MAZE

BY : MOHAMED MURAD 72

Used Data Structures:

1)Stack :

is used to save the tracks in we took to get paths using DFS.

2)DFS :

is used as a first choice to get path from "S" to "E" in our Maze.

3)Queue :

is used to save the all directions of our tracks using BFS.

4)BFS :

-Is used as a second choice to get path from "S" to "E" in our Maze.

Comparisons bet. DFS & BFS :

DFS :

- give us a longer path in our maze.
- use a stack to save nodes.

BFS :

- give us a shorter path in our maze.
- use a queue to save nodes.

Details of comparisons in Code Explanation.

Code Explanation:

the implementation of Maze divided bet. 4 methods:

1 - SolveDFS.
2 - SolveBFS.

1 & 2 have the same task , they get File and extract grids of Maze from it saveing it in grid[][]

besides recognizing coordinates of "S" , number of rows which is saved in M and number of columns which is saved in N.

these to methods also passes throw some validation during extraction of Grids as :

- if there is no exit
- if theres more than entery
- if number of rows or columns more than is specified.

```
public int[][] solveDFS(final File maze) {
    // TODO Auto-generated method stub
    String getSymbol = new String();
    String[] getLines;
    String getLine = new String();
    int i = 0;

    LineNumberReader lNReader;
    try {
        lNReader = new LineNumberReader(new FileReader(maze));
    } catch (FileNotFoundException e2) {
        // TODO Auto-generated catch block
        throw new RuntimeException();
    }
    try {
        lNReader.skip(Long.MAX_VALUE);
    } catch (IOException e1) {
        throw new RuntimeException();
    }
    m = lNReader.getLineNumber();
    getLines = new String[m + 1];
    try {
        lNReader.close();
    } catch (IOException e2) {
        // TODO Auto-generated catch block
        throw new RuntimeException();
    }

    File file = maze;
    FileReader reader;
    try {
        reader = new FileReader(file);
    } catch (FileNotFoundException e1) {
        // TODO Auto-generated catch block
        throw new RuntimeException();
    }
```

```
            BufferedReader bReader = new BufferedReader(reader);
            try {

                while ((getLine = bReader.readLine()) != null) {
                    getLines[i++] = getLine;
                }
            } catch (IOException e) {
                throw new RuntimeException();
            }
            String firstLine = getLines[0];
            firstLine = firstLine + " ";
            for (i = 0; i < firstLine.length(); i++) {
                if (firstLine.charAt(i) != ' ') {
                    getSymbol = new String();
                    while (firstLine.charAt(i) != ' ') {
                        getSymbol += firstLine.charAt(i++);
                    }
                    if (m == 0) {
                        m = Integer.parseInt(getSymbol);
                    } else {
                        n = Integer.parseInt(getSymbol);
                    }

                }
            }
// making the grid
int sX = -1, sY = -1;
boolean exit = false;

grid = new Character[m][n];
String currentLine = new String();
int numberOfCells = 0;
for (i = 1; i < getLines.length; i++) {
    currentLine = new String(getLines[i]);
    currentLine = currentLine.replaceAll(" ", "");
    if (currentLine.contains("E")) {
        exit = true;
    }
    for (int j = 0; j < currentLine.length(); j++) {
        if (currentLine.charAt(j) == '.' || currentLine.charAt(j) == '#'
                || currentLine.charAt(j) == 'E'
                || currentLine.charAt(j) == 'S') {
            if (numberOfCells >= n) {
                throw new RuntimeException();
            }
            if (currentLine.charAt(j) == 'S') {
                if (sX == -1 && sY == -1) {
                    sX = i - 1;
                    sY = numberOfCells;
                } else {
                    throw new RuntimeException();
                }
            }
            grid[i - 1][numberOfCells++] = currentLine.charAt(j);
        } else {
            throw new RuntimeException();
        }

    }
    if (numberOfCells != n) {
        throw new RuntimeException();
    }
```

```
            if (numberOfCells != n) {
                throw new RuntimeException();
            }
            numberOfCells = 0;
        }
        if (!exit) {
            throw new RuntimeException();
        }

        return solveUsingDFS(sX, sY);
    }
```

if everything go ahead , we return the method which will get the path.

