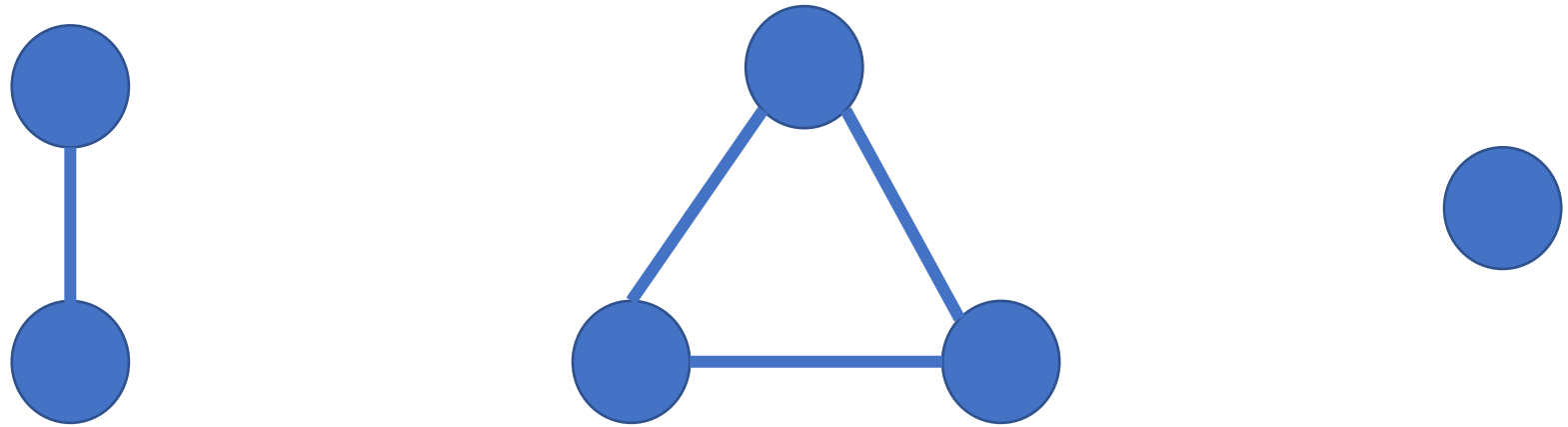
그래프 이론과 구현, DFS와 BFS

손찬영

1. 그래프 기초

그래프란?

- 정점과 간선들로 이루어진 집합을 말한다.

