2020년도 2학기 컴퓨터공학설계및실험Ⅰ

14주차 미로(Maze) 3주차 결과보고서

20161663 허재성

1. 실습 목적

본 실험에서는 2주차의 실습에서 만든 OpenFrameWork 응용 프로그램을 수정하여 DFS 방법(실습)과 BFS 방법(과제)으로 미로를 탐색하여 그 결과를 화면에 표시하는 프로그램을 작성한다..

2. 실습 구현 내용

실습에서는 1주차에서 생성하고 2주차에서 그린 완전 미로를 DFS(깊이 우선 탐색)을 이용하여 시작점(왼쪽 위의 방, 1번 방)으로부터 도착점(오른쪽 아래의 방, WIDTH\*HEIGHT번 방)까지의 경로를 탐색한다. DFS를 위해 스택 자료구조를 사용한다. DFS 과정에서 방문한 방의 정점들을 스택에 삽입하여 쌓고, 그 중 경로에 포함되지 않는 방의 정점을 스택에서 제거한다. 탐색이 끝나고 스택에 남아있는 정점들이 경로에 포함되는 방의 정점들이고, 제거된 정점이 방문은 했지만 경로에는 포함되지 않는(방문하지 않아도 도착점에 도달하는데 상관이 없는) 방의 정점이다.

스택을 사용하기 위해 스택에 저장될 원소인 Node의 구조체를 헤더 파일 ofApp.h에 정의하였다.

|  |
| --- |
| typedef struct \_Node {  int vertex\_name;  \_Node\* link;  } Node; |

스택에 정점 구조체 Vertex를 직접 저장하는 대신 각 방의 번호를 대신 저장한다. 모든 방의 번호(name)은 고유하며 방의 번호를 이용해 2주차의 vertex 배열에서의 방을 나타내는 정점 구조체의 인덱스를 알아낼 수 있다. 인덱스를 알아내는 함수는 returnIdx로 뒤에서 설명한다.

Node의 vertex\_name 변수는 저장할 방(정점)의 번호이다. Node 포인터 link는 다른 Node 구조체를 가리킬 수 있는 포인터이다.

|  |
| --- |
| Node\* stackTop = NULL;  Node\* stackDel = NULL; |

Node 포인터 stackTop과 stackDel를 소스 파일 ofApp.cpp에 선언하였다. stackTop은 스택의 가장 위에 저장된 Node 구조체를 가리키는 포인터로 해당 위치에서 스택의 삽입과 삭제가 일어난다. stackTop이 스택을 의미한다.

stackDel은 스택에서 제거된 Node 구조체들이 저장된 리스트의 가장 앞 구조체를 가리키는 포인터이다. stackDel은 스택에서 제거된 노드들이 저장된 리스트를 의미한다.

스택, 리스트 관련 함수

(1) bool stackEmpty();

스택이 비어 있는지 확인하는 함수이다. stackTop이 NULL일 경우 스택에 저장된 Node 구조체가 없으므로 false를, NULL이 아닐 경우 true를 반환한다.

input : 없음

output : 스택이 비어 있을 경우 true, 비어 있지 않을 경우 false를 반환

(2) void stackPush(int name);

스택에 정점(Vertex)을 삽입한다. Vertex 구조체를 직접 삽입하는 대신 정점 구조체에 저장된 name을 저장한다. 노드를 동적 할당해 정점 구조체의 name을 저장하고, 노드의 link가 기존 스택의 stackTop을 가리키게 하고 stackTop이 새로 삽입될 노드를 가리킨다. 시간 복잡도는 O(1)이다.

input : 저장할 정점의 번호 name

output : 없음

(3) int stackPeek();

스택의 가장 위(top)에 저장된 Node 구조체에 저장된 정점의 번호를 반환한다. 가장 위의 노드는 가장 최근에 삽입된 노드이다. stackPop 함수와 다르게 해당 노드를 스택에서 제거하지 않고, 노드에 저장된 정점의 번호를 반환한다. 시간 복잡도는 O(1)이다.

input : 없음

output : 가장 최근에 삽입된 정점의 번호

(4) int stackPop();

스택의 가장 위(top)에 저장된 Node 구조체에 저장된 정점의 번호를 반환한다. 가장 위의 노드는 가장 최근에 삽입된 노드이다. stackPeek 함수와 다르게 해당 노드를 스택에서 제거한다. 제거된 노드는 insertList 함수를 호출하여 stackDel이 가리키는 리스트에 임시로 저장된다. 노드에 저장된 정점의 번호를 반환한다. 만약 스택이 비었을 경우 -1을 반환한다. 시간 복잡도는 O(1)이다.

input : 없음

output : 가장 최근에 삽입된 정점의 번호

(5) void insertList(Node\*\* list, Node\* node);

스택에서 제거된 노드 node를 리스트 stackDel에 삽입하는 함수이다. 리스트의 가장 앞에 삽입되므로 stackDel의 주소를 가리키는 포인터 list를 이용해 삽입이 이루어질 때마다 stackDel을 변경해준다. 시간 복잡도는 O(1)이다.

input : 스택에서 제거된 노드를 저장할 리스트를 가리키는 이중 포인터 list, 스택에서 제거된 노드 node

output : 없음

(6) void deleteList(Node\*\* list)

스택에서 제거된 노드들이 저장된 리스트를 완전히 비워준다. stackPop 함수가 동적할당한 노드를 단순히 스택에서 연결을 끊어 리스트에 저장한다면 deleteList 함수에서 리스트에 저장된 동적할당 된 노드를 메모리 해제한다.

노드의 삭제는 리스트의 가장 앞의 노드부터 이루어진다. stackDel을 가리키는 이중 포인터 list를 이용하여 제거가 이루어질 때마다 stackDel을 변경하고 stackDel이 NULL일 경우 모든 노드가 해제되었으므로 종료한다.

deleteList 함수는 스택에서 제거된 노드를 저장한 리스트뿐만 아니라 스택에 저장된 노드들도 해제할 수 있다.

deleteList의 시간 복잡도는 리스트에 저장된 노드의 개수에 의존한다. 노드의 최대 개수가 WIDTH\*HEIGHT이므로 시간 복잡도는 O(WIDTH\*HEIGHT)이다.

input : 스택에서 제거된 노드를 저장할 리스트를 가리키는 이중 포인터 list

output : 없음

위에서 정의한 자료구조를 이용하여 DFS를 구현한다.

ofApp.h의 ofApp 클래스의 멤버 변수로 dfs\_flag를 추가한다.

|  |
| --- |
| bool dfs\_flag; |

‘L’ 키를 눌러 파일을 선택해 미로가 그려진 상태에서 ‘D’ 키를 누를 경우 dfs 함수가 호출되어 경로에 포함된 정점은 스택에, 방문했지만 경로에 포함되지 않는 정점은 리스트에 저장되어 분류가 된다. 분류가 된 상태에서 dfs\_flag를 true로 변경하면 draw 함수에서 경로에 포함된 정점과 포함되지 않는 정점을 구분하여 경로는 초록색, 경로가 아닌 부분은 빨간색으로 구분하여 그린다. ofApp 클래스의 setup 함수에서 처음 dfs\_flag를 false로 초기화한다.

DFS를 위해 Vertex 구조체에서 트리에서의 자신의 부모 Vertex 구조체를 알 수 있어야 한다. DFS에서의 부모 정점은 자신을 방문하기 이전에 방문했던 정점을 의미한다. 이를 위해 Vertex 구조체에 다음 변수를 추가한다.

|  |
| --- |
| \_Vertex\* parent; |

DFS를 위해 스택에 저장된 노드의 정점 번호를 통해 실제 정점 구조체에 접근할 수 있어야 한다. 정점의 번호를 이용해 해당 정점이 실제로 정점(Vertex) 구조체가 저장된 2차원 배열 vertex 인덱스(0<= i < HEIGHT, 0 <= j < WIDTH)에 저장된 위치(인덱스)를 알아내야 한다. 이를 위해 다음과 같은 함수를 정의하였다.

