**Background**

Integer.MAX\_VALUE is the maximum value of a Java int: 2147483647. If you want to work with even bigger integers, you have the option of using the type long, which has the maximum value of Long.MAX\_VALUE = 9223372036854775807.

But what if you this is not enough? What if you are working on something like an astronomy application, and need to keep track of things such as number of stars in the universe? This is of the order of 1023, larger than the maximum long value. For situations like this, you need to be able to work with an integer type that can hold arbitrarily large or small positive and negative values, with any number of digits. There is no built-in type in the language for this, so you need to craft your own. In this assignment, you will do exactly this, by implementing a class called BigInteger, with a representative small set of operations.

The trick is to store an integer as a linked list of digits. For example, the integer 754 will be stored as:

4 -> 5 -> 7

Why are the digits stored backward? It's because computations such as adding or multiplying big integers are easier to do if the linked list stores digits in ascending order of positional value. So the least significant digit is in the first node of the linked list, and the most significant digit is in the last node.

This is a simple linked list, NOT circular, with a front pointer. The sign (positive or negative), is stored separately in a boolean field.

Also, there can never be zeros at the end of the list. Such zeros will be insignificant (appearing before the most significant digit in the number) as in 00754.

**Implementation and Point Assignment**

Download the attached biginteger\_project.zip file to your computer. DO NOT unzip it.

Instead, follow the instructions on the Eclipse page under the section "Importing a Zipped Project into Eclipse" to get the entire project into your Eclipse workspace.

You will see a project called BigInteger with classes BigInteger, DigitNode, and BigTest in package math. The DigitNode class implements the linked list node that will hold a digit of a big integer linked list.

You need to fill in the implementation of the following methods in the BigInteger class:

|  |  |
| --- | --- |
| **Method** | **Points** |
| parse | 10 |
|  |  |
| add | 30 |
|  |  |
| multiply | 25 |
|  |  |

Note: When parsing an input string as an integer, you can use the Character.isDigit(char) method to tell if a character is a digit.

Make sure to read the comments that precede classes, fields, and methods for code-specific details that do not appear here.

Observe the following rules while working on **BigInteger.java**:

* You may NOT add any import statements to the file.
* You may NOT add any new classes (you will only be submitting BigInteger.java).
* You may NOT add any fields to BigInteger class.
* You may NOT modify the headers of any of the given methods.
* You may NOT delete any methods.
* You MAY add helper methods if needed, as long as you make them private.

Also, you may NOT make any changes to the DigitNode class (you will only be submitting BigInteger.java). When we test your submission, we will use the exact same version of DigitNode that we shipped to you.

NOTE:

* If your linked list stores digits with most significant digit first, or stores insignificant zeros, you will get ZERO for test cases that examine the resulting linked list in your add and multiply implementations, even if the result is mathematically correct.
* You will NOT get any credit if you convert the linked list representation to an array, work on arrays, then convert back to linked lists. You must work with linked lists ONLY all the way through.
* You will NOT get any credit if you convert the entire linked list representation to an int, do the math on int values, and store the result back in a linked list.

**Running/Testing**

Use the class BigTest to test your implementation. Carefully read the code in the file to get a good idea of how the BigInteger methods are used.

Here's a sample run of BigTest:

(p)arse, (a)dd, (m)ultiply, or (q)uit? => p Enter integer => 125 Value = 125 (p)arse, (a)dd, (m)ultiply, or (q)uit? => p Enter integer => -126 Value = -126 (p)arse, (a)dd, (m)ultiply, or (q)uit? => p Enter integer => +1 Value = 1 (p)arse, (a)dd, (m)ultiply, or (q)uit? => p Enter integer => 005 Value = 5 (p)arse, (a)dd, (m)ultiply, or (q)uit? => p Enter integer => 123xy56 Incorrect Format (p)arse, (a)dd, (m)ultiply, or (q)uit? => a Enter first integer => 12 Enter second integer => -13 Sum: -1 (p)arse, (a)dd, (m)ultiply, or (q)uit? => a Enter first integer => 16756726 Enter second integer => 0 Sum: 16756726 (p)arse, (a)dd, (m)ultiply, or (q)uit? => m Enter first integer => 12 Enter second integer => 200 Product: 2400 (p)arse, (a)dd, (m)ultiply, or (q)uit? => m Enter first integer => 178 Enter second integer => -156 Product: -27768 (p)arse, (a)dd, (m)ultiply, or (q)uit? => m Enter first integer => -16 Enter second integer => -05 Product: 80 (p)arse, (a)dd, (m)ultiply, or (q)uit? => q