2020년도 2학기 컴퓨터공학설계및실험Ⅰ

13주차 미로(Maze) 2주차 결과보고서

20161663 허재성

1. 실습 목적

본 실험은 1주차 미로 만들기 프로젝트의 결과물인 ~.maz 파일을 읽어서 이를 윈도우 화면상에 표시함을 목적으로 한다. 실험목적인 GUI(Graphical User Interface) 구축과 동시에 3주차에서 행할 프로젝트를 위하여 학생들은 창의적이고 효율적인 자료구조를 설계하여야 한다. GUI를 용이하게 설계하기 위하여 OpenFrameWorks를 사용하여 구현하도록 한다.

2. 실습 구현 내용

12주차에서 생성한 완전 미로를 저장하기 위해서는 char형 2차원 배열이 필요하다. 미로의 크기를 미리 알 수 없으므로 2차원 배열을 동적 할당을 해야 하며 이를 위해 동적 할당한 배열을 가리킬 char형 이중 포인터 mazeArray를 소스파일 ofApp.cpp에 전역 변수로 선언한다.

또한 입력받은 미로의 열, 행 크기를 나타내는 변수 MAZE\_WIDTH, MAZE\_HEIGHT와 미로의 한 행의 방의 개수, 한 열의 방의 개수를 나타내는 변수 WIDTH, HEIGHT를 전역 변수로 선언한다.

|  |
| --- |
| char\*\* mazeArray = NULL;  int WIDTH, HEIGHT;  int MAZE\_WIDTH = 0, MAZE\_HEIGHT = 0; |

미로를 읽은 후 미로의 방을 나타내는 정점을 구현한 구조체 Vertex와 미로의 벽을 구현한 구조체 Wall을 헤더파일 ofApp.h에 선언한다.

|  |
| --- |
| typedef struct \_Vertex {  int name;  float x, y;  bool visited;  \_Vertex \*up, \*down, \*left, \*right;  } Vertex; |

Vertex 구조체는 위와 같다. name은 방을 구분하기 위한 방 번호로 1~WIDTH\*HEIGHT까지 방의 개수만큼 존재한다. 모든 방은 정사각형 모양이라 가정하며 x, y는 방의 중심의 좌표이다.

bool형 변수 visited는 미로 찾기를 위한 DFS 및, BFS를 구현했을 때, DFS와 BFS 시 정점을 방문한 적이 있는지 확인하기 위한 변수이다.

마지막으로 Vertex 포인터 up, down, left, right는 다른 Vertex를 가리킬 수 있는 포인터 변수로 각각 미로에서 같은 행의 위의 방, 아랫방, 같은 열의 왼쪽 방, 오른쪽 방을 나타내는 정점 Vertex 구조체를 가리킬 수 있다. 벽이 존재할 경우 벽을 사이에 둔 두 방은 연결되지 않으므로 해당 포인터는 NULL로 설정하고 벽이 존재하지 않을 경우, 해당 방을 나타내는 Vertex 구조체의 주소를 변수에 저장한다.

|  |
| --- |
| typedef struct \_Wall {  float x1, y1;  float x2, y2;  } Wall; |

Wall 구조체는 위와 같다. 벽은 화면에 두 점을 잇는 선분으로 표현된다. x1, y1은 시작점의 x, y 좌표이고 x2, y2는 끝점의 x, y 좌표이다.

Vertex 구조체들을 저장하는 행이 HEIGHT, 열이 WIDTH인 2차원 배열을 동적 할당하기 위한 Vertex형 이중 포인터 vertex와 Wall 구조체를 저장하기 위한 1차원 배열을 동적 할당하기 위한 Wall형 단일 포인터 wall을 ofApp.cpp에 전역적으로 선언한다. 또한 Vertex 구조체를 노드로 가지는 트리를 만들기 위해 트리의 노드를 가리키는 단일 포인터 변수 root를 선언한다.

|  |
| --- |
| Vertex\*\* vertex = NULL;  Wall\* wall = NULL;  Vertex\* root = NULL; |

방의 번호 Vnum과 벽의 번호 Wnum, 그리고 미로를 그릴 때 방의 크기를 결정하는 정수형 변수 r을 ofApp.cpp에 선언한다. r의 경우 예를 들어 r=10일 때, 정사각형 방은 한 변의 길이가 20이 되며 벽의 길이도 20이 된다.

|  |
| --- |
| int Vnum = 0;  int Wnum = 0;  int r = 20; // 적당한 크기 결정 |

.maz 파일을 읽고 미로를 화면에 그리고 적절한 자료구조에 저장하기 위해 다음과 같은 함수를 구현하였다.

Vertex 구조체 관련 함수

(1) Vertex makeVertex(int roomNum, int i, int j)

Vertex 변수 v를 선언 후 방의 이름 v,name을 roomNum, 방의 중심의 좌표 v.x, v.y를 i, j로 설정한다. 아직 해당 방에 방문한 적이 없으므로 v.visited를 false로 설정한다. 또한 아직 주변 방과의 연결 상태를 알 수 없으므로 v.up, v.down, v.left, v.right를 모두 NULL로 설정한다. V의 변수들을 초기화 후 v를 반환한다.

방의 이름으로 전역 변수로 선언된 Vnum을 이용한다.

input : 방의 이름 roomNum, 방의 중심의 x, y 좌표 i, j

output : 방을 나타내는 Vertex 구조체 v

Wall 구조체 관련 함수

(1) Wall makeWall(float x1, float y1, float x2, float y2)

Wall 변수 w를 선언 후 벽을 나타내는 선분의 시작점, 끝점 좌표 w.x1, w.y1, w.x2, w.y2를 각각 x1, y1, x2, y2로 설정 후 w를 반환한다.

