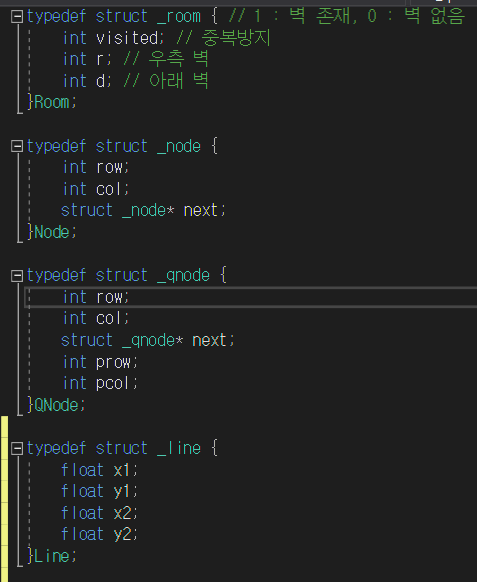
13주차 결과보고서

전공: 기계공학과 학년: 3학년 학번: 20191820 이름: 김형준

**1.**



DFS 및 BFS를 구현하고, 탐색경로와 탈출경로를 표시하기 위한 구조체들을 정의하였다.

Node : (DFS에서 사용되는 연결리스트 스택)

row : 방의 세로 좌표 (i)

col : 방의 가로 좌표 (j)

next : 다음 노드를 가리키는 포인터

QNode : (BFS에서 사용되는 연결리스트 큐, 경로탐색 후 단순 연결리스트로 사용됨)

row : 방의 세로 좌표 (i)

col : 방의 가로 좌표 (j)

next : 다음 노드를 가리키는 포인터

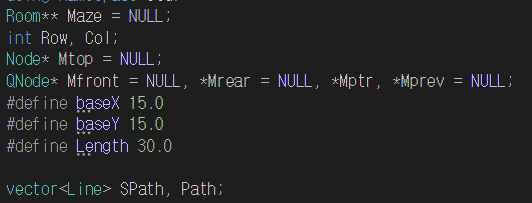
prow : 이전 방의 세로 좌표 (i)

pcol : 이전 방의 가로 좌표 (j)

Line : 경로를 그리기 위해 선분의 좌표를 저장하는 구조체

x1, y1 : 시작점의 x, y좌표

x2, y2 : 끝점의 x, y좌표



Maze : 미로 데이터를 저장하는 포인터 (2차원 배열로 동적 할당됨)

Row, Col : 각각 미로의 세로, 가로 크기

Mtop : 스택에서의 top 포인터

Mfront, Mrear : 큐에서의 front, rear 포인터

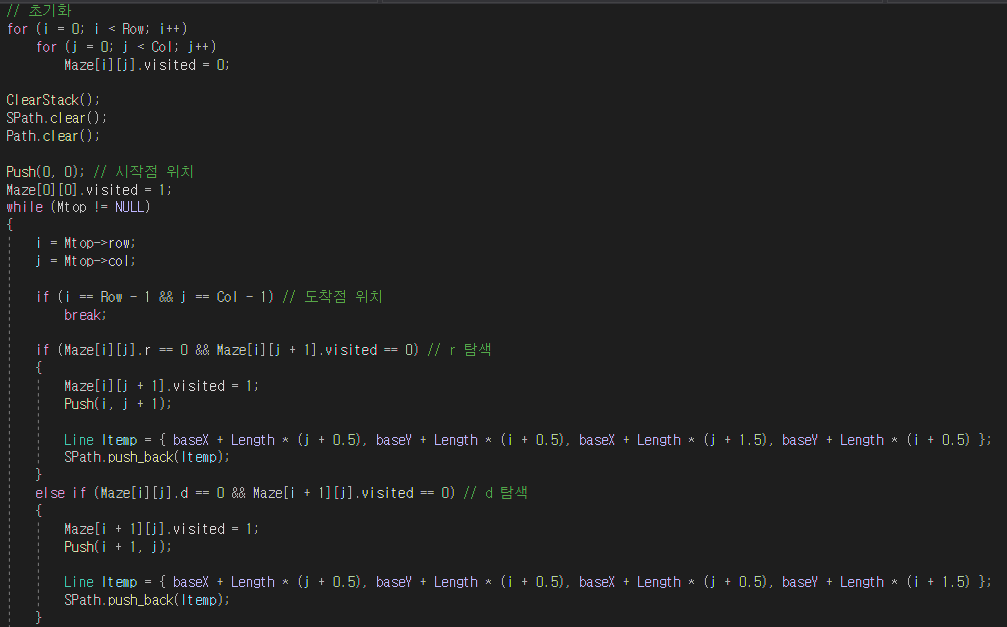
Mptr : 경로탐색 후 큐를 단순 연결리스트로 사용할 때의 첫 번째 노드를 가리키는 포인터

