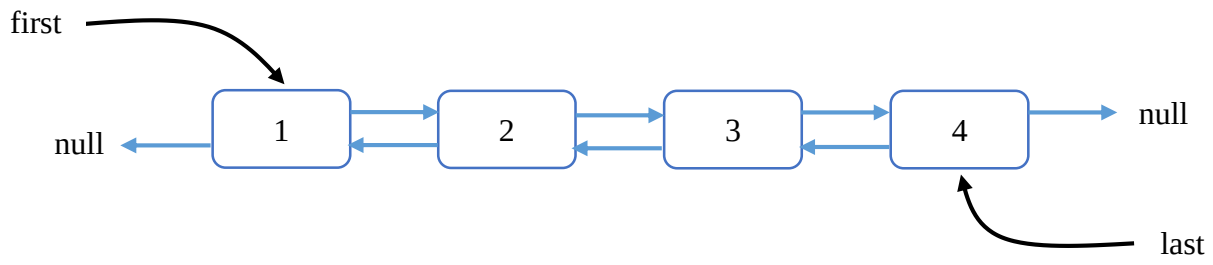
DUE DATE: DECEMBER 9, 2020 AT 11:59PM

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. You will lose marks for not following these standards.
- 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**.
- Automated tests will be used to grade the output of your assignment. If you do not follow precisely the guidelines detailed below, these automated tests can fail and you will lose marks. In particular, make sure that **your methods are spelled exactly as described below**. Also, make sure that **your code produces exactly the example outputs** (including correct spacing) shown in these guidelines.
- **Loops of any kind (for, while, do-while) are not allowed in this assignment.** Every method must be implemented using a recursive approach instead of an iterative approach. **You will lose marks for using loops.**

Assignment overview (please read this carefully before you start)

In the recorded lectures, we saw how to implement regular linked lists. These are also called "singly" linked lists, because each node only has one link to the next node. There exist other types of linked lists. One of them is the "doubly" linked list, in which nodes contain two pointers instead of just one: a link to the previous node, and a link to the next node. The class that represents the doubly linked list usually stores two pointers to both ends of the list: references to the first and last nodes. Please see the figure below for an example of a doubly linked list:



In this assignment, you will have to implement methods in a doubly linked list. The main advantage of a doubly linked list is that you are allowed to go in both directions (backward and forward) from any node. Just like we have seen in the recorded lectures, the most important thing in any linked list is to think about all the special cases (empty list, list with only one node, etc.) and to update the links correctly after each operation. The only difference here is that you have twice the number of links to update compared to a singly linked list. If you forget to update some links, your list will be broken. You will be able to observe this when you print the list in both directions: if the list doesn't contain the same nodes in both directions, it means that the list was not modified properly.

To make things more interesting, we are not allowing you to use loops in this entire assignment (no exception). Everything must be done with **recursion** instead. By the end of this assignment, you will become masters at recursion!

Two template files (**DoublyLinkedList.java** and **Node.java**) are provided to you. They contain the instance variables that you need, accessor/mutator methods for them and one constructor in the Node class. You are not allowed to add any additional instance or class variables in the provided files, nor to modify the lines of code that are already provided. You **must only add the requested methods**, as described below (there shouldn't be a need for any additional methods).

Note that the data that we store in the Node class is of type **Number**. The Number class is provided by Java (you can see the documentation here: https://docs.oracle.com/javase/8/docs/api/java/lang/Number.html). It is an abstract class, which serves as a superclass to all the objects representing numbers. In this assignment, we will only store these possible subclasses as possible data types in our Nodes: **Integer**, **Long** and **Double**.

Unless specified otherwise, 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 role will be to call the **private** recursive version of the method with appropriate parameters. In most cases, you will have to find which parameters to use to make the recursion work in those private recursive versions of the methods (since they are private anyway, we will not test them directly in the automated tests, so you can define any parameter(s) that you need). Note that the public interface methods can also do some checks or deal with some easy cases, if you find it useful or appropriate.

Remember as usual to keep all of your methods short and simple. Make sure to call methods already created when appropriate, instead of duplicating code for no reason. There are no methods in this entire assignment that should require more than about **20** lines of code, not counting comments, blank lines, or {} lines (and many of them need much less than that). Also, unless specified otherwise, there should be ***no* println** statements in your methods. Objects usually do not print anything, they only return **String** values from methods.

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. As usual, 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:

First, implement these methods in the DoublyLinkedList class:

- A **public void addFront(Number)** method that adds a new Node containing the Number received as a parameter to the beginning (front) of the list. This method does not require recursion.
- A **public void addEnd(Number)** method that adds a new Node containing the Number received as a parameter to the end of the list. This method does not require recursion.

