Computer Science I Program 3: Strands (Linked Lists)

Please Check Webcourses for the Due Date

Please read the whole assignment before you start coding

Objective

Give practice with linked lists.
Give practice with functions in C.

Strands Queries - Modified

In the original NY Times Strands game, users are given a 2D grid of letters, and are asked to partition the grid into "strands", where each strand is a path formed by starting at one letter and moving in one of the eight directions (NW, N, NE, W, E, SW, S, SE) to get to the next letter, continuing the process until each letter in the grid is part of precisely one word.

Here is an example puzzle with a theme (school related terms):

C↓	E→	X↓	G	R→	E
O↓	M	←A	N↑	←I	↑←U
L→	L→	E↓	D→↑	↓←E	T↑
R	Q↓	G↓	A↑	R↑	C↑
E↑	U→↓	E	C→	O↓	E↑
P↑	M→↓	I→	Z	U↓	L↑
A↑	O↑	E→	W→↓	R→	S↓
P↑	H↑	K	←R	←O	E

A few of the squares are start squares (no incoming arrows), which store the first letter of a word, and few other squares are end squares (no outgoing arrys), which store the last letter of a word.

For this program, you'll be given information about the board and be required to store the board as a 2D array of pointers to structs, each of which are nodes for a doubly linked list. For each cell, you'll be given both the where the next letter for that word is and the previous letter, if those exist. Thus, each word will be stored in a doubly linked list, while each of the physical nodes will reside in an indexed array.

Note: In the illustration above, only next pointers are shown with arrows. (The previous pointers can be inferred.)

Problem

Given the initial structure of the board as described above, write a program that builds the structure in the manner designated (there are many ways to store this data differently, but the point of the assignment is to practice linked lists), then handles the following query types in sequence:

1. Given a position in the grid (0-based row and column numbers), prints out the word that grid position is part of in the current Strands configuration.

2. Given a position in the grid, reverse the word that grid position is part of.

3. Given a position in the grid that is an ending position, as well as an adjacent position that is a starting position for a different word, concatenate those two words together.

High Level Hints

Here are some basic hints explaining how to implement each of the listed tasks:

1. From the given node, continue following the previous pointers until a node's previous pointer is NULL. (Don't go to NULL, just the node that exists but has its previous pointer pointing to NULL.) Then, traverse the list in forward's order via the next pointers, printing each letter one by one. (Alternatively, you can copy these letters into a string, null terminate that string and return it to a function that prints it.

2. This is the hardest of the tasks. Use auxiliary pointers and make sure not to lose any part of the list. Ultimately, for each node in the word, its previous and next pointers just have to be swapped. One trick might just be to store an array of pointers to each pointer in the word first, then for each node pointed to by one of the pointers in the array, just swap the two aforementioned pointers.

3. This one's easy: just link the next pointer of the last node of the first word (this pointer will be originally NULL) to point to the first node of the second word, and set the previous pointer for the first node in the second word to point to the last node in the first word.

Required Data Structures/Constants/Variables

You must use the following doubly linked list struct:

```
typedef struct dllnode {
    char ch;
    struct dllnode* prev;
    struct dllnode* next;
} dllnode;
```

In your main function, please declare the following variables to store the grid:

```
int numRows;
int numCols;
dllnode*** strandsGrid;
```

The way to interpret this is that strandsGrid will be a two dimensional array of type dllnode*. Each of the dllnode*'s will point to a single dllnode struct. Those actual dllnode structs will contain pointers (previous and next) that will build the structure of each of the words.

There will be 8 possible directions of movement, using the following "codes":

NW = 0, N = 1, NE = 2, W = 3, E = 4, SW = 5, S = 6, SE = 7.

These codes correspond to the following two constant arrays and directions for the previous and next pointers, which you must use:

```
const int NUMDIR = 8;
const int NULLPTR = -1;
const int DR[NUMDIR] = {-1, -1, -1, 0, 0, 1, 1, 1};
const int DC[NUMDIR[ = {-1, 0, 1, -1, 1, -1, 0, 1};
```

The way to make sense of this is that moving NW on the grid is the same as moving up one row DR[0] = -1, and to the left by one column, DC[0] = -1. Essentially, the DR array is storing the relative north/south movement (also movement in rows) and the DC array is storing the relative west/east movement (also movement in cols).

The reason you will need to use these fixed arrays is that the input will refer to the link directions for both the previous letter and the next letter using the 0 through 7 codes provided above. If a pointer is NULL, then the code of -1 will be used.

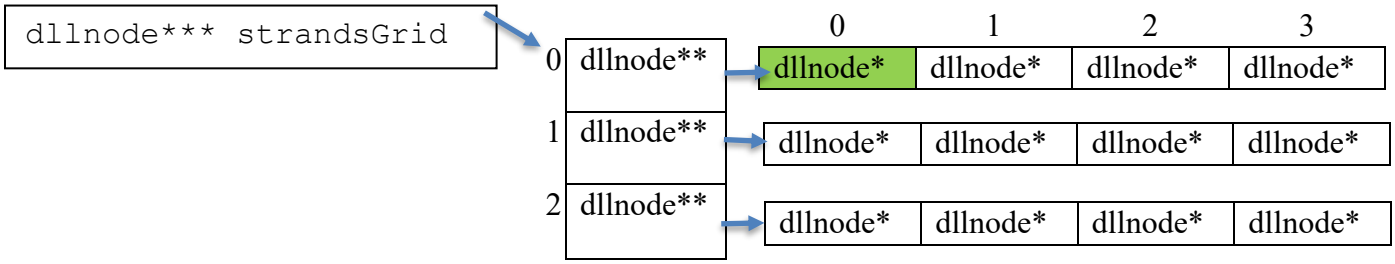
Here is a breakdown with the purpose of dllnode*:**

Our target is to create a 2D array of dllnode* with r rows and c columns. From our knowledge of dynamic memory allocation, we know that a 2D array is just like an array of arrays. So, we kind of need r number of arrays of dllnode*. So, we need r number of dllnode** to point those r number of arrays.