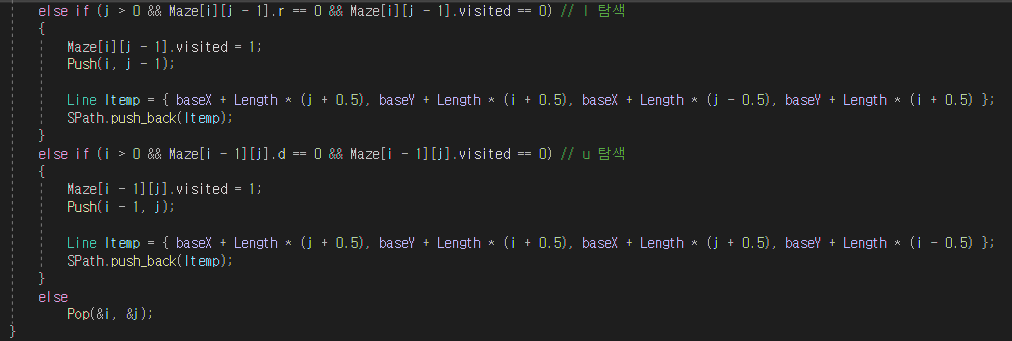
Mprev : 큐에서 Dequeue로 삭제된 노드의 next를 잇기 위해 사용하는 임시 포인터

SPath : 탐색 경로의 좌표를 저장하는 Line타입 벡터

Path : 탈출 경로의 좌표를 저장하는 Line타입 벡터

DFS :



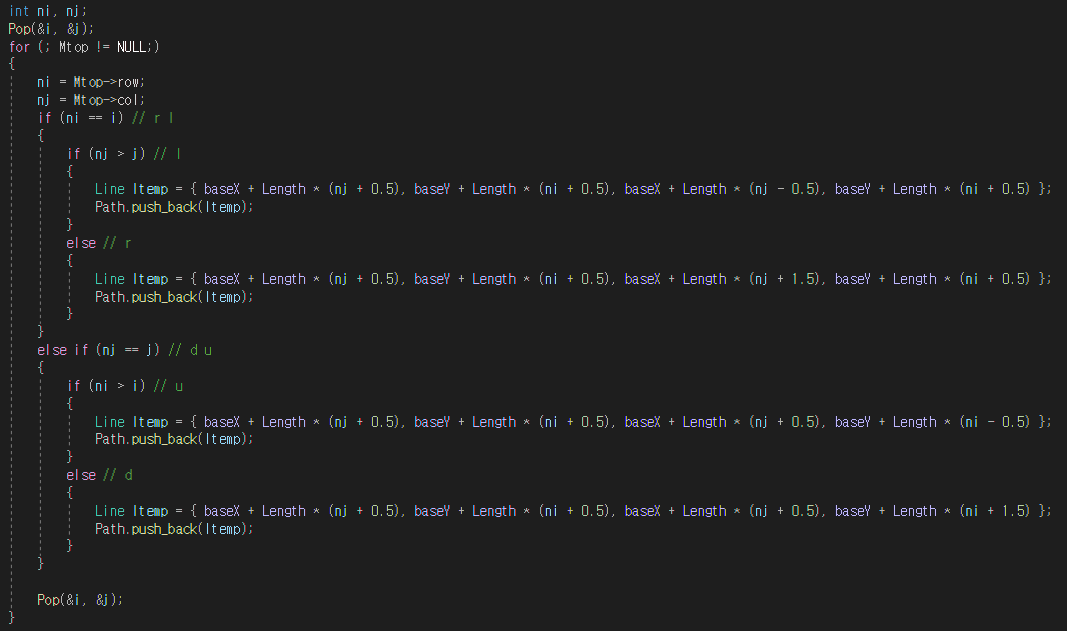


DFS는 예비보고서에 있는 내용과 거의 유사하게 구현하였다.