(1) void returnIdx(int name, int\* hidx, int\* widx)

정점의 번호 name을 이용해 2차원 배열 vertex에 정점이 저장된 인덱스를 알아낸다. 가로 5, 세로 4인 미로의 방들을 저장한 정점 배열 vertex의 경우 인덱스 (i, j)는 아래와 같다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| (0, 0) | (0, 1) | (0, 2) | (0, 3) | (0, 4) |
| (1, 0) | (1, 1) | (1, 2) | (1, 3) | (1, 4) |
| (2, 0) | (2, 1) | (2, 2) | (2, 3) | (2, 4) |
| (3, 0) | (3, 1) | (3, 2) | (3, 3) | (3, 4) |

방 번호를 같이 표시하면 다음과 같다.(굵은 숫자가 방 번호)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **1** (0, 0) | **2** (0, 1) | **3** (0, 2) | **4** (0, 3) | **5** (0, 4) |
| **6** (1, 0) | **7** (1, 1) | **8** (1, 2) | **9** (1, 3) | **10** (1, 4) |
| **11** (2, 0) | **12** (2, 1) | **13** (2, 2) | **14** (2, 3) | **15** (2, 4) |
| **16** (3, 0) | **17** (3, 1) | **18** (3, 2) | **19** (3, 3) | **20** (3, 4) |

이중 for문을 이용하여 (0, 0)부터 (HEIGHT-1, WIDTH-1)까지 탐색한다. 가로 인덱스를 w, 세로 인덱스가 h일 때 name과 WIDTH \* h + w + 1과 같을 경우 해당 h, w가 찾으려는 인덱스이므로 입력으로 받은 포인터 hidx, widx가 가리키는 변수에 h, w를 저장한다.

input : 인덱스를 찾으려는 정점의 번호, 찾으려는 세로 인덱스, 가로 인덱스 변수를 가리키는 포인터

output : 없음

위의 함수를 이용해 DFS를 구현할 수 있다.

(2) void dfs(Vertex\* root, int target)

2주차에서 제작했던 정점들을 연결해 구현한 Vertex 트리에서 dfs를 수행하는 함수이다. 입력인 Vertex 포인터 root는 Vertex 트리의 루트 노드를 가리키며 1번 방을 나타내는 정점이다. 1번 방의 정점은 루트이므로 부모 정점이 없다. 1번 방 이전에 방문한 방 또한 존재하지 않는다. 따라서 root의 parent를 root로 설정하고 root를 방문한 것으로 처리한 뒤에 root의 번호를 스택에 삽입한다.

stackEmpty 함수로 확인해 스택이 비어 있지 않을 경우, stackPeek 함수를 이용해 스택에 가장 최근에 저장된 정점의 번호를 확인한다. 만약 해당 번호가 도착점의 번호 target과 같다면 해당 정점이 도착점이므로 탐색을 종료한다.

그렇지 않을 경우 returnIdx 함수를 이용해 해당 정점의 인덱스 hidx, widx를 알아내어 해당 정점 구조체에 직접 접근할 수 있게 한다. 정점 구조체는 vertex[hidx][widx]가 된다.

시작점의 번호가 1이고 도착점의 번호가 WIDTH\*HEIGHT이므로 탐색의 우선 순위를 번호가 큰 방부터 탐색하도록 한다. 따라서 아래 방, 오른쪽 방, 왼쪽 방, 위 방 순서로 탐색한다.

먼저 vertex[hidx][widx].down을 확인한다. 해당 값이 NULL이 아닐 경우 아래 방이 존재한다. 해당 방이 존재하고 아직 방문하지 않았을 경우, 해당 방 vertex[hidx][widx].down의 visited를 true로 설정하여 해당 방을 방문했다고 표시한다. 또한 vertex[hidx][widx].down 방의 부모를 vertex[hidx][widx]로 설정하여 이전에 방문한 방을 알 수 있도록 한 뒤에 vertex[hidx][widx].down의 방 번호를 스택에 삽입한다.

만약 vertex[hidx][widx].down이 NULL이거나 이미 방문했을 경우 다음 우선 순위인 오른쪽 방을 탐색하야 한다. vertex[hidx][widx].right를 위와 같은 방법으로 확인한다.

마찬가지로 vertex[hidx][widx].left, vertex[hidx][widx].up을 확인한다.

만약 down, right, left, up을 모두 확인했는데 더 이상 방문하지 않은 인접한 방이 없을 경우, 해당 방의 정점을 가진 노드를 stackPop을 이용해 스택에서 제거한다.

위와 같은 과정을 스택이 빌 때까지 계속한다. 함수가 종료되면 경로에 포함된 정점들은 스택에 남아 있고, 방문했지만 포함되지 않은 정점들은 스택에서 제거되어 리스트에 저장된다.

dfs의 의사코드는 다음과 같다.

|  |
| --- |
| void dfs(Vertex\* root, int target) {  시작점(root)의 부모를 시작점으로 설정;  시작점을 방문;  stackPush(시작점 번호); // 스택에 삽입  while (!stackEmpty()) {  int u = stackPeek();  if (u가 도착점과 같을 경우) return;  u에 해당하는 정점 구조체를 가져옴;  if (아랫방을 방문할 수 있을 경우) {  아랫방을 방문함;  방을 아랫방의 부모로 설정;  stackPush(아랫방 번호);  }  else if (오른쪽 방을 방문할 수 있을 경우) {  오른쪽방을 방문함;  방을 오른쪽방의 부모로 설정;  stackPush(오른쪽방 번호);  }  else if (왼쪽 방을 방문할 수 있을 경우) {  왼쪽방을 방문함;  방을 왼쪽방의 부모로 설정;  stackPush(왼쪽방 번호);  }  else if (위쪽 방을 방문할 수 있을 경우) {  위쪽방을 방문함;  방을 위쪽방의 부모로 설정;  stackPush(위쪽방 번호);  }  방문할 수 있는 방이 없을 경우 현재 방을 스택에서 제거함;  }  } |

DFS에서는 우선순위에 따라 현재 방을 기준, 아랫방->오른쪽방->왼쪽방->윗방의 순서대로 탐색하는데, 더 높은 우선순위의 방을 방문할 수 없을 때 다음 우선순위의 방문을 고려한다.(if-else if 문으로 구현되어 있다.) 더 높은 우선순위의 방을 방문할 수 있을 경우 해당 방에 방문하여 이동한 방을 기준으로 인접한 방을 탐색한다.

최악의 경우 모든 정점을 다 한번씩 방문해야 하므로 시간 복잡도는 O(WIDTH\*HEIGHT)이다. 모든 정점을 방문해야 하는 경우 모든 정점이 스택에 삽입되어야 한다. 따라서 공간 복잡도는 O(WIDTH\*HEIGHT)이다.

input : DFS로 경로를 찾을 미로를 나타내는 Vertex 트리 포인터 root, 찾으려는 방 번호 target

output : 없음

ofApp 클래스에서 수정한 함수

(1) void ofApp::draw()

draw\_flag가 true여서 미로가 그려진 상태에서 ‘D’ 키가 눌려서 dfs\_flag가 true일 경우 미로에 경로와 경로에 포함되지 않은 길을 구분하여 그린다. 스택 stackTop에 저장되어 있는 노드들을 모두 탐색해 노드에 저장된 정점의 번호와 returnIdx 함수를 이용해 실제 Vertex 구조체에 접근한다. Vertex 구조체의 x, y와 parent의 x, y를 ofDrawLine 함수를 이용해 연결하여 두 방의 중심을 잇는 선분을 그린다. 이때 ofSetColor 함수를 이용해 경로의 색상을 초록색으로 설정한다.

마찬가지로 리스트 stackDel에 저장되어 있는 노드들을 모두 탐색해서 방문했지만 경로에 포함되지 않은 길을 그린다. 이 때 경로와 구분되게 하기 위해 ofSetColor 함수를 이용하여 빨간색으로 설정한다.

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

draw\_flag가 true일 때, 즉 이미 미로가 그려진 상태에서 대소문자 관계 없이 ‘D’ 키를 입력하면 콘솔에 Depth-First-Search를 출력 후 dfs 함수를 호출한다. dfs 함수를 호출 후 dfs\_flag를 true로 설정하여 draw 함수에서 경로가 그려질 수 있도록 한다.

(2) void ofApp::freeMemory()

dfs\_flag를 false로 설정하여 더 이상 경로가 그려지지 않게 한 다음, deleteList 함수를 호출하여 스택 stackTop과 리스트 stackDel에 저장된 모든 동적 할당된 Node 구조체들을 해제한다.

3. 실습 환경

OS : Windows 10

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

RAM : 8.00GB

IDE : Visual Studio 2019 community

4. 과제

과제에서는 1주차에서 생성하고 2주차에서 그린 완전 미로를 BFS(너비 우선 탐색)을 이용하여 시작점(왼쪽 위의 방, 1번 방)으로부터 도착점(오른쪽 아래의 방, WIDTH\*HEIGHT번 방)까지의 경로를 탐색한다. BFS를 위해 큐 자료구조를 사용한다. BFS 과정에서 방문한 방의 정점을 큐에 삽입한다. 큐가 비지 않았을 경우 큐에서 정점을 제거하여 가져온 뒤, 해당 정점이 도착 정점일 경우 탐색을 종료한다. 그렇지 않을 경우 해당 정점의 인접 정점들 중 방문하지 않은 정점을 모두 방문하여 큐에 삽입한다. 큐가 비게 되거나 도착점을 찾게 되면 BFS는 종료된다.