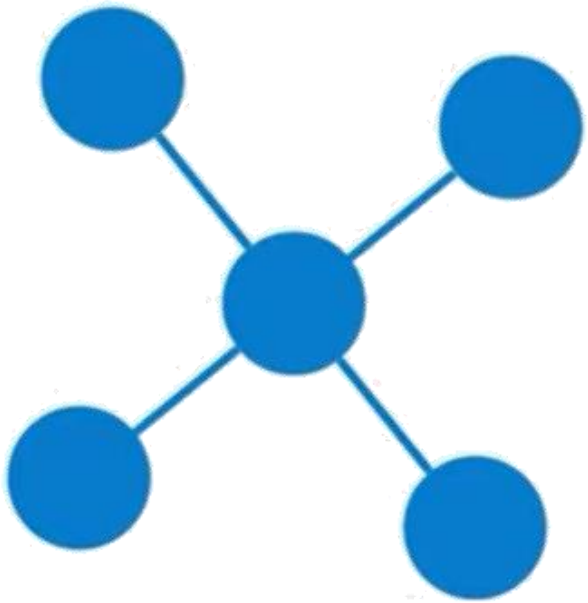


기초 용어 정리

정점(Node)　　간선(Edge)　　단방향 간선　　양방향 간선

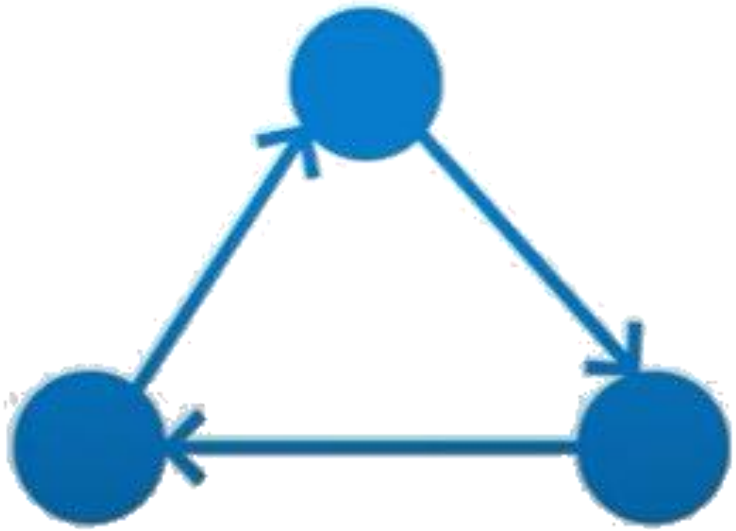
기초 용어 정리

- 차수(Degree)
  어떤 정점의 차수, 어떤 꼭지점의 차수



기초 용어 정리

- 사이클(Cycle)



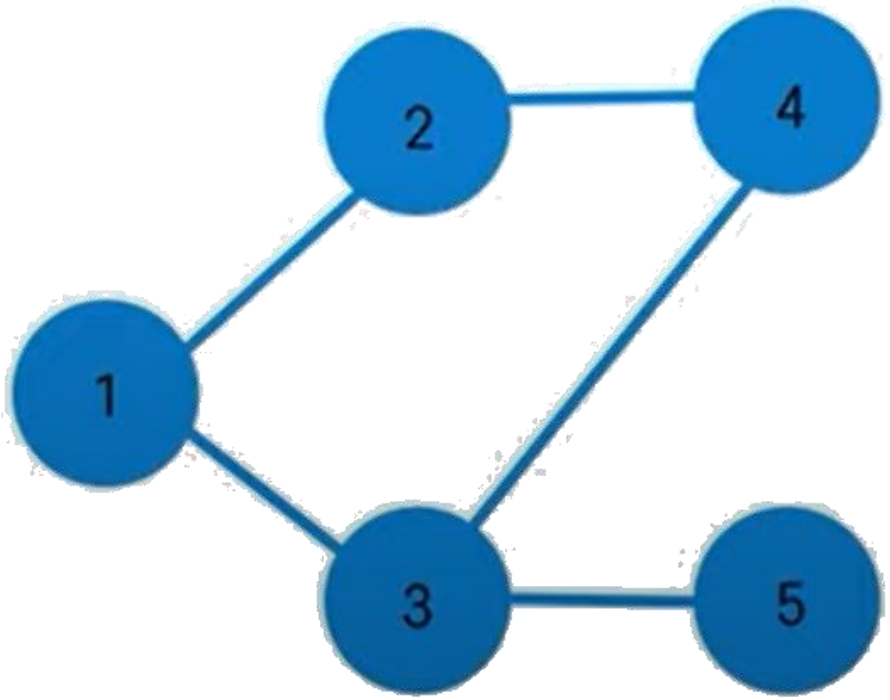
기초 용어 정리

- 경로

  1에서 4로 가는 경로

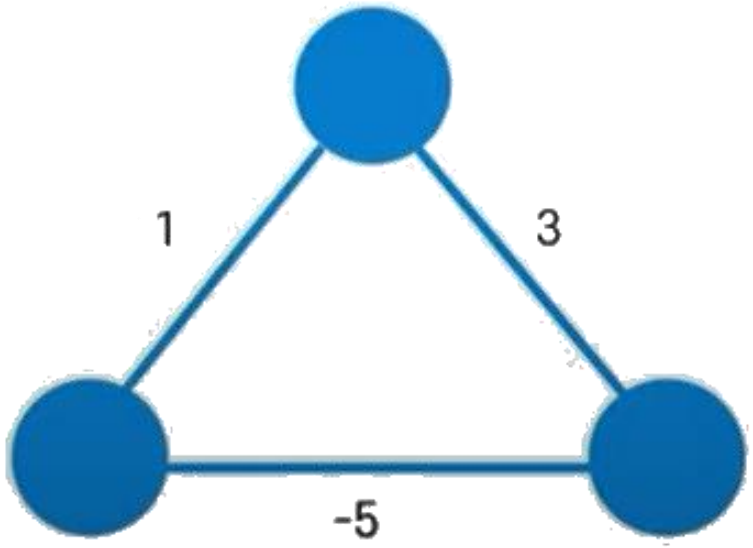
  1->2->4

  1->3->4



기초 용어 정리

- 가중치
  간선(Edge)이 가지는 고유값
  문제에 따라 거리, 비용 등 다양한 값을 갖는다.