3 - solveUsingDFS.

-here we search for a path from S to E by followed the directions "up , right , down , left"

-here we use : (a stack : to save points of nodes

p Point : to save x and y in formal way.

Boolean visited[][] : to mark every node as visited or not.

flag : which will be valued as true as soon as we reach to "E")

simply in this code we loop `while (!flag && !mS.isEmpty()) {`

Then take peak value of the stack and search in all 4 dirctions

if there is no any valid direction we pop this point

but if we found a valid direction we push it using an if ladder

that's so that we insure existence of only the path in the stack which we pop it finally in the path .

```
public final int[][] solveUsingDFS(final int x, final int y) {
    boolean[][] visited = new boolean[m][n];

    MyStack mS = new MyStack();
    Point p = new Point();
    p.setLocation(x, y);
    mS.push(p);
    visited[x][y] = true;
    boolean flag = false;
    /**
     * xC and yC is the current coordinates.
     */
    int xC , yC;
    while (!flag && !mS.isEmpty()) {
        p = (Point) mS.peek();
        xC = (int) p.getX();
        yC = (int) p.getY();
        if (grid[xC][yC] == 'E') {
            flag = true;
        }
        if (!flag) {
            if (xC - 1 >= 0 && grid[xC - 1][yC] != '#'
                    && !visited[xC - 1][yC]) {
                Point pp = new Point(xC - 1, yC);
                mS.push(pp);
                visited[xC - 1][yC] = true;
            } else if (yC + 1 < n && grid[xC][yC + 1] != '#'
                    && !visited[xC][yC + 1]) {
                Point pp = new Point(xC, yC + 1);
                mS.push(pp);
                visited[xC][yC + 1] = true;
            } else if (xC + 1 < m && grid[xC + 1][yC] != '#'
                    && !visited[xC + 1][yC]) {
                Point pp = new Point(xC + 1, yC);
                mS.push(pp);
                visited[xC + 1][yC] = true;
            } else if (yC - 1 >= 0 && grid[xC][yC - 1] != '#'
                        && !visited[xC][yC - 1]) {
                    Point pp = new Point(xC, yC - 1);
                    mS.push(pp);
                    visited[xC][yC - 1] = true;
                } else {
                    mS.pop();
                }

            }
        }
        int size = mS.size();
        if (size == 0) {
            return null;
        }
        path = new int[size][2];
        for (size = size - 1; size >= 0; size--) {
            p = (Point) mS.pop();
            path[size][0] = (int) p.getX();
            path[size][1] = (int) p.getY();
        }
        return path;
    }
```

4)solveUsingBFS.

here we use the same codes with some – not slight – changes

we use Queue to save all nodes of all directions , every time we dequeue we check for all directions "is not sufficient a one as DFS"

we enqueue these all direction which we dequeue one after one until we reach "E" .

we used an array list as a necessary matter , because we don't know the size of valid nodes and we need to recall specific nodes in it , so it was a necessary to use ArrayList.

finally we extract the array list from back to insure not saveing irrelevant nodes (which its path has been cut from the queue)