DFS 탐색 실행시 미로 데이터의 모든 원소의 visited를 0으로 초기화 하고 (방문 안함으로 초기화), 미로의 시작점을 스택에 넣고 (Push(0,0)) 시작 방의 visited를 1로 설정한다.

그 다음 스택의 맨 위쪽 원소의 값으로 i, j(i=세로, j=가로)의 값을 갱신하고, 상( (i-1,j)의 아래 벽이 없는 경우), 하( (i,j)의 아래 벽이 없는 경우), 좌( (i,j-1)의 우측 벽이 없는 경우), 우( (i,j)의 우측 벽이 없는 경우) 방향을 각각 탐색하면서 탐색하는 방향의 방이 아직 방문하지 않은 방인 경우 (visited가 0인 경우) 해당 방의 visited를 1로 설정, 좌표를 스택에 넣고 (Push() 사용, 이동할 방의 위치로 Push함), 현재 방-다음 방의 좌표를 각각 꼭짓점으로 가지는 선분의 좌표를 계산하여 SPath에 추가한다. (탐색 경로에 추가시킴)

모든 탐색 방향에서 더 이상 진행 할 수 없는 경우 (현재 방에서 상하좌우 방향으로 ‘벽으로 막혀있지 않고, 이미 방문하지 않은 방’이 하나도 없는 경우) Pop하여 스택에서 원소를 하나 제거한다. 이를 스택에 남은 노드가 없거나 현재 위치가 도착점 (Row-1,Col-1)에 도달할 때까지 반복한다.



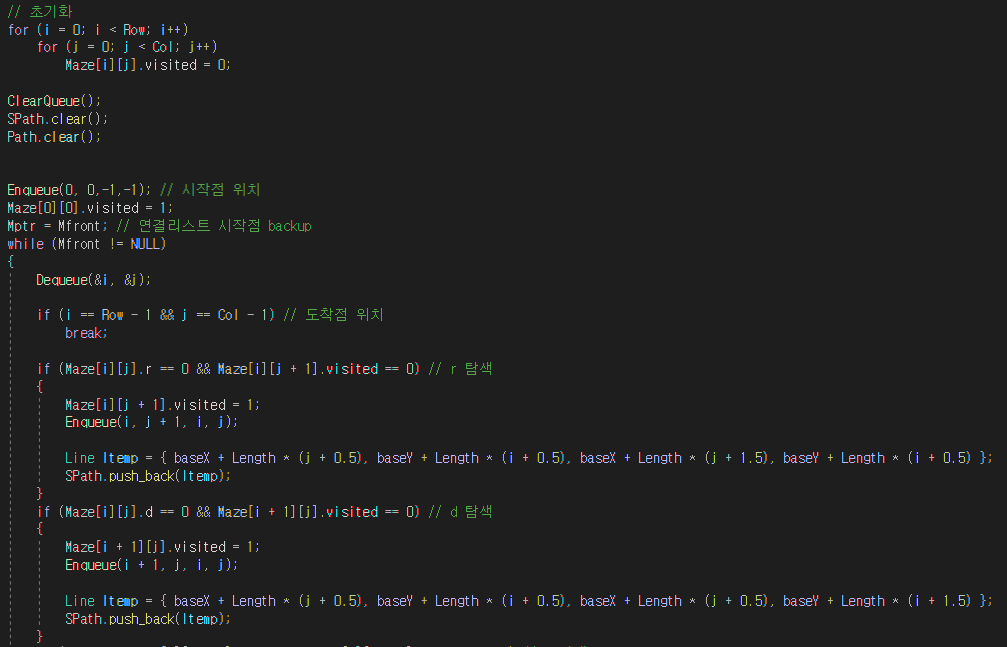
주어지는 미로는 완전미로이므로 DFS를 사용했을 때 항상 탈출 경로를 찾을 수 있다.

