DUE DATE: DECEMBER 4, 2019 AT 11:59PM

Instructions:

- You must complete the "**Blanket Honesty Declaration**" checklist on the course website before you can submit any assignment.
- Only submit the **java files**. Do ***not*** submit any other files, unless otherwise instructed.
- To submit the assignment, upload the specified files to the **Assignment 4** folder on the course website.
- Assignments must follow the **programming standards** document published on UMLearn.
- After the due date and time, assignments may be submitted but will be subject to a late penalty. Please see the ROASS document published on UMLearn for the course policy for late submissions.
- If you make multiple submissions, only the **most recent version** will be marked.
- These assignments are your chance to learn the material for the exams. Code your assignments independently. We use software to compare all submitted assignments to each other, and **pursue academic dishonesty vigorously**.
- Your Java programs must compile and run upon download, **without requiring any modifications**.
- **Loops of any kind (for, while, do-while) are not allowed in this assignment, except in the createLinkedList method.** Every method must be implemented using a recursive approach instead of an iterative approach. **You will lose marks for using loops**.

Assignment overview

In this assignment, you will receive two template files, in which you will implement a set of methods. The two template files, that you will modify are:

- **`Node`**: A node of a LinkedList, containing some data (stored as an Object) and a reference to the next link (a Node).
- **`LinkedList`**: A class representing a LinkedList of Nodes. It has a reference to the first Node of the list (top).

Keep all of your methods short and simple. In a properly-written object-oriented program, the work is distributed among many small methods, which call each other. Most methods in this entire assignment require no more than 10-12 lines of code, not counting comments, blank lines, or {} lines. The swap method (Phase 5) is the only exception (this one might require about 30 lines of code).

Unless specified otherwise, all instance variables must be **`private`**, and all interface methods should be **`public`**. Helper methods (not supposed to be used by outside code) should normally be **`private`**. Since this assignment will be using a lot of recursion, and since recursive methods often need extra parameters, we will often implement a **`public`** interface method, which is meant to be used by outside code, and its only role will be to call the **`private`** recursive version of the method with appropriate parameters.

Testing Your Coding

We have provided sample test files for each phase to help you see if your code is working according to the assignment specifications. These files are *starting points* for testing your code. Part of developing your skills as a programmer is to think through additional important test cases and to write your own code to test these cases. ***For marking, we will use longer and more comprehensive tests***.

Phase 1: add and toString

Implement these methods in the LinkedList class:

The **`add(Object data)`** method:

- This method is the **`public`** interface to add a new piece of data at the end of the list.
- This method will call the **`addRec`** method defined below, with appropriate parameters.

The **`addRec(Node start, Node newNode)`** method:

- This method is the **`private`** recursive method used internally only.
- It will use recursive calls (to itself) to move to the end of the list, and add the **`newNode`** at the end of the list.

The **`String toString()`** method:

- This method is the **`public`** interface to return a String representation of the list.
- This method will call the **`toStringRec`** method defined below, with appropriate parameters.

The **`String toStringRec([parameters of your choice])`** method:

- This method is the **`private`** recursive method used internally only.
- It will use recursive calls (to itself) to move through the list, and build a String representation of the list.
- The resulting String will have each data from each Node on a separate line (see example below for an example).

You can test your code using the supplied test program **`TestingPhase1.java`**. You should get the output shown below.

```
Hello
World
1
2
3
```

Phase 2: Factory method to build a LinkedList from a file

Implement this method in the LinkedList class:

The **`LinkedList createLinkedList(String filename)`** method:

- This is a **`public`** factory method that builds and returns a new LinkedList.
- This is the only method of this assignment in which loops are allowed.
- This method must open the file corresponding to the filename parameter, read it and build the corresponding LinkedList.
- You can assume that there is only one token (representing the data to be stored in a Node of the LinkedList) on each line in the file.
- Tokens that are convertible to Integers, must be stored in the list as Integers.
- Tokens that are not convertible to Integers, must be stored in the list as Strings.
- This method must handle the exceptions related to reading a file internally (do not add a throws statement at the end of the signature of this method).

You can test your code with **`TestingPhase2.java`**. The output should be equivalent to the text in the three supplied files: list1.txt, list2.txt and list3.txt.

Phase 3: Sum

Implement these methods in the LinkedList class:

The **`int getSum()`** method:

- This method is the **`public`** interface to return an int representing the sum of all data in the list (see description of the next method for details).
- This method will call the **`getSumRec`** method defined below, with appropriate parameters.

The **`int getSumRec([parameters of your choice])`** method:

- This method is the **`private`** recursive method used internally only.