- A **public int size()** method. This method is the public interface to the **sizeRec** method described below. This method must return the size of the list (i.e. how many nodes are in the list).
- A **private int sizeRec([parameter(s) of your choice])** method. This method will use recursive calls (to itself) to move through the list and count the number of nodes. Choose parameter(s) that will allow you to do that.

- A **public String toString()** method. This method is the public interface to the **toStringRec** method described below. This method must return a String representation of the list, enclosed within the << and >> pairs of characters (as in the example output shown below).
- A **private String toStringRec([parameter(s) of your choice])** method. This method will use recursive calls (to itself) to move through the list and build a String representation of it. Choose parameter(s) that will allow you to do that.

- A **public String toStringReversed()** method. This method is the public interface to the **toStringReversedRec** method described below. The goal of this method is to return a String representation of the list (also using the << and >> pairs of characters), but in reverse order (i.e. starting from the end of the list).
- A **private String toStringReversedRec([parameter(s) of your choice])** method. This method will use recursive calls (to itself) to move through the list (in the reverse order) and build a String representation of it. Choose parameter(s) that will allow you to do that.

You can test your classes using the supplied test program **`TestPhase1.java`**. You should get the output shown below.

```
<< 111 >>
<< 111 222.2 >>
<< 111 222.2 3000000000 >>
<< 0.777 111 222.2 3000000000 >>
<< 3000000000 222.2 111 0.777 >>
The list has a size of: 4
```

Phase 2:

Next, implement these methods in the **DoublyLinkedList** class:

- A **public static DoublyLinkedList createList(String)** method. The ultimate goal of this method is to create and return a new DoublyLinkedList based on the parsing of a file (the filename is the received String parameter).

  This method must open the file represented by the filename parameter using a Scanner (this is important for the next steps). Then, it must call the recursive method **parseScanner** described below. Your method must handle the type of exception thrown by doing this here, with an appropriate try-catch (do not add a "throws" statement to the signature of this method, otherwise the automated tests will fail).

  Fun fact: This kind of method is often called a "factory method". It is a design pattern that is used a lot in programming, or in other words, a general solution that is used often to solve a common problem (we're not going to get into the details of design patterns here; you will see them in 2nd and 3rd year courses). This is completely out of the scope of this course, but if you are interested, you can read more about this here:
  https://www.geeksforgeeks.org/design-patterns-set-1-introduction/
  https://www.geeksforgeeks.org/design-patterns-set-2-factory-method/

- A **private static void parseScanner(Scanner, DoublyLinkedList)** method. This method will use recursive calls (to itself) to move through the Scanner received as a parameter and add to the list received as a parameter. This method must add to the list only the tokens read from the file (through the Scanner) that correspond to either an Integer, a Long or a Double. All other tokens must simply be ignored. In the end, the list must be in the same order as in the file.

You can test your class with **`TestPhase2.java`** and the text file **`list.txt`**. You should get the output shown below.

```
<< 1 2 3 4 5 6.567 7000 8000 9000000 10000000000 11111111111 12.12 >>
<< 12.12 11111111111 10000000000 9000000 8000 7000 6.567 5 4 3 2 1 >>
The list has a size of: 12
```

Phase 3:

Next, implement these methods in the **DoublyLinkedList** class:

- A **private void removeNode(Node)** method. This method does not use recursion. The goal of this method is to remove from the current list the Node that is received as a parameter (you can assume that the Node is for sure in this list). This method will be useful for other methods in this assignment, including the methods below.
- A **public Node remove(int)** method. This method is the public interface to the **removeRec** method described below. The goal of this method is to remove the Node that is at the index (i.e. position) received as a parameter

(note that index 0 corresponds to the first Node). It must also return the Node that was just removed. If there is no Node at that index, or if the index received is a negative number, an IndexOutOfBoundsException must be thrown (you can do that here and/or in the recursive method below).

- A **private Node removeRec([parameter(s) of your choice])** method. This method will use recursive calls (to itself) to move through the list and remove the correct Node (and return it). Choose parameter(s) that will allow you to do that.

You can test your class with TestPhase3.java. You should get the output shown below:

```
<< 1 2 3 4 5 6.567 7000 8000 9000000 10000000000 11111111111 12.12 >>
The list has a size of: 12
The removed Node contained: 3
The list has a size of: 11
The removed Node contained: 4
The list has a size of: 10
The removed Node contained: 5
The list has a size of: 9
The removed Node contained: 6.567
The list has a size of: 8
The removed Node contained: 7000
The list has a size of: 7
The removed Node contained: 8000
The list has a size of: 6
The removed Node contained: 9000000
The list has a size of: 5
The removed Node contained: 10000000000
The list has a size of: 4
List (forward): << 1 2 11111111111 12.12 >>
List (backward): << 12.12 11111111111 2 1 >>
Seems like we have an invalid index!
```

Phase 4:

In this phase, we will add one method in the **Node** class, and two methods in the **DoublyLinkedList** class.

