Data Structure and Algorithm [Assignment 1]

Submission to be done on UCP PORTAL
Deadline: Sunday, November 17, 2024

Task 1

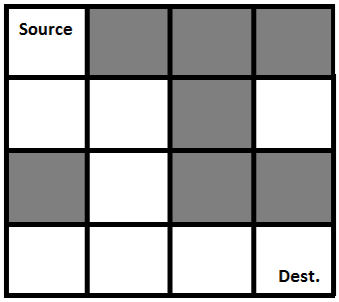
Let us implement Rat in a MAZE as example problem that can be solved using a Stack. A Maze is given as N*N binary matrix of blocks where source block is the upper left most block i.e., maze[0][0] and destination block is lower rightmost block i.e., maze[N-1][N-1]. A rat starts from source and has to reach destination. The rat can move in four directions: back, forward, up and down.

In the maze matrix, 0 means the block is dead end and 1 means the block can be used in the path from source to destination. Visited path (visited index) should be marked as -1. Note that this is a simple version of the typical Maze problem. For example, a more complex version can be with limited number of moves.

All inputs and outputs will be done using filing. Input file name will be ***input.txt*** and output will be in ***output.txt***.

Following is an example maze.

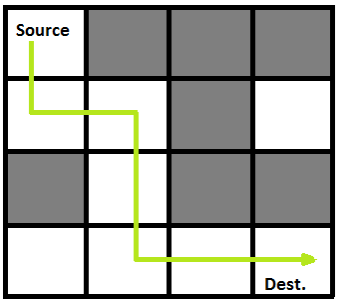
```
Gray blocks are dead ends (value = 0).
```



Following is binary matrix representation of the above maze (in file "***input.txt***"). Each row is in a new line; numbers in each row are separated by white spaces.

```
1    0    0    0
1    1    0    1
0    1    0    0
1    1    1    1
```

Following is maze with highlighted solution path.



Following is the solution matrix (output of program) for the above input matrix. The output will be placed in "***output.txt***"

```
1    0    0    0
1    1    0    0
0    1    0    0
0    1    1    1
```

Note: In case of multiple path, the very first path found should be saved. If the solution does not exist, that is, you are not able to find a path, write: **PATH NOT FOUND** in the file "***output.txt***".

Task 2

The compiler scans the expression either from left to right or from right to left.

Consider the expression: **a + b * c + d**

The compiler first scans the expression to evaluate the expression "b * c", then again scan the expression to add "a" to it. The result is then added to "d" after another scan. The repeated scanning makes it very inefficient. It is better to convert the expression to postfix (or prefix) form before evaluation. The corresponding expression in postfix form is: "a b c*+d+". The postfix expressions can be evaluated easily using a stack.

Implementation

You have to implement the following functionality using stack which takes input of fully parenthesized **infix expressions** and **convert it to postfix form.**

Sample Input:

(((12 + 13) * (20 - 30)) / (811 + 99))

Sample Output: (there is a single space in between each operator and operand)

12 13 + 20 30 - * 811 99 + /

Hint:

Take input in a character array. If there is a space in between 2 digits, consider them as separate numbers. For example: 12 will be considered as TWELVE, however, 1 2 will be ONE and TWO. Our algorithm cannot contain negative numbers, otherwise, you will have to pop multiple items till a starting bracket. We are popping 2 values only.