CS 0445 Spring 2022 Assignment 2

Online: Monday, February 7, 2022
Due: All files (see below for list) zipped into a single .zip file and submitted properly to the submission site by **11:59PM on Tuesday, February 22, 2022** (Note: See submit.html for submission instructions / requirements)
Late Due Date: Saturday, February 26, 2021 by 11:59PM

Purpose: To give you experience implementing and using linked lists
Goal: To implement a subclass of a doubly linked-list that adds some functionality to it, and to implement a subclass of that class that can store and manipulate arbitrary length integers.

Details: We can think of a decimal integer as a sequence of digits. For example, the number 1234 could be stored as the digit '1' followed by the digit '2' followed by the digit '3' followed by the digit '4'. We can store these digits in an array, or, as is required in this assignment, in a linked list. Clearly, to perform operations on a number that is stored in this fashion, we must access the digits one at a time in some systematic way. More specific details follow below.

Part 1: The first part of this assignment is to write a new subclass of **A2LList<T>** called **LinkedListPlus<T>**. Class A2LList<T> is a class that I have written for you which is a doubly-linked version of the author's LList<T> class. Note that in addition to making the list doubly-linked, I have also removed the ListInterface and a few methods and I have changed the data from private to protected – this will allow you to access the instance variables directly from the subclass. **See the code in A2LList.java to see how the data is stored and how the methods are implemented.** LinkedListPlus will **not have any new instance variables** but will extend A2LList by adding the following methods:

`public void leftShift(int num)`
Shift the contents of the list num places to the left (assume the beginning is the leftmost node), removing the leftmost num nodes. For example, if a list has 8 nodes in it (numbered from 1 to 8), a leftShift of 3 would shift out nodes 1, 2 and 3 and the old node 4 would now be node 1. If num <= 0 leftShift should do nothing and if num >= the length of the list, the result should be an empty list.

`public void rightShift(int num)`
Same idea as leftShift above, but in the opposite direction. For example, if a list has 8 nodes in it (numbered from 1 to 8) a rightShift of 3 would shift out nodes 8, 7 and 6 and the old node 5 would now be the last node in the list. If num <= 0 rightShift should do nothing and if num >= the length of the list, the result should be an empty list.

`public void leftRotate(int num)`
In this method you will still shift the contents of the list num places to the left, but rather than removing nodes from the list you will simply change their ordering in a cyclic way. For example, if a list has 8 nodes in it (numbered from 1 to 8), a leftRotate of 3 would shift nodes 1, 2 and 3 to the end of the list, so that the old node 4 would now be node 1, and the old nodes 1, 2 and 3 would now be nodes 6, 7 and 8 (in that order). The rotation should work modulo the length of the list, so, for example, if the list is length 8 then a leftRotate of 10 should be equivalent to a leftRotate of 2. If num < 0, the rotation should still be done but it will in fact be a right rotation rather than a left rotation.

`public void rightRotate(int num)`
Same idea as leftRotate above, but in the opposite direction. For example, if a list has 8 nodes in it (numbered from 1 to 8), a rightRotate of 3 would shift nodes 8, 7 and 6 to the beginning of the list, so that the old node 8 would now be node 3, the old node 7 would now be node 2 and the old node 6 would now be node 1. The behavior for num > the length of the list and for num < 0 should be analogous to that described above for leftRotate.

Note: The fact that the A2LList<T> class is a circular, doubly-linked list makes the rotate methods above relatively simple. Think about why this is so.

Note that in the methods above **you may not create any new Node objects**. The purpose of them is to **rearrange the Nodes that already exist**. To see how these should work, I strongly recommend drawing one or more pictures.

You will also need to write the following constructors:

```
public LinkedListPlus()
public LinkedListPlus(LinkedListPlus<T> oldList)
```

The first contructor simply initializes the list to an empty state, and the second generates a new list that is a copy of the argument list (copying **all of the nodes inside the old list – thus this copy is "deepish"**).

Finally, you will need to override the following method:

```
public String toString();
```

This method will return a String that is the result of all of the data in the list being appended together, separated by spaces.

Important Notes:

1) **All of your LinkedListPlus methods must be implemented in an efficient way, utilizing the underlying linked list. For example, a poor implementation of a left rotation could be done via repeated calls to `remove(1)` and `add(this.getLength(),X)`. However, this would remove / add many Nodes to the list, adding overhead to the method call. This and similar implementations are not allowed and if implemented in this way you will not receive credit.**
2) **The leftRotate() and rightRotate() methods should not actually change the position of any Nodes in the list. Since the list is circular you can accomplish these simply by re-assigning the instance variable references within your LinkedListPlus object. Think about how this would be done.**

To verify that your LinkedListPlus class works properly, you will use it with the program **LLPTest.java**, which will be provided for you on the Assignments Web page. Your output must match that shown in **LLPTest.txt**.

