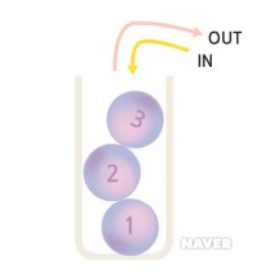
**BFS와 DFS 방식을 이용한 미로 해결**

**2018036029 김민정**

**1. 스택과 큐**

스택(stack)과 큐(queue)는 컴퓨터 프로그램에 많이 사용되는 가장 기본적인 데이터 타입으로 리스트의 특별한 경우이다.

먼저 스택(stack)은 “입력과 출력을 한 방향으로 제한한 알고리즘” 이라고 표현할 수 있다. 다시 말하면, “top” 이라고 하는 한쪽 끝에서 모든 삽입과 삭제가 일어나는 순서리스트다. stack에서 발생하는 삽입을 push (=add) 라고 하고 삭제를 pop(=remove) 라고 한다.

주어진 스택에서 a0는 가장 아래에 있는 원소이고 an-1은 가장 위쪽에 있는 원소이다. 만약 A, B, C, D, E 순서로 스택에 삽입했다면, 제일 먼저 삭제되는 원소는 E이다. 즉 후입선출(LIFO, Last-In First-Out)리스트 라고도 한다.

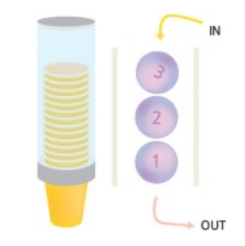
스택을 종이컵으로 비유해볼 수 있다. 아래 사진처럼 선반에 겹겹이 쌓인 종이컵은 한쪽은 막혀 있기 때문에 다른 한쪽으로만 물건을 넣거나 꺼내서 사용할 수 있다. 즉, 종이컵 기준으로 위쪽이 “top”이라고 지칭할 수 있다.

그렇다면 이러한 스택 구조를 컴퓨터에서 언제 사용하는지 알아보았다.

가장 대표적인 사례는 [되돌리기] 기능이다. 이 기능은 [프로그램](https://terms.naver.com/entry.nhn?docId=4369091&ref=y)을 사용하여 문서 편집 혹은 그림 편집을 하면서 방금 한 작업을 취소하기 위해 사용된다. 또한 [웹 브라우저](https://terms.naver.com/entry.nhn?docId=4369083&ref=y)에도 방금 전 방문했던 [사이트](https://terms.naver.com/entry.nhn?docId=4369082&ref=y) 기록을 저장해 두었다가 이전 페이지로 돌아갈 수 있다.

다음으로 큐(queue)는 한쪽 끝에서 삽입이 일어나고 그 반대쪽 끝에서 삭제가 일어나는 순서 리스트이다. 새로운 원소가 삽입되는 끝을 “rear”, 원소가 삭제되는 끝을 “front”라고 한다. 큐에서 발생하는 삽입을 put (=insertion, addition, push)라고 하고 삭제를 pop(=removal, deletion) 라고 한다.

만약 A, B, C, D, E 순서로 큐에 삽입했다면 제일 먼저 삭제되는 원소는 A이다. 즉, 먼저 입력된 작업을 먼저 처리하는 방식으로 선입선출(FIFO, First-In First-Out)리스트라고도 한다. 또한, 우리가 자주 사용하는 “선착순” 과 같은 의미이다.



큐(queue)도 종이컵에 비유해 쉽게 설명할 수 있다. 종이컵을 그림과 같은 디스펜서에 넣어서 하나씩 꺼내서 사용하는 경우, 가장 먼저 들어온 종이컵을 꺼내 쓰게 된다. 즉, 종이컵 기준으로 위쪽이 “rear”. 아래쪽을 “front” 이라고 지칭할 수 있다.

이러한 큐(queue) 구조를 컴퓨터는 어떤 작업에 사용하는지 알아보았다.