큐를 구현하기 위해 스택에서 사용한 Node 자료구조를 큐의 노드로 그대로 사용할 수 있다.

|  |
| --- |
| Node\* queueFront = NULL;  Node\* queueRear = NULL;  Node\* queueDel = NULL; |

Node 포인터 queueFront, queueRear, queueDel를 소스 파일 ofApp.cpp에 선언하였다. queueFront는 큐의 가장 앞에 저장된 노드를 가리키는 포인터로 큐에서 가장 앞에 저장된 노드는 큐에 가장 먼저 저장된 노드를 가리키며 큐에서 노드를 제거할 때 가장 먼저 제거되는 노드이다. 큐의 제거는 queueFront에서 일어나며 queueFront가 큐를 의미한. queueRear는 큐의 가장 뒤에 저장된 노드를 가리키는 포인터로 큐에 가장 최근에 삽입된 노드를 가리킨다. 큐에 삽입은 queueRear에서 이루어진다.

queueDel은 큐에서 제거된 노드들을 임시로 저장하는 리스트를 가리키는 포인터이다. 큐에 저장되었던 모든 정점들은 BFS 과정에서 적어도 한 번 방문한 정점이고 큐에서 제거된 정점들은 큐에 삽입되었던 정점들의 부분 집합이다. 따라서 큐에서 제거된 노드들을 저장한 queueDel의 노드들 또한 방문했던 적이 있는 정점들의 집합이다. 이를 이용해 미로 찾기의 경로를 그릴 때 방문한 적 있는 정점의 길을 빨간 색으로 그린다.

큐를 사용하기 위해 다음과 같은 함수를 정의한다. insertList, deleteList 함수는 스택을 사용할 때 정의했던 함수를 그대로 사용할 수 있다.

큐 관련 함수

(1) bool queueEmpty();

큐가 비어 있는지 확인하는 함수이다. queueFront가 NULL일 경우 큐에 저장된 Node 구조체가 없으므로 false를, NULL이 아닐 경우 true를 반환한다.

input : 없음

output : 큐가 비어 있을 경우 true, 비어 있지 않을 경우 false를 반환

(2) void enqueue(int name);

큐에 정점(Vertex)을 삽입한다. 스택과 마찬가지로 큐에 Vertex 구조체를 직접 삽입하는 대신 정점 구조체에 저장된 name을 저장한다. 노드를 동적 할당해 정점 구조체의 name을 저장한다. 만약 기존 큐가 비었을 경우 queueFront가 새로 삽입될 노드를 가리키게 하고, 큐가 비어있지 않을 경우 queueRear의 link가 새로 삽입될 노드를 가리키게 하여 노드가 큐의 끝에 저장되도록 한다. 마지막으로 queueRear가 삽입된 노드를 가리키게 한다.

input : 저장할 정점의 번호 name

output : 없음

(3) int dequeue();

큐의 가장 앞(front)에 저장된 Node 구조체에 저장된 정점의 번호를 반환한다. 가장 앞의 노드는 가장 먼저 삽입된 노드이다. 제거된 노드는 insertList 함수를 호출하여 queueDel이 가리키는 리스트에 임시로 저장된다. 만약 큐가 비었을 경우 -1을 반환한다. 큐에 저장된 노드가 1개일 경우, queueRear를 NULL로 설정한다. queueFront가 queueFront의 link로 가리키던 다음 노드를 가리키고, 제거할 노드를 insertList 함수를 호출하여 queueDel에 저장한다. 마지막으로 제거된 노드에 저장된 정점의 번호를 반환한다. 시간 복잡도는 O(1)이다.

input : 없음

output : 가장 먼저 삽입된 정점의 번호

같은 미로에 대해서 DFS와 BFS 탐색이 동시에 가능하기 위해서는 한 가지 탐색으로 설정된 트리의 노드의 방문 상태와 부모를 재설정해줘야 다른 탐색을 정상적으로 수행할 수 있다. 만약 재설정 과정이 없이 DFS를 수행한 상태에서 BFS를 다시 수행하면 DFS 과정에서 트리 노드들의 방문 상태가 true로 변경된 상태에서 탐색을 하기 때문에 BFS에서 아직 방문하지 않은 노드를 방문할 수 없게 된다. BFS 후 DFS를 수행할 경우도 마찬가지이다. 이를 방지하기 위해 다음 함수를 정의한다.

(1) void resetVertices()

정점이 저장된 Vertex 배열 vertex의 모든 Vertex 구조체의 visited 변수를 false로 설정하고, 모든 Vertex 포인터 parent를 NULL로 초기화한다. 모든 Vertex 구조체에 대해 수행하므로 시간 복잡도는 O(WIDTH\*HEIGHT)이다.

input : 없음

output : 없음

BFS를 구현하면 다음과 같다.

(2) void bfs(Vertex\* root, int target)

2주차에서 제작했던 정점들을 연결해 구현한 Vertex 트리에서 bfs를 수행하는 함수이다. 입력인 Vertex 포인터 root는 Vertex 트리의 루트 노드를 가리키며 1번 방을 나타내는 정점이다. 1번 방의 정점은 루트이므로 부모 정점이 없다. 1번 방 이전에 방문한 방 또한 존재하지 않는다. 따라서 root의 parent를 root로 설정하고 root를 방문한 것으로 처리한 뒤에 root의 번호를 큐에 삽입한다.

queueEmpty 함수로 확인해 큐가 비어 있지 않을 경우, dequeue 함수를 이용해 큐에 가장 먼저 저장된 노드를 제거하여 가장 먼저 저장된 정점의 번호를 얻어온다. 만약 해당 번호가 도착점의 번호 target과 같다면 해당 정점이 도착점이므로 탐색을 종료한다.

그렇지 않을 경우 returnIdx 함수를 이용해 해당 정점의 인덱스 hidx, widx를 알아내어 해당 정점 구조체에 직접 접근할 수 있게 한다. 정점 구조체는 vertex[hidx][widx]가 된다.

시작점의 번호가 1이고 도착점의 번호가 WIDTH\*HEIGHT이므로 탐색의 우선 순위를 번호가 큰 방부터 탐색하도록 한다. 따라서 DFS의 경우와 마찬가지로 아래 방, 오른쪽 방, 왼쪽 방, 위 방 순서로 탐색하며 DFS와 다르게 아래 방, 오른쪽 방, 왼쪽 방, 위 방 모두 확인하여 존재하고 방문하지 않았을 경우 모두 방문하고 큐에 삽입한다.

먼저 vertex[hidx][widx].down을 확인한다. 해당 값이 NULL이 아닐 경우 아래 방이 존재한다. 해당 방이 존재하고 아직 방문하지 않았을 경우, 해당 방 vertex[hidx][widx].down의 visited를 true로 설정하여 해당 방을 방문했다고 표시한다. 또한 vertex[hidx][widx].down 방의 부모를 vertex[hidx][widx]로 설정하여 이전에 방문한 방을 알 수 있도록 한 뒤에 vertex[hidx][widx].down의 방 번호를 스택에 삽입한다.

vertex[hidx][widx].down과 관계 없이 다음 우선 순위인 오른쪽 방을 탐색하야 한다. vertex[hidx][widx].right를 위와 같은 방법으로 확인하여 존재하고 방문하지 않았을 경우에는 큐에 삽입한다.

마찬가지로 vertex[hidx][widx].left, vertex[hidx][widx].up을 확인 후 존재하고 방문하지 않았을 경우에는 큐에 삽입한다..

위와 같은 과정을 스택이 빌 때까지 계속한다. 목적 정점을 찾아서 함수가 종료되면 큐에서 제거된 정점과 큐에 남은 정점 모두 방문한 정점이 된다.

bfs의 의사코드는 다음과 같다.

|  |
| --- |
| void bfs(Vertex\* root, int target) {  시작점(root)의 부모를 시작점으로 설정;  시작점을 방문;  enqueue(시작점 번호); // 큐에 삽입  while (!queueEmpty()) {  int w = dequeue();  if (w가 도착점과 같을 경우) return;  w에 해당하는 정점 구조체를 가져옴;  if (아랫방을 방문할 수 있을 경우) {  아랫방을 방문함;  방을 아랫방의 부모로 설정;  enqueue(아랫방 번호);  }  if (오른쪽 방을 방문할 수 있을 경우) {  오른쪽방을 방문함;  방을 오른쪽방의 부모로 설정;  enqueue(오른쪽방 번호);  }  if (왼쪽 방을 방문할 수 있을 경우) {  왼쪽방을 방문함;  방을 왼쪽방의 부모로 설정;  enqueue(왼쪽방 번호);  }  if (위쪽 방을 방문할 수 있을 경우) {  위쪽방을 방문함;  방을 위쪽방의 부모로 설정;  enqueue(위쪽방 번호);  }  }  } |

BFS에서는 우선순위에 따라 현재 방을 기준, 아랫방->오른쪽방->왼쪽방->윗방의 순서대로 탐색하는데, 더 높은 우선순위의 방을 방문할 수 없을 때 다음 우선순위의 방문을 고려한다.(여러 개의 if 문으로 구현되어 있다.) 현재 방을 기준으로 우선 순위에 따라 인접한 모든 방을 차례로 방문한 후 다음 방으로 이동한다.