input : 벽을 나타내는 선분의 시작 점의 좌표 x1, y1, 끝 점의 좌표 x2, y2

output : 벽을 나타내는 Wall 구조체 w

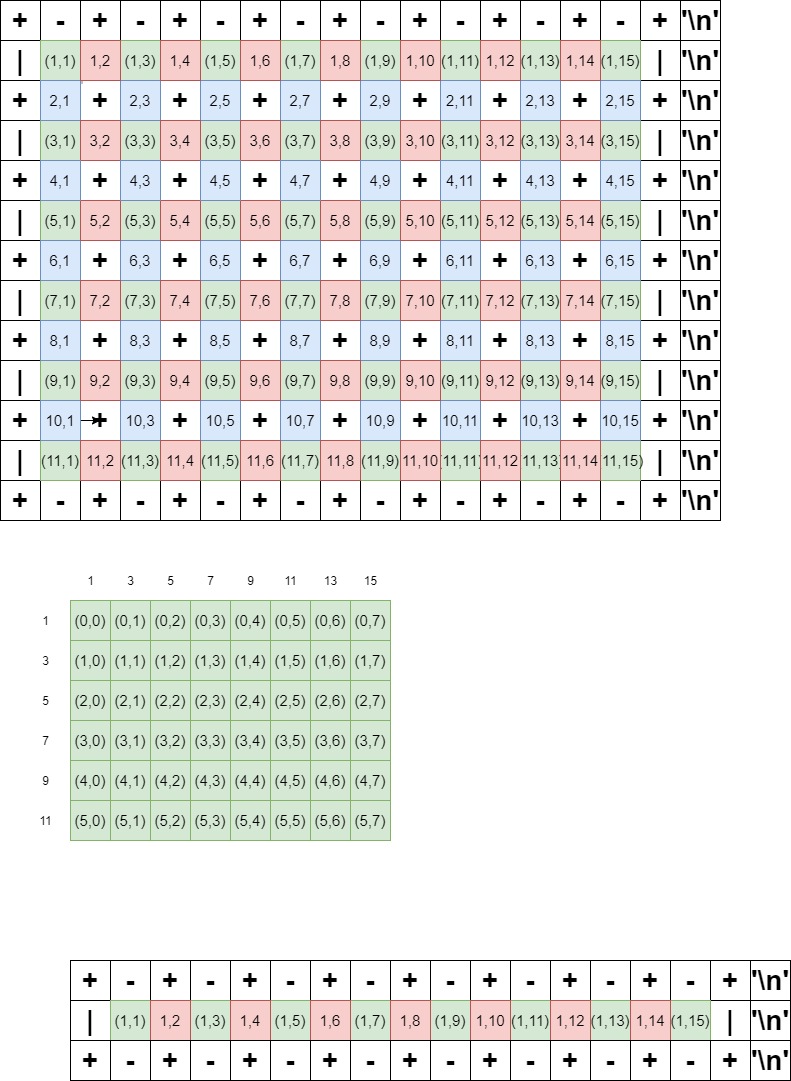
ofApp 클래스에 추가한 함수

(1) bool ofApp::readFile(const char\* filename)

12주차에서 생성한 .maz 파일을 읽어와 미로를 mazeArray 배열에 저장하고, 미로의 방의 정보들을 동적 할당된 Vertex 배열 vertex에 저장하며, 미로의 벽의 정보를 동적 할당된 Wall 배열 wall에 저장한다. 또한 Vertex 배열에 저장된 Vertex 구조체들을 트리 형태로 구성한다.

먼저 파일을 읽기 전에 이전에 할당된 메모리를 해제하기 위해 freeMemory 함수를 호출한다.

미로의 크기를 알아내기 위해 filename이라는 maz 파일을 읽기 모드로 연다. 하나의 행에 방이 8개, 하나의 열에 방이 6개인 8\*6 미로를 예로 들어 설명한다.



.maz 파일에 미로는 위와 같이 저장되어 있다. 흰색 칸은 미로의 모서리 ‘+’와 외곽 벽 ‘-‘, ‘|’를 저장하는 공간이고 초록색 칸은 방이 존재하는 칸이다. 빨간색 칸은 같은 행의 방 사이의 세로 벽 ‘|’이 존재할 수 있는 칸이고, 파란색 칸은 같은 열의 방 사이의 가로 벽 ‘-‘이 존재할 수 있는 칸이다. 미로 파일을 열어 줄 바꿈 문자 ‘\n’이 나타날 때까지 읽어 처음 ‘\n’까지의 문자의 개수를 계산 후 MAZE\_WIDTH에 저장한다. 위의 경우 MAZE\_WIDTH는 18이다.

또한 파일에 저장된 전체 문자의 개수를 세서 length에 저장하고 length를 MAZE\_WIDTH로 나누어 주면 MAZE\_HEIGHT를 알아낼 수 있다. 위의 경우 13개이다.

방의 가로, 세로 개수(WIDTH, HEIGHT)와 미로의 가로, 세로(MAZE\_WIDTH, MAZE\_HEIGHT)는 다음과 같은 관계가 성립한다.

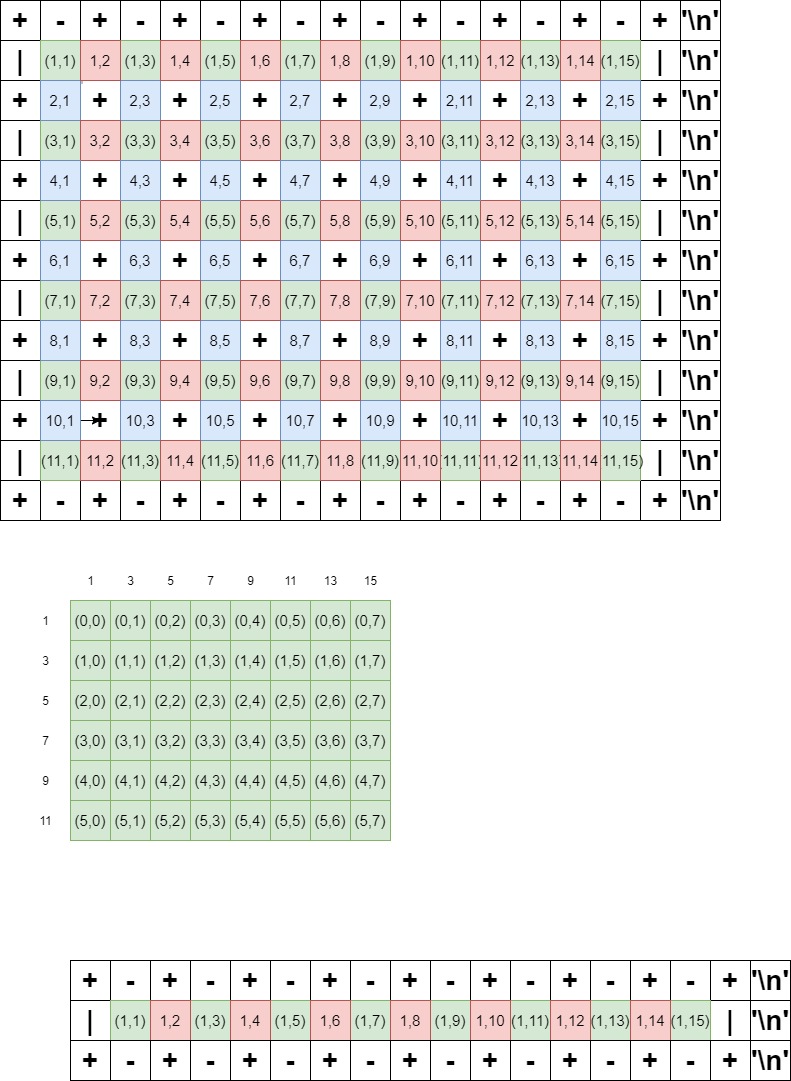
|  |
| --- |
| MAZE\_WIDTH = 2 \* WIDTH + 2;  MAZE\_HEIGHT = 2 \* HEIGHT + 1; |

위의 관계식을 이용해 WIDTH와 HEIGHT를 구할 수 있다. 위의 그림에서는 각각 8, 6이다. 필요한 변수의 크기를 구한 후 일단 파일을 닫아 준다.