- It will use recursive calls (to itself) to move through the list, and calculate the sum of the data of all Nodes of the list.
- For each Node, if the data is an Integer, it considers the value of the int and adds it to the sum.
- If the data is a String, it counts the number of characters in the String and adds it to the sum.
- If the data is any other kind of Object (not an Integer, not a String), it simply adds 0 to the current sum.

You can test your code with **`TestingPhase3.java`**. The output of this code tells you what values are expected.

Phase 4: compareTo and getSmallestNode

First, implement this method in the **Node** class:

The **`int compareTo(Node other)`** method:

- This method is a traditional compareTo method, which will return an int that is either < 0, == 0 or > 0.
- The data instance variable in each Node is going to be used to determine which Node is bigger than the other.
- Since many different types of data can be contained in a Node, we will follow the following rules:
  - Integer data is considered to be smaller than String data
  - If the two Nodes being compared contain Integer data, the difference between them will be returned.
  - If the two Nodes being compared contain String data, the alphabetical difference between them will be returned (you can simply use compareTo of Strings).
  - Any data that is not an Integer and not a String is considered to be bigger than Integers and Strings. No difference needs to be calculated when dealing with those and no specific int value needs to be returned as long as it is either a positive or negative number (you could use +1 or -1 for example). See TestingPhase4.java for examples.

Second, implement this method in the **LinkedList** class:

The **`Node getSmallestNode(Node start, Node smallest)`** method:

- This method is a **`public`** method that will also be recursive (no separate interface for this one).
- It will use recursive calls (to itself) to move through the list, and find the smallest Node in the list, which will be returned. The **`start`** Node represents where to start the search, and the **`smallest`** Node represents the smallest Node found so far.
- Using the compareTo method created previously in the Node class, this should be straightforward.

You can test your code with **`TestingPhase4.java`**. The output of this code tells you what values are expected.

Phase 5: getPreviousNode and swap

Implement these methods in the **LinkedList** class:

The **`Node getPreviousNode(Node toFind)`** method:

- This method is the **`public`** interface to return a Node representing the node that comes right before the Node **`toFind`** in the list (see description of the next method for details).
- This method will call the **`getPreviousNodeRec`** method defined below, with appropriate parameters.

The **`Node getPreviousNodeRec([parameters of your choice])`** method:

- This method is the **`private`** recursive method used internally only.
- It will use recursive calls (to itself) to move through the list, and find and return the Node that immediately precedes the Node to find.
- If the Node to find is the very first Node of the list, return null.
- You can assume that the Node to find is always present, that no duplicates are in the list, and that we will never call this method on an empty list.

The **`swap(Node n1, Node n2)`** method:

- This method is a **`public`** method that swaps two given Nodes in the list. Nothing needs to be returned.
- You have to swap the entire Nodes, not just the data (you are not allowed to add a mutator in the Node class for the data).
- You can assume that Nodes **`n1`** and **`n2`** are going to be different.
- Using the **`getPreviousNode`** method might be useful.
- This is the most complex method of the assignment (it could take up to about 30 lines of code). Make sure to think about all the links that must be modified in order to have a successful swap that will preserve the integrity of the list. Make sure to also think about all the special cases that could arise.

You can test your code with **`TestingPhase5.java`**. The output of this code should allow you to see if everything worked correctly.

Phase 6: Bonus phase (+20% bonus)

If you are completing this bonus phase, you must mention it when submitting your assignment in the dropbox using the comments field. **This bonus phase will not be evaluated if you do not specify, in the UMLearn dropbox, that you completed it.**

Implement these methods in the **LinkedList** class (if recursion is not used as described below, you will not get the bonus points):

The **`selectionSort()`** method:

- This method is the **`public`** interface to sort the list using a selection sort algorithm, as seen in class.
- This method will call the **`selectionSortRec()`** method defined below, with appropriate parameters.

The **`selectionSortRec([parameters of your choice])`** method:

- This method is the **`private`** recursive method used internally only.
- It will use recursive calls (to itself) to move through the list, and sort it in place using a selection sort algorithm.
- The same rules defined in Phase 4 are to be used to order the Nodes.

You can test your code with **`TestingPhase6.java`**. The output of this code should allow you to see if everything worked correctly.

Hand in

Submit your four Java files (**`LinkedList.java`** and **`Node.java`**). ***Do not submit .class or .java~ files!*** You do ***not*** need to submit the **`TestingPhaseN.java`** files that were given to you. If you did not complete all phases of the assignment, **use the Comments field when you hand in the assignment to tell the marker which phases were completed**, so that only the appropriate tests can be run. For example, if you say that you completed Phases 1-2, then the marker will compile your files, and compile and run the tests for phases 1 and 2. If it fails to compile and run, you will lose ***all*** of the marks for the test runs. The marker will ***not*** try to run anything else, and will ***not*** edit your files in any way. ***Make sure none of your files specify a package at the top!***