To help you out with the assignment, I have implemented some of the methods above for you, with comments. See the code in LinkedListPlus.java.

Part 2: The second part of this assignment is to write the **ReallyLongInt** class with the specifications as given below. **You may assume all numbers will be non-negative**.

Inheritance: ReallyLongInt must be a subclass of LinkedListPlus. However, since LinkedListPlus is generic while ReallyLongInt is not generic, you should use the following header:

```
public class ReallyLongInt extends LinkedListPlus<Integer>
                           implements Comparable<ReallyLongInt>
```

Note that rather than T the underlying element data is now Integer. This means that the individual digits of your ReallyLongInt will be Integer objects.

Data: The data for this class is inherited and **you may not add any additional instance variables**. You will certainly need method variables for the various operations but the only instance variables that you need are those inherited from A2LList (via LinkedListPlus).

Operations: Your ReallyLongInt class must implement the methods shown below. Note that the compareTo() method is necessary for the Comparable interface.

Important Note: All of your operations must be implemented on the underlying linked-list of digits. You may NOT cast your ReallyLongInt into a BigInteger or any other type and

you may not convert the linked-list of digits into an array of digits. If you do these things you will not get credit for the methods.

`private ReallyLongInt()`

The default constructor will create an "empty" ReallyLongInt. Note that this leaves the number in an inconsistent state (having no actual value), so it should only be used within the class itself as a utility method (for example, you will probably need it in your add() and subtract() methods). For this reason it is a private method.

`public ReallyLongInt(String s)`

The string s consists of a valid sequence of digits with no leading zeros (except for the number 0 itself – special case). Insert the digits as Integer objects into your list, such that the least significant digit is at the beginning of the list. For example, the String "456202" would be stored in a ReallyLongInt as:

firstNode → 2 → 0 → 2 → 6 → 5 → 4 (note: actual Nodes are not shown but implicit)

Note that because we are using a circular, doubly-linked list for the numbers, it is just as easy to store the number with the most significant digit first. You may certainly implement it in that way if you wish.

`public ReallyLongInt(ReallyLongInt rightOp)`

This just requires a call to super. However, it is dependent upon a correct implementation of the copy constructor for the LinkedListPlus class.

To help you out with the assignment, I have implemented the methods above for you, with comments. See the code in ReallyLongInt.java.

`public ReallyLongInt(long x)`

The argument x is a valid, non-negative long value. Parse this into digits and create a new ReallyLongInt from the digits. For this method you may NOT convert x into a String and use the String constructor above – rather you must parse the individual digits from the argument and create your ReallyLongInt in this method digit by digit. Think how you can do this.

`public String toString()`

Return a string that accurately shows the integer as we would expect to see it. Based on the way we have stored the integer, this can be accomplished by going backward through the list. Note: The TA may test your output by comparing it character by character to my output. In order for this to work, your toString() method must show the number correctly, as a sequence of digits with no spaces. Make sure your toString() method does this.

`public ReallyLongInt add(ReallyLongInt rightOp)`

Return a **NEW** ReallyLongInt that is the sum of the current ReallyLongInt and the parameter ReallyLongInt, without altering the original values. For example:

```
ReallyLongInt X = new ReallyLongInt("123456789");
ReallyLongInt Y = new ReallyLongInt("987654321");
ReallyLongInt Z;
Z = X.add(Y);
System.out.println(X + " + " + Y + " = " + Z);
```

should produce the output:

```
123456789 + 987654321 = 1111111110
```

Be careful to handle carries correctly and to process the nodes in the correct order. Since the numbers are stored with the least significant digit at the beginning, the add() method can be implemented by traversing both numbers in a systematic way. This must be done efficiently using references to traverse the lists. In other words, you should start at the beginning of each ReallyLongInt and **traverse one time** while doing the addition. **You may NOT use**

getNodeAt() or getEntry() for this method. Think how you can do this with reference variables. Also be careful to handle numbers with differing numbers of digits.