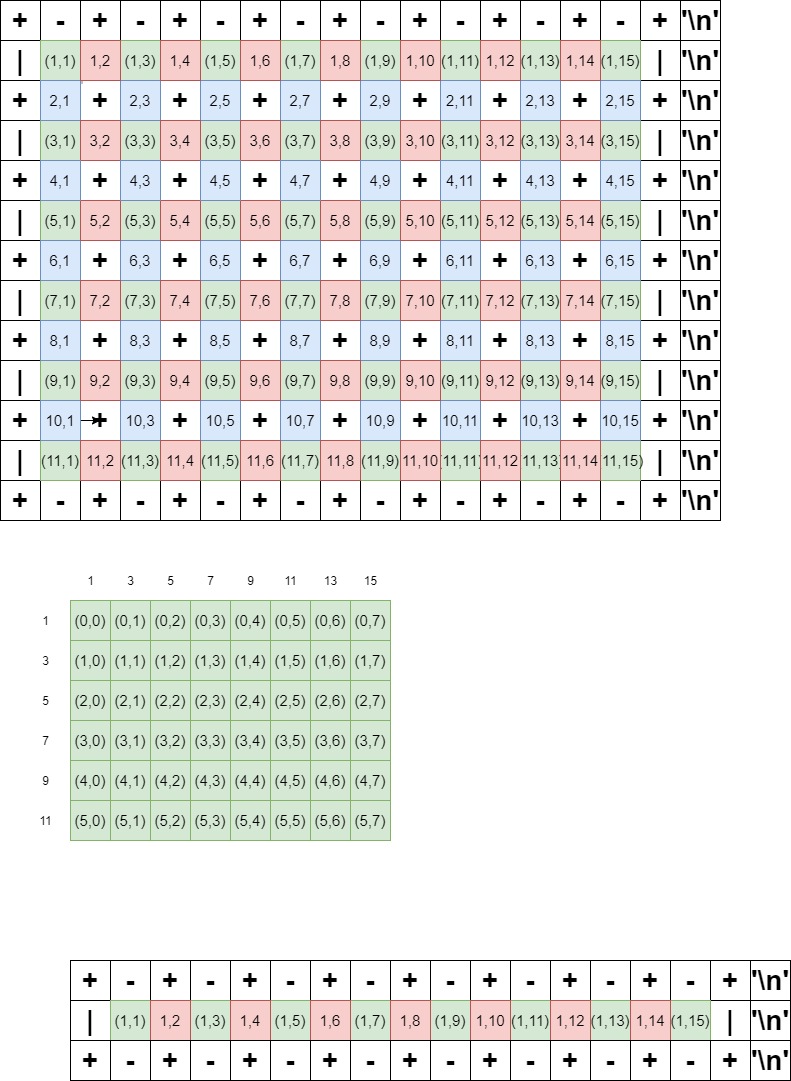
방의 개수가 WIDTH\*HEIGHT일 경우 미로에 존재 가능한 외곽벽을 포함한 최대 벽의 개수는 방들사이 벽이 하나도 존재하지 않을 때, 2\*WIDTH\*HEIGHT + WIDTH + HEIGHT이다. 입력 미로는 완전 미로이므로 미로를 나타내는 그래프는 그래프의 모든 정점을 잇는 신장 트리(Spanning Tree)가 되며 그래프의 정점의 개수는 WIDTH\*HEIGHT이므로 신장 트리의 간선의 개수는 WIDTH\*HEIGHT-1개가 된다. 미로를 나타내는 그래프에서 간선의 개수는 미로에서 벽이 지워진 개수와 같으므로 완전 미로에서 외곽벽을 포함한 벽의 개수는 2\*WIDTH\*HEIGHT + WIDTH + HEIGHT에서 WIDTH\*HEIGHT-1를 뺀 값인 WIDTH\*HEIGHT + WIDTH + HEIGHT + 1이다. 해당 값을 벽의 최대 개수 maxWall에 저장하고 Wall 구조체를 저장할 배열 wall을 동적할당한다.

방의 개수는 WIDTH \*HEIGHT이므로 해당 개수만큼 방을 나타내는 정점 Vertex 구조체를 저장할 2차원 배열 vertex를 동적 할당한다.

또한 미로를 저장할 MAZE\_HEIGHT\*MAZE\_WIDTH의 크기를 가지는 char형 2차원 배열 mazeArray를 동적 할당한다.



모든 동적 할당이 끝난 후 maz 파일을 다시 한번 읽기 모드로 열고 파일의 미로 정보를 mazeArray에 저장한다. 미로의 정보를 읽을 때 방일 경우(행 인덱스 i, 열 인덱스 j가 모두 홀수이고 mazeArray[i][j]가 ‘ ‘일 때 방이다.) 해당 방이 Vertex 구조체를 저장할 2차원 배열 vertex에 저장될 인덱스 newI, newJ를 구한다.



방의 인덱스가 i, j일 때 대응하는 2차원 배열 vertex에서의 인덱스 newI, newJ는 각각 (i-1)/2, (j-1)/2이다. makeVertex 함수를 호출하여 방 번호와 방의 중심의 좌표를 건내 주어 정점 v를 만든다.

Vnum은 처음에 0으로 설정되어 있으므로 makeVertex를 호출하기 전에 1씩 증가시키고 makeVertex 함수에 전달해 준다.

만약 j가 1일 경우 해당 방은 가장 왼쪽 열에 존재하므로 연결될 왼쪽 방이 없으므로 v.left=NULL로 설정한다. 반대로 j = 2 \* WIDTH-1일 경우 방이 가장 오른쪽 열에 존재하므로 v.right=NULL로 설정한다.

비슷한 방법으로 i=1일 경우와 i=2 \*HEIGHT-1일 경우 각각 방이 가장 위쪽 열과 가장 아래쪽 열에 존재하므로 v.up = NULL, v.down=NULL로 설정한다.

Vertex 구조체 v를 2차원 배열 vertex의 대응되는 인덱스 vertex[newI][newJ]에 저장한다.

미로의 정보를 읽을 때 벽일 경우(i가 홀수이고 j는 짝수이며 공백 ‘ ‘이 아니라 ‘|’일 경우, i가 짝수이고 j가 홀수이며 공백 ‘ ‘나 ‘\n’이 아니고 ‘-‘일 경우) Wall 구조체 배열 wall에 저장해준다. Wnum은 벽을 저장할 인덱스 역할을 한다. Wnum은 처음에 0이므로 wall[Wnum]에 makeWall 함수를 호출해 생성한 벽을 저장 후 Wnum 변수를 1 증가시켜 준다.

Vertex 2차원 배열 vertex를 완성 후 makeVertexTree 함수를 호출하여 Vertex 구조체를 노드로 가지는 트리를 만든다.

Wall 배열에 벽의 정보가 저장되었으므로 벽을 그릴 수 있게 되었다. draw\_flag를 true로 설정하여 draw 함수에서 벽을 그릴 수 있게 한 후, 파일을 닫고 true를 반환한다.

만약 읽으려는 파일을 여는데 실패할 경우 false를 반환한다.

파일을 두 번 열어 각각 미로 정보를 읽으므로 시간 복잡도는 각각 O(MAZE\_HEIGHT \*MAZE\_WIDTH)이다. MAZE\_HEIGTH = 2 \* HEIGHT + 1, MAZE\_WIDTH = 2 \* WIDTH + 2이므로 시간 복잡도는 각각 O(HEIGHT\*WIDTH)이다. readFile에서 호출하는 makeVertexTree 함수의 시간 복잡도 또한 O(HEIGHT\*WIDTH)이다. 따라서 readFile의 전체 시간 복잡도는 O(HEIGHT\*WIDTH)이다.

mazeArray의 공간 복잡도는 O(MAZE\_HEIGHT\*MAZE\_WIDTH)이므로 O(HEIGHT\*WIDTH)이다. Vertex의 공간 복잡도는 O(HEIGHT\*WIDTH)이고 wall의 공간 복잡도는 O(maxWall) = O(WIDTH\*HEIGHT + WIDTH + HEIGHT + 1) = O(HEIGHT\*WIDTH)이다. 따라서 전체 공간 복잡도는 O(HEIGHT\*WIDTH)이다.

input : 읽어오려는 미로 파일의 이름(확장자 .maz 포함)을 저장한 문자열 filename

output : 없음

(2) void ofApp::freeMemory()

readFile에서 동적 할당된 메모리들을 해제하는 함수이다. ‘Q’ 버튼을 눌러 종료될 때, 또는 ‘L’ 버튼을 눌러 새로운 미로를 그릴 때 이전에 할당된 메모리를 해제하기 위해 호출된다. 만약 freeMemory 함수가 처음 호출되었을 경우, 해제할 메모리가 없으므로 아무 일도 하지 않고 함수를 종료한다. 이를 위해 ofApp 클래스의 멤버 변수 start\_flag를 이용한다. start\_flag는 false로 설정되어 있다. start\_flag가 false일 경우 해제할 메모리가 없다는 의미이므로 start\_flag를 true로 설정하고 함수를 종료한다. 다음에 freeMemory 함수가 호출되었을 때에는 동적 할당된 메모리가 존재하므로 동적 할당된 메모리를 해제한다. 동적 할당된 배열 vertex, wall, mazeArray,를 모두 해제 해준다.