최악의 경우 모든 정점을 다 한번씩 방문해야 하므로 시간 복잡도는 O(WIDTH\*HEIGHT)이다. 모든 정점을 방문해야 하는 경우 모든 정점이 큐에 삽입되어야 한다. 따라서 공간 복잡도는 O(WIDTH\*HEIGHT)이다.

input : BFS로 경로를 찾을 미로를 나타내는 Vertex 트리 포인터 root, 찾으려는 방 번호 target

output : 없음

ofApp.h의 ofApp 클래스의 멤버 변수로 bfs\_flag를 추가한다.

|  |
| --- |
| bool bfs\_flag; |

‘L’ 키를 눌러 파일을 선택해 미로가 그려진 상태에서 ‘B’ 키를 누를 경우 bfs 함수가 호출되어 BFS 과정에서 방문한 정점들이 큐와 리스트에 남아있다. 이를 이용해 draw 함수에서 방문한 길과 그 중에서 경로를 따로 그려야 한다. bfs\_flag를 true로 변경하면 draw 함수에서 경로는 초록색, 경로가 아닌 부분은 빨간색으로 구분하여 그린다. ofApp 클래스의 setup 함수에서 처음 bfs\_flag를 false로 초기화한다.

ofApp 클래스에서 수정한 함수

(1) void ofApp::draw()

draw\_flag가 true여서 미로가 그려진 상태에서 ‘B’ 키가 눌려서 bfs\_flag가 true일 경우 미로에 경로와 경로에 포함되지 않은 길을 구분하여 그린다.

먼저 큐에 저장되어 있는 모든 노드를 dequeue 함수를 이용해 queueDel 리스트에 저장한다. 따라서 queueDel에 저장된 노드들이 미로 탐색 중 방문한 노드들이 된다. queueDel에 저장된 노드들을 모두 탐색하여 정점의 번호와 returnIdx 함수를 이용해 실제 Vertex 구조체에 접근한다. Vertex 구조체의 x, y와 parnet의 x, y를 ofDrawLine 함수를 이용해 연결하여 두 방의 중심을 잇는 선분을 그린다. 이 때 ofSetColor 함수를 이용해 경로의 선분의 색상을 빨간색으로 그리면 방문한 정점들을 잇는 길이 그려진다. 방문한 정점의 길에 미로의 경로가 포함된다.

방문한 길을 그린 후 도착 정점에서부터 시작해 설정된 parent를 따라 시작 정점이 나올 때까지 탐색하여 경로를 그린다. 이 때 경로는 ofSetColor 함수를 이용해 초록색으로 그린다. 따라서 방문한 모든 방을 이은 길과, 그 중 최단경로가 구분되어 나타난다.

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

draw\_flag가 true일 때, 즉 이미 미로가 그려진 상태에서 대소문자 관계 없이 ‘B’ 키를 입력하면 콘솔에 Breadth-First-Search를 출력 후 bfs 함수를 호출한다. bfs 함수를 호출 후 bfs\_flag를 true로 설정하여 draw 함수에서 경로가 그려질 수 있도록 한다.

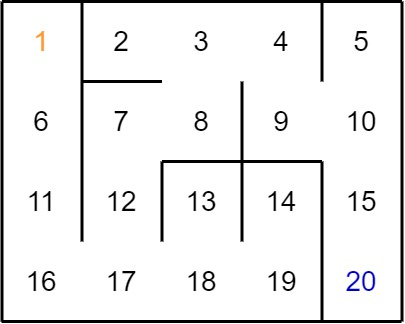
만약 ‘B’ 키를 눌렀는데 dfs\_flag가 true일 경우 이미 DFS 탐색이 이루어진 상태로 미로에 DFS로 찾은 경로가 그려져 있다. 이 경우, dfs\_flag를 false로 바꾸고 resetVertices 함수를 호출하여 트리의 Vertex 구조체의 visited와 parent를 모두 재설정해준 뒤에 bfs 함수를 호출한다. 반대로 ‘D’ 키를 눌렀는데 bfs\_flag가 true일 경우에도 같은 방법으로 처리한다. 따라서 ‘D’ 키와 ‘B’ 키를 번갈아 누르면 DFS로 찾은 미로 탐색 결과와 BFS로 찾은 미로 탐색 결과를 번갈아 확인할 수 있다. 이를 위해 스택에서 제거된 노드들은 stackDel 리스트에, 큐에서 제거된 노드들은 queueDel 리스트에 따로 저장하였다.

(2) void ofApp::freeMemory()

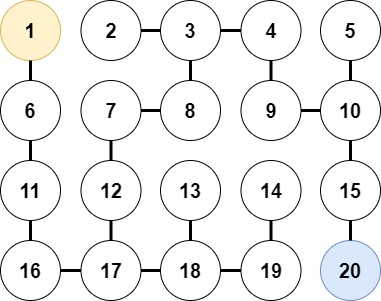
dfs\_flag와 마찬가지로 bfs\_flag를 false로 설정하여 더 이상 경로가 그려지지 않게 한 다음, deleteList 함수를 호출하여 큐 queueFront과 리스트 queueDel에 저장된 모든 동적 할당된 Node 구조체들을 해제한다.

4. 실습, 과제 결과 및 분석

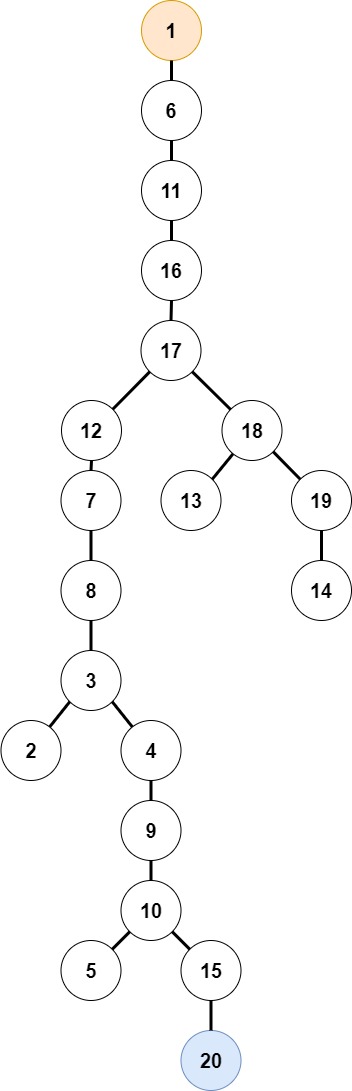
실습 결과



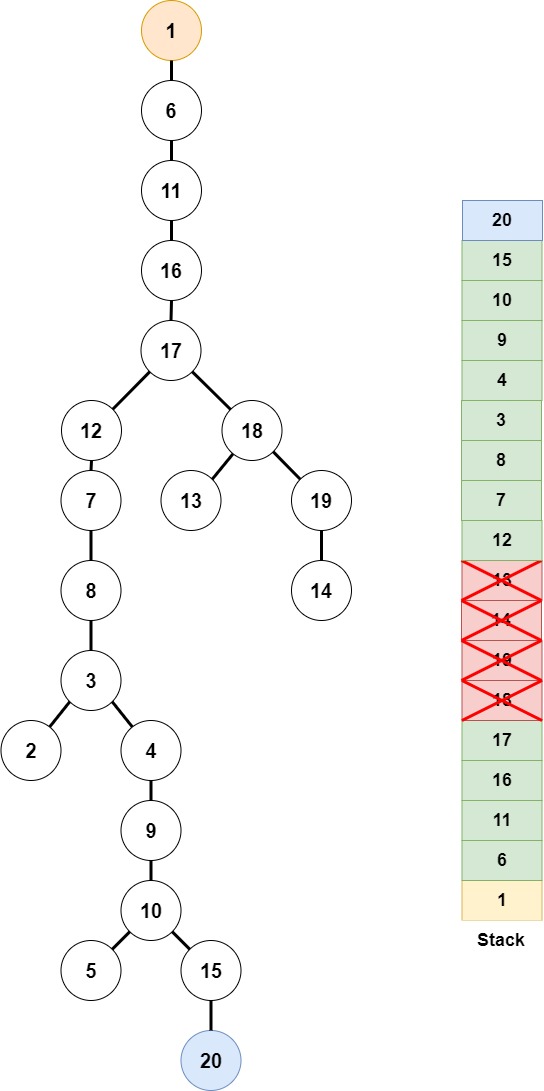
위와 같은 미로를 그래프로 나타내면 다음과 같다.



