Homework 4 :: Recursion

15-121 Fall 2015
Due: September 29, 2015 (11:50pm)
Basecode, Hand-In

Overview

This assignment consists of a set of exercises meant to give you practice writing recursive methods. Because of this, a significant penalty will be imposed if any of your solutions use loops (e.g. `for`, `while`, `for-each`, `do-while`) or call Java library methods that solve the problem for you. Even though the code for each solution may only be a few lines, writing recursive code (especially for the first time) takes a lot more thought per line. It gets easier with practice, but because this is new to you, you should give yourself time to mull over the problems for several days before asking for help.

The methods you need to implement require different data structures, or no data structures at all. In some of them you will use LinkedLists, ArrayLists, and Strings. Some of the methods will be identical in functionality to the ones you wrote in Homework #3 but implemented recursively. Finally, some of the methods will require "helper" methods. You may write as many helper functions as necessary unless specified otherwise. Make sure your solutions work correctly under all possible cases, especially edge cases, like null objects, lists of any size, empty strings, etc.

As with all assignments, you will be graded in part on your coding style. Your code should be easy to read, organized, and well-documented. Be consistent in your use of indentation and braces. See the style guide for more details.

Background :: Code

For this homework, you will be completing a variety of methods of the `Recursion` class. Read the `Node` class carefully -- it is different from the `Node` class you used in homework 3.

To create a new `Node`, do the following:

```
Node n = new Node("foo"); // creates a new Node
```

One you have a Node object created, you can set the `next` and `data` variables of the `Node`:

```
n.next = new Node("bar"); // add another Node to the list after n
n.data = "foo"; // change the data in the Node
```

There is no nice wrapper for our `Nodes` this time (as in Homework #3); you will be working directly with the `Nodes`. Make sure you fully understand the `Node` class before proceeding. You will be using it heavily throughout this exercise.

Exercise :: Identify

Download the base code and fill in the required fields like so:

```
/**
 * @author [First Name] [Last Name] <[Andrew ID]>
 * @section [Section Letter]
 */
```

```
/**
 * @author Jess Virdo <jvirdo>
 * @section A
 */
```

Exercise :: Examine

Inspect each of the methods that have been provided to you. You are not responsible for understanding each line of code, but you should understand the overall algorithm and concept. You can work on the methods in this homework in any order you like.

Exercise :: `count`

Complete the `count` method which searches through each `Node` in the `LinkedList` and returns the number of `Nodes` whose data exactly equals the `findData` parameter. If there are no matches, this method should `return 0`.

```
Node head = new Node("foo");
head.next = new Node("bar");
head.next.next = new Node("fish");
int ret = Recursion.count(head, "fish"); // ret = 1
```

```
Node head = new Node("foo");
head.next = new Node("bar");
head.next.next = new Node("fish");
int ret = Recursion.count(head, "here"); // ret = 0
```

Exercise :: isReverse

Complete the `isReverse` method which takes two String parameters and returns `true` if `string1` is the exact reverse of `string2`.

```
String string1 = "foo";
String string2 = "oof";
boolean ret = Recursion.isReverse(string1, string2); // ret = true
```

```
String string1 = "abc";
String string2 = "def";
boolean ret = Recursion.isReverse(string1, string2); // ret = false
```

```
String string1 = "";
String string2 = "";
Recursion.isReverse(string1, string2); // ret = true
```

Exercise :: insertAfter

Complete the `insertAfter` method which inserts a new `Node` into a `LinkedList` after the first occurrence of a `Node` in the `LinkedList` whose `data` matches the `findData` parameter. If there are multiple occurrences of `findData` in the original `LinkedList` only the closest to the front of the list should be used. If there are no occurrences of `findData`, the `LinkedList` should remain unchanged.

```
Node head = new Node("foo");
head.next = new Node("bar");
head.next.next = new Node("zip");
// [HEAD] => foo => bar => zip [TAIL]

Recursion.insertAfter(head, "cookie", "bar"); // insert "cookie" after "bar"
// [HEAD] => foo => bar => cookie => zip => [TAIL]

Node head = new Node("foo");
head.next = new Node("bar");
head.next.next = new Node("bar");
head.next.next.next = new Node("zip");
// [HEAD] => foo => bar => bar => zip [TAIL]

Recursion.insertAfter(head, "cookie", "bar"); // insert "cookie" after "bar"
// [HEAD] => foo => bar => cookie => bar => zip => [TAIL]

Node head = new Node("foo");
head.next = new Node("bar");
head.next.next = new Node("zip");
// [HEAD] => foo => bar => zip [TAIL]

Recursion.insertAfter(head, "cookie", "zoo"); // insert "cookie" after "zoo"
// [HEAD] => foo => bar => zip => [TAIL]
```

Exercise :: itAddsUp

Complete the `itAddsUp` method with returns `true` if all the `Integers` in the given `ArrayList<Integer>` sum to the specified target. It should return `false` otherwise.

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(2);
list.add(3);

boolean ret = Recursion.itAddsUp(list, 5); // ret = true
ret = Recursion.itAddsUp(list, 2); // ret = false

ArrayList<Integer> list = new ArrayList<Integer>();
list.add(-2);
list.add(3);

Recursion.itAddsUp(list, 1); // returns true
Recursion.itAddsUp(list, 2); // returns false

ArrayList<Integer> list = new ArrayList<Integer>();

Recursion.itAddsUp(list, 0); // returns true
Recursion.itAddsUp(list, 8); // returns false
```

Exercise :: removeDuplicates

Complete the removeDuplicates method which removes all duplicate consecutive characters from the given String parameter. All other characters should be left in their same relative positions. If there are no consecutive duplicate letters, the String should remain unchanged.

