CS 201, Spring 2024
Homework Assignment 4

Due: 23:59, May 17, 2024

In this homework, you will help a group of students find their way through a complex university building to reach their TA for an office hour. The building plan consists of cubicles of the same size where some cubicles are separated from each other by walls. Figure 1 illustrates an example building plan that has a grid structure. Each cell in this grid corresponds to a cubicle that is represented by its (row,column) coordinates. For example, the top-left cubicle is at (0,0) and the bottom-right cubicle is at (4,5). A student can go from one cubicle to a neighboring cubicle either horizontally or vertically. No diagonal movements are allowed and, obviously, a student cannot go through a wall.

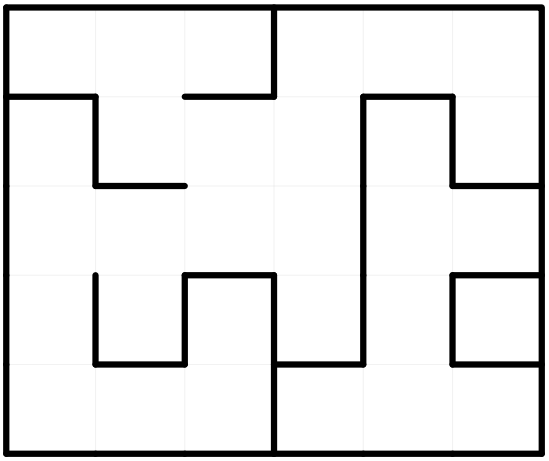

Figure 1: Sample building plan.

Your task is to find all paths from an origin cubicle to a target cubicle under various conditions. Your solution **MUST** utilize a non-recursive implementation of the exhaustive depth-first search algorithm, employing a **stack** for searching the desired paths. A student is allowed to pass through a particular cubicle only once. One example path for the sample building plan is illustrated in Figure 2.

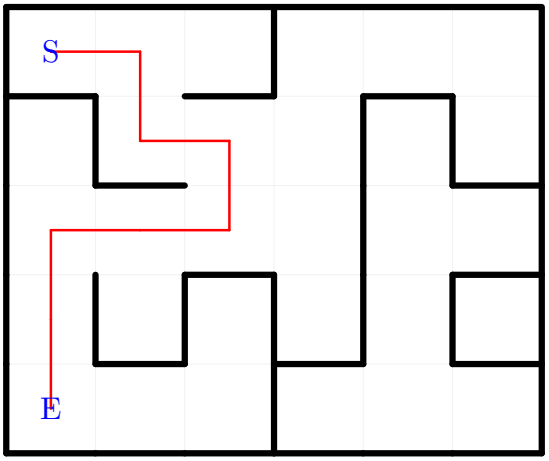

Figure 2: Example path for the sample building plan. The goal is to go from the origin cubicle (represented as S at coordinate (0,0)) to the target cubicle (represented as E at coordinate (4,0)).

We will provide you with a `buildingMapFile`. The `buildingMapFile` describes the building structure with the following encoding. Each cubicle is represented with one of the following numbers that denote if that cubicle has a wall on the left or at the bottom that separates it from neighboring cubicles:

- 0: cubicle has neither a left wall nor a bottom wall
- 1: cubicle has a left wall but no bottom wall
- 2: cubicle has no left wall but a bottom wall
- 3: cubicle has both a left wall and a bottom wall

Note that, we assume that the top row of cubicles all have top walls and the last column of cubicles all have right walls.

The `buildingMapFile` starts with the number of rows and the number of columns in the grid structure (two numbers separated by a space in the first line). Then, each line describes a row with each cubicle encoded as defined above (one number for each cubicle and no spaces in between). The file that corresponds to the sample building plan is as follows:

```
5 6
302120
130013
102012
131313
322322
```

Your solution must be implemented in a class called **BuildingMap**. Below is the required public part of the `BuildingMap` class. The interface for the class must be written in a file called **BuildingMap.h** and its implementation must be written in a file called **BuildingMap.cpp**. You can define additional public and private member functions and data members in this class. You can also define additional classes in your solution.

```
class BuildingMap {

public:
    BuildingMap(const string buildingMapFile);
    ~BuildingMap();

    void displayAllCubicles() const;
    void displayAdjacentCubicles(const int row, const int col) const;
    void displayBuildingMap() const;

    void findPaths(const int startRow, const int startCol, const int endRow, const int endCol);
    void findPaths(const int startRow, const int startCol, const int endRow, const int endCol,
                   const int avoidRow, const int avoidCol);
};
```

