Data Structures & Algorithms

Assignment Problem

Write a program that implements the path:

- Solve the maze using stack:

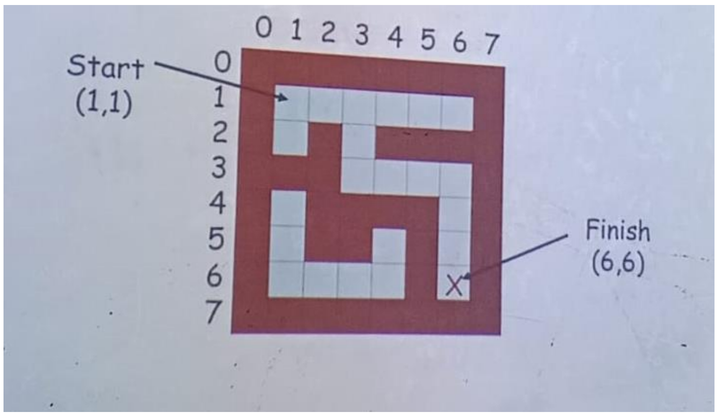


Fig: 01

Code:

The code begins by including necessary header files iostream and stack for input/output and stack functionality, respectively.

```
#include <iostream>

#include <stack>

using namespace std;
```

Declare Point Structure: Defines a structure named Point with two integer members x and y, representing coordinates.

```
struct Point {

    int x, y;

};
```

Functions:

1. `isSafetoMove( )` Checks whether a move to a given position in the maze is safe. It returns true if the move is within the bounds of the maze and leads to an empty cell.

```
bool isSafetoMove(int** maze, int M, int x, int y) {

    return (x >= 0 && x < M && y >= 0 && y < M && maze[x][y]
== 1);

}
```

2. `printMazeOutput( )` Prints the maze with some symbols representing different states of the cells:
   i. Represents walls or obstacles.
   ii. Represents empty cells.
   iii. Represents cells that are part of the path.
   iv. Represents the starting point.
   v. Represents the ending point.

```
void printMazeOutput(int** maze, int M) {
    for (int i = 0; i < M; ++i) {
        for (int j = 0; j < M; ++j) {
            if (maze[i][j] == 0)
                cout << " 0 ";
            else if (maze[i][j] == 1)
                cout << " 1 ";
            else if (maze[i][j] == 2)
                cout << " - ";
            else if (maze[i][j] == 3)
                cout << " S ";
            else if (maze[i][j] == 4)
                cout << " E ";
        }
        cout << endl;
    }
}
```

3. `printMazeIutput( )` Prints the initial state of the maze, similar to printMazeOutput, but without marking the start and end points.

```
void printMazeIutput(int** maze, int M) {
    for (int i = 0; i < M; ++i) {
        for (int j = 0; j < M; ++j) {
            if (maze[i][j] == 0)
                cout << " 0 ";
            else if (maze[i][j] == 1)
                cout << " 1 ";
            else if (maze[i][j] == 2)
                cout << " - ";
            else if (maze[i][j] == 3)
```

```
                    cout << ″ 1 ″;
                else if (maze[i][j] == 4)
                    cout << ″ 1 ″;
            }
            cout << endl;
        }
    }
```

4. `solveMaze( )` Solves the maze using depth-first search (DFS) algorithm with backtracking. It takes the maze, its size, start and end points as input arguments. It returns true if there exists a path from start to end, otherwise false. The function does the following:

