Module 5 Lab -- Big Numbers

Due Saturday by 11:59pm **Points** 100

> The goal of this lab is to practice the implementation of a linked-sequential structure through the application of union data types.

Big Numbers

When we represent a number on a computer we must allocate a fixed number of bits to it. The number of bits dictates the range of numbers that can be stored. This data limitation applies to all data types, and prevents us from storing large numbers.

One way to represent a number is in terms of its digits, similar to how `int` is stored in terms of bits, or how a `String` represents a sequence of characters. In this assignment you will represent large integral numbers in a linked list representation. Instead of storing bits, we store individual digits (0-9) in a single node of such a list. We can store them in any order (corresponding to big-endian and little-endian in bit representation). For example, the number 32411 can be stored as `3 -> 2 -> 4 -> 1 -> 1` or `1 -> 1-> 4 -> 2 -> 3`.

Given an integral number, we can shift its digits to the left or right. For example 32411 can be *left-shifted* to get the number 324110. Thus left-shifting by one position is equivalent to multiplying the number by 10 (similar to how left-shifting by one bit in a binary representation multiplies the number by 2). Similarly, 32411 can be *right-shifted* to get the number 3241, which is the integer-division by 10.

We can support basic addition of a single digit to a number. For example 324115 + 7 = 324122. Shifting and adding single digits can allow us to create arbitrarily large numbers, one digit at a time. For example 32411 can be created by:

1. Start with 0.
2. Left-shift by 1 position, and add 3
3. Left-shift by 1 position (to get 30) and add 2
4. Left-shift by 1 position (to get 320) and add 4
5. Left-shift by 1 position (to get 3240) and add 1
6. Left-shift by 1 position (to get 32410) and add 1

Numbers can also be added by using simple arithmetic: start from the right-most digits and add them. Record the sum and carry, and add the carry to the next pair of digits, and so on. Note that the numbers may be of different lengths.

What to do

Package: bignumber

Design and implement an interface `BigNumber` that defines the above operations, specifically with the following method signatures:

- A method `length` that returns the number of digits in this number.
- A method `shiftLeft` that takes the number of shifts as an argument and shifts this number to the left by that number. A negative number of left-shifts will correspond to those many right shifts.
- A method `shiftRight` that takes the number of shifts as an argument and shifts this number to the right by that number. The number 0 can be right-shifted any positive number of times, yielding the same number 0. A negative number of right-shifts will correspond to those many left shifts.
- A method `addDigit` that takes a single digit as an argument and adds it to this number. This method throws an `IllegalArgumentException` if the digit passed to it is not a single non-negative digit.
- A method `getDigitAt` that takes a position as an argument and returns the digit at that position. Positions start at 0 (rightmost digit). This method throws an `IllegalArgumentException` if an invalid position is passed.
- A method `copy` that returns an identical and independent copy of this number.
- A method `add` that takes another `BigNumber` and returns the sum of these two numbers. This operation does not change either number.
- A method to compare two `BigNumber` objects for ordering.

Now implement this interface in a class `BigNumberImpl`. This implementation represents non-negative numbers of arbitrary lengths. Beyond implementing the `BigNumber` interface, this implementation should have the following features/obey these constraints:

- You are not allowed to use any of Java's existing list implementations, interfaces, **arrays**, or otherwise any collection classes or maps to implement big numbers. You must create your own list implementation. The implementation may be customized for this application.
- You are not allowed to use the `BigInteger` or any similar existing classes from JDK in your implementation; however you may use `BigInteger` in your tests.
- This class should have a constructor with no parameters that creates the number 0.
- This class should have another constructor that takes a number as a string and represents it. This constructor should throw an `IllegalArgumentException` if the string passed to it does not represent a valid number.
- This representation should not contain any unnecessary digits. That is, if representing a 5-digit number, there should be exactly 5 digits represented in this object.
- This class must provide a way to compare two numbers.
- This class should include a `toString` method that returns a string representation of this number, as simply the number itself.

We recommend that you select an operation and implement it end-to-end before starting the next operation:

1. Add it to the interface
2. Add an empty implemetation
3. Write test cases
4. Complete the implementation until all of the tests cases pass

Testing

Whenever you write a class, you should also write tests for that class that proves not only that your code CAN work but that it WILL ALWAYS work. Your goal here is to write tests to achieve as close to 100% coverage as possible. But even more importantly, your tests should be sufficient to ***convince someone else that your code works correctly***.

What to Submit

1. Create a zip file that directly contains only your src/ and test/ folders. When you unzip the file, you must see only these two folders.
2. Log on to the **Handins server (https://handins.ccs.neu.edu/)**
3. Navigate to this lab and submit the zip file.
4. Wait for a few minutes for the grader's feedback to appear, and take action if needed.

Grading Criteria

This assignment will be assessed via the automatic testing available on the Handins server. You should ensure that your code does not have any code style violations and that it passes all of the test cases. This is worth 1% of your final grade.

Additionally, your code will be assess via peer evaluation. Each student will be required to review one other student's code for an additional 0.5% of their final grade. During the peer review your code will be evaluated based on three criteria:

1. Design and organization of classes, interfaces, and methods including
    - How appropriately you captured the data and operations in the implementation
    - Whether code looks well-structured and clean (not unnecessarily complicating things or using unwieldy logic)
    - How well your design and implementation embraces various design principles that you have learned so far
2. Understandability and quality of documentation including

- Whether the documentation enhances the understandability of the code rather than just repeating how something is implemented
- Whether it is stand alone, meaning whether the documentation is sufficient for use without reading the implementation

3. Quality and coverage of tests including
    - Whether tests are modular -- do you write one big, monolithic test method or does your test class have separate methods for each test
    - Whether tests are written in such a way so that the reader knows what is being tested
    - Whether the tests convince the reader that if all tests pass then the code it is testing works correctly