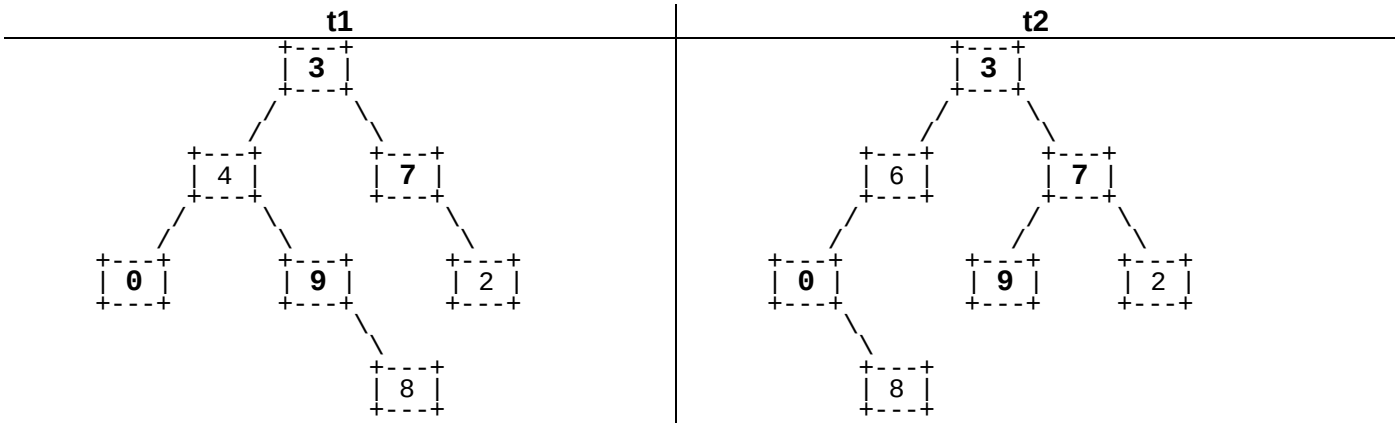
Your method should throw an `IllegalArgumentException` if passed the value 0. You may define private helper methods to solve this problem, but otherwise you may not call any other methods of the `IntTree` class nor create any data structures such as arrays, lists, etc. Your method should not change the 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 `matches` that could be added to the `IntTree` class from lecture and section. The method returns a count of the number of nodes in one tree that match nodes in another tree. A match is defined as a pair of nodes that are in the same position in the two trees relative to their overall root and that store the same data.

   For example, suppose `IntTree` variables `t1` and `t2` refer to the following trees (matches underlined):



   The calls of `t1.matches(t2)` and `t2.matches(t1)` would each return `4`. The overall root of the two trees match (both are 3). The nodes at the top of the left subtrees of the overall root do not match (one is 4 and one is 6). The top of the right subtrees of the overall root match (both are 7). The next level of the tree has 2 matches for the nodes storing 0 and 2 (there are two nodes that each store 9 at this level, but they are in different positions relative to the overall root of the tree). The nodes at the lowest level both store 8, but they aren't a match because they are in different positions.

   You may define private helper methods to solve this problem, but otherwise you may not call any other methods of the `IntTree` 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 examined.