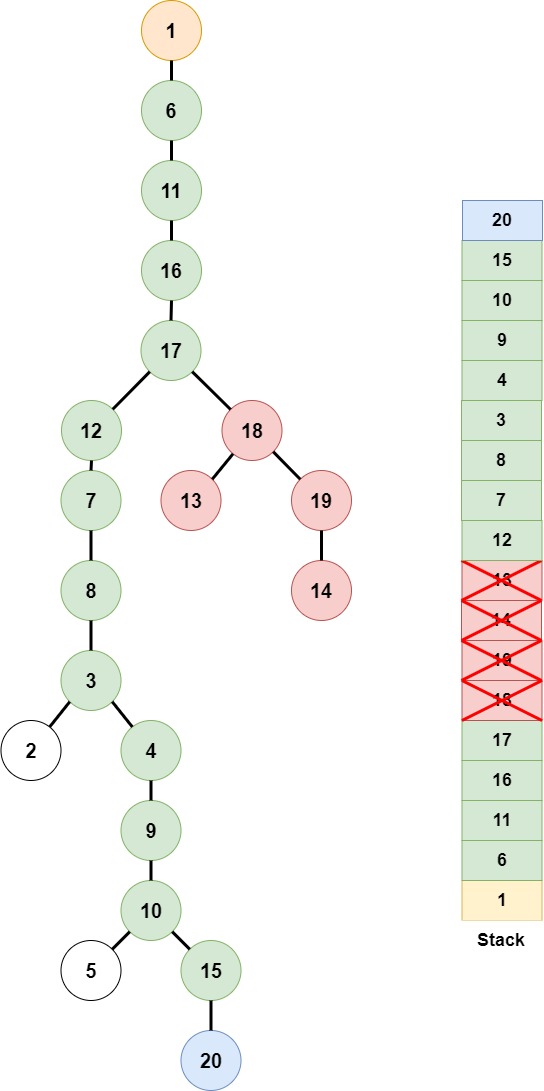
그래프를 알아보기 쉽게 트리 형태로 변형하면 다음과 같다.



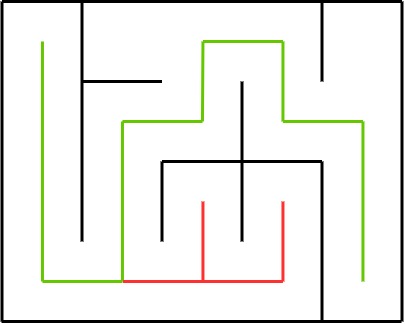
아래, 오른쪽, 왼쪽, 위 순서로 방문할 경우, 스택에 쌓이는 과정은 다음과 같다. 가위표 된 노드는스택에서 제거된 노드를 나타내며 그 위에 존재하는 노드는 제거 된 후 새로 삽입된 노드이다.



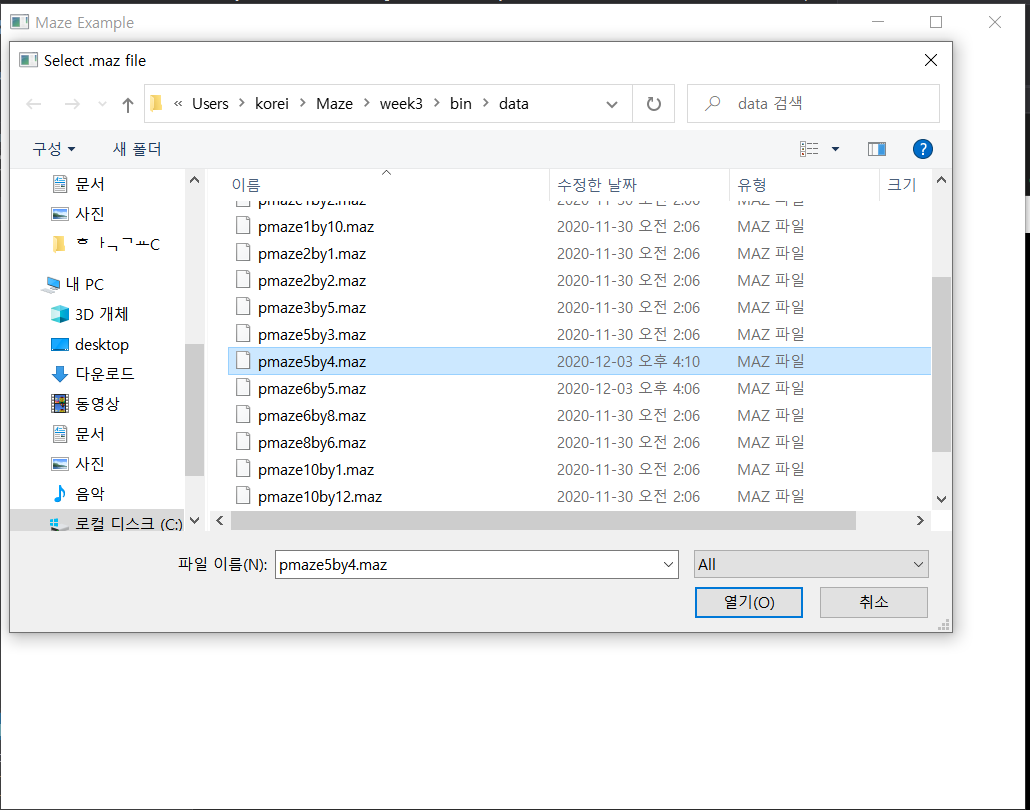
스택에 남은 정점과 제거된 정점을 각각 초록색과 빨간색으로 구분하여 트리에 표시하면 다음과 같다. 시작 정점과 도착 정점은 편의 상 다른 색으로 표시했지만 초록색과 같다.



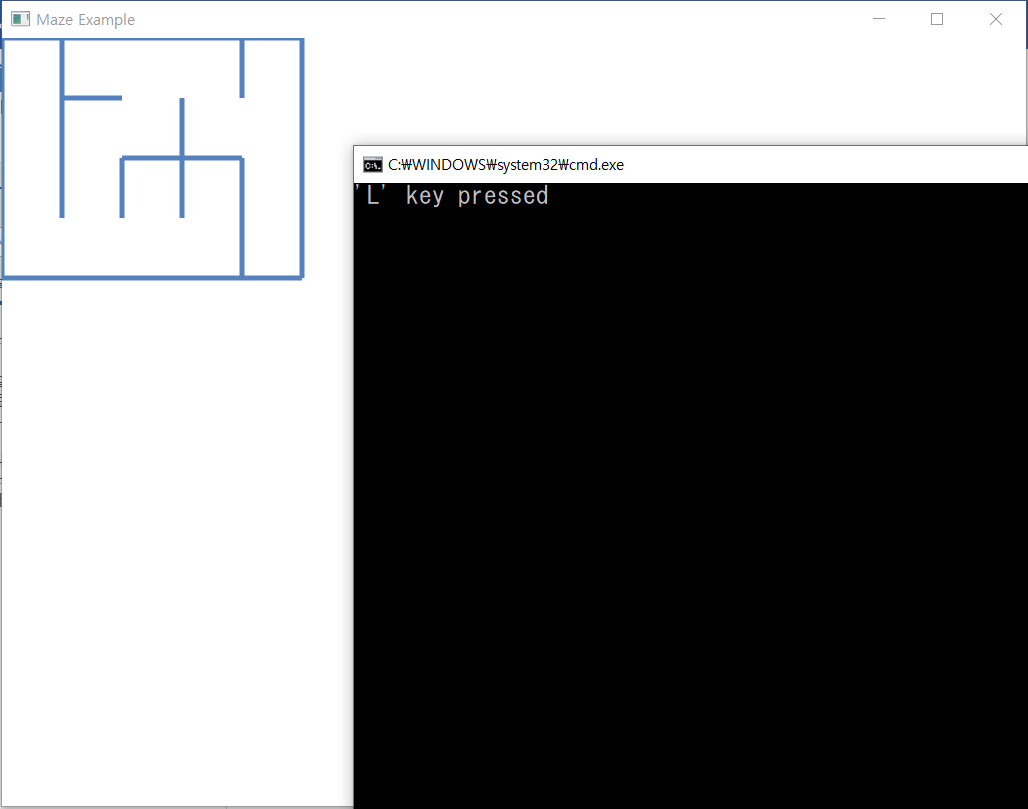
칠해지지 않은 정점은 방문하지 않은 정점이며 빨간 정점은 방문했지만 경로에 포함되지 않은 정점이다. 미로에 표시하면 다음과 같다.



