HW1. Doubly Linked Lists and math operations with infinite digits
COSC 2320 Data Structures

Instructor: Boanerges Aleman-Meza

Updates

1. Corrected error for sample output (removed "#8" that should not had been there). Added a note about requirement for folder name (`hw1`), and an indication about compiling via g++.

1 Introduction

You will create a C++ program that can evaluate arithmetic operations with floating point numbers having any number of digit. These numbers do not have the limitation of maximum number of accurate digits that some data types have (normally dependent on size of CPU registers).

2 Input and Output

The input is a regular plain-text file. Each line is terminated with an end-of-line character(s). Each line will contain an arithmetic operation between two floating point numbers.

Input example:

```
0*0

890+0

0*400

7650.4+100.26

160008800999008800+4

976976*863586589

1247.00127#8
```

The program should display as output the input expression and the results, separated with this symbol: =

Output example:

```
0*0=0.0

890+0=890.0

0*400=0.0

7650.4+100.26=7750.66

160008800999008800+4=160008800999008804.0
```

```
976976*863586589=843703371374864.0

1247.00127#8=124700127000.0
```

Advanced functions example:

```
median
```

Input example with Advanced functions:

```
12+4000
10+0.0
median
300000+123
median
4+0.976
4.0+0.00000006765
0.0+0
4.0+0.00000000000000000000214340000001
4.0+0.012
median
swap(9800043.000888401)
swap(23400000.56780001)
```

Output example with Advanced functions:

```
12+4000=4012.0
10+0.0=10.0
median=11.0
300000+123=300123
median=12.0
4+0.976=4.976
4.0+0.00000006765=4.00000006765
0.0+0=0.0
4.0+0.00000000000000000000214340000001=4.00000000000000000000214340000001
4.0+0.012=4.012
median=4.0
swap(9800043.000888401)=888401.9800043
swap(23400000.56780001)=56780001.234
```

3 Program Input and Output Specification

The main program should be called: groovymath
The output should be written to the console (via printf, or cout), but the TA will redirect it to create some output file.
Call syntax at the OS prompt (notice the double quotes):

```
groovymath "inputfile=filename.txt;digitsPerNode=number"
```

or:

```
./groovymath "inputfile=filename.txt;digitsPerNode=number"
```

Place your codes in a folder named `hw1` on your home folder. Failure to do so will cause your program to have a zero grade due to inability for automated grading. Your codes will be compiled via: `g++ *.cpp`
The TA may or may not specify the name of the executable, such as `g++ -o groovymath *.cpp`

It is OK to assume the following:

- The input file is a a small plain-text file (say < 20000 bytes). It is not needed to handle binary files.
- Operators: $+*$
- Limits:
  You can assume that input text lines can have up to 256 characters but that should not be a limit for the linked lists.
  You can not assume a maximum number of lines for the input file (for example, a file may have many blank lines).
- The homework will be graded from 0 to 100. A program that compiles but does nothing gets 10 points. Successfully passing extra credit operations can make the grade to be over 100 (the extra points can only be added to other homeworks having less than 100 points anywhere in the semester).

4 Requirements

- Top requirement: correctness. You are encouraged to test your program with many input files. Programs that crash or produce exceptions will get zero points on each test case when this happens.
- Doubly linked lists are required. That is, you need to implement your own doubly linked lists. A program using arrays to store long numbers will receive a grade of zero. However, arrays for parameters or other auxiliary variables are acceptable.
  A program that does the operations/calculations in arrays instead of via the doubly linked list is not acceptable.
- Output of numbers must be without leading zeroes because automatic grading may cause your program to fail a test case.
  For example, if the `digitsPerNode` is 4 and the operation was 3+10 and your output has leading zeroes such as 0013.0, then it will fail the test case and there will be zero points on such failed test cases.
- Always output at least one digit after the decimal point (default is zero). Example: $2 + 4 = 6.0$
- When an operation causes infinite number of digits after the decimal point such as 1/3, output exactly 20 such digits after the decimal point (no rounding on the last digit)
- Operation #
  This operation shifts elements to the left of the decimal point by as many times as indicated by the second operand; similar to multiplying the number by 10 at the power of the second operand.
- Output of numbers must be without unnecessary trailing zeroes after the decimal point because automatic grading may cause your program to fail a test case.
  For example, if the result of an operation is 9.7762 and your output has trailing zeroes such as 9.7762000 then it will fail the test case and there will be zero points on such failed test cases.
- Breaking an arithmetic expression into multiple lines displayed to the console is not allowed (because it will mess testing). Each test case when this happens will get zero points.
- Breaking a number into a list of nodes. Each node will store the number of digits specified in the parameter named `digitsPerNode`
  It is acceptable to "align" digits after reading the entire number so that the rightmost node before the decimal point has all the digits.
  Example of numbers stored as a list of nodes of 2 digits:

```
859 stored as {8,59} or {85,9}
0 stored as {0}
3.145 stored as {3} for the part before decimal point,
    and {14,5} or {1,45} for the part after the decimal point
```

- Operation `swap`
  The `swap` operation exchanges the digits before the decimal point with those after the decimal point. If a number has the form abc.dfg then the result of the operation is dfg.abc (where the rules for leading zeroes and trailing zeroes still apply).

- Doubly linked list features:
  Numbers should be read and inserted manipulating the list forward starting with the most significant digit (leftmost digit). Eeach node represents values multiplied by some power of 10, depending on its position within the list. If it is a 3-digits per node list, then the powers would be $1, 1000, 1000000, ...$

- Addition operation:
  Numbers must be added starting on the least significant digit, keeping a carryover from node to node. The implementation must be done via traversals of your own linked lists.

- Multiplication operation:
  Numbers must be multiplied with the traditional method you likely learned in elementary school starting from the rightmost digit. However, your multiplication algorithm must handle the number of digits per node specified (where the simplest case is that of 1 digit per node). You must use linked lists to store partial results.

- Storage of numbers:
  Both of the input numbers (that is, operands) must be stored in lists. The result of the arithmetic operation must be stored on a third list as well. Output to the console must be done traversing the list forward (from leftmost digit to rightmost digit). After an operation is complete and the result has been output to the console, the program must deallocate (`delete`) the lists (the exception being the lists for the first operand in case that you implement the optional advanced function `median`).

- Memory:
  Your program must use `new` to allocate each node and must free memory of each allocated via `delete`

- `digitsPerNode`:
  Each node will store a fixed number of digits as specified by the `digitsPerNode` parameter. You can assume that such parameter will be between the values of 1 to 8.

- Advanced function: `median`
  Calculate the median of all "first operand" values entered so far. For example, on operations such as $16 * 20$ it will only consider the 16 (first operand).
  Optional to implement: 10 points extra credit.