대표적으로 버퍼에 사용된다. [키보드](https://terms.naver.com/entry.nhn?docId=4369033&ref=y)로 글자를 [입력](https://terms.naver.com/entry.nhn?docId=4369038&ref=y)하면, 입력한 순서대로 화면에 나온다. 또한 인쇄물을 프린터로 [출력](https://terms.naver.com/entry.nhn?docId=4369044&ref=y)할 때, 출력 버튼을 누른 순서대로 출력된다. 이렇게 [컴퓨터](https://terms.naver.com/entry.nhn?docId=4369002&ref=y)가 입출력 장치들이 요구하는 작업을 처리하기 위해 작업들을 순서대로 모아두는 곳을 버퍼라고 합니다. 이러한 버퍼는 큐의 구조를 이용하는 대표적인 사례이다.

큐는 구현하는 과정이나 사용방식이 앞서 설명한 스택과 유사하다. 하지만 스택이 후입선출(LIFO, Last-In First-Out) 방식이라면 큐는 선입선출(FIFO, First-In First-Out) 방식이라는 점에서 차이가 있다.

**2. BFS와 DFS탐색 방식**

그래프 순회 란 **그래프**가 주어졌을 때, **그래프**의 모든 정점(=노드)들을 빠짐없이, 중복없이 한 번씩 찾아서 방문하는 일이다.

대표적인 방법으로는 노드를 방문하는 순서에 따라 너비 우선 탐색(BFS)과 깊이 우선 탐색(DFS) 이 두가지가 있다.

먼저, 너비 우선 탐색인 BFS(Breadth First Search)는 같은 레벨을 먼저 방문한 뒤 그 다음 레벨을 방문한다. 이러한 BFS(Breadth First Search)는 큐(queue )를 사용한 방법으로 생성된 순서에 따라 노드를 확정한다. 시작 정점을 출발로 방문한 정점을 큐 구조에 넣어가면서 탐색을 하는 방식이다. 한 정점에 연결된 모든 정점의 탐색이 끝나면 그 정점을 큐에서 제거한다. 그리고 큐의 다음 정점을 꺼내서 그 정점에 연결된 모든 정점을 탐색한다. 따라서 많은 기억 공간이 요구되는 탐색 방식이다.

너비 우선 탐색 방식 과정을 간략하게 다음과 같이 살펴볼 수 있다.

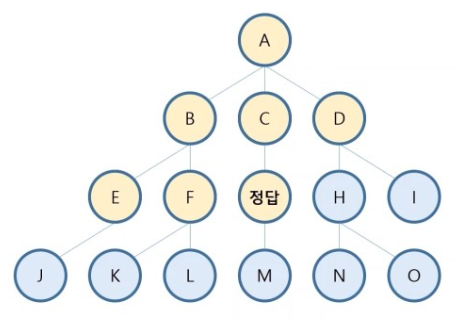
1. 시작 정점에서 출발

2. 인접한 노드를 모두 방문.

3. 인접한 노드를 모두 방문했다면, 다른 노드로 이동

4. 그 다른 노드에서 인접한 노드들 탐색.

위 과정을 노드가 전부 탐색될 때까지 반복한다.

​ 만약 왼쪽 그림과 같은 구조에서 너비 우선 탐색을 이용하면,

  “A -> B -> C -> D -> E -> F -> 정답” 의 과정으로 노드를 탐색한다.

너비 우선 방식 BFS(Breadth First Search)은 최단 경로에 유용하고 무한히 깊거나 무한에 가까운 트리인 경우에 사용하면 효과적이다. 또한, 특정 A 지점으로부터 B 지점까지의 거리, 또는 그래프를 전 탐색하는데 걸리는 최소 시간 등을 도출하고자 할 때 효과적이다. 하지만 반면 목표 노드로 가는 경로가 모두 같은 거리일 때는 비효율적인 방법이다.

깊이 우선 탐색인 DFS(Depth First Search)는 하나의 경우에 대하여 모든 경우의 수를 조사하고 다음 경우의 수를 조사하면서 해를 찾는 과정이다. 즉 현재 정점에 연결된 정점이 존재하면 계속해서 깊숙이 이동한다. 연결된 노드가 더 이상 없으면 나아갈 곳이 있는 곳으로 이동하고, 만약 너무 깊게 들어가면 Overflow가 발생할 수도 있다.

이러한 DFS(Depth First Search)방식은 스택(stack)을 이용한다. 재귀호출을 이용해 방문한 정점들을 스택에 저장한다. 재귀호출을 이용할 때마다 더 깊숙한 위치로 이동한다.

