CS 445 – Data Structures – Assignment#2[1]

Due: Tuesday Oct. 17th@ 11:59pm

All source files plus a completed Assignment Information Sheet zipped into a single .zip file and submitted properly to CourseWeb.

Late submission: Thursday Oct. 19th @11:59pm with 10% penalty per late day

OVERVIEW

Purpose: To give you experience implementing and using linked data structures.

Goal 1: To design and implement a class LinkedDS<T> that will act as a linked data structure for accessing Java Objects. Your LinkedDS<T> class will primarily implement two interfaces – PrimQ<T> and Reorder. The details of these interfaces are explained in the files PrimQ.java and Reorder.java. **Read these files over very carefully before implementing your LinkedDS<T> class.**

Goal 2: To utilize your LinkedDS<T> class to store and manipulate arbitrary length integers. 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.

DETAILS

Details 1: For the details on the functionality of your LinkedDS<T> class, carefully read over the files PrimQ.java, Reorder.java and Assig2A.java provided on the CourseWeb folder where you find this assignment description. You must use these files as specified and **cannot remove/alter any of the code that is already written in them**. There are different ways of implementing the PrimQ<T> and Reorder interface methods, some of which are more efficient than others. Use only the best way of implementing these methods in this assignment. A lot of pencil and paper work is recommended before actually starting to write your code. Your LinkedDS<T> class header should be:

public class LinkedDS<T> **implements** PrimQ<T>, Reorder

The only instance variables allowed inside the LinkedDS<T> class are

```
private Node firstNode;
```

[1] Assignment adapted from Dr. John Ramirez's class.

private int numOfEntries;

Important Note: The primary data within your LinkedDS<T> class must be a linked list. You may not use any predefined Java collection class (e.g., ArrayList) for your LinkedDS<T> data.

Note that the following methods are added to the Reorder interface in this assignment as compared to Assignment 1.

`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.

`public void reverse()`

This method should reverse the nodes in the list.

Note that in the methods above **you may not create any new Node objects**. The purpose of these methods 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 LinkedDS()

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

The first constructor 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**).

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.

After you have finished your coding of LinkedDS<T>, the **Assig2A.java** file provided for you should compile and run correctly, and should give output identical to the output shown in the sample executions.

Important Notes:

1) **All of your LinkedDS 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 `X = removeItem()` and `addItem(X)`. However, doing this would require multiple traversals of the list and has a very bad run-time (think arithmetic series). This and similar implementations are not allowed and if implemented in this way you will not receive credit.**
2) **The leftRotate(), rightRotate() amd reverse() methods should NOT create any new Node objects. Rather, they should move the Node objects currently in the list into other locations. You may use temporary Node variables for these methods, but you may not create any new Nodes.**

Details 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 LinkedDS. However, since LinkedDS is generic while ReallyLongInt is not generic, you should use the following header:

```
public class ReallyLongInt extends LinkedDS<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 LinkedDS.

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

`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)

`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 LinkedDS class.

`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 **should** be accomplished by going backward through the list. Since we cannot actually do that, we will reverse() the list, then traverse it, then reverse() it again.

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 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. The KEY is that for add() (and subtract()) you should access each Node in the list only 1 time TOTAL. This is true for other methods as well. 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. 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 LinkedDS 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 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 cannot just iterate through the digits as given. We will use what is given in the LinkedDS class – first reverse() the Nodes in both numbers, then do the comparison in a sequential way, then reverse() again to restore the original numbers.

`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.

`public void multTenToThe(int num)`

Multiply the current ReallyLongInt by 10^{num} . Note that this can be done very simply through adding of nodes containing 0's.

`public void divTenToThe(int num)`

Divide the current ReallyLongInt by 10^{num} . Note that this can be done very simply through shifting.

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) When implementing these operations you may discover that it would be much easier to do them if the underlying linked list were bidirectional (i.e. a doubly-linked list). This is absolutely true but unfortunately you are required to use the singly-linked list that is provided!
2) Both the add() and subtract() 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 multiply() method to your ReallyLongInt class. If you implement this you should also submit a separate driver program to test / demonstrate the multiply() method.

SUBMISSION REQUIREMENTS

You must submit, in a single .zip file, at least the following six complete, working source files for full credit:

1. PrimQ.java
2. Reorder.java
3. Assig2A.java
4. RLITest.java

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

5. LinkedDS.java
6. ReallyLongInt.java

The **above two files** must be created so that they work as described. If you create any additional files, be sure to include those as well.

The idea from your submission is that your TA can unzip your .zip file, then compile and run both of the main programs (Assig2A.java and RLITest.java) **from the command line** WITHOUT ANY additional files or changes, so be sure to test it thoroughly before submitting it.

If you cannot get the programs working as given, clearly indicate any changes you made and clearly indicate why (ex: "I could not get the reverse() method to work, so I eliminated code that used it") on your Assignment Information Sheet. You will lose some credit for not getting it to work properly, but getting the main programs to work with modifications is better than not getting them to work at all. A template for the Assignment Information Sheet can be found in the assignment's CourseWeb folder. You do not have to use this template but your sheet should contain the same information.

Note: If you use an IDE such as NetBeans, Eclipse, or IntelliJ, to develop your programs, make sure they will compile and run on the command line before submitting – this may require some modifications to your program (such as removing some package information).

RUBRICS

Please check the grading rubric on CourseWeb.

HINTS / NOTES

1. See file A2Out.txt to see how your output for Assig2A.java should look. As noted, your output when running Assig2A.java should be identical to this.
2. See file RLITest.txt to see how your output for RLITest.java should look. As noted, your output when running RLITest.java should be identical to this.

3. For Javadoc comments and code style, please refer to Appendix A of the textbook.