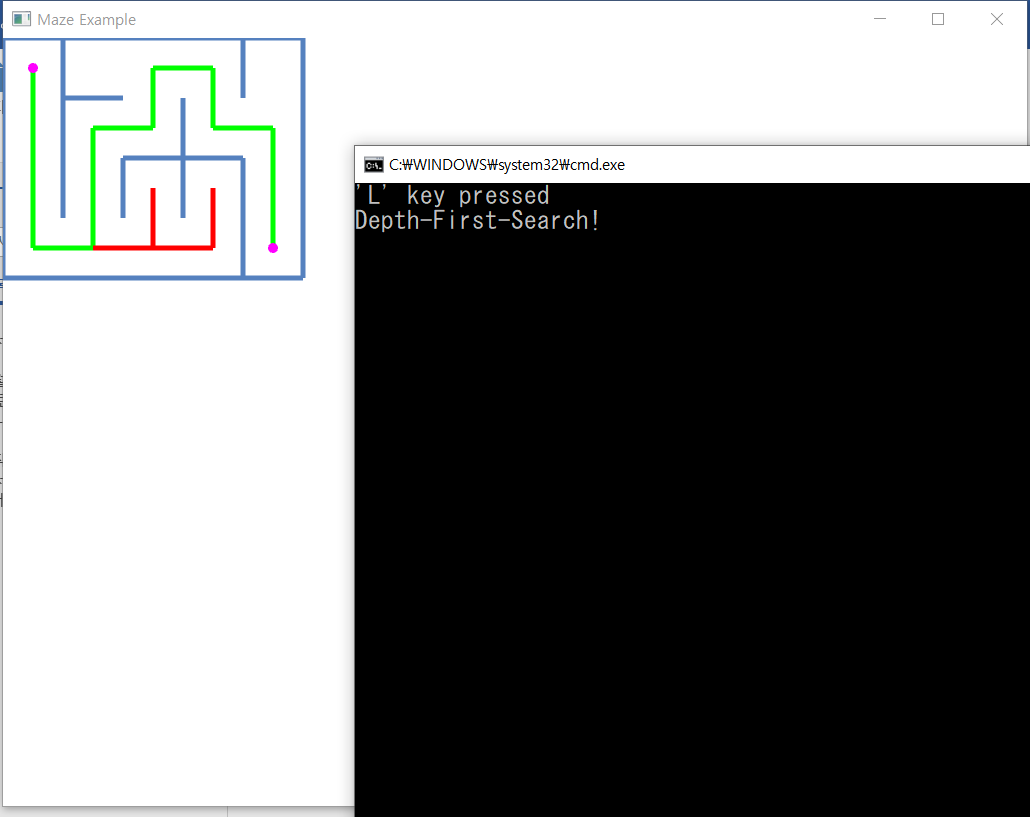
실제로 위와 같이 그려지는지 확인해보면 다음과 같다.



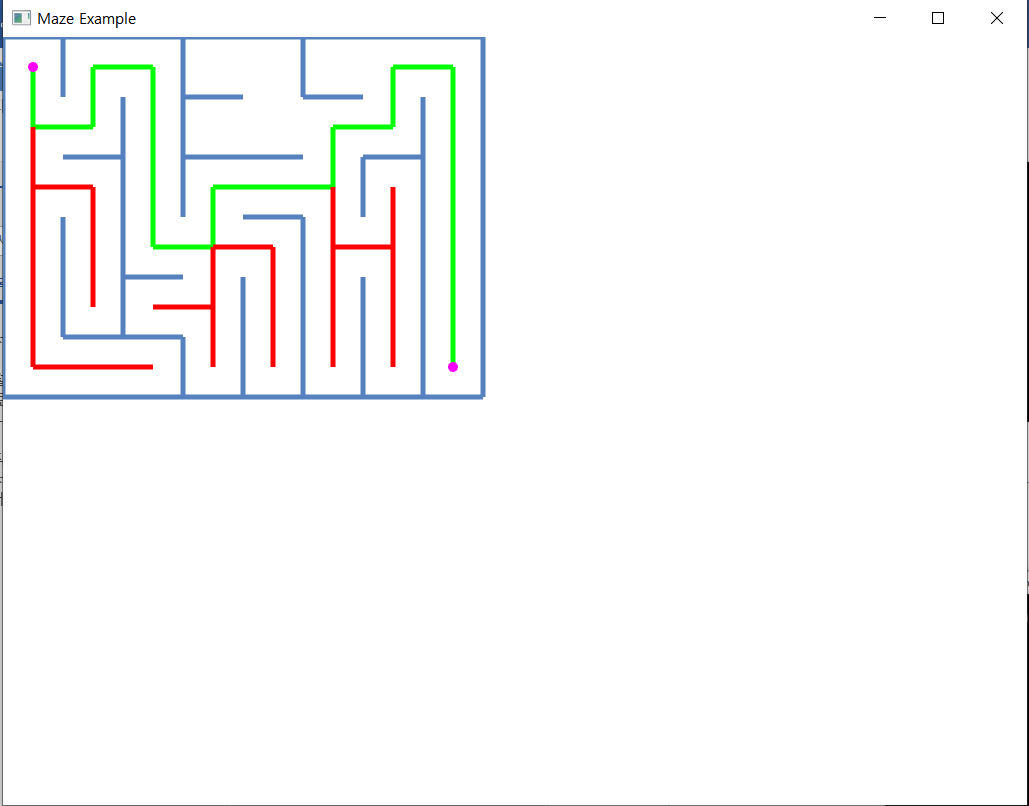
‘L’ 키를 눌러 해당 미로를 선택한 후 미로를 그린다.



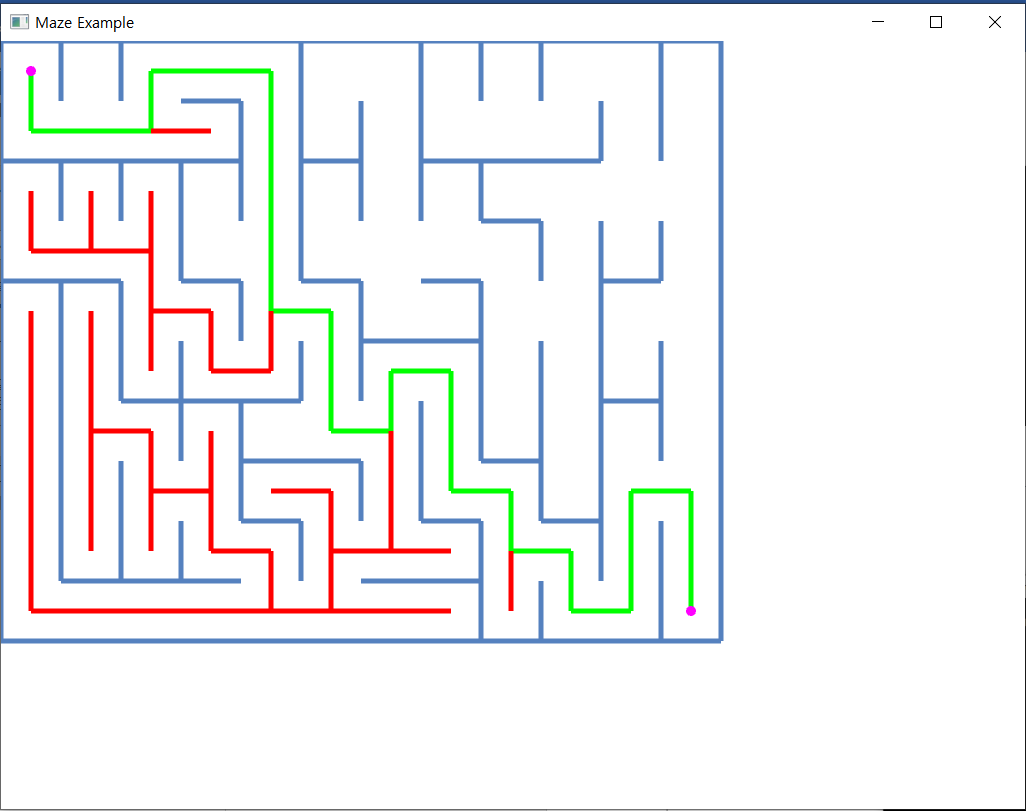
미로가 그려진 상태에서 ‘D’ 키를 누르면



콘솔에 메시지가 출력되며 위와 같이 경로가 그려진다. 경로는 초록색으로, 경로가 아닌 길은 빨간색으로 그려진다. 추가적으로 출발점과 도착점은 보라색 점으로 표현되도록 했다.