마지막으로 전역 변수 MAZE\_HEIGHT, MAZE\_WIDTH, Vnum, Wnum를 0으로, root를 NULL로 재설정하여 다음 파일을 열 때 오류가 발생하지 않도록 한다.

동적 할당된 2차원 배열을 해제하기 위한 시간 복잡도는 O(HEIGHT)이고, mazeArray를 해제하기 위한 시간 복잡도는 O(MAZE\_HEIGHT)이다. MAZE\_HEIGHT = 2\*HEIGHT + 1이므로 마찬가지로 시간 복잡도는 O(HEIGHT)이다. 따라서 freeMemory의 시간 복잡도는 O(HEIGHT)이다.

input : 없음

output : 없음

(3) void ofApp::makeVertexTree()

2차원 배열 vertex에 저장된 구조체들을 이용하여 Vertex 트리를 만든다. 이 때 트리의 루트 노드는 미로에서 첫 번째 방(왼쪽 위의 방)을 나타내는 정점 vertex[0][0]이 된다.

mazeArray에 저장된 미로의 정보 중 외곽 벽과 ‘\n’을 제외한 정보들을 탐색한다. 이 때 인덱스 i, j가 세로 벽 ‘|’이 존재할 수 있는 인덱스(i가 홀수, j가 짝수)일 경우 해당 위치의 왼쪽 방의 vertex 배열에서의 인덱스 prevX, Y, 벽의 오른쪽 방의 vertex 배열에서의 인덱스 nextX, Y는 다음과 같다.

|  |
| --- |
| prevX = (j-2)/2;  nextX = j/2;  Y = (i-1)/2; |

mazeArray[i][j]가 ‘|’일 경우 벽이 존재하므로 해당 벽의 좌우 방은 서로 연결될 수 없다. 따라서 다음과 같이 두 방의 정점 노드가 서로 연결되지 않도록 한다.

|  |
| --- |
| vertex[Y][prevX].right = NULL;  vertex[Y][nextX].left = NULL; |

반대로 mazeArray[i][j]가 공백 ‘ ‘일 경우 벽이 존재하지 않으므로 해당 위치의 좌우 방은 연결될 수 있으므로 다음과 같이 연결한다.

|  |
| --- |
| vertex[Y][prevX].right = &vertex[Y][nextX];  vertex[Y][nextX].left = &vertex[Y][prevX]; |

비슷한 방법으로 인덱스 i, j가 가로 벽 ‘-’이 존재할 수 있는 인덱스(i가 짝수, j가 홀수)일 경우 해당 위치의 왼쪽 방의 vertex 배열에서의 인덱스 X, prevY, 벽의 오른쪽 방의 vertex 배열에서의 인덱스 X, nextY는 다음과 같다.

|  |
| --- |
| X = (j-1)/2;  prevY = (i-2)/2;  nextY = i/2; |

mazeArray[i][j]가 ‘-’일 경우 벽이 존재하므로 해당 벽의 위 아래 방은 서로 연결될 수 없다. 따라서 다음과 같이 두 방의 정점 노드가 서로 연결되지 않도록 한다.

|  |
| --- |
| vertex[prevY][X].down = NULL;  vertex[nextY][X].up = NULL; |

반대로 mazeArray[i][j]가 공백 ‘ ‘일 경우 벽이 존재하지 않으므로 해당 위치의 위 아래 방은 연결될 수 있으므로 다음과 같이 연결한다.

|  |
| --- |
| vertex[prevY][X].down = &vertex[nextY][ X];  vertex[nextY][X].up = &vertex[prevY][ X]; |

마지막으로 전역 변수 root에 첫 번째 방의 Vertex 구조체의 주소를 대입한다.

|  |
| --- |
| root = &vertex[1][1]; |

mazeArray에서 외곽벽과 ‘\n’ 문자를 제외한 배열의 크기는 (2\*HEIGHT-1)\*(2\*WIDTH-1)이므로 반복의 횟수는 O(HEIGHT\*WIDTH)이다. 따라서 시간 복잡도는 O(HEIGHT\*WIDTH)이다.

input : 없음

output : 없음

ofApp 클래스에서 수정한 함수

(1) void ofApp::setup()

프로그램 실행 시 창의 이름을 바꿔주고 창의 색을 흰색으로 설정한다. 미로를 그릴 선의 굵기를 5로 설정한다. start\_flag와 draw\_flag를 false로 초기화한다.

(2) void ofApp::draw()

draw\_flag가 false일 경우 아무 일도 하지 않는다. readFile에서 파일을 읽어와 벽의 정보를 저장한 wall 구조체 배열이 만들어지면 draw\_flag가 true가 되어 이 때 wall에 저장된 벽의 정보를 이용해 그린다.

벽을 나타내는 선분의 색을 ofSetColor를 이용해 설정하고 ofDrawLine 함수를 이용해 선분을 그려 미로를 표현한다.

(3) void ofApp::keyPressed(int key)

key 입력으로 ‘l’ 또는 ‘L’이 들어왔을 경우 readFile 함수를 호출해 미로 파일을 읽어 미로를 그릴 수 있게 한다.

입력으로 ‘q’ 또는 ‘Q’가 들어왔을 경우 freeMemory 함수를 호출해 동적 할당된 메모리를 해제 후 프로그램을 종료한다.

draw\_flag가 true일 때, 즉 이미 미로가 그려진 상태에서 대소문자 관계 없이 ‘D’ 또는 ‘B’를 입력하면 각각 DFS와 BFS를 이용해 미로를 탐색한다. DFS와 BFS는 14주차에서 구현하므로 13주차에서는 콘솔에 버튼이 눌렸음을 알리는 메시지만 출력한다.