Assume, r = 3 and c = 4

So, first we need an array of dllnode** of size r. As we need to store the address of a dllnode**, we need to use dllnode*** type, which is our strandGrid in this case.

Now, for each *strandsGrid[i]*, we need to create an array of dllnode* of size c (in this example c=4)



If you write strandsGrid[0][0]->, you will be able to access the node pointed by the green colored node pointer shown in the above picture. Of-course, the node has not been created yet. You need to create the node while processing your inputs, and then you can access it.

Input

The first line of input contains two space separated integers, r ($2 \leq r \leq 100$), and c ($2 \leq c \leq 100$), the number of rows and columns, respectively, in the input grid.

The following r lines will each contain c lowercase letters. The i^{th} of these lines will indicate the contents of row **i** ($0 \leq i < r$), from left to right. The indexing to the grid will be 0-based, so the first character in each row is column 0, the second character in each row is column 1, etc.

The second set of r lines will each contain c space separated integers, each in between -1 and 7, inclusive, representing the direction for the next pointer from that letter to the next letter in the word it is part of. If this number is in between 0 and 7, inclusive, then a next letter exists, and it's in the relative direction specified previously by the direction codes. Namely, the DR/DC arrays store the relative movement to get to the next letter in this index location. If this number is -1, then this position in the grid represents the last letter in the word it forms in the Strands configuration.

The third set of r lines will each contain c space separated integers, each in between -1 and 7, inclusive, representing the direction for the previous pointer from that letter to the previous letter in the word it is part of. If this number is in between 0 and 7, inclusive, then a previous letter exists, and it's in the relative direction specified previously by the direction codes. If this number is -1, then this position in the grid represents the first letter in the word it forms in the Strands configuration.

The next line of input contains a single positive integer, q ($1 \leq q \leq 1000$), representing the number of queries that follow, each query on a single line.

There are three query types:

1. Print the word that the node in a particular grid square is part of.
2. Reverse the word that the node in a particular grid square is part of.
3. Given the position of the last letter in one word, and an adjacent position that is the starting letter in a different word, concatenate the word in the first position to the word in the second listed position.

For each query, the first integer, t ($1 \leq t \leq 3$), will represent the query type corresponding to the numeric list above.

If t is 1 or 2, then the line will contain 2 more space-separated integers, y ($0 \leq y < r$), and x ($0 \leq x < c$), respectively, representing the 0-based row number and 0-based column number of the grid square for the query.

If t is 3, then the line will contain 4 more space separated integers, y_1 ($0 \leq y_1 < r$), x_1 ($0 \leq x_1 < c$), y_2 ($0 \leq y_2 < r$), and x_2 ($0 \leq x_2 < c$), respectively, where y_1 and x_1 are the row and column numbers, of the last grid square in the first word for the query, and y_2 and x_2 are the row and column numbers, of the first grid square in the second word for the query. In addition, it will be guaranteed that these two positions are adjacent in the grid.

Output

For each query of type 1, on a line by itself, print out the word that grid position is part of. No output should be produced for queries of types 2 or 3.

Sample Input	Sample Output
8 6 cexgre omaniu lledet rqgarc euecoe pmiaul aoewrs phkroe 6 4 6 -1 4 -1 6 -1 3 1 3 0 4 4 6 2 5 1 -1 6 6 1 1 1 1 7 -1 4 6 1 1 7 4 -1 6 1 1 1 4 7 4 6 1 1 -1 3 3 -1 -1 -1 3 6 7 3 1 4 1 4 5 6 1 3 3 6 6 6 6 -1 1 2 -1 6 6 1 1 -1 3 6 6 6 0 3 1 -1 6 6 0 3 1 3 -1 -1 4 4 0 1 10 1 1 0 1 6 2 1 5 4 2 7 4 1 7 1 3 4 2 4 3 1 2 2 1 6 5 1 4 0 1 2 4	college homework course krowemoh collegecourse collegecourse paper reading

Implementation Requirements/Run Time Requirements

Note: This problem can be solved without a linked list, but the intention of the assignment is to practice linked lists, so please follow the following requirements:

1. **You must store the grid as previously described, initially creating a 2d array of pointers to dllnode, then linking the dllnodes to each other as described.**

2. The total run-time of your code should be O(qrc) for full credit, where q, r and c are the variables described in the input section.

3. **Do not use global variables at all.**

4. Your code must compile and execute on the Eustis system. The C compiler on this system is the one that the graders will be using to grade/evaluate your submissions.

5. Make sure your program is broken down into separate functions in a reasonable, modular design.

Deliverables

1. Please submit a source file `strands.c`, via Webcourses, for your solution to the problem.