The member functions are defined as follows:

BuildingMap: Constructor. Reads the building description from the file `buildingMapFile` and stores the building map as a graph. The building graph will not change once the `BuildingMap` object is constructed by using the information provided in the given file. You can assume that the file is correctly formatted.

displayAllCubicles: Displays the coordinates of all cubicles in the building. The cubicles must be displayed as (row,column) coordinates separated by commas with each row displayed on a separate line.

displayAdjacentCubicles: Displays the coordinates of the cubicles adjacent to the given cubicle. A cubicle is considered to be adjacent to another cubicle if it is directly accessible from that cubicle (i.e., there is no wall in between). The adjacent cubicle coordinates must be displayed in a comma separated format in lexicographic order. The order is defined as follows:

$$(r_1, c_1) < (r_2, c_2) \text{ if and only if } r_1 < r_2 \text{ or } (r_1 = r_2 \text{ and } c_1 < c_2)$$

You can assume that the cubicle with the given coordinates exists in the building.

displayBuildingMap: Displays all adjacent cubicles for all cubicles in the building. In other words, this function displays the complete building graph. All cubicles are denoted with their (row,column) coordinates and must be displayed in lexicographic order.

findPaths: Finds and displays all paths from the start cubicle to the end cubicle. Uses the (row,column) coordinates to display the cubicles and "`->`" to display the connections. The paths must be displayed in lexicographic order.

You are also asked to implement an overloaded version of this function which gets the coordinates of a cubicle to be avoided. A TA is currently in a meeting in this cubicle so that cubicle must be avoided. Therefore, even if a path from the origin to the destination exists by passing through this cubicle, you must avoid it in your search algorithm. That is, it is not sufficient to find all paths that use this cubicle and then not simply print those. You need to avoid that cubicle and do not generate those paths. You can also assume that the given cubicle coordinates exist in the building.

Here is an example test program that uses this class and the corresponding output. We will use a similar program to test your solution so make sure that the name of the class is **BuildingMap**, its interface is in the file called **BuildingMap.h**, and the required functions are defined as shown above. Your implementation must use the **exact** format given in the example output to display the expected messages as the result of the defined functions.

Example test code:

```
#include "BuildingMap.h"

int main() {

    BuildingMap bm("building.txt");
    cout << endl;

    bm.displayAllCubicles();
    cout << endl;

    bm.displayAdjacentCubicles(1,1);
    cout << endl;

    bm.displayAdjacentCubicles(2,2);
    cout << endl;

    bm.displayAdjacentCubicles(3,5);
    cout << endl;

    bm.displayBuildingMap();
    cout << endl;

    bm.findPaths(0,0,4,0);
    cout << endl;

    bm.findPaths(0,0,4,0,1,3);
    cout << endl;
```

```
    bm.findPaths(0,0,4,0,2,1);
    cout << endl;

    bm.findPaths(3,1,2,4);

    return 0;
}
```

Output of the example test code:

```
5 rows and 6 columns have been read.

The cubicles in the building are:
(0,0),(0,1),(0,2),(0,3),(0,4),(0,5)
(1,0),(1,1),(1,2),(1,3),(1,4),(1,5)
(2,0),(2,1),(2,2),(2,3),(2,4),(2,5)
(3,0),(3,1),(3,2),(3,3),(3,4),(3,5)
(4,0),(4,1),(4,2),(4,3),(4,4),(4,5)

The cubicles adjacent to (1,1) are:
(1,1) -> (0,1),(1,2)

The cubicles adjacent to (2,2) are:
(2,2) -> (1,2),(2,1),(2,3)

The cubicles adjacent to (3,5) are:
(3,5) ->

The building map is as follows:
(0,0) -> (0,1)
(0,1) -> (0,0),(0,2),(1,1)
(0,2) -> (0,1)
(0,3) -> (0,4),(1,3)
(0,4) -> (0,3),(0,5)
(0,5) -> (0,4),(1,5)
(1,0) -> (2,0)
(1,1) -> (0,1),(1,2)
(1,2) -> (1,1),(1,3),(2,2)
(1,3) -> (0,3),(1,2),(2,3)
(1,4) -> (2,4)
(1,5) -> (0,5)
(2,0) -> (1,0),(2,1),(3,0)
(2,1) -> (2,0),(2,2),(3,1)
(2,2) -> (1,2),(2,1),(2,3)
(2,3) -> (1,3),(2,2),(3,3)
(2,4) -> (1,4),(2,5),(3,4)
(2,5) -> (2,4)
(3,0) -> (2,0),(4,0)
(3,1) -> (2,1)
(3,2) -> (4,2)
(3,3) -> (2,3)
(3,4) -> (2,4),(4,4)
(3,5) ->
(4,0) -> (3,0),(4,1)
(4,1) -> (4,0),(4,2)
(4,2) -> (3,2),(4,1)
(4,3) -> (4,4)
(4,4) -> (3,4),(4,3),(4,5)
(4,5) -> (4,4)
```

```
Paths from (0,0) to (4,0) are:
(0,0) -> (0,1) -> (1,1) -> (1,2) -> (1,3) -> (2,3) -> (2,2) -> (2,1) -> (2,0) -> (3,0) -> (4,0)
(0,0) -> (0,1) -> (1,1) -> (1,2) -> (2,2) -> (2,1) -> (2,0) -> (3,0) -> (4,0)

Paths from (0,0) to (4,0) avoiding (1,3) are:
(0,0) -> (0,1) -> (1,1) -> (1,2) -> (2,2) -> (2,1) -> (2,0) -> (3,0) -> (4,0)

Paths from (0,0) to (4,0) avoiding (2,1) are:
None

Paths from (3,1) to (2,4) are:
None
```