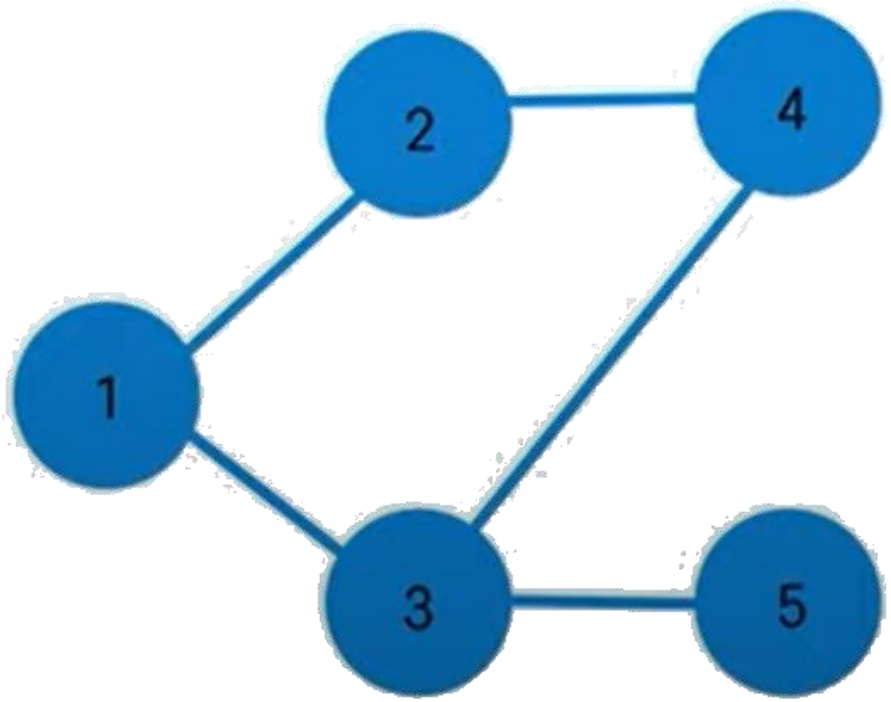
복습

2. 그래프 표현 방법

1) 인접 행렬
2) 인접 리스트

인접 행렬

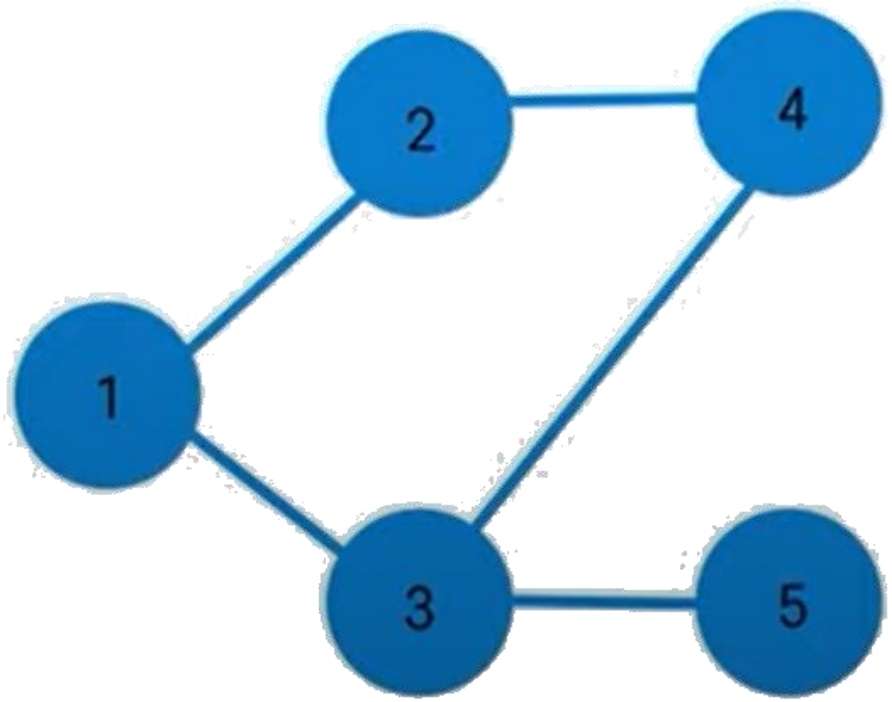
- 정점의 개수 N개에 대해 NxN행렬(배열)을 이용하여 표현한다.
  N[i][j] : i번 노드 ~ j번 노드를 잇는 간선의 여부 확인
  (1: 있다, 0: 없다)