`public ReallyLongInt subtract(ReallyLongInt rightOp)`

Return a NEW ReallyLongInt that is the difference of the current ReallyLongInt and the parameter ReallyLongInt. Since ReallyLongInt is specified to be non-negative, if rightOp is greater than the current ReallyLongInt, you should throw an ArithmeticException. Otherwise, subtract digit by digit (borrowing if necessary) as expected. As with the add() method, you must implement this efficiently via a **single traversal of both lists**. **In other words, you may NOT use getNodeAt() or getEntry() for this method.** This method is tricky because it can result in leading zeros, which we don't want. Be careful to handle this case (and consider the tools provided by LinkedListPlus that will allow you to handle it). For example:

```
ReallyLongInt X = new ReallyLongInt("123456");
ReallyLongInt Y = new ReallyLongInt("123455");
ReallyLongInt Z;
Z = X.subtract(Y);
System.out.println(X + " - " + Y + " = " + Z);
```

should produce the output:

```
123456 - 123455 = 1
```

As with the add() method, be careful to handle numbers with differing numbers of digits. Also note that borrowing may extend over several digits. See RLITest.java for some example cases.

`public ReallyLongInt multiply(ReallyLongInt rightOp)`

Return a NEW ReallyLongInt that is the product of the current ReallyLongInt and the parameter ReallyLongInt. Multiplication of ReallyLongInt objects can be done using nested loops – think of how you learned to multiply in gradeschool. For example, consider the following "long" multiplication:

```
  1234
    56
------
  7404
 6170
------
 69104
```

Note that each digit in the "bottom" number is multiplied by each digit in the "top" number and the sub-products are added together to get the final answer. Note also that each subsequent sub-product is shifted by one location to the left to account for the higher power of 10 in the multiplying digit. Think how you can implement this algorithm using your ReallyLongInt objects. There are some very important requirements for this algorithm – **minimal credit will be earned if they are not followed**:

1) You must access both ReallyLongInt objects in a sequential fashion – from one end to the other. You **may NOT** use any indexing operations such as getNodeAt() or getEntry() as these will lead to a very inefficient solution.
2) An arbitrary number of sub-products can be generated in this algorithm, one for each digit in the "bottom" number. However, **you should NOT use an array or any collection** to store these sub-products – this will be very wasteful in terms of memory usage. Rather, you should keep a running "sum" of the sub-products and just add the next sub-product to it at each step. Note that you can use your add() method to add the sub-products.

`public int compareTo(ReallyLongInt rightOp)`

Defined the way we expect compareTo to be defined for numbers. If one number has more digits than the other then clearly it is bigger (since there are no leading 0s). Otherwise, the numbers must be compared digit by digit. Since this requires the most significant digit to be processed first (which is at the end of the list), we should go backward through the list (starting at the end) in order to implement this method.

`public boolean equals(Object rightOp)`

Defined the way we expect equals to be defined for objects – comparing the data and not the reference. Don't forget to cast rightOp to ReallyLongInt so that its nodes can be accessed (note: the argument here is Object rather than ReallyLongInt because we are overriding equals() from

the version defined in class Object). Note: This method can easily be implemented once compareTo() has been completed.

To verify that your ReallyLongInt class works correctly, you will use it with the program **RLITest.java**, which will be provided for you on the Assignments page. Your output should match that shown in RLITest.txt.

More Important Notes:

1) Using a doubly-linked list makes many of these operations simpler than if a singly-linked list were used. However, it also adds additional special cases. Make sure you handle all of the special cases in your code.
2) The add(), subtract() and multiply() methods are tricky and have different cases to consider. For these methods especially I recommend working out some examples on paper to see what needs to be considered before actually coding them.

Extra Credit: Here are a couple non-trivial extra credit ideas. Either one done well could get you the full 10 extra credit points. However, don't attempt either until you are confident that your required classes are working correctly.

- Allow the numbers to be signed, so that we can have both positive and negative numbers. This may require an extra instance variable (for the "sign") and will clearly affect many of your methods. If you choose this extra credit, you must submit it as a separate class (ReallyLongInt2) in addition to your original ReallyLongInt class. You must also submit a separate driver program to test / demonstrate your signed ReallyLongInt2 class.
- Add a non-trival operation to your ReallyLongInt class. The full 10 points will be given only if the operation is reasonably complex.

Submission:

You must submit the following 5 complete, working source files for full credit:

A2LList.java
LLPTest.java
RLITest.java

The **above three files** are given to you and must not be altered in any way.

LinkedListPlus.java
ReallyLongInt.java

The **above two files** must written by you so that they work as described. **Note that I have provided partial implementations of both – download those as a starting point.**

The idea from your submission is that your TA can compile and run your program WITHOUT ANY additional files OR PROCESSING. This means compilation and execution on the command line using the javac and java commands. Test your programs from the command line before submitting – especially if you use an IDE such as NetBeans or Eclipse to develop your project. If you cannot get the programs working as given, clearly indicate any changes you made and clearly indicate why on your Assignment Information Sheet.