3. 실습 환경

OS : Windows 10

CPU : Intel® Core(TM) i3-5005U CPU @ 2.00GHz

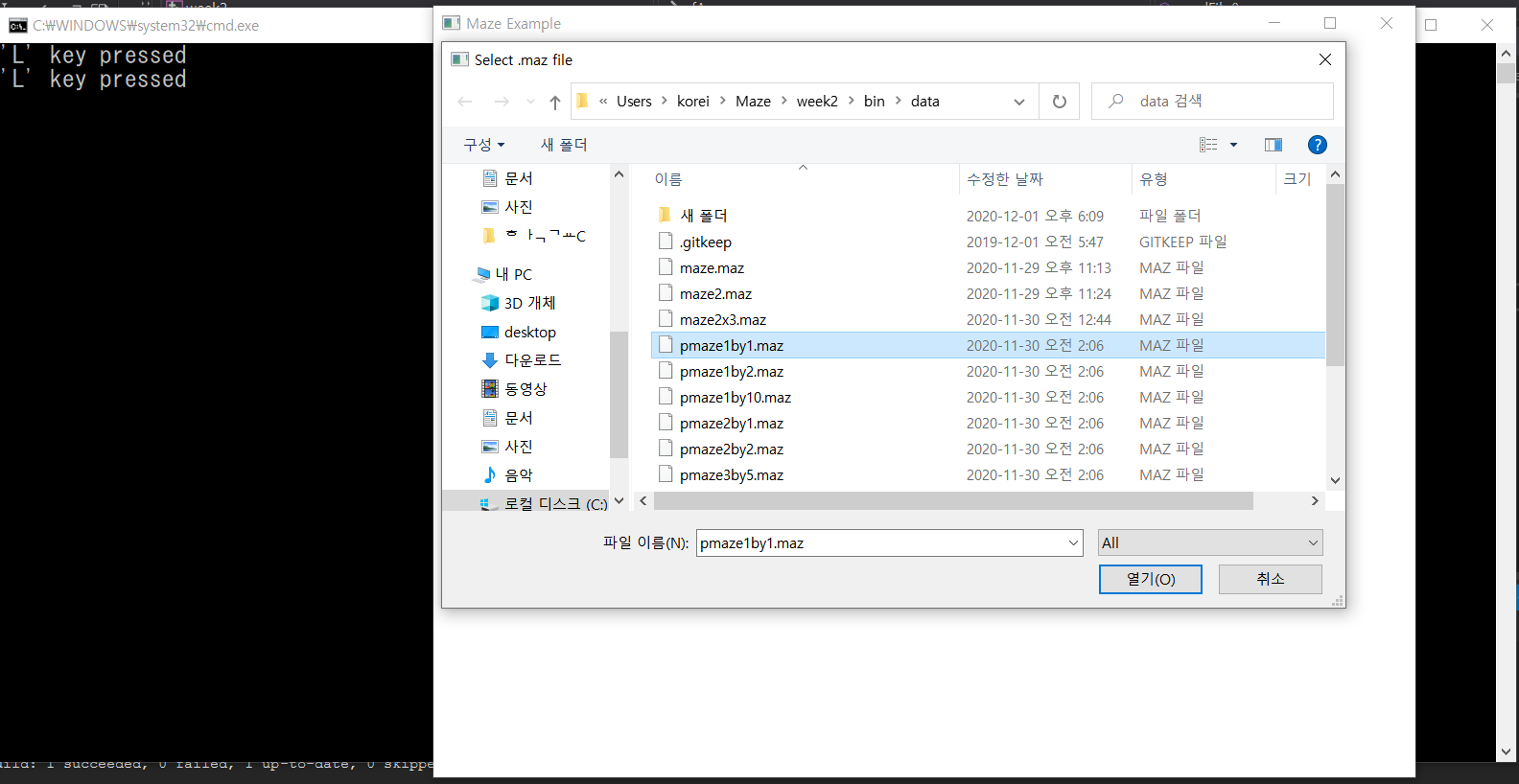
RAM : 8.00GB

IDE : Visual Studio 2019 community

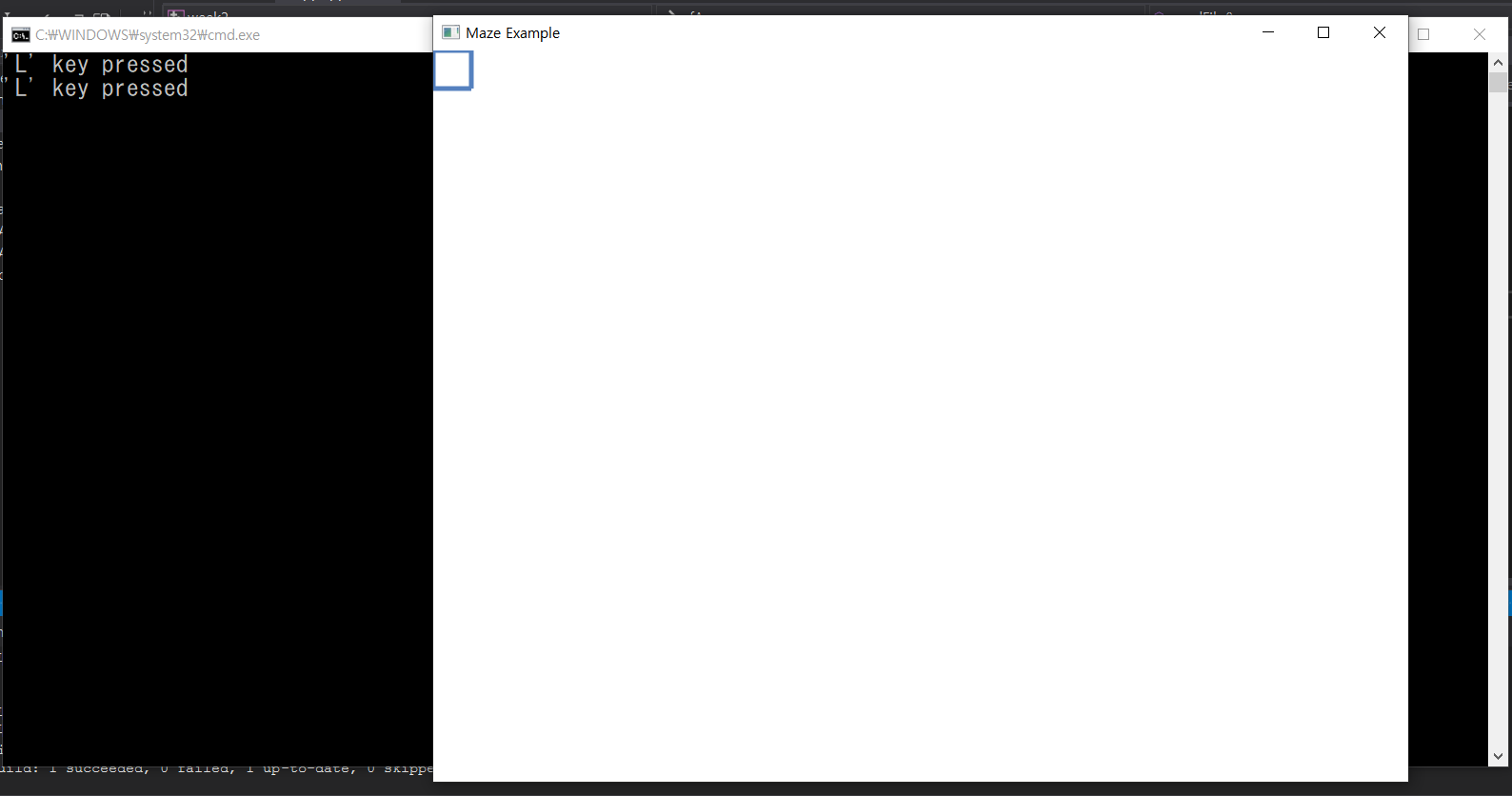
4. 실습 결과 및 분석

13주차에서 그리고 14주차에서 미로 찾기에 사용할 미로는 완전 미로이다. 따라서 입력할 .maz 파일도 완전 미로만을 가지고 있다.