인접 행렬

가중치가 없는 경우

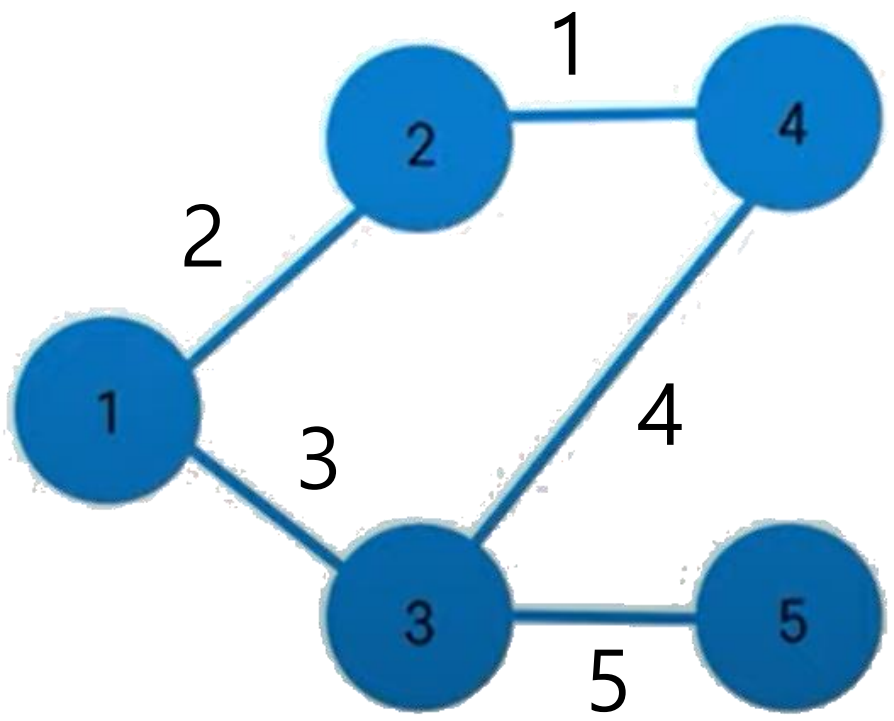
	1	2	3	4	5
1	0	1	1	0	0
2	1	0	0	1	0
3	1	0	0	1	1
4	0	1	1	0	0
5	0	0	1	0	0



인접 행렬

가중치가 있는 경우
? : 절대 나올 수 없는 수

	1	2	3	4	5
1	?	2	3	?	?
2	2	?	?	1	?
3	1	?	?	1	1
4	?	1	1	?	?
5	?	?	1	?	?



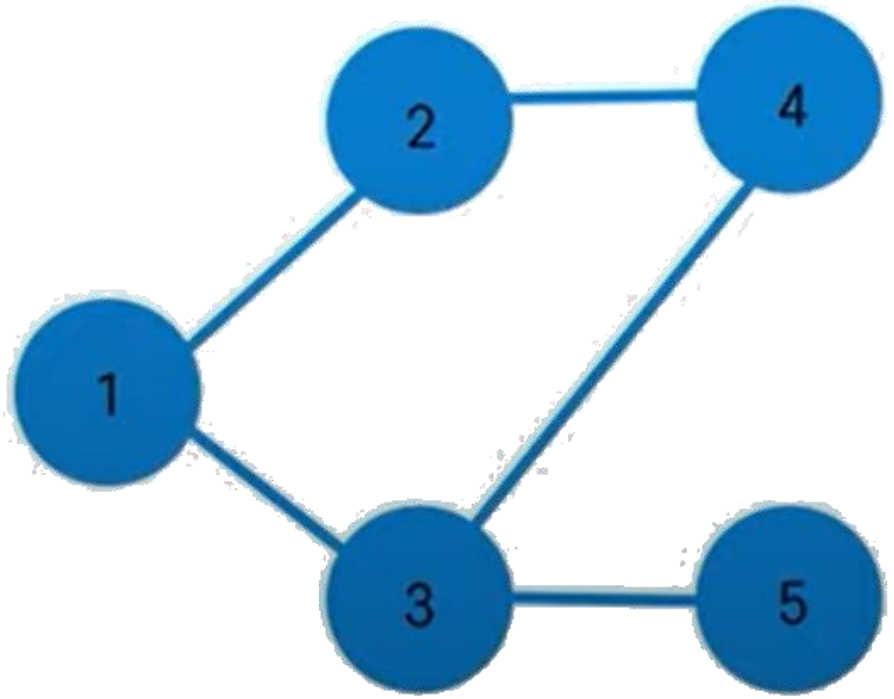
인접 리스트

- 가변 길이의 벡터를 통해 구현한다.
- 정점 N개인 그래프의 인접 리스트 초기 선언 – vector<int>[N]
- V[i] = {i와 연결되어 있는 모든 정점}