- Checks if the start and end points are safe to move.
- Initializes a stack to store the current path.
- While the stack is not empty:
- Pops the top element from the stack.
- Marks the current position as part of the path.
- Checks if the current position is the end point. If yes, marks the start and end points, prints the solved maze, and returns true.
- If not at the end point, it checks neighboring cells (down and right) and pushes them onto the stack if they are safe to move.
- If the stack becomes empty and the end point is not reached, prints a message indicating no path exists and returns false.

```
bool solveMaze(int** maze, int M, Point start, Point end) {
    if (!isSafetoMove(maze, M, start.x, start.y) ||
!isSafetoMove(maze, M, end.x, end.y)) return false;


    stack<Point> solve;
    solve.push(start);


    while (!solve.empty()) {
        Point temp = solve.top();
        solve.pop();


        maze[temp.x][temp.y] = 2; // Mark as part of the path


        if (temp.x == end.x && temp.y == end.y) {
            maze[start.x][start.y] = 3; // Mark start
            maze[end.x][end.y] = 4; // Mark end
            printMazeOutput(maze,M);
            return true;
        }

        // Move down
        if (isSafetoMove(maze, M, temp.x + 1, temp.y)) {
            Point down = {temp.x + 1, temp.y};
            solve.push(down);
        }

        // Move right
        if (isSafetoMove(maze, M, temp.x, temp.y + 1)) {
            Point right = {temp.x, temp.y + 1};
            solve.push(right);
        }
    }

    cout << ″I am stuck in it !!!!!″ << endl;
    return false;
}
```

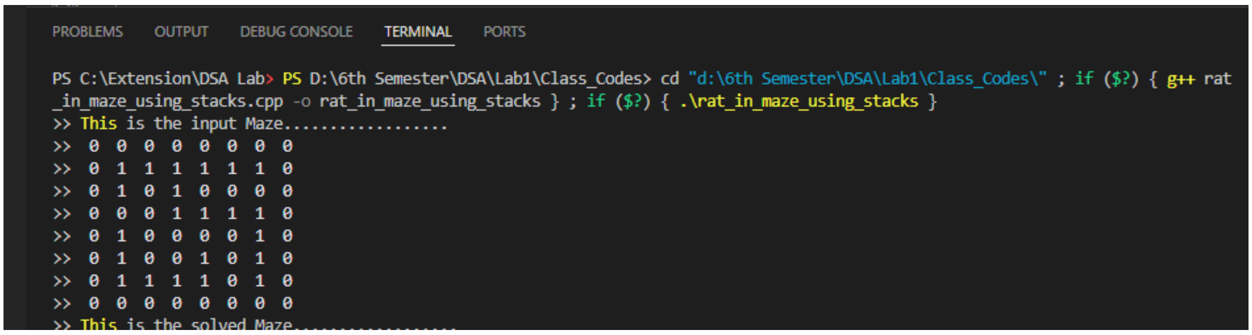
A main( ) function that:

- Defines the maze as a 2D array of integers.
- Dynamically allocates memory for the maze.
- Defines start and end points for the maze.
- Prints the initial maze using printMazeIutput.
- Solves the maze using solveMaze function.
- Prints a message if no path is found.
- Frees the dynamically allocated memory for the maze.

Code:

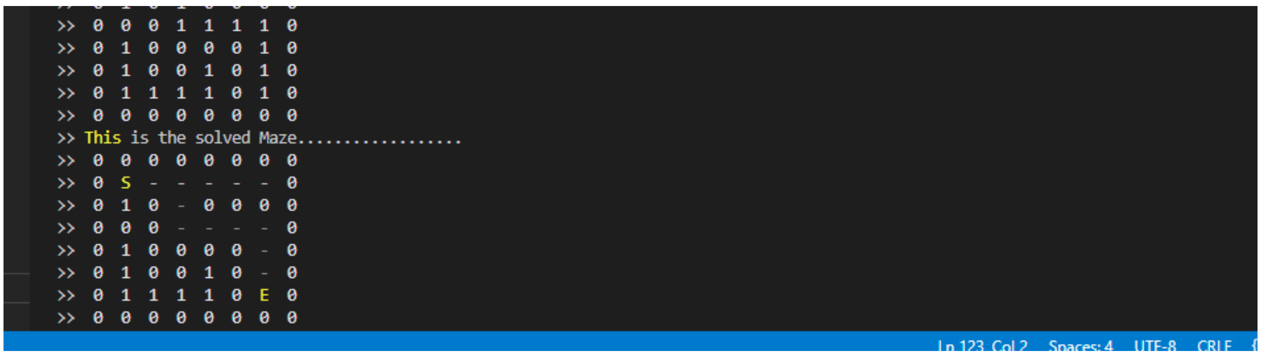
```
    int main () {
        const int size = 8;
        int mazeData[size][size] = {
            {0, 0, 0, 0, 0, 0, 0, 0},
            {0, 1, 1, 1, 1, 1, 1, 0},
            {0, 1, 0, 1, 0, 0, 0, 0},
            {0, 0, 0, 1, 1, 1, 1, 0},
            {0, 1, 0, 0, 0, 0, 1, 0},
            {0, 1, 0, 0, 1, 0, 1, 0},
            {0, 1, 1, 1, 1, 0, 1, 0},
            {0, 0, 0, 0, 0, 0, 0, 0}
        };

        int** problem = new int*[size];
        for (int i = 0; i < size; ++i) {
            problem[i] = new int[size];
            for (int j = 0; j < size; ++j) {
                problem[i][j] = mazeData[i][j];
            }
        }

        Point start = {1, 1}; // Start point
        Point end = {6, 6};   // End point
        cout<<"This is the input Maze.................."<<endl;
        printMazeIutput(problem, size);

        cout<<"This is the solved Maze.................."<<endl;
        if (!solveMaze(problem, size, start, end)) {
            cout << "There is no path to escape!" << endl;
        }

        // Free memory
        for (int i = 0; i < size; ++i) {
            delete[] problem[i];
        }
        delete[] problem;

        return 0;
    }
```