먼저 특수한 크기의 미로(가로, 세로 중 어느 하나가 1인 경우, 2\*2 미로)가 잘 그려지는 지 확인한다.

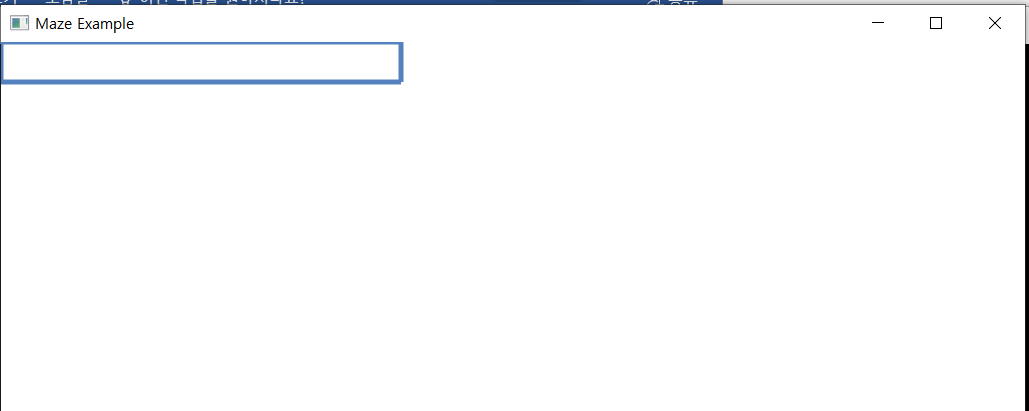


대소문자 구분 없이 L 키를 누르면 미로 파일을 선택하는 창이 뜬다. 가로 세로 모두 1인 미로를 선택해본다.



방이 하나인 미로가 그려졌다.

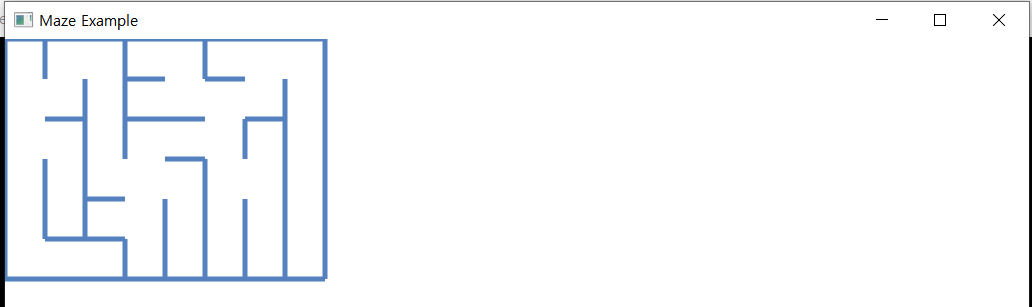
마찬가지로 방의 크기가 1\*10, 10\*1, 2\*2인 미로를 그려본다.



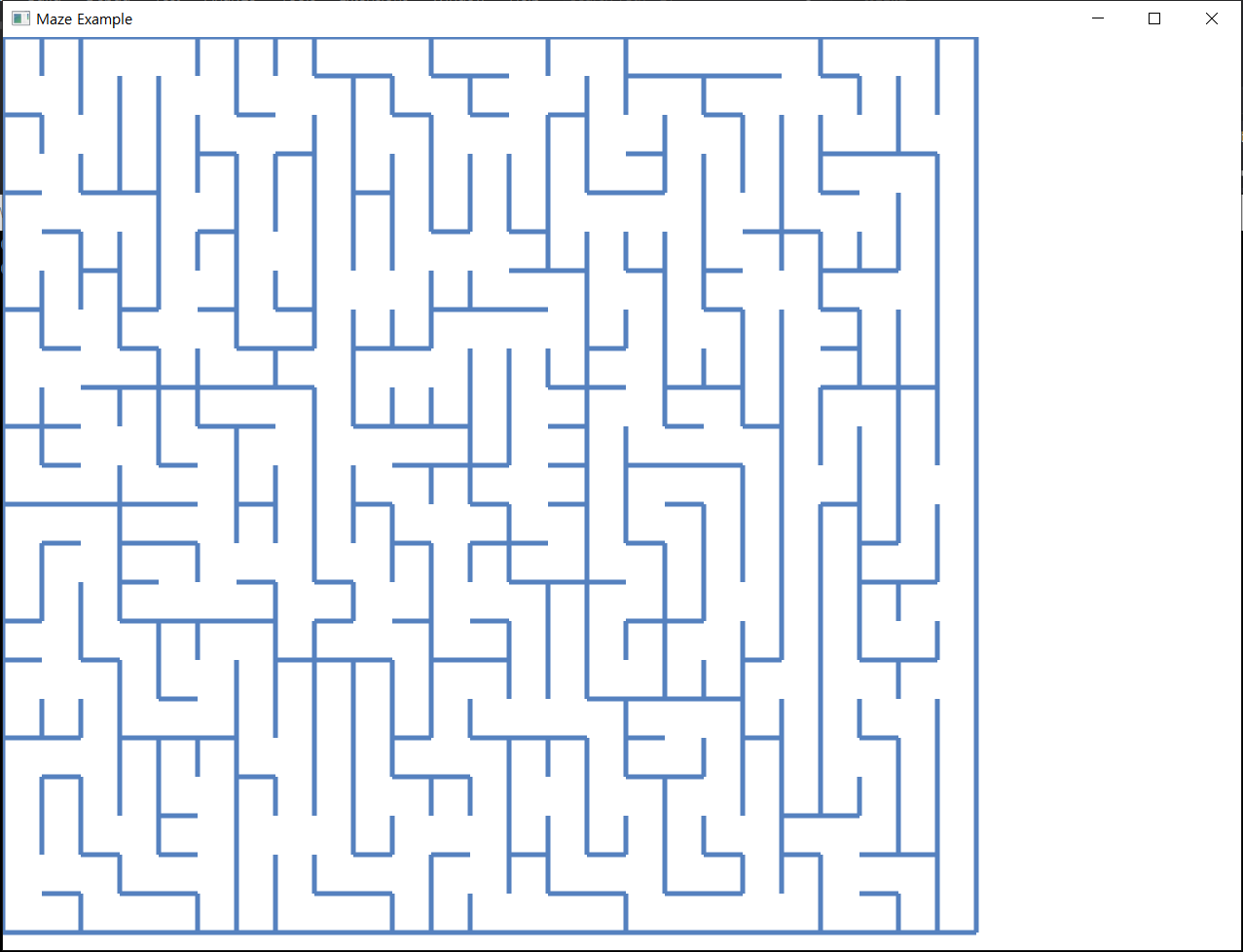


특수한 크기의 미로가 잘 그려진다.

일반적인 크기의 미로 8\*6 미로를 그려본다.

