CSE 143 Sample Final Exam #6

1. Inheritance and Polymorphism

Consider the following classes:

```
public class Computer extends Mineral {
  public void b() {
    System.out.println("Computer b");
    super.b();
  }

  public void c() {
    System.out.println("Computer c");
  }
}

public class Mineral extends Vegetable {
  public void b() {
    System.out.println("Mineral b");
    a();
  }
}

public class Animal extends Mineral {
  public void a() {
    System.out.println("Animal a");
  }

  public void c() {
    b();
    System.out.println("Animal c");
  }
}

public class Vegetable {
  public void a() {
    System.out.println("Vegetable a");
  }

  public void b() {
    System.out.println("Vegetable b");
  }
}
```

and that the following variables are defined:

```
Vegetable var1 = new Computer();
Mineral   var2 = new Animal();
Vegetable var3 = new Mineral();
Object    var4 = new Mineral();
```

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.a();`	_____
`var1.b();`	_____
`var1.c();`	_____
`var2.a();`	_____
`var2.b();`	_____
`var3.a();`	_____
`var3.b();`	_____
`var4.a();`	_____
`((Computer) var1).b();`	_____
`((Computer) var1).c();`	_____
`((Computer) var2).c();`	_____
`((Animal) var2).c();`	_____
`((Computer) var3).b();`	_____
`((Vegetable) var4).b();`	_____
`((Animal) var4).c();`	_____

2. Inheritance and `Comparable` Programming

You have been asked to extend a pre-existing class `Student` that represents a college student. A student has a name, a year (such as 1 for freshman and 4 for senior), and a set of courses that he/she is taking.

Member	Description

`private String name`
`private int year`
`private Set<String> courses`
	private data of each student

`public` **`Student`**`(String name, int year)`	constructs a student with the given name and year; initially not taking any courses
`public void` **`addCourse`**`(String name)`	adds the given course to this student's set of courses
`public void` **`dropAll`**`()`	removes the student from all of his/her courses
`public int` **`getCourseCount`**`()`	returns number of courses the student is taking
`public String` **`getName`**`()`	returns student's name
`public double` **`getTuition`**`()`	returns the amount of money the student pays for tuition each year, which is $1,234.50 per course
`public int` **`getYear`**`()`	returns student's year, from 1 (freshman) to 4 (senior)

You are to **define a new class called `GradStudent` that extends `Student` through inheritance**. A `GradStudent` should behave like a `Student` except for the following differences:

- A grad student keeps track of a **research advisor**, which is a professor working with the student.
- Grad students are considered to be **4 years further ahead** than typical students. So, for example, a grad student in year 1 of grad school is really in year 5 of school overall. The `getYear` method should still return the student's overall year in school; for example, a 3rd-year grad student should return 7 from `getYear`.
- Grad students can enroll in a **maximum of 3 courses** at a time. If a grad student tries to add additional courses beyond 3, the course is not added to the student's set of courses.
- Grad students **pay tuition differently** than other students. If the grad student is enrolled in 1 course or no courses, the grad student pays a flat rate of tuition of $500.00. If the grad student is enrolled in 2 or 3 courses, he/she pays twice as much as the normal student rate for all of his/her courses.
- If grad students work too much, they become "**burnt out**." A burnt-out student is one who is in his/her 5th or higher year of grad school (9th or higher year of school overall) or one who is taking 3 courses.

You should provide the same methods as the superclass, as well as the following new behavior.

Constructor/Method	Description
`public` **`GradStudent`**`(String name, int year, String advisor)`	constructs a graduate student with the given name, given year (where 1 is the first year of grad school, 5th year of school overall; etc.), and research advisor
`public String` **`getAdvisor`**`()`	returns this grad student's research advisor
`public boolean` **`isBurntOut`**`()`	returns true if grad student is "burnt out" (in at least the 5th year of grad school, and/or taking 3 courses)

You must also **make `GradStudent` objects comparable to each other using the `Comparable` interface**. `GradStudent`s are compared by year in ascending order, breaking ties by count of courses in ascending order, and further breaking ties by advisor in ascending alphabetical order. In other words, a `GradStudent` object with a lower year is considered to be "less than" one with a higher year. If two students have the same year, the one with a lower count of courses is considered to be "less than" one with a higher course count. If the two students have the same year *and* the same number of courses, the one whose advisor's name comes first in alphabetical order is considered "less." (You should compare the strings as-is and not alter their casing, spacing, etc.) If the two objects are the same by all three of these criteria, they are considered to be "equal."