인접 행렬

가중치가 없는 경우

V[1] = {2,3}
V[2] = {1,4}
V[3] = {1,4,5}
V[4] = {2,3}
V[5] = {3}



인접 행렬 vs 인접 리스트

- 공간 복잡도
  인접 행렬 : N x N행렬 사용 -> O(n^2)
  인접 리스트 : 간선 개수 x 2만큼의 공간 필요 -> O(E)

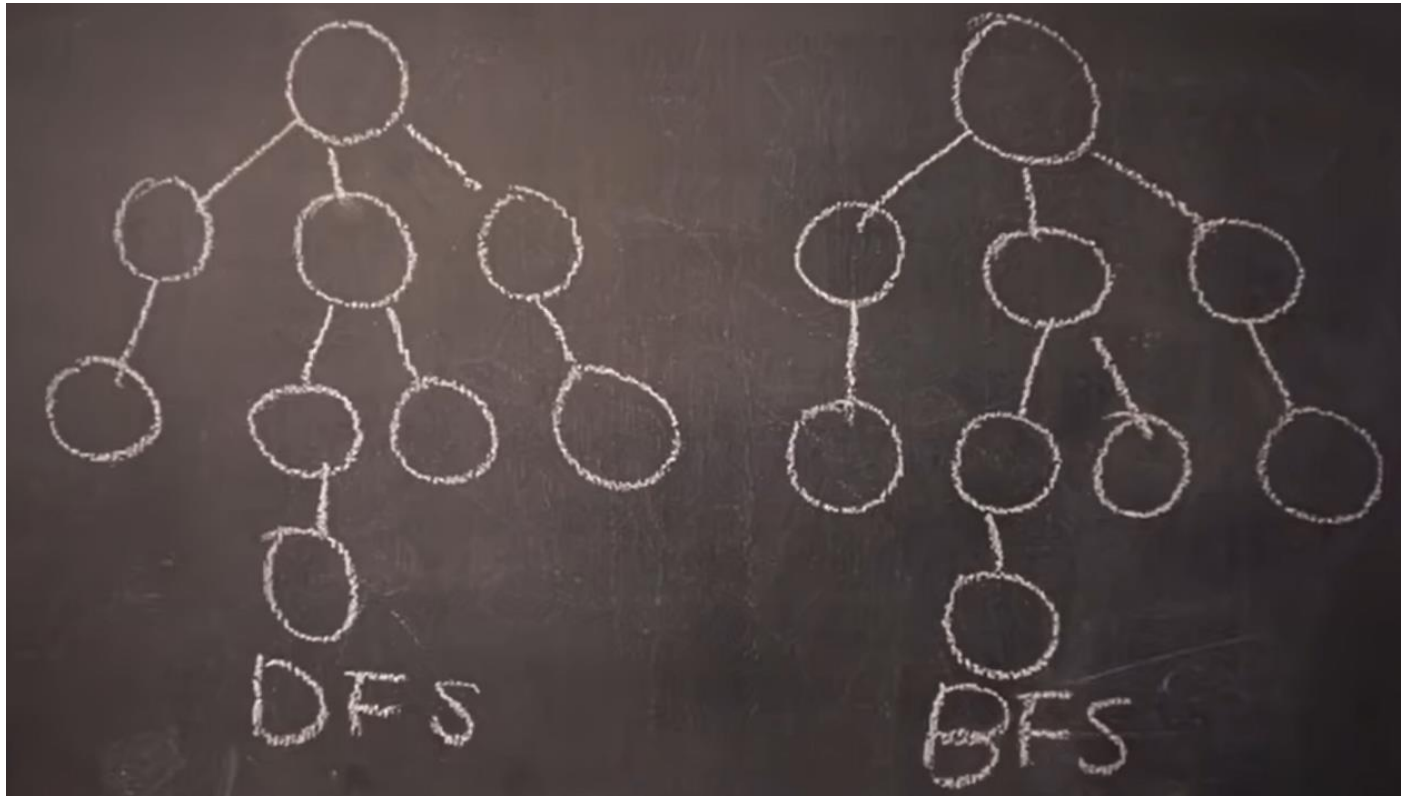