8\*6 미로

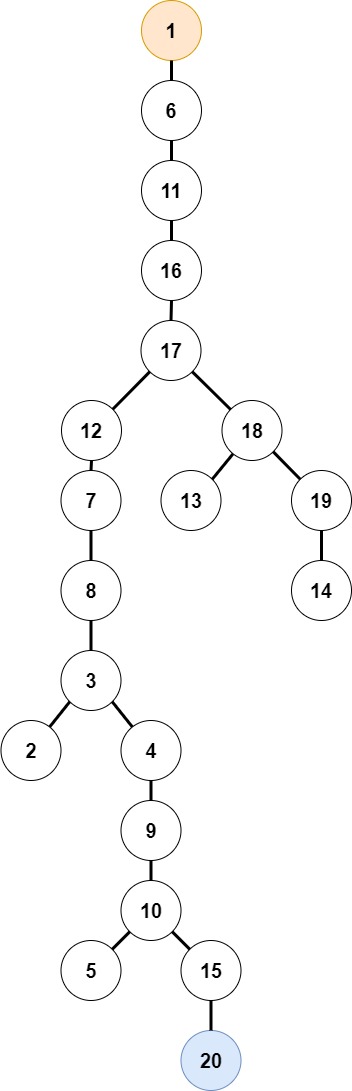


12\*10 미로

더 큰 미로에서도 경로가 잘 그려진다.

과제 결과

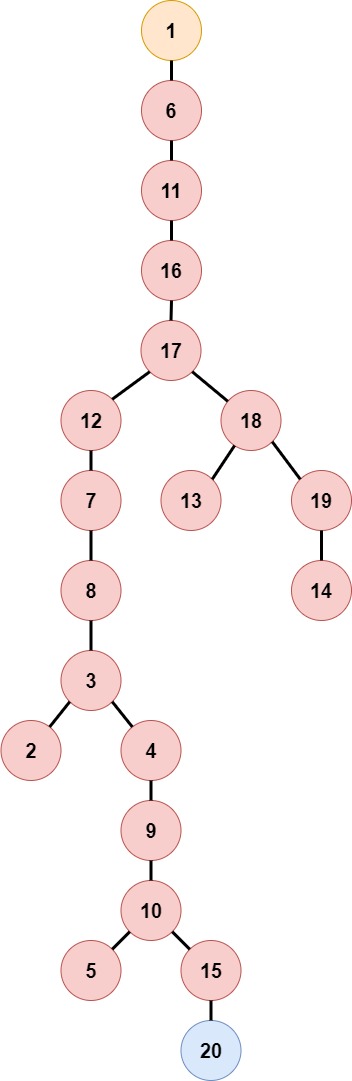
실습때와 같은 미로를 이용해 결과를 확인한다. 미로를 트리로 나타내면 다음과 같다.



위의 트리를 BFS를 이용하여 DFS 때와 같은 우선 순위로 탐색하면 방문 순서는 다음과 같다.

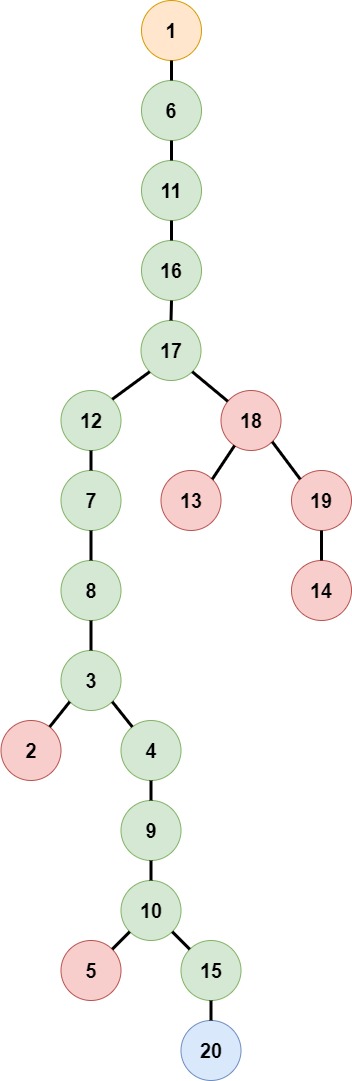
|  |
| --- |
| 1->6->11->16->17->18->12->19->13->7->14->8->3->4->2->9->10->15->5->20 |

DFS 때와 다르게 모든 정점을 방문하게 된다. 방문한 정점을 빨갛게 칠하면 다음과 같다.

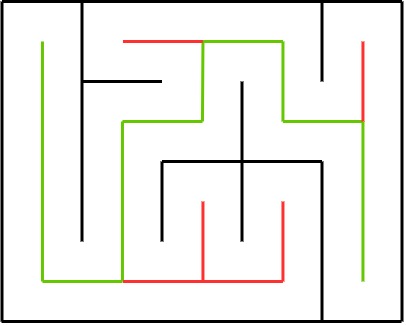


시작 정점과 도착 정점, 그리고 빨갛게 칠한 정점들은 모두 큐에 삽입된 적이 있는 정점들이다.

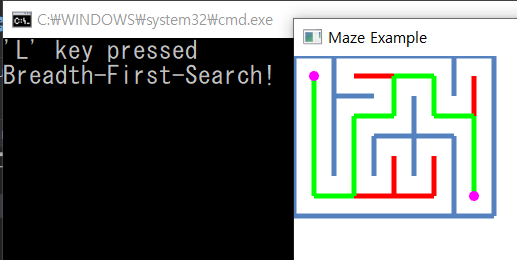
이 중에서 실제 경로에 포함되는 정점들은 도착 정점에서부터 시작 정점까지 부모 자식으로 연결된 정점들이다. 초록색으로 표시하면 다음과 같다.



미로에 표시하면 다음과 같다.ㅇ

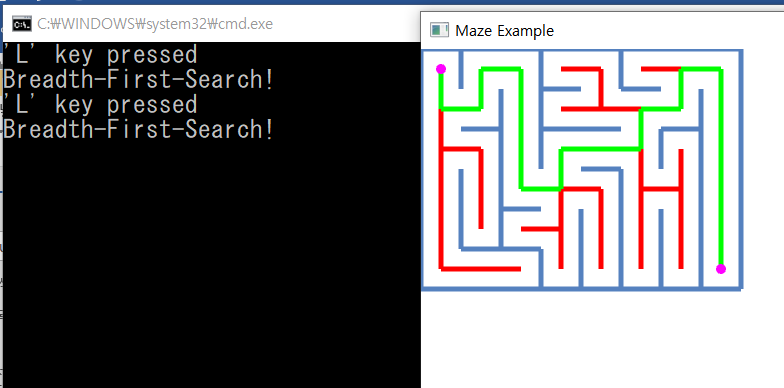


실제로 위와 같이 그려지는 지 확인해보면 다음과 같다.

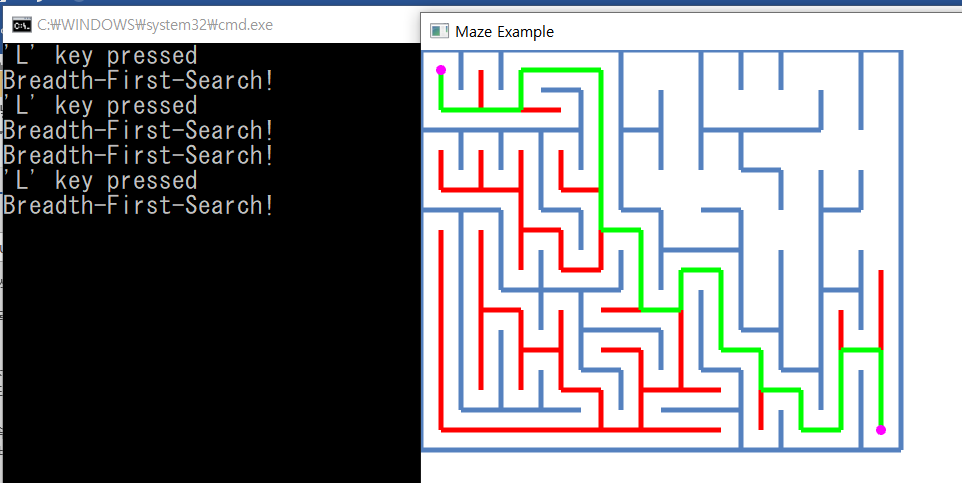


‘L’ 키를 눌러 파일을 열어 미로를 그린 후 ‘B’ 키를 누르면 BFS로 찾은 결과가 그려진다. 위에서 설명한 것과 같이 그려지는 것을 알 수 있다.

DFS의 경우와 마찬가지로 더 큰 미로 8\*6 미로와 12\*10 미로에 대해서도 BFS 탐색 결과를 그려보면 다음과 같다.