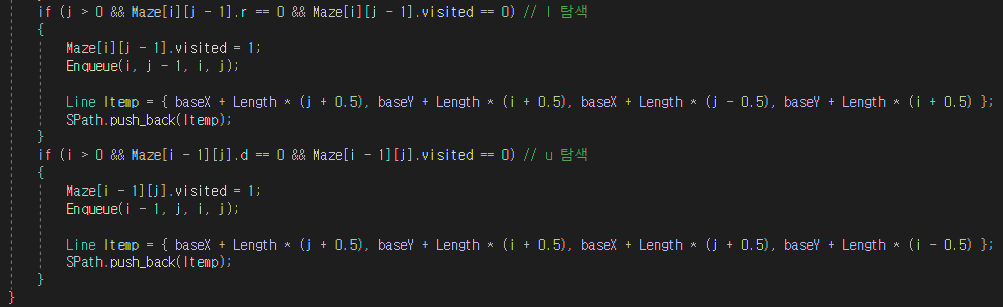
따라서 스택에는 맨 위부터 맨 아래의 노드에 차례대로 도착점부터 시작점까지의 탈출경로가 그대로 저장되어 있으므로, Mtop (스택의 top 포인터)를 사용해 스택에 남아있는 모든 노드들을 Pop()시키면서 경로를 좌표를 계산하여 Path에 추가한다. (탈출경로에 추가함)

DFS 탐색 종료시 SPath와 Path에 있는 선분의 정보를 사용해 ofDrawLine로 탈출경로와 탐색경로 각각을 색상을 구분해서 그린다.

미로의 가로, 세로 크기를 각각 Col, Row라 하면 방의 개수 및 가능한 경로의 총 길이는 O(Row\*Col)이므로 구현한 DFS 탐색 알고리즘의 시간복잡도는 O(Row\*Col)이고, 공간복잡도는 O(Row\*Col)이다.

BFS :





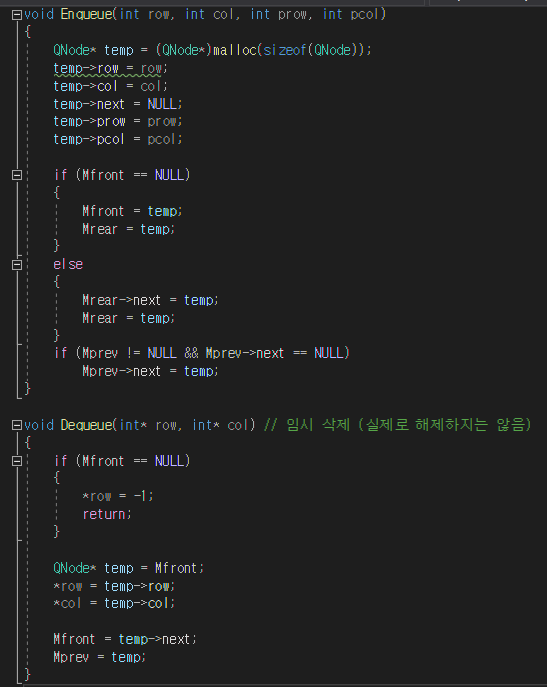
BFS도 예비보고서에 있는 내용과 거의 유사하게 구현하였으나, 탈출 경로를 그리기 위하여 큐로 사용하는 구조체에 prow, pcol 멤버변수를 추가했고, 몇 가지 연산을 추가하였다.

BFS 탐색 실행시 미로 데이터의 모든 원소의 visited를 0으로 초기화 하고 (방문 안함으로 초기화), 미로의 시작점을 큐에 넣고 (Enqueue(0,0,-1,-1), 시작점의 이전 방은 없으므로 -1, -1를 이전 방의 좌표로 설정함) 시작 방의 visited를 1로 설정한 다음 큐를 탐색 후 단순 연결리스트로 사용하기 위해 Mptr에 Mfront의 값을 저장한다.

