Indian Statistical Institute

Semester-I 2019-2020

M.Tech.(CS) - First Year

Lab Test 2 (27 September, 2019)

Subject: Data and File Structures Laboratory

Total: 60 marks Duration: 4 hrs.

**SUBMISSION INSTRUCTIONS**

1. Naming convention for your programs: `cs19XX-test2-progY.c`
   **IMPORTANT: Insert a single alpha-numeric string of your choice, 6-8 characters long, in the name given above as shown in the examples below. Think of this string as something like a security password, except that you are not required to remember the string. Examples: `cs19XX-test2-abcdef-prog1.c`, `mnopqr-cs19XX-test2-prog2.c`, `cs19XX-test2-prog2-uvwxyz.c`**
2. When you have finished, copy all your files to `~dfslab/2019/labtest2/cs19XX/` on 192.168.64.35.

1. **(20 marks)** The power of any arbitrary square matrix $A$, denoted as $A^n$, for a nonnegative integer $n$, is defined as the matrix product of $n$ copies of A. This means

$$A^n = \underbrace{A \cdots A}_{n \text{ times}}.$$

Note that, a matrix to the zeroth power is defined to be the identity matrix of the same dimensions, i.e. $A^0 = I$.

Given the square matrix $A$ and the nonnegative integer $n$, write a program that can recursively compute $A^n$ involving minimal multiplications.

**Input Format**

The first line of input (to be read from stdin) is a pair of integers representing order (number of rows/columns) of the matrix $A$ and the power (i.e., $n$) to which it has to be multiplied, respectively. This is followed by the elements of the square matrix $A$. The successive rows are provided in successive lines. The elements in each row are separated by spaces.

**Output Format**

The output (to be printed to stdout) will show the value of $A^n$.

**Sample Input 0**

3 4
1 1 1
1 1 1
1 1 2

**Sample Output 0**

34 34 48

34 34 48
48 48 68

**Sample Input 1**
2 0
2 2
2 2

**Sample Output 1**
1 0
0 1

**Sample Input 2**
2 17
1 2
3 4

**Sample Output 2**
617852597821 900475124662
1350712686993 1968565284814

2. **(20 marks)** Let us define an *overspill stack* as a special kind of stack from which data items may automatically spill over. An overspill stack has a *spill tolerance* level of $S$. If $S$ stack operations (PUSH, POP or DISPLAY) are completed since the last spill, the top element of the stack will spill over immediately **after** the completion of the $S$-th operation. Moreover, if the number of elements in the stack crosses its capacity, then the top element in the stack will automatically overflow because of its inherent nature.

   NOTES:

   (a) If the reason for a spill (i.e., the $S$-th operation after a spill) coincides with a natural overflow operation, only *one* element will spill over followed by the overflow.

   (b) Pushing into a full stack (overflow) and popping from an empty stack (underflow) are counted as operations because they are consequences of PUSH and POP, respectively, even though they do not change the stack.

   Write a program to implement the PUSH, POP and DISPLAY operations on a **generic** overspill stack, given its maximum capacity (in terms of number of elements) and spill tolerance level.

   **Input Format**
   The first line of input (taken from stdin) consists of three fields:

- a single character from the set $\{c, f, i\}$ which specifies the element-type of the stack (`char`, `float` or `int`); and
- a pair of integers, specifying the capacity of the overspill stack and its spill tolerance level $S$, respectively.

The subsequent lines specify a sequence of PUSH, POP, and DISPLAY operations to be performed on the stack. A push operation is denoted by a leading '`+`' sign, a pop operation is denoted by a leading '`-`' sign, and a display operation is denoted by a leading '`=`' sign.

**Output Format**

Your program should print the contents of the stack for each DISPLAY operation, however, the result of subsequent spills, if any, are not to be shown. For the other operations (PUSH and POP, and their consequences), the final contents of the stack (including the subsequent spills, if any, have been completed) are to be shown. The stack contents should be printed on one single line, in order from top to bottom, as a list of single-space-separated elements.