The majority of your grade comes from implementing the correct behavior. Part of your grade also comes from appropriately utilizing the behavior you have inherited from the superclass and not re-implementing behavior that already works properly in the superclass. You may assume valid arguments are passed to all of your class's methods and constructors.

3. Linked List Programming

Write a method `removeEvens` that could be added to the `LinkedIntList` class from lecture and section. The method removes all elements from the list that store even numbers (numbers divisible by 2, such as 14, -6, 48, 0, 22, etc). Your method should also add a value to the end of the linked list that stores a count of how many elements were removed. (This might be an even number; that's okay.)

Suppose a `LinkedIntList` variable named `list` stores the following values:

```
[4, 2, 1, 10, 15, 8, 7, 4, 20, 36, -3, 40, 5]
```

The call of `list.removeEvens();` would change the list to store the following elements:

```
[1, 15, 7, -3, 5, 8]
```

Note that the `8` on the end of the list signifies that the call removed eight elements. If the list is empty or does not contain any even elements, it is unchanged except that a new node will be added to the end containing the value `0` (because no nodes were removed).

For full credit, obey the following restrictions in your solution:

- The method should run in no worse than O(*N*) time, where *N* is the length of the list.
  For full credit, you can make up to 2 passes over the list at most.
- Do not call any methods of the linked list class to solve this problem.
  Note that the list does not have a `size` field, and you are not supposed to call its `size` method.
- Do not use auxiliary data structures such as arrays, `ArrayList`, `Queue`, `String`, etc.
- Do not modify the `data` field of any nodes; you must solve the problem by changing the links between nodes.
- You may not create new `ListNode` objects, except for the single new node that you are to add to the end of the list. You may create as many `ListNode` variables as you like.

You are using the `LinkedIntList` and `ListNode` class as defined in lecture and section.

---

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 = {-5, -1,  0,  3,  9, 14, 19, 24, 33, 41, 56, 62, 70, 88, 99};

int index = binarySearch(numbers, 18);
```

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 elements of the array below after each of the first 3 passes of the outermost loop of a **selection sort**.

```
int[] numbers = {29, 17, 3, 94, 46, 8, -4, 12};
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 = {29, 17, 3, 94, 46, 8, -4, 12};
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:

- Pikachu, Slowpoke, Vulpix, Golbat, Blastoise, Jigglypuff, Tentacool, Charizard, Weedle, Squirtle, Mew

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

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

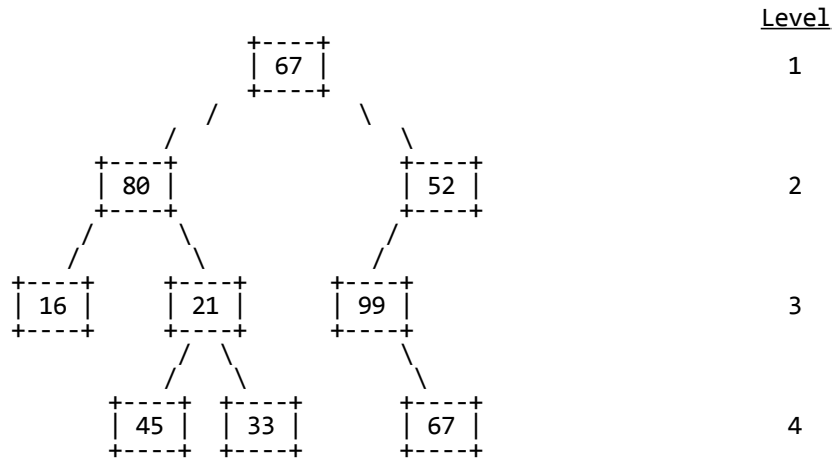
6. Binary Tree Programming

Write a method `countBelow` that could be added to the `IntTree` class from lecture and section. The method accepts an integer parameter representing a given level of the tree and returns an integer representing the count of the number of nodes in exist in the tree at that level or deeper. For example, if 3 is passed, you should count the number of nodes at levels 3, 4, 5, 6, ... For this problem, the root of a tree is defined to be at level 1, its children are at level 2, etc.

If the tree is empty or does not have any nodes at the given level or deeper, you should return 0. If a negative value or 0 is passed, your method should throw an `IllegalArgumentException`.

