**MAZE PUZZLE GAME**

BASIC DESCRIPTION:

We have created a maze solving game using the C++ language. The code has been designed to randomly generate a maze using ‘#’ (wall) and ‘.’ (available path) Symbols of the desired size, or as per the difficulty level chosen by the user. It then allows the user to play a maze – solving game by navigating through the maze beginning at the entry point and ending at the exit, which lies at the bottom left corner of the maze. When the user either gives up or successfully completes the maze, the computer – generated solution is displayed on the screen. We now discuss the random maze generator and the maze solver. It works very well for mazes of large sizes (small console font size and full screen view is recommended for large mazes to avoid overlapping. This can be done by right – clicking and making changes in the console properties section).

RANDOM MAZE GENERATOR:

The first part of the code generates a maze of the desired size by employing the Recursive Back – Tracking algorithm. It generates ‘perfect’ mazes. These are mazes devoid of loops, closed circuits and inaccessible or detached paths. They have exactly one solution and only one path connecting any two points in the maze and are also called ‘simply – connected’ mazes.

The Recursive Back – Tracking algorithm generally uses the Stack concept, but we have executed it using Linked Lists as we are more acquainted with it and have learnt about it as part of one of the first – year courses. The maze was initialized to form a square matrix of the given dimensions containing only ‘#’ and then the paths were ‘carved’ into it using ‘.’ to create the final maze.

The basic concept is to start from one point and randomly grow the maze in size, by keeping track of the points already visited and making sure not to visit them again. We accomplished this by simultaneously updating two Linked Lists: one for remembering the path followed to reach the current location, and the other to keep track of all the visited locations.

While growing randomly, if the current cell has only previously visited locations as neighbors, it retraces its path until a point that has at least one un – visited neighbor is reached and continues growing from there by taking a different route. This process is repeated using a while loop which terminates when all positions (except the walls) have been visited. This results in the complete maze.

USER INPUT:

A function is created to take user input of direction and ‘@’ are filled accordingly showing the solved extent of puzzle. Switch case is used after taking user input of direction to do and display the proper changes. Screen is cleared after each input thus making the game user friendly. Finally the solution is printed.

MAZE SOLVER:

The solution to the maze is found using the ‘wall – follower’ or ‘right – hand’ or ‘left – hand’ rule, which basically says that simply – connected mazes can be successfully completed by keeping the right hand or left hand in contact with the walls throughout the maze – traversal and moving in the forward direction. This method, however, does not always generate the shortest path; it only guarantees successful completion of the simply – connected maze. It won’t work if you start from a location that contains a loop as you will go in circles.

It runs by identifying straight paths, bends, corners, dead – ends, etc. by checking for ‘#’ symbols (representing the wall) to its right, front and left; and making the next step, accordingly, following the left or right – hand rules (rules remain fixed once chosen in the beginning). This is repeated using a while loop, which terminates when the current location falls on one of the four borders. The print function has been designed to omit parts where the program entered and finally exited ‘dead – ends’ so as to print the shortest path solution. Since this is a simply – connected maze, both methods yield the same solution.