IMPORTANT NOTES:
Do not start your homework before reading these notes!!!

NOTES ABOUT IMPLEMENTATION:

1. You ARE NOT ALLOWED to modify the given parts of the header file. You MUST use the nonrecursive solution using a stack to implement the exhaustive depth-first search algorithm for finding the requested paths between the cubicles. You will get no points if you use any other algorithm for the solution of the search problem. You can use other data structures to help with your implementation but the main search algorithm must be implemented using a stack.
2. You MUST implement the stack and any additional container (e.g., array, list) by yourself. You ARE NOT ALLOWED to use the data structures and related functions in the C++ standard template library (STL) or any external library.
3. Moreover, you ARE NOT ALLOWED to use any global variables or any global functions.
4. Output message for each operation MUST match the format shown in the output of the example code.
5. Your code MUST NOT have any memory leaks. You will lose points if you have memory leaks in your program even though the outputs of the operations are correct. To detect memory leaks, you may want to use Valgrind which is available at `http://valgrind.org`.

NOTES ABOUT SUBMISSION:

1. In this assignment, you must have separate interface and implementation files (i.e., separate `.h` and `.cpp` files) for your class. Your class name MUST BE `BuildingMap` and your file names MUST BE `BuildingMap.h` and `BuildingMap.cpp`. Note that you may write additional class(es) in your solution.
2. The code (`main` function) given above is just an example. We will test your implementation using different scenarios, which will contain different function calls. Thus, do not test your implementation only by using this example code. We recommend you to write your own driver files to make extra tests. However, you MUST NOT submit these test codes (we will use our own test code). In other words, do not submit a file that contains a function called `main`.
3. You should put all of your `.h` and `.cpp` files into a folder and zip the folder (in this zip file, there should not be any file containing a `main` function). The name of this zip file should conform to the following name convention: secX-Firstname-Lastname-StudentID.zip where X is your section number. The submissions that do not obey these rules will not be graded. Please do not use Turkish letters in your file and folder names.
4. Make sure that each file that you submit (each and every file in the archive) contains your name, section, and student number at the top as comments.

5. You are free to write your programs in any environment (you may use Linux, Windows, MacOS, etc.). On the other hand, we will test your programs on "dijkstra.ug.bcc.bilkent.edu.tr" and we will expect your programs to compile and run on the dijkstra machine. If we could not get your program properly work on the dijkstra machine, you would lose a considerable amount of points. Thus, we recommend you to make sure that your program compiles and properly works on dijkstra.ug.bcc.bilkent.edu.tr before submitting your assignment.

6. This assignment is due by 23:59 on Friday, May 17, 2024. You should upload your work to Moodle before the deadline. No hardcopy submission is needed. The standard rules about late homework submissions apply. Please see the course home page for further discussion of the late homework policy.

7. We use an automated tool as well as manual inspection to check your submissions against plagiarism. Please see the course home page for further discussion of academic integrity and the honor code for programming courses in our department.

8. This homework will be graded by your TA **Aqsa Shabir** (aqsa.shabir@bilkent.edu.tr). Thus, you may ask your homework related questions directly to her. There will also be a forum on Moodle for questions.