그 다음 Dequeue로 큐의 원소를 하나 제거하고, 제거된 원소의 값으로 i, j의 값을 갱신한 다음 상( (i-1,j)의 아래 벽이 없는 경우), 하( (i,j)의 아래 벽이 없는 경우), 좌( (i,j-1)의 우측 벽이 없는 경우), 우( (i,j)의 우측 벽이 없는 경우) 방향을 각각 탐색하면서 탐색하는 방향의 방이 아직 방문하지 않은 방인 경우 (visited가 0인 경우) 해당 방의 visited를 1로 설정, 좌표를 큐에 넣고 (Enqueue() 사용, 이동할 방의 위치로 row, col의 값을, 현재 방의 위치로 prow, pcol의 값을 설정함), 현재 방-다음 방의 좌표를 각각 꼭짓점으로 가지는 선분의 좌표를 계산하여 SPath에 추가한다. (탐색 경로에 추가시킴)

DFS와 다르게 한 탐색방향에서 조건을 만족해 큐에 원소를 추가하더라도 나머지 탐색방향으로 계속 탐색을 진행한다.

이를 큐에 남은 노드가 없거나 현재 위치가 도착점 (Row-1,Col-1)에 도달할 때까지 반복한다.

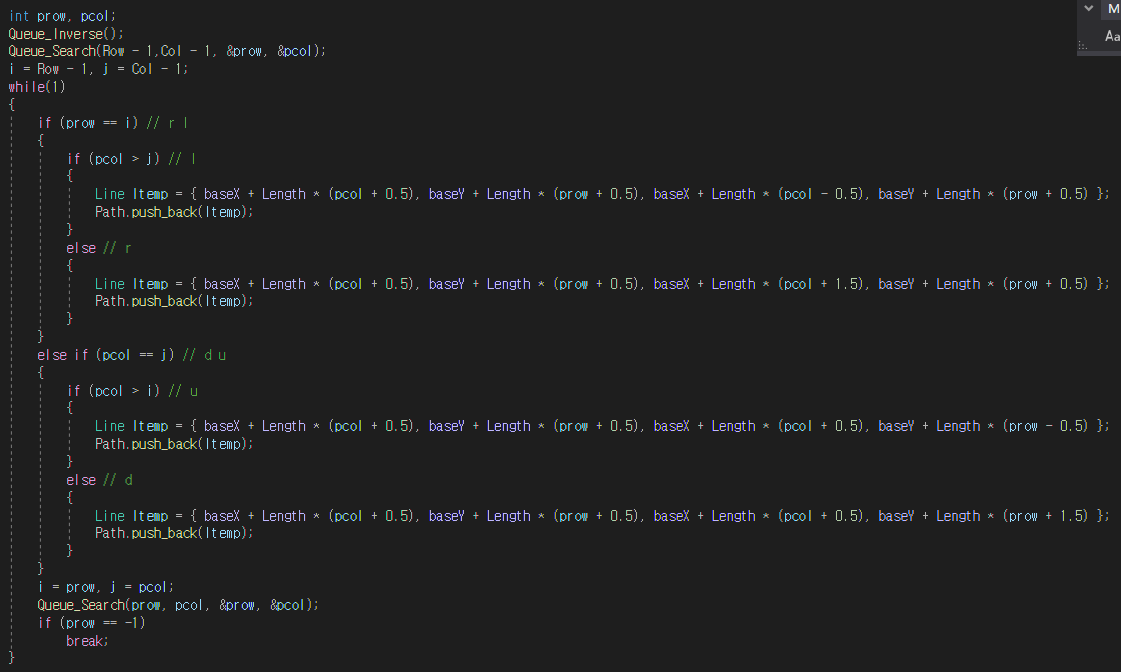


탈출경로를 저장하기 위해 일반적인 연결리스트 큐와 다르게 Enqueue(), Dequeue()를 구현하였다. Dequeue()의 경우 연결리스트에서 노드를 바로 삭제하는 대신, Mfront를 다음 노드로 옮기는 것으로 ‘큐에서의 삭제’를 구현하였다. 삭제 후, MPrev를 삭제한 노드의 주소로 설정한다.

Enqueue()의 경우 연결리스트에 노드를 추가하고, Mprev가 가리키는 노드가 존재하고, 해당 노드의 next가 NULL인 경우 연결리스트를 한 줄로 모두 연결하기 위해 Mptr->next = temp로 (마지막으로 삭제처리한 노드)와 (새로 생성한 노드)를 연결한다.

Dequeue()를 연속으로 2회이상 호출하는 경우, 삭제 처리된 노드들 끼리는 next가 다음 노드로 연결되어 있으므로 마지막으로 삭제처리한 노드와 새로 생성한 노드만을 연결해주면 된다.