and eventually return this path .

```
public final int[][] solveUsingBFS(final int x, final int y) {

    // basic declaration.
    boolean[][] visited = new boolean[m][n];
    Integer[][] dist = new Integer[m][n];
    MyQueue2 mQ = new MyQueue2();
    ArrayList<Point> outPuts = new ArrayList<Point>();
    int i = 0;
    boolean flag = false;
    boolean valid = false;
    Point p = new Point();
    Point e = new Point();
    /**
     * xC and yC is the current coordinates.
     */
    int xC , yC;
    // basic initialization.
    p.setLocation(x, y);
    mQ.enqueue(p);
    visited[x][y] = true;
    dist[x][y] = i;

    while (!flag) {
        try {
            p = (Point) mQ.dequeue();
        } catch (Exception e1) {
            // TODO Auto-generated catch block
            return null;
        }
        xC = (int) p.getX();
        yC = (int) p.getY();
        i = dist[xC][yC] + 1;

        if (!flag && (xC - 1 >= 0 && grid[xC - 1][yC] != '#'
                && !visited[xC - 1][yC])) {
            Point pp = new Point(xC - 1, yC);
            mQ.enqueue(pp);
            visited[xC - 1][yC] = true;
```

```
            dist[xC - 1][yC] = i;
            valid = true;
            if (grid[xC - 1][yC] == 'E') {
                flag = true;
                e.setLocation(xC - 1, yC);
            }
        }
        if (!flag && (yC + 1 < n && grid[xC][yC + 1] != '#'
                && !visited[xC][yC + 1])) {
            Point pp = new Point(xC, yC + 1);
            mQ.enqueue(pp);
            visited[xC][yC + 1] = true;
            dist[xC][yC + 1] = i;
            valid = true;
            if (grid[xC][yC + 1] == 'E') {
                flag = true;
                e.setLocation(xC, yC + 1);
            }
        }
        if (!flag && (xC + 1 < m && grid[xC + 1][yC] != '#'
                && !visited[xC + 1][yC])) {
            Point pp = new Point(xC + 1, yC);
            mQ.enqueue(pp);
            visited[xC + 1][yC] = true;
            dist[xC + 1][yC] = i;
            valid = true;
            if (grid[xC + 1][yC] == 'E') {
                flag = true;
                e.setLocation(xC + 1, yC);
            }
        }
             if (valid) {
                 outPuts.add(p);
             }
             valid = false;
             if (flag) {
                 outPuts.add(e);
             }

         }
         int size = dist[(int) e.getX()][(int) e.getY()] + 1;
         path = new int[size][2];
         int counter = size - 1;

         for (i = outPuts.size() - 1; i >= 0; i--) {
             xC = (int) outPuts.get(i).getX();
             yC = (int) outPuts.get(i).getY();
             if (counter == dist[xC][yC]) {
                 path[counter][0] = xC;
                 path[counter--][1] = yC;
             }

         }
         return path;

    }
```

Sample Runs:

1)

```
6 6
# # # E . .
# # . . . #
. . . . . .
# . . # # .
. . # . . #
. . . . S .
```

DFS

```
5 4
4 4
4 3
5 3
5 2
5 1
4 1
3 1
2 1
2 2
1 2
1 3
0 3
```

BFS

```
5 4
5 3
5 2
5 1
4 1
3 1
2 1
2 2
2 3
1 3
0 3
```

2)

```
6 6
# # # E . .
# # . . . #
. . . . . .
# . . # # .
S . # . . #
. . . . S .
```

```
Exception in thread "main" java.lang.RuntimeException
        at eg.edu.alexu.csd.datastructure.maze.cs72.MyMaze.solveDFS(MyMaze.java:231)
        at eg.edu.alexu.csd.datastructure.maze.cs72.MyMaze.main(MyMaze.java:447)
```

3)

```
6 6
.  .  .  #  E  .
.  #  .  .  #  .
.  .  #  .  .  .
#  .  .  #  #  .
.  .  #  .  .  #
.  .  .  .  S  .
```

DFS

```
5 4
4 4
4 3
5 3
5 2
5 1
4 1
3 1
2 1
2 0
1 0
0 0
0 1
0 2
1 2
1 3
2 3
2 4
2 5
1 5
0 5
0 4
```

BFS

```
5 4
5 3
5 2
5 1
4 1
3 1
2 1
2 0
1 0
0 0
0 1
0 2
1 2
1 3
2 3
2 4
2 5
1 5
0 5
0 4
```