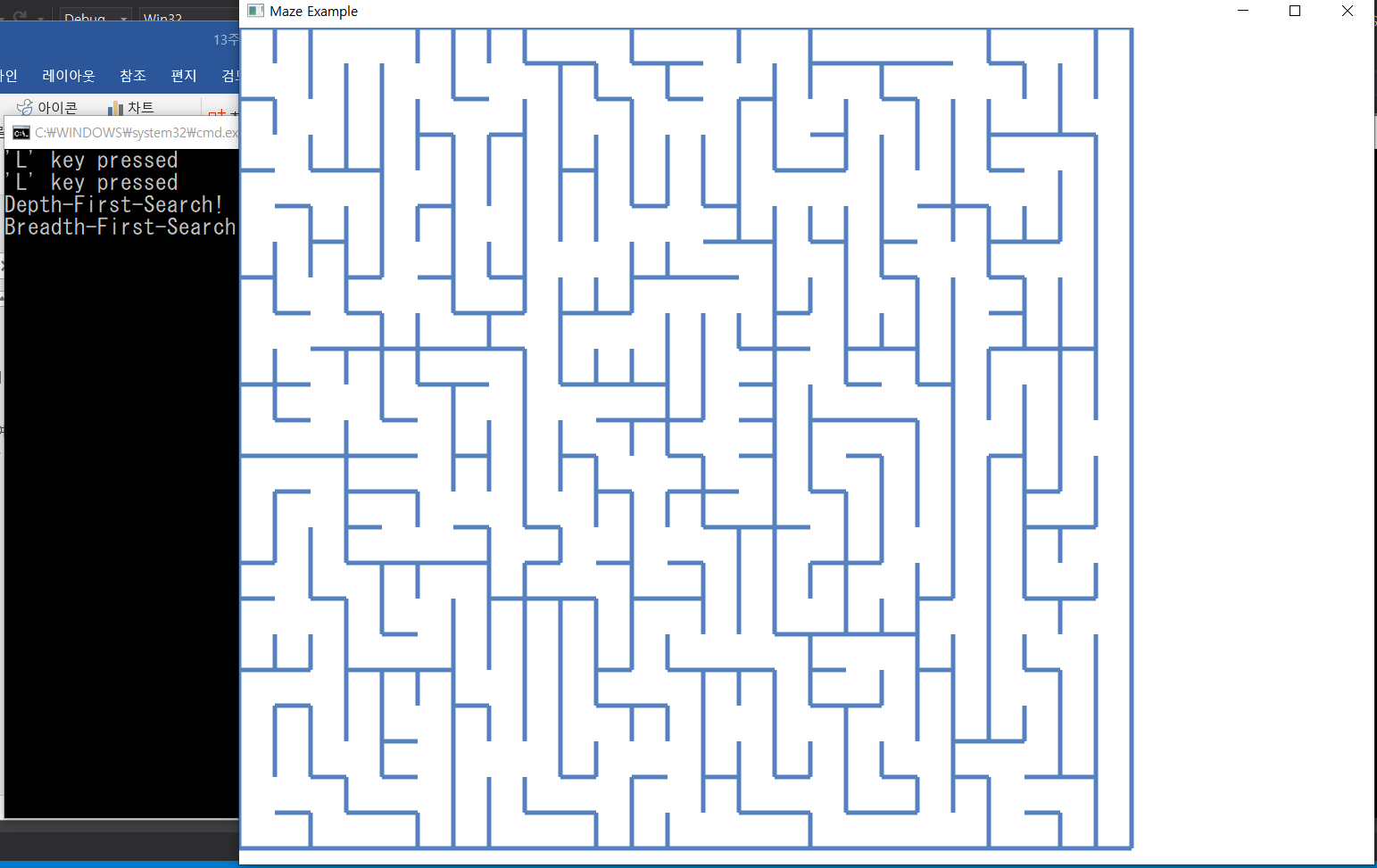
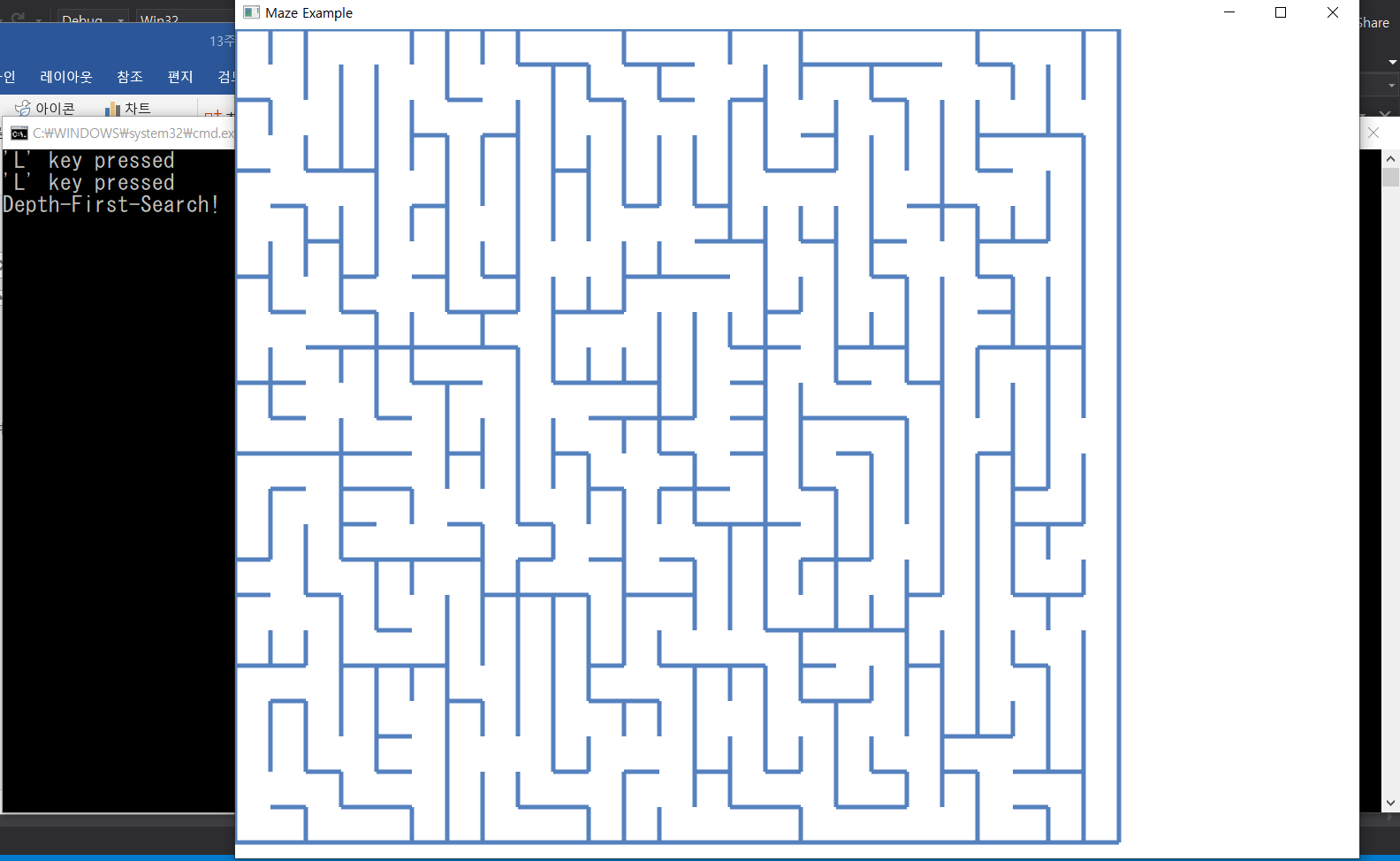


최대한 큰 미로를 그려본다. 25\*23 크기의 미로이다.

