Infinite Precision Arithmetic

Assignment:

In most programming languages, integer values are of a fixed size, e.g., 32 bits in **C++**, 64 bits in Java. 32 bits will allow integers of approximately 10 decimal digits. For most purposes this is sufficient, but when it is not, infinite precision integer arithmetic can be implemented in software. For this project you will implement a "bignum" or infinite precision arithmetic package for non-negative integers.

Input and Output:

The input to this program should be **read from standard input** and the output should be **directed to standard output**. The name of the program should be "bignum".

The input file will consist of a series of arithmetic expressions. To simplify the project, all expressions will be in Reverse Polish Notation (RPN). In RPN, the operands come before the operator. For example, you normally see multiplication written $(a * b)$. In RPN, this expression would appear as $(a\ b\ *)$. For expressions with more than one operator, each time an operator is encountered, it operates on the two immediately preceding sub-expressions. For example, $(a+b*c)$ in "normal" notation would be $(a\ b\ c\ *\ +)$ in RPN. This means that b and c are multiplied, then the results of the multiplication are added to a (this follows the normal rules of arithmetic precedence in which $*$ operations take precedence over $+$ operations). In RPN, there is no operator precedence; operators are evaluated left to right. Thus, the expression $((a + b) * c)$ is written $(a\ b\ \ + c\ *)$ in RPN.

Input operands consist of a string of digits, and may be of any length. You should trim off any excess zeros at the beginning of an operand. Expressions may also be of any length. Spaces and line breaks may appear within the expression arbitrarily, with the rule that a space or line break terminates an operand, and a blank line (i.e., two line breaks in a row) terminates an expression. The operations to be supported are addition $(+)$, multiplication $(*)$, and exponentiation $(\hat{})$. An error should be reported whenever the number of operators and operands don't match correctly – i.e., if there are no operands left for an operator, or if the expression ends without providing sufficient operators. When an error is detected, processing should stop on the current expression, and your program should proceed to the next expression.

For each expression, echo the input as it is read, followed by the result of the operation, or an error message, as appropriate. Be sure that the result is clearly marked on the output for readability.

Implementation:

RPN simplifies the project since it can be easily implemented using a stack. As the expression is read in, operands are placed on the stack. Whenever an operator is read, the top two operands are removed from the stack, the operation is performed, and the result is placed back onto the stack. You may assume that the stack will never hold more than 100 operands.

The main problem in this project is implementation of infinite precision integers and calculation of the arithmetic operations $+$, $*$, and $\hat{}$. **NOTE:** These three operations must be implemented **recursively**. Integers are to be represented in your program by a linked list of digits, one digit per list node. You must maintain a freelist of linked list nodes as discussed in class, and allocate new link nodes with the **new** operator as needed. Each digit may be represented as a character,

or as an integer, as you prefer. You will likely find that operations are easier to perform if the list of digits is stored backwards (low order to high order).

Addition is easily performed by starting low order digits of the two integers, and add each pair of digits as you move to the high order digits. Don't forget to carry when the sum of two digits is 10 or greater!

Multiplication is performed just as you probably learned in elementary school when you multiplied two multi-digit numbers. The low order digit of the second operand is multiplied against the entire first operand. Then the next digit of the second operand is multiplied against the first number, with a "0" placed at the end. This partial result is added to the partial result obtained from multiplying the previous digit. The process of multiplying by the next digit and adding to the previous partial result is repeated until the full multiplication has been completed.

Exponentiation is performed by multiplying the first operand to itself the appropriate number of times, decrementing the exponent once each time until done. In order to simplify implementation, you are guaranteed the the exponent will be small enough to fit in a regular `int` variable. You should write a routine to convert a number from the list representation to an integer variable, and use this to convert the top operand on the stack when the exponentiation operator is encountered. Be careful with exponents of 0 or 1.