**Sample Input 0**

```
i 5 2
+ 2
-
-
+ 1
+ 7
=
+ 3
```

**Sample Output 0**

```
2
   <-- stack is empty, spill has no effect
   <-- stack is empty
   <-- 1 was pushed, but immediately spilled over
7
7  <-- stack has been displayed, but subsequently 7 has spilled over
3
```

**Sample Input 1**

```
c 10 3
+ s
+ f
```

```
+ d
-
+ l
```

**Sample Output 1**

```
s
f s
f s   <-- d has spilled over
s
l s
```

**Sample Input 2**

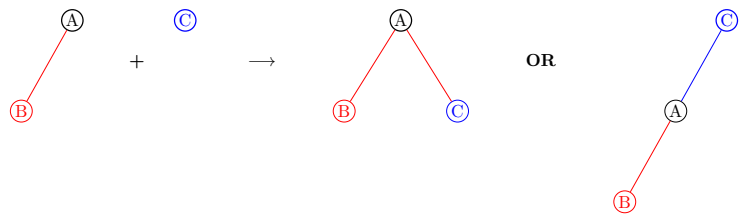
```
f 4 3
+ 1.2
+ 2.9
+ 3.0
+ 4
+ 5.7
+ 6
+ 7.3
```

**Sample Output 2**

```
1.2
2.9 1.2
2.9 1.2   <-- 3.0 has spilled over
4 2.9 1.2
5.7 4 2.9 1.2
4 2.9 1.2   <-- overflow, and 5.7 has spilled over
7.3 4 2.9 1.2
```

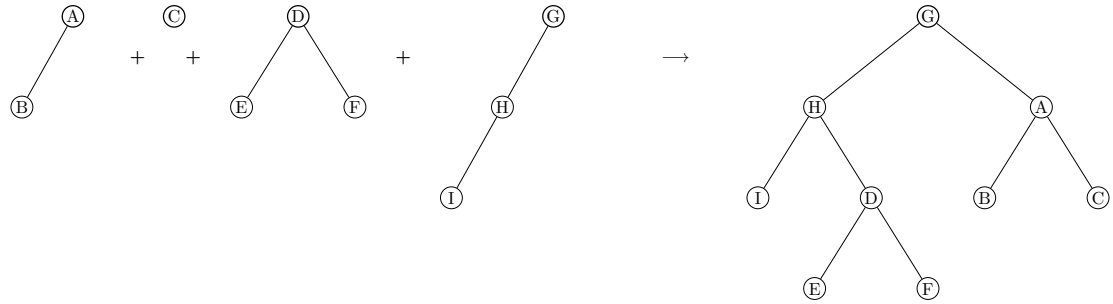
3. **(20 marks)** Suppose you are given a collection of binary trees, $T_1$, $T_2$, ..., $T_k$. Your task is to join all the trees to form one single binary tree that has the minimum possible height. In order to join two trees $T_i$ and $T_j$, the root of one of the trees must be made a child of some node of the other tree that has less than two children.

   **Example 1:** Given the two trees in the left part of the figure below, two ways in which they can be joined are shown on the right side.




The tree in the middle corresponds to the correct way of joining, since the resultant tree has minimum height. Write a program to accomplish this task.

**Example 2:** Note that there may be multiple correct answers.




**Input Format**

Input will be provided via standard input in the following format. The first line of input will contain $k$, denoting the number of binary trees to be given as user input. This will be followed by the successive entries for each binary tree of which the first line is an integer $n_i$, representing the number of nodes in the $i^{th}$ binary tree. This will be followed by $n_i$ more lines. Each of these remaining lines will correspond to one node in the tree and will consist of 3 integers: the data (an integer to be stored in the node), the line number (count starts from 1 representing the line where the root appears) of the node corresponding to the left child (-1 if there is no left child), and the line number corresponding to the right child (-1 if there is no right child).

**Output Format**

The output will print a single combined binary tree, having $\sum_{i=1}^{k} n_i$ number of nodes, in the following format spread over $\sum_{i=1}^{k} n_i$ number of lines. Each of these lines will correspond to one node in the tree and will consist of 3 integers: the data (an integer to be stored in the node), the line number (count starts from 1 representing the line where the root appears) of the node corresponding to the left child (-1 if there is no left child), and the line number corresponding to the right child (-1 if there is no right child).

**Sample Input 0**

```
2
2
10 2 -1   <-- line number 1 starts from here
20 -1 -1
1
```

```
30 -1 -1   <-- line number 1 starts from here
```

**Sample Output 0**

```
10 2 3
20 -1 -1
30 -1 -1
```

**Sample Input 1**

```
4
2
2 2 -1   <-- line number 1 starts from here
4 -1 -1
1
6 -1 -1   <-- line number 1 starts from here
3
8 2 3   <-- line number 1 starts from here
10 -1 -1
12 -1 -1
3
14 2 -1   <-- line number 1 starts from here
16 3 -1
18 -1 -1
```

**Sample Output 1**

```
14 2 3
16 4 5
2 6 7
18 -1 -1
8 8 9
4 -1 -1
6 -1 -1
10 -1 -1
12 -1 -1
```