경로 탐색이후 큐로 사용했던 연결리스트의 구조는 최종적으로 Mptr -> (0,0,-1,-1) (시작점 노드) -> … -> (마지막 노드) -> NULL의 순서로 이루어지게 된다.



이 상태에서 Queue\_Inverse()를 호출해 Mptr이 가리키는 연결리스트를 뒤집고 (모든 노드를 반대방향으로 연결 후, Mptr을 뒤집기 전의 마지막 노드를 가리키게 함)

Queue\_Search()로 도착지점의 좌표가 저장된 노드를 찾고 해당 노드에 저장된 prow, pcol값으로 prow, pcol의 값을 갱신한다. (Queue\_Search()는 Mptr의 첫 번째 노드부터 노드의 row와 col값이 각각 입력된 row와 col의 값과 일치하는 노드에 도달할 때까지 노드를 Mptr에서 한 개씩 삭제(할당 해제)하면서 연결리스트를 순회한다, 도달시 해당 노드에 저장된 prow, pcol값을 입력된 포인터에 리턴하고, 연결리스트에 남은 노드가 없는 경우 각각 -1을 리턴함)

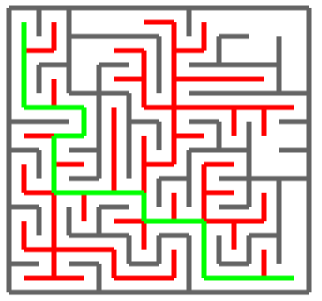
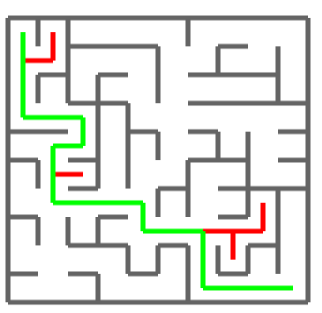
그 다음 i(row값), j(col값)과 prow, pcol값을 사용해 탈출경로의 선분 좌표를 계산하고 Path에 추가한다. (탈출경로에 추가시킴)

i, j에 각각 prow, pcol값을 저장해 값을 갱신하고 prow, pcol값을 Queue\_Search()의 row, col값으로 넣어서 호출한 다음 prow, pcol의 값을 갱신해, prow가 -1이 될 때까지 반복한다. (경로를 탐색하면서 생성된 노드들의 이전 위치를 도착점부터 따라가다 보면 탈출경로가 1개이상 존재하므로 반드시 시작점에 도달하게됨)

BFS 탐색 종료시 SPath와 Path에 있는 선분의 정보를 사용해 ofDrawLine로 탈출경로와 탐색경로 각각을 색상을 구분해서 그린다.

미로의 가로, 세로 크기를 각각 Col, Row라 하면 방의 개수 및 가능한 경로의 총 길이는 O(Row\*Col)이므로 구현한 BFS 탐색 알고리즘의 시간복잡도는 O(Row\*Col)이고, 공간복잡도는 O(Row\*Col)이다.

**2.**



여러 개의 미로를 생성해서 실험 해본 결과 위 그림(좌 DFS, 우 BFS)과 같이 Eller’s 알고리즘으로 만들어진 미로에서는 BFS의 탐색 횟수가 DFS의 탐색 횟수보다 많다는 것을 알 수 있었다.

두 방법의 시간 복잡도는 같으나, Eller’s 알고리즘을 사용하여 생성된 미로에서는 이러한 양상을 보이는 것 같다. 또한, 정답 경로를 그릴 때 DFS의 경우에는 도착지점에 도달시 스택에 남아있는 원소들의 좌표대로 쭉 따라가면 도착점부터 시작점까지의 경로를 바로 그릴 수 있는데, BFS의 경우에는 진행하면서 도착지점까지 도달만 할뿐, 경로를 저장하지 않지는 않기 때문에 경로를 그리기 위해 추가적인 연산이 필요하였다. 따라서 이번주차 실습과 과제를 해결하는데는 DFS가 더 적합하다. 만약, 완전미로가 아닌 불완전 미로를 탐색할 경우, DFS는 최단 경로를 보장할 수 없으므로 그 경우에는 BFS가 적합할 것 같다.