CSE 143 Sample Final Exam #5

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

```
public class Blue extends Green {
    public void one() {
        System.out.println("Blue 1");
        super.one();
    }
}

public class Red extends Yellow {
    public void one() {
        super.one();
        System.out.println("Red 1");
    }

    public void two() {
        System.out.println("Red 2");
        super.two();
    }
}

public class Yellow extends Blue {
    public void two() {
        System.out.println("Yellow 2");
    }

    public void three() {
        two();
        System.out.println("Yellow 3");
    }
}

public class Green {
    public void one() {
        System.out.println("Green 1");
    }

    public void three() {
        System.out.println("Green 3");
    }
}
```

The following variables are defined:

```
Green var1 = new Blue();
Green var2 = new Red();
Blue var3 = new Yellow();
Object var4 = new Green();
```

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.one();`	______________________________
`var1.two();`	______________________________
`var1.three();`	______________________________
`var2.one();`	______________________________
`var2.two();`	______________________________
`var2.three();`	______________________________
`var3.two();`	______________________________
`var3.three();`	______________________________
`var4.one();`	______________________________
`((Blue) var1).one();`	______________________________
`((Yellow) var1).two();`	______________________________
`((Red) var2).three();`	______________________________
`((Yellow) var2).two();`	______________________________
`((Green) var4).three();`	______________________________
`((Yellow) var4).two();`	______________________________

2. **Inheritance and Comparable**. You have been asked to extend a pre-existing class `Ticket` that represents a ticket to attend an on-campus event. A ticket has a price and also stores how many days early the ticket was bought (how many days before its event takes place). Some tickets have "promotion codes" that can allow special access and benefits for the ticket holder.

The `Ticket` class includes the following members:

Member	Description

`private double price`
`private int daysEarly`
`private String promotionCode`
	private data of each ticket

`public` **`Ticket`**`(double price, int daysEarly)`	constructs a ticket with the given price, purchased the given number of days early, with no promotion code
`public int` **`getDaysEarly`**`()`	returns how many days early the ticket was bought
`public double` **`getPrice`**`()`	returns the ticket's price
`public String` **`getPromotionCode`**`()`	returns the ticket's promotion code (`""` if none)
`public void` **`setPromotionCode`**`(String code)`	sets the ticket's promotion code to the given value (throws an exception if `null` is passed)
`public String` **`toString`**`()`	returns a string representation of the ticket

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

- Student tickets are always bought by the campus ticket sales agency, **two weeks (14 days)** ahead of the event.
- Students always get a **1/2 price discount** off the initial price of any ticket to any event.
- Honor students get an **additional $5 off** the price after the 1/2 discount, down to a minimum cost of $0 (free).
- Student tickets have special promotion codes. Any promotion code that is set on a student ticket should be modified to have the **suffix `" (student)"` attached**. For example, if the client sets the promotion code to `"KEXP call-in winner"`, a student ticket should actually store `"KEXP call-in winner` **`(student)`**`"`.

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

Constructor/Method	Description
`public` **`StudentTicket`**`(double price, boolean honors)`	constructs a student ticket with the given base (non-discounted) price, possibly for an honor student
`public boolean` **`isHonorStudent`**`()`	returns `true` if ticket is for an honor student

Note that some of the existing behaviors from `Ticket` should behave differently on `StudentTicket` objects, as described previously.

You must also **make `StudentTicket` objects comparable to each other using the `Comparable` interface**. `StudentTicket`s are compared by price, breaking ties by promotion code. In other words, a `StudentTicket` object with a lower price is considered to be "less than" one with a higher price. If two tickets have the same price, the one whose promotion code 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 have the same price and promotion code, 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.

3. **Linked List Programming**. Write a method `trimEnds` that could be added to the `LinkedIntList` class from lecture and section. The method accepts an integer parameter *k* and removes *k* elements from the front of the list and *k* elements from the back of the list. Suppose a `LinkedIntList` variable `list` stores the following values:

   `[10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110]`

   The call `list.trimEnds(3);` would change the list to store the following elements:

   `[40, 50, 60, 70, 80]`

   If we followed this by a second call of `list.trimEnds(1);` , the list would store the following elements:

   `[50, 60, 70]`

   If the list is not large enough to remove *k* elements from each side, throw an `IllegalArgumentException`. If the list contains exactly 2*k* elements, it should become empty as a result of the call. If *k* is 0 or negative, the list should remain unchanged.

   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.
     - (You can make more than one pass over the list, but you may not make *k* or *N* passes over it.)
   - Do not call <u>any</u> 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, though you may create as many `ListNode` variables as you like.