깊이 우선 탐색 방식 과정을 간략하게 다음과 같이 살펴볼 수 있다.

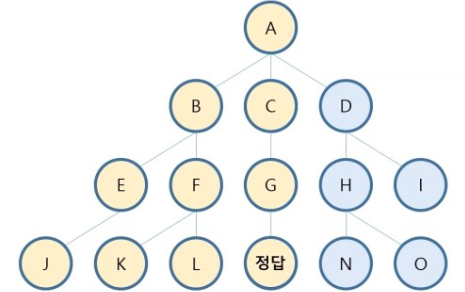
1. 시작 노드에서 시작

2.인접한 다른 노드 탐색

3.탐색한 노드에서 인접한 노드 중 탐색되지 않은 노드 탐색

4.인접한 노드 중 탐색되지 않은 노드가 없으면 다시 되돌아오면서 탐색되지 않은 노드 확인

5.만약 탐색되지 않은 노드가 있다면 탐색

위와 같은 과정을 모든 노드가 탐색될 때까지 반복한다.

만약 왼쪽 그림과 같은 구조에서 깊이 우선 탐색을 이용하면, 노드 탐색 과정은

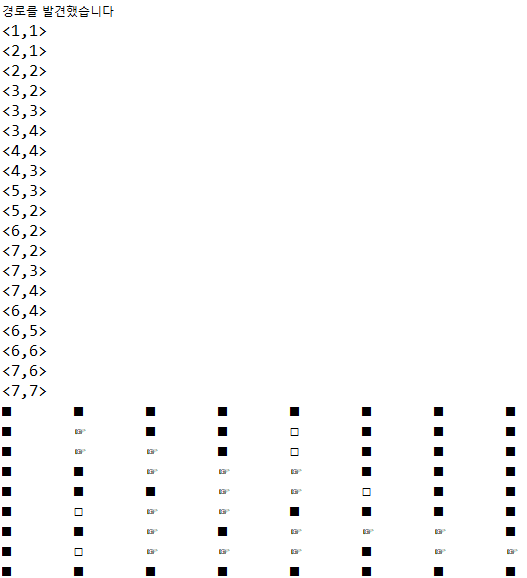
“A -> B -> E- > J -> F -> K -> L -> C -> G -> 정답”

이 될 것이다.

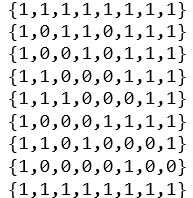
**이러한 깊이 우선 탐색 방식은 무한히 넓은 트리에 효과적인 방식이다. 하지만 목적지가 없는 경로에 깊이 빠지면 overflow가 발생 할 수도 있다는 단점이 있다.**

**최종적으로 너비 우선 방식** BFS(Breadth First Search)**과 깊이 우선 방식** DFS(Depth First Search)**을 정리하면,** 너비 우선 탐색 방식은 현재 노드에서 탐색할 수 있는 모든 노드를 탐색한 뒤에 다음으로 넘어가지만, 깊이 우선 탐색 방식은 새로운 노드가 탐색될 때마다 그 노드로 이동하고, 다시 돌아와서 새로운 노드를 탐색한다. 또한 너비 우선 탐색 방식은 큐(queue )를 사용하고, **깊이 우선 방식은 스택(stack)을 사용해 구현할 수 있다.**

**3. 미로 해결**

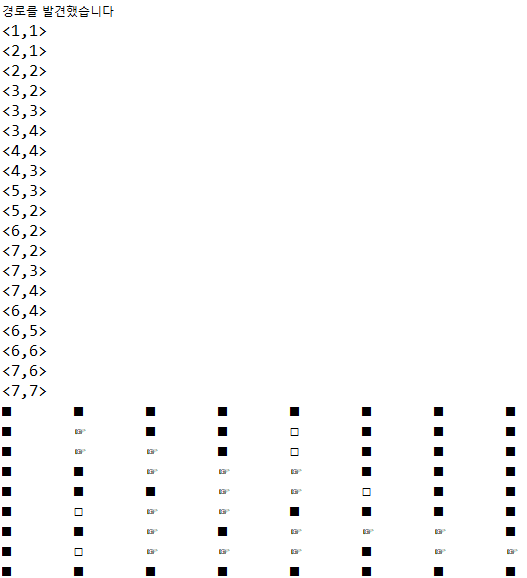
 **너비 우선 방식** BFS(Breadth First Search)**과 깊이 우선 방식** DFS(Depth First Search)**을 활용하여** 미로문제를 해결할 수 있다. 다음과 같은 미로가 주어졌을 때 이를 해결하기 위해 1. MazePath() 2. displayMaze( ) 3. BreadthFirstTraversal( )