In the **Node** class, implement:

- A **public int compareTo(Node)** method that will compare this Node with the Node received as a parameter. As usual, the compareTo method must return an int value that is either a negative int, a positive int or 0, depending on whether this Node is smaller, greater or equal (respectively) to the Node received as a parameter.

  We will use the 'data' instance variable to compare Nodes. Note that the type of the 'data' instance variable is Number, and the Number class does not define a compareTo method. You will have to follow these rules to compare the different possible subtypes of Numbers that can be stored in 'data':
  - When comparing Nodes that contain different subtypes, we will always follow this ordering of subtypes: **Integer < Long < Double**. So for example, a Node containing an Integer will always be smaller than a Node containing a Long, no matter what the actual values are (e.g. 2 < 1L in our system).
  - When comparing Nodes that contain the same subtype of Number, an actual comparison of the values must be made to determine the order. (recursion)

In the **DoublyLinkedList** class, implement:

- A **public Node findMax()** method. This method is the public interface to the **findMaxRec** method described below. The goal of this method is to return the largest Node in the list, using the compareTo method of the Node class. If the list is empty, it should simply return null.
- A **private Node findMaxRec([parameter(s) of your choice])** method. This method will use recursive calls (to itself) to move through the list and return the largest Node in the list (using the compareTo method of the Node class). Choose parameter(s) that will allow you to do that.

You can test your class with TestPhase4.java. You should get the output shown below:

```
<< 1 2 3 4 5 6.567 7000 8000 9000000 10000000000 11111111111 12.12 >>
The Node with the max value contains: 12.12
true
<< 3 4 5 6.567 7000 8000 9000000 10000000000 11111111111 12.12 >>
false
<< 3 4 5 8000 9000000 10000000000 11111111111 12.12 >>
true
<< 3 4 5 8000 9000000 10000000000 >>
The Node with the max value contains: 10000000000
<< 9.99 3 4 5 8000 9000000 10000000000 >>
The Node with the max value contains: 9.99
```

Phase 5:

Finally, implement these methods in the **DoublyLinkedList** class:

- A **public void orderedInsertRec(Node toInsert, Node current)** method. This method is recursive, but we will not create an interface for this one. We will keep this method public here, because we want to be able to test it independently. For that reason, we also provide the required parameters: one Node that must be inserted, and the current Node, representing where to start the ordered insertion process.

  The goal of this method is to make an ordered insertion of the toInsert Node, following the rules of the compareTo method defined in the Node class, following an ascending order from left to right. A call to this method would only affect the portion of the list starting from the first Node (of the list) to the current Node (the parameter). In other words, this recursive method is moving through the list backwards starting from the given current Node, and inserts the toInsert Node as soon as it finds the right spot for it.

  This method is defined this way because it will be useful for the next two methods described below.

- A **public void insertionSort()** method. This method is the public interface to the **insertionSortRec** method described below. The goal of this method is to implement a recursive insertion sort algorithm, which will result in the full list being sorted in ascending order.
- A **private void insertionSortRec([parameter(s) of your choice])** method. This method will use recursive calls (to itself) to move through the list, take the current Node and insert it in the right spot inside the sorted part of the list (it will be on the left), following the traditional insertion sort algorithm. Of course you should make use of the orderedInsertRec method here, and maybe other method(s) defined previously in this assignment. This method should be quite short. If you have more than 6-8 lines of code here, rethink your approach.

You can test your class with TestPhase5.java (make sure list.txt and list2.txt are in the same folder). You should get the output shown below:

```
<< 22.1 >>
<< 5.55 22.1 >>
<< 1 5.55 22.1 >>
<< 1 111121212121212 5.55 22.1 >>
<< 1 600 111121212121212 5.55 22.1 >>
<< 1 600 111121212121212 0.5 5.55 22.1 >>
<< 1 600 111121212121212 0.5 5.55 22.1 77.789 >>

Before sorting:
<< 1 2 3 4 5 6.567 7000 8000 9000000 10000000000 11111111111 12.12 >>
After sorting:
<< 1 2 3 4 5 7000 8000 9000000 10000000000 11111111111 6.567 12.12 >>

Before sorting:
<< 10.1 1.5 0.25 4.75 11111111111111111 99999999999999999 55555555555555555 8 9 1 2 5 >>
After sorting:
<< 1 2 5 8 9 11111111111111111 55555555555555555 99999999999999999 0.25 1.5 4.75 10.1 >>
```

Hand in

Submit your two Java files (**`DoublyLinkedList.java`**, **`Node.java`**). ***Do not submit .class or .java~ files!*** You do ***not*** need to submit the **`TestPhaseN.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 your submitted code 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, otherwise your code will not compile on the marker's computer!***