Output:



```
PS C:\Extension\DSA Lab> PS D:\6th Semester\DSA\Lab1\Class_Codes> cd "d:\6th Semester\DSA\Lab1\Class_Codes\" ; if ($?) { g++ rat
_in_maze_using_stacks.cpp -o rat_in_maze_using_stacks } ; if ($?) { .\rat_in_maze_using_stacks }
>> This is the input Maze.................
>>  0  0  0  0  0  0  0  0
>>  0  1  1  1  1  1  1  0
>>  0  1  0  1  0  0  0  0
>>  0  0  0  1  1  1  1  0
>>  0  1  0  0  0  0  1  0
>>  0  1  0  0  1  0  1  0
>>  0  1  1  1  1  0  1  0
>>  0  0  0  0  0  0  0  0
>> This is the solved Maze.................
```

Fig 02: code firstly print Input maze



```
>>  0  0  0  1  1  1  1  0
>>  0  1  0  0  0  0  1  0
>>  0  1  0  0  1  0  1  0
>>  0  1  1  1  1  0  1  0
>>  0  0  0  0  0  0  0  0
>> This is the solved Maze.................
>>  0  0  0  0  0  0  0  0
>>  0  S  -  -  -  -  -  0
>>  0  1  0  -  0  0  0  0
>>  0  0  0  -  -  -  -  0
>>  0  1  0  0  0  0  -  0
>>  0  1  0  0  1  0  -  0
>>  0  1  1  1  1  0  E  0
>>  0  0  0  0  0  0  0  0
```

Fig 03: After then a Solved maze

Methodology:

Initialization: The code initializes a 2D array representing the maze with some cells as walls (0) and other s as empty spaces (1). It also defines the start and end points of the maze.

Maze Solving Algorithm: The maze solving algorithm is implemented using depth-first search (DFS) with backtracking. It starts from the start point and explores neighbouring cells recursively until it reaches the end point or finds no possible path. During exploration, it marks the cells it visits as part of the path.

If it reaches the end point, it marks the start and end points, prints the solved maze, and returns true. If no path is found, it prints a message indicating the inability to escape and returns false.

Maze Printing Functions: Two functions are provided to print the maze : `printMazeIutput:` Prints the initial state of the maze without marking the start and end points. `printMazeOutput:` Prints the solved maze, marking the start, end, and path cells with appropriate symbols.

Safety Check Function: The `isSafetoMove` function ensures that a move to a given position in the maze is safe by checking if it falls within the maze boundaries and if the cell is empty.

Main Function:

The main function initializes the maze, defines start and end points, prints the initial maze state, and calls the maze solving function. It prints a message if no path is found and frees dynamically allocated memory for the maze. Overall, the code systematically solves a maze by traversing through its cells, marking the path, and backtracking when necessary, until the end point is reached or all possible paths are exhausted.