Assignment 2: Big Integers

Out: 4th February 2015 **Due**: 23rd February 2015 at 11:55pm

In this assignment you will write a class that represents big integers (i.e. very large integer numbers) and implement some operations on it.

A Tester class has been provided to you. You must write all the code necessary to successfully run this tester program. No changes to this tester program are allowed. Your class can have as many private helpers as you want, but the only public functions should be those required to run the tester and the destructor.

Big Integers

You must implement a big integer as a doubly-linked list. Each node in this list stores a single digit (0-9). The head of the linked list represents the most significant digit, while the tail represents the least significant digit. For example,1234567890 is represented as 1 <-> 2 <-> 3 <-> 4 <->5 <->6 <-> 7 <-> 8 <-> 9 <-> 0, with "head" pointing to 1 and "tail" pointing to 0. Negative big integers can be represented by maintaining a separate variable that signifies whether this number is positive or not.

Addition of Big Integers

Addition of big integers is performed similar to how we add manually.

1. Start from the least significant digit of each number.
2. Add the respective digits and a carry from the previous addition (carry starts out as 0).
3. The sum modulo 10 gives the digit in the answer, and the sum divided by 10 gives the next carry.
4. The number of digits in the result may be one more than the number in the greater of the two operands, as the last sum may produce a carry.

Recall that the above algorithm assumes that both integers have the same number of digits. We can guarantee this by padding 0's to the shorter of the two numbers.

Adding numbers that may be positive or negative makes the math a bit more involved.

Negative Big Integers

Recall that in binary math, a negative integer is represented by reserving the most significant bit for the sign (0 for positive, 1 for negative). Negative binary numbers are stored using 2's complement notation. For example, to represent -17 using 8 bits, we first represent +17 in binary (00010001), then reverse all the bits (11101110) and add 1, to produce 11101111. To go back to separately representing the sign and the value, we first look at the most significant bit. In this case it is 1, which means the number is negative. We take the 2's complement of this number in the same way (reverse all bits to produce 00010000 and add 1 to produce 00010001 which is the value of 17).

The decimal equivalent of this is the "10s complement". To represent -17, we would need 3 digits. We start with representing +17 (017). Then, we subtract each digit from 9, producing 982, and finally add 1 to produce 983. Thus, -17 is represented in 10s complement as 983. To go back to separately representing the sign and the value, we look at the most significant digit. In this case it is 9, which means the number is negative. We take the

10s complement of this number in the same way (subtract each digit from 9 to produce 016 and add 1 to produce 017 which is the value of 17).

The advantage of using the complements method of representing negative numbers (2s complement for binary, 10s complement for decimal) is that addition becomes easier. We do not have to handle cases separately where one or more operands may be negative.

Addition algorithm with negative operands

In order to use the above addition algorithm to add possibly negative numbers, we must allot an extra digit for the sign. We must also ensure that overflow is handled properly (i.e. when the result is longer than the two numbers such as 9+8=17). The modified addition algorithm can be summarized as follows:

Let ‘a’ and ‘b’ be the two numbers to be added, with l_a and l_b number of digits respectively.

1. Pad extra leading zeros to each so that their lengths is $2+\max(l_a,l_b)$ (1 for the sign, 1 for possible overflow).
2. Convert both operands to “signed” form:
    a. If ‘a’ is negative, take 10s complement.
    b. If ‘b’ is negative, take 10s complement.
3. Add using the above algorithm. Ignore any carry that is produced by the most significant digits.
4. Convert ‘a’, ‘b’ and result back to “digit” form:
    a. If the most significant digit is 9, take 10s complement and note the sign as negative. If 0, do not change the digits, but note the sign as positive.
5. Remove any leading zeros in the result, ‘a’ and ‘b’.

Rules and Hints for the assignment

1. You must use a doubly-linked representation throughout, including the addition operation. You are NOT allowed to convert the number into a string to implement the math.
2. It is recommended that you implement the doubly-linked list inside the BigInteger class itself, instead of implementing it separately and using it as a black box in the BigInteger class.
3. Use the C++ string class wherever possible, instead of C-style strings.
4. Before using the provided tester, please write your own main function to test. Use the Windows or Linux calculator to verify your answers. Start with small numbers before moving on to big numbers. Make sure you test with one or both operands being negative!

What/How to submit

1. Set up a Makefile that creates two executable files (MyTester and BigIntegerTest): one with your main function and one with the provided main function.
2. Submit all source code files (*.h/*.cpp) **and a Makefile** as “assignment-2.zip” and submit on ReggieNet. Make sure that your source file(s) are suitably commented.