8\*6 미로

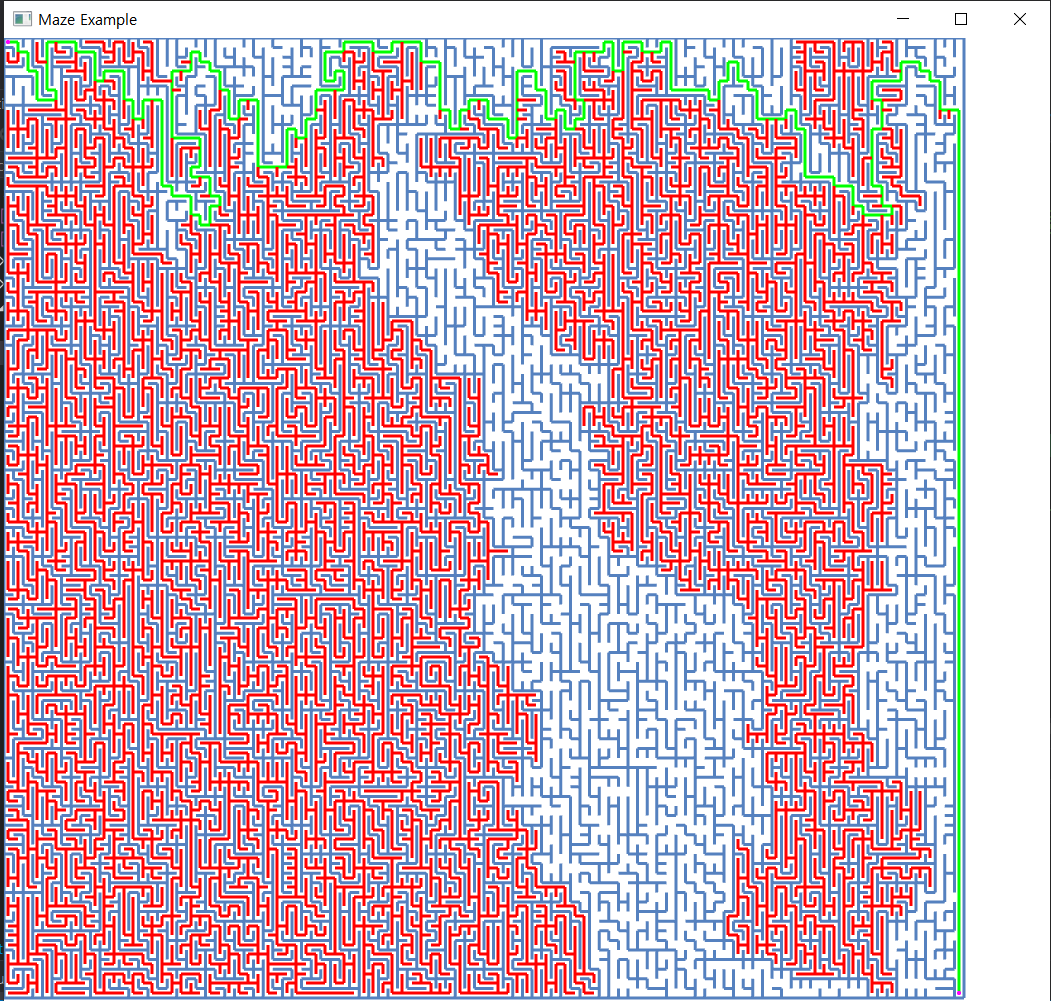


12\*10 미로

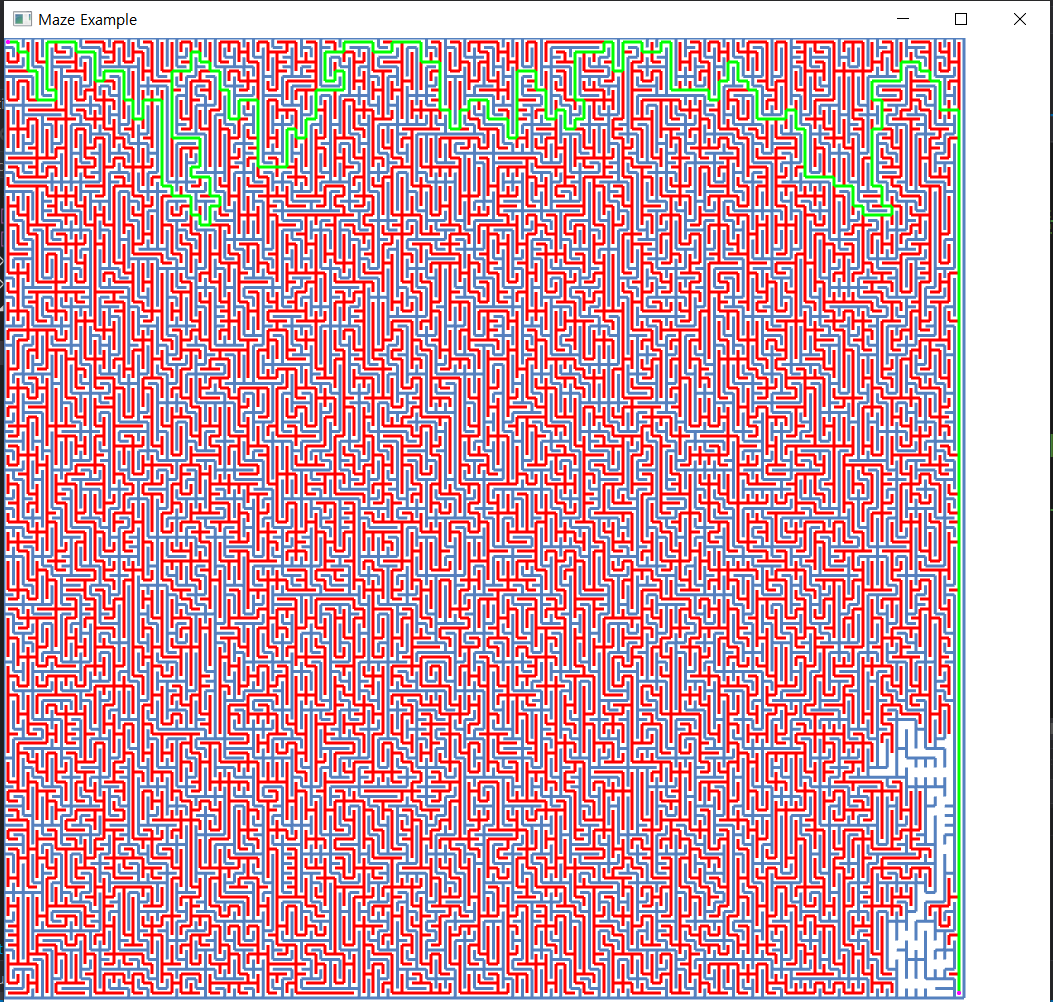
BFS로 찾은 결과도 잘 그려진다.

완전 미로에서 DFS와 BFS로 찾은 미로 탐색 결과를 비교해보면 같은 탐색 우선 순위로 했을 경우 BFS가 경로에 포함되지 않는 방을 더 많이 탐색하는 것을 알 수 있다. DFS의 경우 현재 위치에서 방문할 수 있는 방이 여러 개일 경우 우선 순위가 높은 방을 먼저 방문 후 이동하여 해당 방에서 이동할 수 있는 방들을 고려한다. 우선 순위가 높은 방의 정점은 스택에 삽입되고 바로 스택에서 제거되어 우선 순위가 높은 방으로 이동하기 때문에 우선 순위가 낮은 방들은 나중에 고려된다. 따라서 우선 순위를 따라 이동한 경로 상에 목적지가 있을 경우 다른 방들은 고려하지 않고 상대적으로 빠르게 찾을 수 있다. 하지만 BFS는 현재 위치에서 방문할 수 있는 방이 여러 개일 경우 우선 순위에 따라 방문 후 큐에 삽입하기는 하지만 결국 모든 인접 방을 방문 후, 큐에서 제거된 다음 방으로 이동하기 때문에 한 길로 집중적으로 탐색하지 않는다. 따라서 DFS처럼 잘못된 길로 깊게 빠지는 일은 일어나지 않지만 일반적으로 더 많은 방을 탐색해야 한다.

더 큰 미로(100\*100 미로)에서 두 방법의 차이를 확인해보면 다음과 같다.



DFS로 임의의 100\*100 미로를 탐색한 결과이다. 100\*100 미로가 한 화면에 나올 수 있도록 ofApp.cpp 에 선언된 미로의 방의 중심을 결정하는 변수 r을 4.8로 설정하였다. DFS로 탐색할 경우 불필요한 방을 많이 방문하긴 했지만 방문하지 않은 방도 많다는 것을 알 수 있다. 아래의 BFS의 결과와 비교해보면 상대적으로 효율적인 탐색이 이루어졌음을 알 수 있다.



반면 BFS로 탐색했을 경우 거의 모든 방을 방문하였다. 위의 DFS 결과와 비교했을 때 비효율적인 탐색이 이루어 진 것을 알 수 있다.