The following diagram and table show the results of several calls of your method on a particular tree:

```
IntTree tree = new IntTree();
...
```



call	returns
`tree.countBelow(3)`	`6` (16, 21, 99, 45, 33, and 67)
`tree.countBelow(4)`	`3` (45, 33, and 67)
`tree.countBelow(5)`	`0`
`tree.countBelow(1)`	`9` (all of the nodes in the tree)
`tree.countBelow(0)`	`IllegalArgumentException`
`tree.countBelow(-2)`	`IllegalArgumentException`

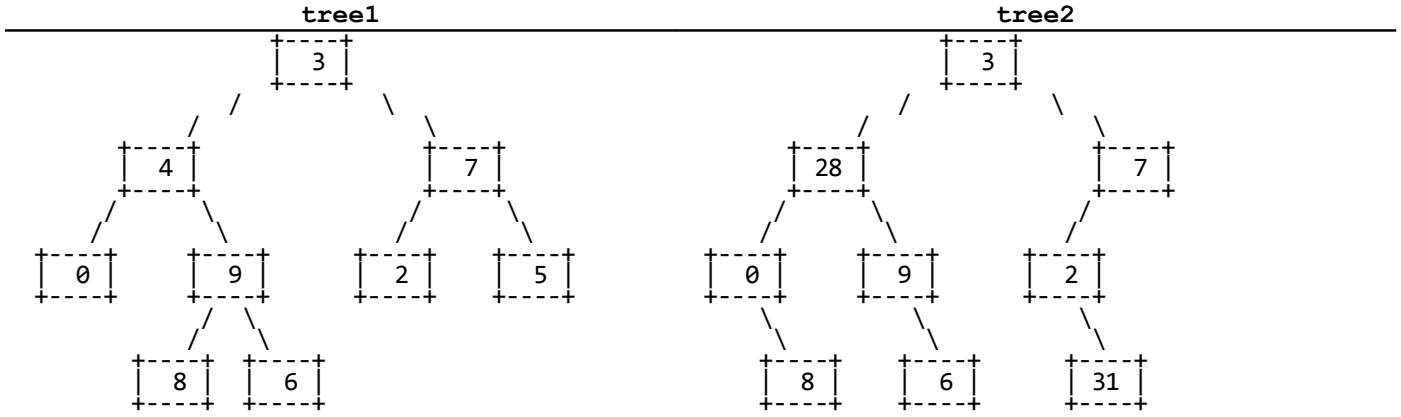
You may define private helper methods to solve this problem, but otherwise you may not call any other methods of the tree class nor create any data structures. Do not construct any new node objects or change the `data` of any nodes.

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

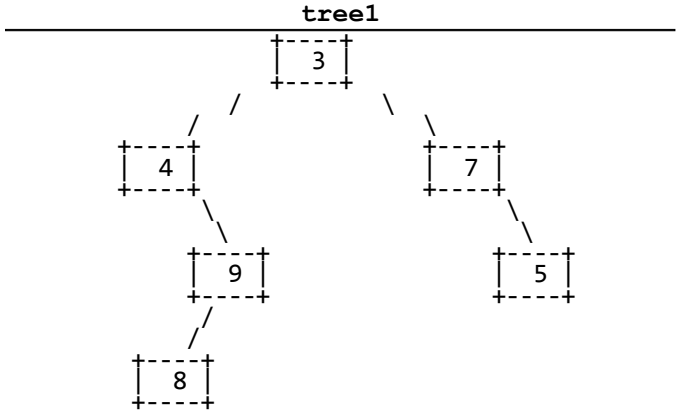
7. Binary Tree Programming

Write a method `removeMatchingLeaves` that could be added to the `IntTree` class from lecture and section. The method accepts a second `IntTree` as its parameter and removes any leaf nodes (nodes without children) that "match" with a corresponding node in the other tree. For this problem, two nodes "match" if they contain the same `data` value and are located in exactly the same place in the tree relative to the trees' overall roots. For a node to be removed by your method, it must be a leaf in your tree; but it need not be a leaf in the other tree passed as a parameter. If the second tree doesn't contain a node at that corresponding location or the value there is different, it does not match.

For example, suppose two variables of type `IntTree` called `tree1` and `tree2` store the following elements:



The call of `tree1.removeMatchingLeaves(tree2);` would modify `tree1` to store the elements below. The leaf nodes containing 0, 2, and 6 have been removed because they match the corresponding nodes in `tree2`. The other leaves from `tree1`, which contain the values 8 and 5, are not removed because they aren't matched in `tree2`.



You may define private helper methods, but otherwise you may not call any other methods of the class nor create any data structures. You should not construct any new node objects or modify the tree passed in as the parameter. It may be helpful for you to note that your method can directly access the other tree's `overallRoot` field with an expression such as `tree2.overallRoot` (this is allowed in Java because they are objects of the same class).

For full credit, your solution must be recursive and utilize the *x = change(x)* pattern discussed in lecture.