Inheritance & Comparable

Consider the following class:

```
public class BankAccount {
    private String name;
    private double balance;

    public BankAccount(String name, double balance) {
        this.name = name;
        this.balance = balance;
    }

    public BankAccount(String name) {
        this(name, 0);
    }

    public double getBalance() {
        return this.balance;
    }

    public String toString() {
        return name + "\'s balance: " + getBalance();
    }
}
```

Write a new class called SavingsAccount that extends BankAccount and allows the Client to do the following:

- SavingsAccount now has a yearly interest rate specified in the constructor as an double
- SavingsAccount has a getInterestRate() that returns the interest rate
- SavingsAccount has a nextYearBalance() that does the following:
    - Calculates next year's interest with the simple interest formula - $SI = P * r$, where SI is Simple Interest, P is the starting Balance and r is the Yearly Interest Rate
    - Updates Balance by increasing the current amount by the interest amount
- If the balance SavingsAccount becomes 0 or less, the string representation should return "no money D:"
    - The rest of the string representation is the same as any other BankAccount
- SavingsAccount implements the Comparable interface; SavingsAccounts are first compared by the balance (bigger balances are "greater than" smaller ones), then by interest rate (bigger interest rates are "greater than" smaller rates).

To earn an E on this problem, your SavingsAccount class must not duplicate any code from the BankAccount class.

Write your solution on the next page.

LinkedList

Write a method called `removeDuplicates()` that exists within the following `LinkedIntList` class

```
public class LinkedIntList {
    private ListNode head;
    private int size;

    public LinkedIntList(int[] elements) {
        for (int i = elements.length - 1; i >= 0; i--) {
            this.head = new ListNode(elements[i], this.head);
        }
        this.size = elements.length;
    }

    public int size() {
        return this.size;
    }

    [...]

    private static class ListNode {
        public int data;
        public ListNode next;

        public ListNode(int data, ListNode next) {
            this.data = data;
            this.next = next;
        }

        public ListNode(int data) {
            this(data, null);
        }
    }
}
```

that removes any duplicate values within the current sorted LinkedIntList.

(continued on next page)

A more concrete example can be seen in the following JUnit tests:

```
public class Testing {
    @Test
    @DisplayName("General Example")
    public void general() {
        LinkedIntList l = new LinkedIntList(new int[]{1, 2, 3});
        l.removeDuplicates();

        assertEquals(3, l.size());
        assertEquals("[1, 2, 3]", l.toString());
    }

    @Test
    @DisplayName("Duplicate Example")
    public void duplicate() {
        LinkedIntList l = new LinkedIntList(new int[]{1, 2, 2, 3, 3});
        l.removeDuplicates();

        assertEquals(3, l.size());
        assertEquals("[1, 2, 3]", l.toString());
    }

    @Test
    @DisplayName("Complex Example")
    public void complex() {
        LinkedIntList l = new LinkedIntList(new int[]{1, 1, 2, 3, 3, 3, 4, 5, 5});
        l.removeDuplicates();

        assertEquals(5, l.size());
        assertEquals("[1, 2, 3, 4, 5]", l.toString());
    }
}
```

You may define private helper methods to solve this problem, but otherwise you may not call any other methods of the `LinkedIntList` class. You may not construct any new nodes, and you may not use any auxiliary data structure to solve this problem (no array, ArrayList, stack, queue, String, etc). You also may not change any data fields of the nodes.

Write your solution on the next page.

Binary Trees

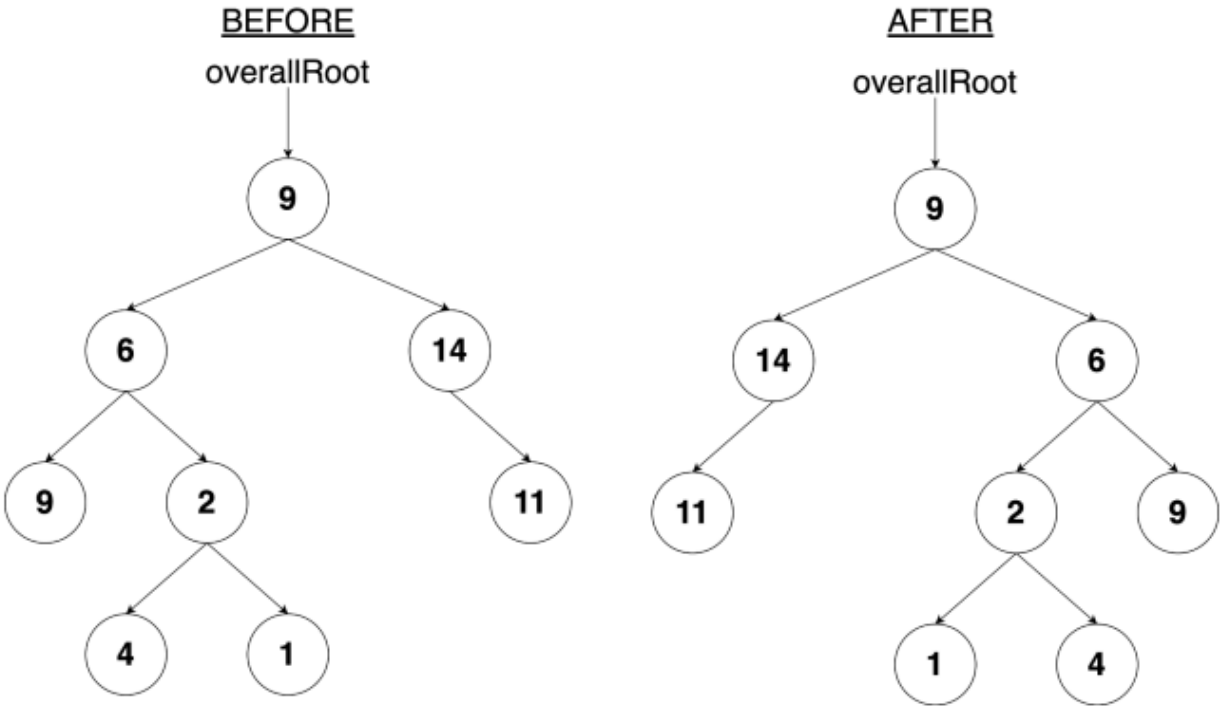
Write a method called `mirror()` that exists within the following `IntTree` class

```
private static class IntTree {
    private IntTreeNode overallRoot;

    [...]

    private static class IntTreeNode {
        public int data;
        public IntTreeNode left;
        public IntTreeNode right;

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

        public IntTreeNode(int data) {
            this(data, null, null);
        }
    }
}
```

that converts a binary tree of integers to its mirror image. For example, if a variable called `t` stores a reference to a binary tree then after calling `t.mirror()` the links of the tree should be rearranged such that `t` stores the mirror image of what it stored before the call.

(continued on next page)

Below is a specific example of how a tree would change:



After the call is made, the tree stores the structure that you would see if you were to hold the original tree's diagram up to a mirror. More precisely, the overall root (if it exists) remains in the same place in the mirror image and every other node is moved so that its new path from the overall root is composed of the old path from the overall root with left and right links exchanged. For example, the node that stores 4 in the example above has the path (left, right, left) in the original tree. In the mirror image, it has the path (right, left, right). The node that stores 1 has the path (left, right, right) in the original tree. In the mirror image it has the path (right, left, left).

You may define private helper methods to solve this problem, but otherwise you may not call any other methods of the `IntTree` class. You may not construct any new nodes, and you may not use any auxiliary data structure to solve this problem (no array, ArrayList, stack, 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 tree.

Write your solution on the next page.