Programming Assignment 10-1

Use the `MyBST` class in the `bst` package and the classes in the `employeebst` package to write a little application in which `Employee` instances are stored in a BST, ordered by name.

To begin, create a class `EmployeeBST` that behaves essentially the same way as the `MyBST` class, except instead of `Integers`, the data in each tree node is an `Employee` instance. `Employee` instances should be compared using the `NameComparator` class; pass this in as an argument to the constructor of the `EmployeeBST` class. The `compareTo` method in `MyBST` will now be replaced by calls to `compare` in the `NameComparator` instance.

The other classes have been created for you. The code you will need in the `EmployeeDriver` class is in the main method, but has been commented out, since it refers to the `EmployeeBST` class; once you have written the code for `EmployeeBST`, you can uncomment and run the code. The `EmployeeDriver` class populates your BST and then prints the tree; the values will appear in sorted order.

Programming Assignment 10-2

A *rational number* is a real number that can be represented as a fraction of the form p/q, where q > 0 and both p and q are integers. Although Java has data types for integers and floating point numbers, it does not have an explicit data type for rational numbers. Create a Java class `Rational` that meets this need. Design so that it meets the following requirements:

a. The constructor accepts two integers p, q; if q is not positive, an error is indicated with a statement printed to the console
b. The class has the following methods, with these signatures:

```
//adds the rational rat to this Rational
public Rational add(Rational rat);

//multiplies rat by this Rational
public Rational multiply(Rational rat);

//returns -1 if this rational is less than rat
//returns 0 if this rational equals (see equals
//          method discussion below) rat
//returns 1 if this rational is greater than rat
public int compareTo(Rational rat)

Mutators and accessors for numerator and denominator
```

c. It overrides the `equals()` method. Recall that two rationals p/q and r/s are equal if and only if qr = ps. (Therefore, you must override `hashCode()` too.)

d. It overrides the `toString()` method. The return value of the `toString()` method should represent the Rational in the usual format; for instance, the rational having numerator 2 and denominator 3 should look like this:

```
2/3
```

e. In the main method of your class, test your methods by performing the following computations:

```
(2/3 * -17/5) + 1/3,
2/3 * (-17/5 + 1/3),
```

and then stating which of the two values is larger. All computations and output values should involve *fractions only* – no floating point numbers allowed.

Expected output:

```
(2/3 * -17/5) + 1/3 is greater than 2/3 * (-17/5 + 1/3)
```

Hint:

a/b < c/d if and only if ad < bc

Programming Assignment 10-3

In this exercise, you will create a sorting program based on BSTs, in the way that was described in the Lesson 10 slides, and you will compare its performance to the `MinSort` program; the performance test will be carried out in a test harness that has been provided for you.

To write your code so that it can be used in the test harness, copy the project `Sorting` (you will find this in your lab folder) to your local drive and import it into your workspace as a new Java project. You will add your new classes to the `sortroutines` package. In that package, you should place `MyBST.java` (from assignment 10-1) and also a new class called `BSTSort.java`, which you will create.

First, you will need to create two methods in MyBST.java like following –
// It takes as input an array and builds a BST tree from it.
public void insertAll(int[] array){…}

//It traverses the BST and returns all its elements in a sorted array
public int[] readIntoArray() {…}

Your class `BSTSort` should inherit from the class `Sorter` (which is in the runtime package of the `Sorting` project). When you inherit from `Sorter`, you will be required to override the `abstract` method `int[] sort(int[] arr)`. Your code will accept any input array of `int`s, load them into an instance of MyBST (by calling the `insertAll` method you just created), and then, will use the `readIntoArray` method to obtain a return value, which will be the original array of `int`s, now in sorted order.

Remember that by autoboxing, Java will automatically convert between `int` and `Integer` types.

To compare your `BSTSort` program with `MinSort`, type the string "BSTSort" below "MinSort" in the text file `sorters_to_be_run.txt` (be careful not to change the location of this file in the project). Then run the class `SortTester`. `SortTester` will read the names of the classes specified in `sorters_to_be_run.txt`, and will (by reflection) create instances of each and run them through thousands of sorting tests. In the console window, you will see how well each sorter performed, from fastest to slowest.