   Assume that you are using the `LinkedIntList` and `ListNode` class as defined 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;  // a link to the next node in the list

    public ListNode() { ... }
    public ListNode(int data) { ... }
    public ListNode(int data, ListNode next) { ... }
}
```

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, 37);
```

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 = {22, 88, 44, 33, 77, 66, 11, 55};
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 = {22, 88, 44, 33, 77, 66, 11, 55};
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:

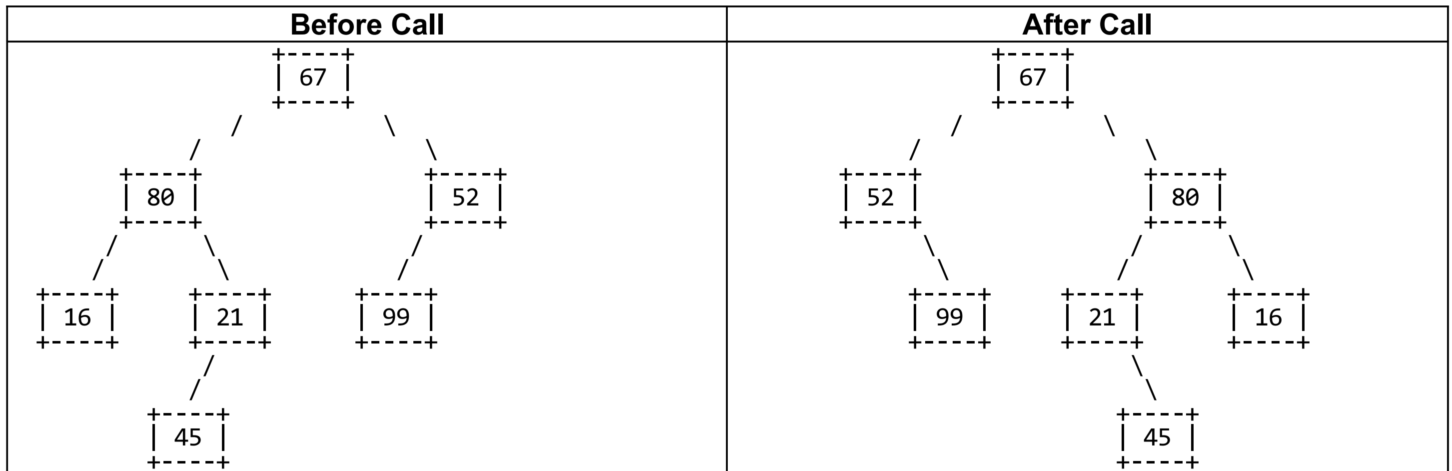
- Hill, Barbs, Stat, Dudley, Lopez, Rich, Frye, Nash, Dragic, Amund

(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 `flip` that could be added to the `IntTree` class from lecture and section. The method reverses the tree about its left/right axis so that any node that used to be its parent's left child will become its parent's right child and vice versa. The table below shows the result of calling this method on an `IntTree` variable `tree`.

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



If a tree has no nodes or is a leaf, it should not be affected by a call to your method.

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 such as arrays, lists, etc. You should 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:

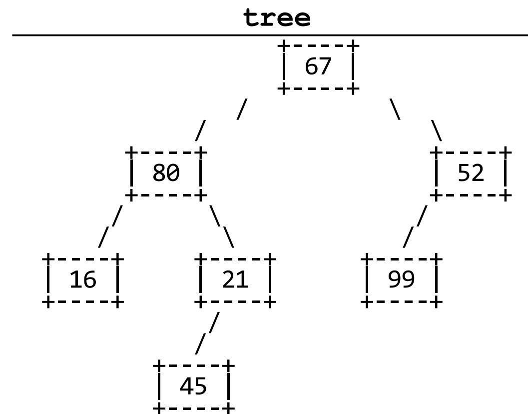
```
public class IntTreeNode {
    public int data;           // data stored in this node
    public IntTreeNode left;   // reference to left subtree
    public IntTreeNode right;  // reference to right subtree

    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 `hasPath` that could be added to the `IntTree` class from lecture and section. The method accepts *start* and *end* integers as parameters and returns `true` if a path can be found in the tree from *start* down to *end*. In other words, both *start* and *end* must be found in the tree, and end must be in one of *start*'s subtrees; otherwise the method returns `false`. The result is trivially `true` if *start* and *end* are the same; in such a case, you are simply checking whether a node exists in the tree with that value. For example, suppose a variable of type `IntTree` called `tree` stores the following elements:



The table below shows what the state of the tree would be if various calls were made:

Call	Result	Reason
`tree.hasPath(67, 99)`	`true`	path exists 67 → 52 → 99
`tree.hasPath(80, 45)`	`true`	path exists 80 → 21 → 45
`tree.hasPath(67, 67)`	`true`	node exists with `data` of 67
`tree.hasPath(16, 16)`	`true`	node exists with `data` of 16
`tree.hasPath(52, 99)`	`true`	path exists 52 → 99
`tree.hasPath(99, 67)`	`false`	nodes do exist, but in wrong order
`tree.hasPath(80, 99)`	`false`	nodes do exist, but there is no path from 80 to 99
`tree.hasPath(67, 100)`	`false`	end of 100 doesn't exist in the tree
`tree.hasPath(-1, 45)`	`false`	start of -1 doesn't exist in the tree
`tree.hasPath(42, 64)`	`false`	start/end of -1 and 45 both don't exist in the tree

An empty tree does not contain any paths, so if the tree is empty, your method should return `false`. You should not assume that your tree is a binary search tree (BST); its elements could be stored in any order.

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 (arrays, lists, etc.). You should not construct any new node objects or modify the tree in any way in your code.