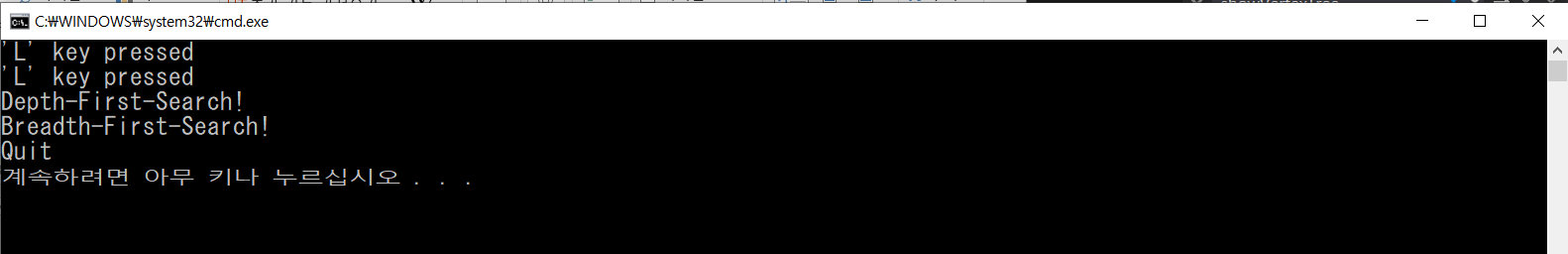


화면을 초과할 경우 화면을 늘리면 확인할 수 있다.

위의 경우 미로가 잘 그려지는 것을 알 수 있다.

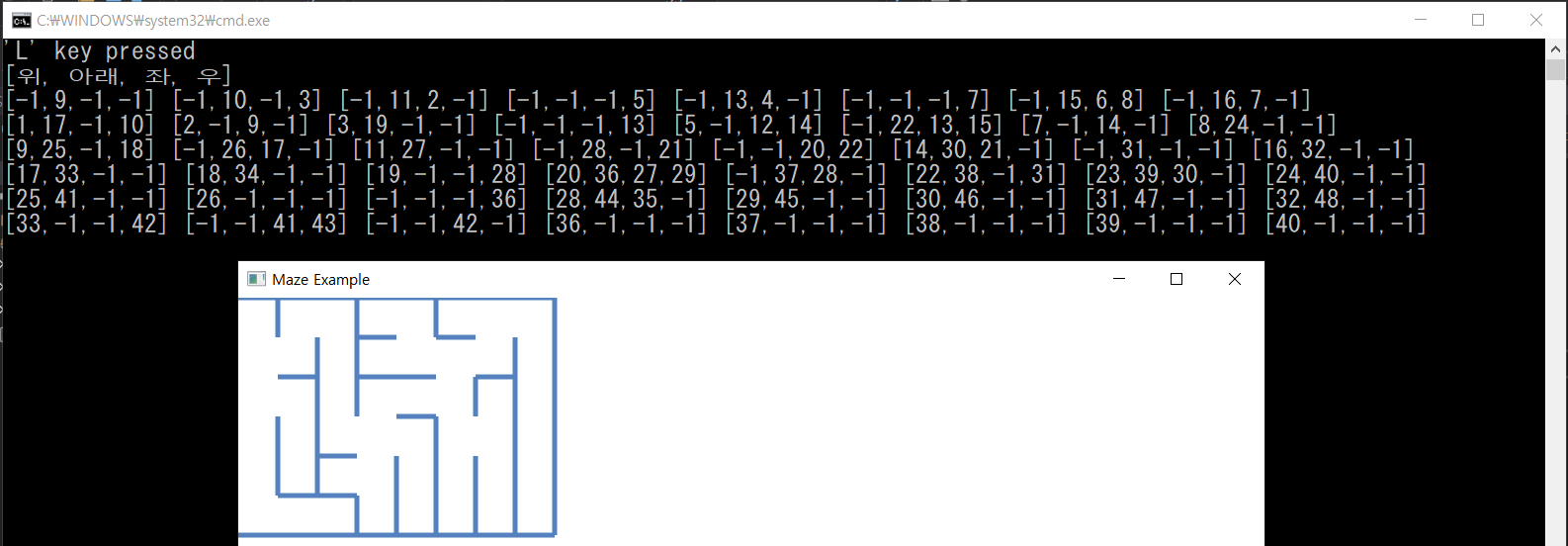


미로가 그려진 상태에서 d, b 버튼을 누르면 각각 depth-first-search!, breadth-first-search! 메시지가 콘솔에 출력되도록 하였다. 실제 탐색은 14주차에서 구현한다.



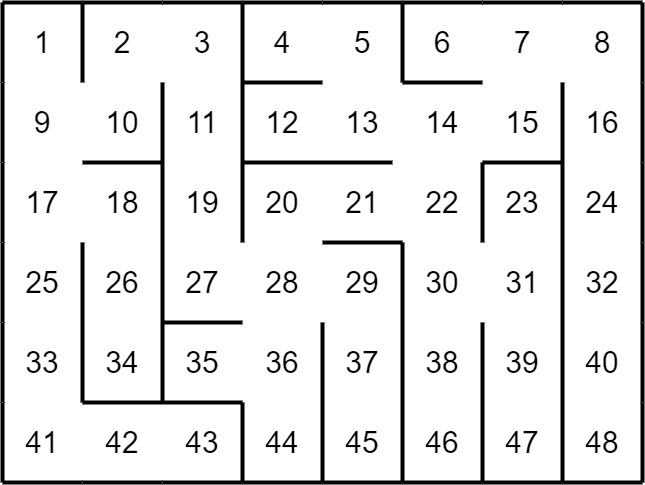
마지막으로 Q를 누르면 프로그램이 종료된다.

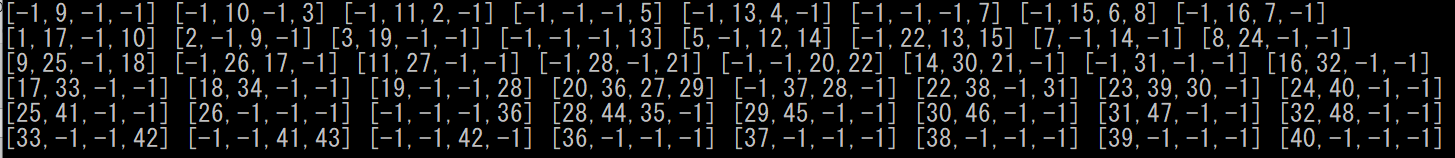
트리가 잘 만들어졌는지 확인하기 위해 임의로 함수 showVertexTree를 만들어 확인해본다. 각 노드들이 잘 연결되었는지를 확인하기 위해 vertex 배열의 원소들을 하나씩 확인한다. 각 원소의 up, down, left, right를 확인하여 값이 NULL일 경우 -1을 출력하고 NULL이 아닐 경우 연결된 Vertex 구조체의 name(방 번호)를 출력하도록 했다. 출력은 [윗방, 아랫방, 왼쪽방, 오른쪽방] 형식으로 출력된다.



각 방의 연결 상태는 위와 같다.

미로를 좀 더 알아보기 쉽게 방 번호와 함께 표현하면 다음과 같다.





위의 미로와 출력 결과를 비교해보면 정점끼리 연결된 형태가 트리가 되는 것을 알 수 있다. 이때 트리의 루트 노드는 1번 방이다.

기타

트리 외에도 인접 행렬을 이용해 그래프를 표현할 수 있고 인접 행렬로 표현한 그래프에서도 DFS와 BFS를 수행할 수 있다.