```
String ret = Recursion.removeDuplicates("pizza"); // ret =  "piza"
ret = Recursion.removeDuplicates("shell"); // ret =  "shel"
ret = Recursion.removeDuplicates("mississippi"); // ret =  "misisipi"
ret = Recursion.removeDuplicates("looooooooooooooool"); // ret =  "lol"
ret = Recursion.removeDuplicates("desk"); // ret =  "desk"
```

Exercise :: stringNumbers

Complete the stringNumbers method which returns a String comprised of all the numbers from 1 to n separated by dashes ("-"). The string must start with the values of the even integers in descending order first, and then follow with the odd integers in ascending order. This method does not print anything! It returns the generated String. You may assume that n will be a positive whole number, and you may <u>not</u> use helper methods for this problem.

```
String ret = Recursion.stringNumbers(4); // ret =  "4-2-1-3"
ret = Recursion.stringNumbers(1); // ret =  "1"
ret = Recursion.stringNumbers(7); // ret =  "6-4-2-1-3-5-7"
ret = Recursion.stringNumbers(0); // ret =  ""
```

Note: The quotation marks around the strings about method above indicate that the return type is a String. You should not add the literal quotation marks in your answer. This is just how we syntactically represent a String in Java on "paper".

Exercise :: removeAll

Complete the removeAll method which recursively removes all Nodes from a LinkedList whose data is exactly equal to the given length parameter. All other Nodes should remain in their same relative positions. If there are no Nodes with data of the given length, this method should leave the list entirely unchanged.

```
Node head = new Node("foo");
head.next = new Node("bar");
head.next.next = new Node("zipper");
// [HEAD] => foo => bar => zipper => [TAIL]
Recursion.removeAll(head, 3);
// [HEAD] => zipper => [TAIL]


Node head = new Node("foo");
head.next = new Node("bar");
head.next.next = new Node("zipper");
// [HEAD] => foo => bar => zipper => [TAIL]
Recursion.removeAll(head, 6);
// [HEAD] => foo => bar => [TAIL]


Node head = new Node("foo");
```

```
        head.next = new Node("bar");
        head.next.next = new Node("zipper");
        // [HEAD] => foo => bar => zipper => [TAIL]
        Recursion.removeAll(head, 1);
        // [HEAD] => foo => bar => zipper => [TAIL]
```

Exercise::Reflect

In Homework 3, you worked with instance methods of your custom `LinkedList` class. You created a new `LinkedList` object and called methods on that object like this:

```
        LinkedList list = new LinkedList();
        list.add("foo");
        list.add("bar");
```

However, in this homework, we never create a `Recursion` object. Instead we call methods directly on the `Recursion` class. These are called class methods. In Java, they are also called static methods.

```
        Recursion.removeDuplicates("fooooooots");
```

Comparing the code between these two homeworks, you will see that there's a subtle difference in the method signatures:

```
        // in LinkedList.java
        public void add(String newData) {
          ...
        }

        // in Recursion.java
        public static int count(Node head, String findData) {
          ...
        }
```

Note the additional `static` keyword in `Recursion`. Class methods, i.e. methods meant to be called without an object (an instance of the class), are static. You can expect an exam question on this topic, so please see a TA if you do not understand the difference between class methods/static methods and instance methods/non-static methods.

Exercise :: mirrorList

Complete the `mirrorList` method which given an `ArrayList<Integer>` of length n returns a new `ArrayList<Integer>` of length 2n.such that the last n elements are copies of the first n elements in the original ArrayList, but the order is reversed.

```
    ArrayList<Integer> al = new ArrayList<Integer>();
    for (int i = 1; i < 5; i++) al.add(i);
    // al = [1, 2, 3, 4]
    ArrayList<Integer> almirror = new ArrayList<Integer>();
    almirror = Recursion.mirrorList(al);
    // almirror = [1, 2, 3, 4, 4, 3, 2, 1]


    ArrayList<Integer> al = new ArrayList<Integer>();
    al.add(10);
    // al = [10]
    ArrayList<Integer> almirror = new ArrayList<Integer>();
    almirror = Recursion.mirrorList(al);
    // almirror = [10, 10]
```

Exercise :: sumDigits

Complete the `sumDigits` method which returns the sum of the digits in the given integer parameter. This method should throw an `IllegalArgumentException` if the integer passed in is negative.

```
int result = sumDigits(15121); // result = 10

result = sumDigits(123); // result = 6

result = sumDigits(8); // result = 8
```

Exercise :: countBinary

Complete the `countBinary` method that returns the number of binary strings of length `n` (passed in as a parameter to the function) which do not have two consecutive zeros.

```
/*
 * There are 8 binary strings of length 4 that do not have two consecutive
 * zeros:
 * 1111, 1110, 1101, 1011, 1010, 0111, 0110, 0101
 */


int result = countBinary(4); // result = 8



/*
 * There are 5 binary strings of length 3 that do not have two consecutive
 * zeros:
 * 111, 110, 101, 011, 010
 */


int result = countBinary(3); //result = 5
```

Submitting your Work

When you have completed the assignment and tested your code thoroughly, create a `.zip` file on your work. Only include the following file(s):

1. `Recursion.java`

Do not include any `.jar` or `.class` files when you submit, and zip only the files listed above. Do not zip not an entire folder containing the file(s).

Once you’ve zipped your files, visit the Autolab site -- there is a link on the top of this page -- and upload your zip file.

Keep a local copy for your records.