4. DepthFirstTraversal( ) 5. main( ) 를 사용하였다. 하나씩 살펴보면,

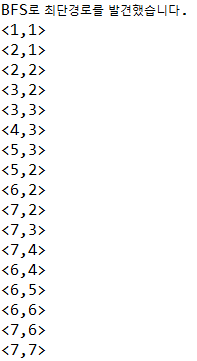


먼저, 1. MazePath()에 미로 탈출 경로를 찾을 수 있는 코드가 있다. 이를 실행하면 다음과 같은 경로가 출력된다. 하지만 이는 최단경로는 아니다.

2. displayMaze( ) 함수는 1번으로 얻은 경로를 문자로 나타내는 함수이다. 경로가 위 사진과 같을 때, 이 함수를 호출하면 그림과 같은 미로 모양이 출력된다.

여기서 “■” 이 표시는 막힌 벽을 나타낸다. 즉, 1 을 표현한다. “□” 표시는 이동 할 수 있는 블록을 나타낸다. 즉, 0을 표현한다.

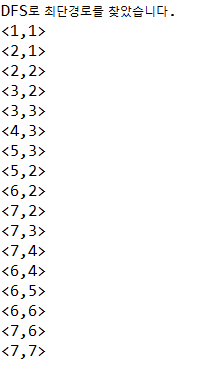
마지막으로 ”☞” 표시는 이동한 경로를 나타낸다.

3. BreadthFirstTraversal( ) 함수는 너비 우선 탐색을 이용해 미로를 통과하는 함수이다.

이 함수는 BFS를 위한 Queue를 생성해 구현한다. 최단 경로를 발견했을 시, "BFS로 최단경로를 발견했습니다." 라는 문구를 출력하고 발견하지 못했다면 "BFS로 경로를 발견하지 못했습니다.” 를 출력하도록 한다.

위 미로에 대해 BreadthFirstTraversal( )함수를 실행하면 다음과 같은 출력 결과를 얻을 수 있다. 이 방법과 아까 최단경로가 아닌 1번 방법과 경로를 비교해 보면, 이 방법은 불필요한 경로는 거치지 않는 최단 경로임을 알 수 있다.

4. DepthFirstTraversal( ) 함수는 깊이 우선 탐색을 이용해 미로를 통과하는 함수이다.

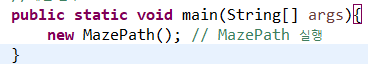


이 함수는 1.BreadthFirstTraversal( )와 마찬가지로, 최단 경로를 발견했을 시, " DFS로 최단경로를 발견했습니다." 라는 문구를 출력하고 발견하지 못했다면 DFS로 경로를 발견하지 못했습니다.” 를 출력하도록 한다. 하지만 BreadthFirstTraversal( )와 다르게 stack을 생성해 구현한다.

앞선 미로에 대해 왼쪽과 같은 경로를 얻을 수 있다. 마찬가지로 1번 방법에 비해 경로가 적고, 이것이 즉, 최단 경로임을 알 수 있다.

마지막으로 이렇게 구한 함수들을 5. main( )에서 호출을 해주어야 한다.

2. displayMaze( ) 3. BreadthFirstTraversal( ) 4. DepthFirstTraversal( ) 함수들을 먼저 1. MazePath()에서 호출한 뒤 1. MazePath() 자체를 5. Main()에서 호출 하였다.

 자바에서는 class 달랑 하나로 구현해서 작동되는 것이 아니라 이러한 class들을 다시 실행해줄 수 있는 별도의 class 필요하다. 그래서 메인 함수에서 이 클래스를 인스턴스화해서 실행할 수 있도록 한다.

new MazePath( )는 MazePath 라는 함수의 생성자 부분이 자동으로 실행하도록 한다. 생성자 함수가 메인 함수에서 new해줄 때 자동으로 실행된다.