-> 인접 리스트가 훨씬 효율적

3. BFS / DFS

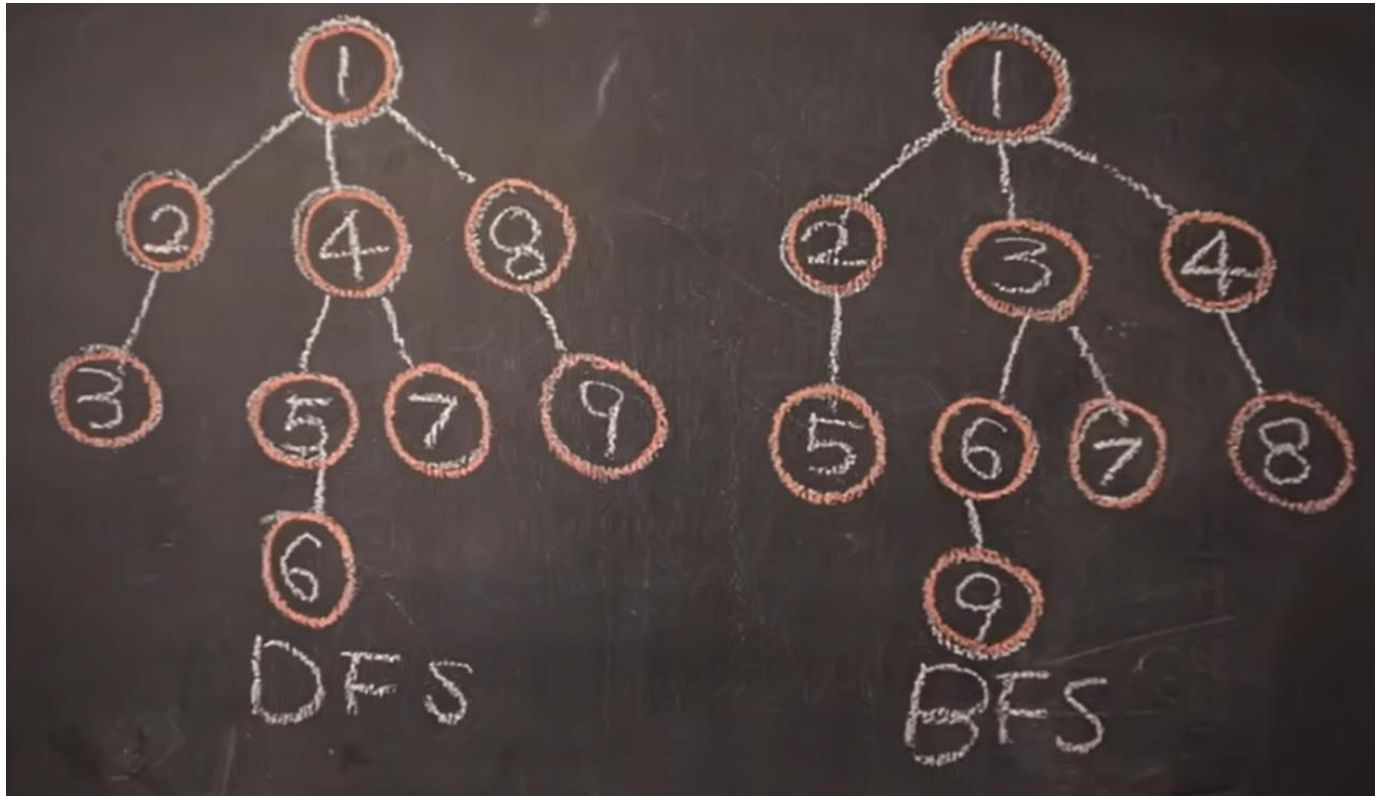
BFS, DFS 란?

- 그래프의 탐색 기법의 일종
- 목적 : 임의의 한 정점에서 시작하여 모든 장점을 방문하는 것.
- BFS : Breadth First Search, 너비 우선 탐색
- DFS: Depth First Search, 깊이 우선 탐색

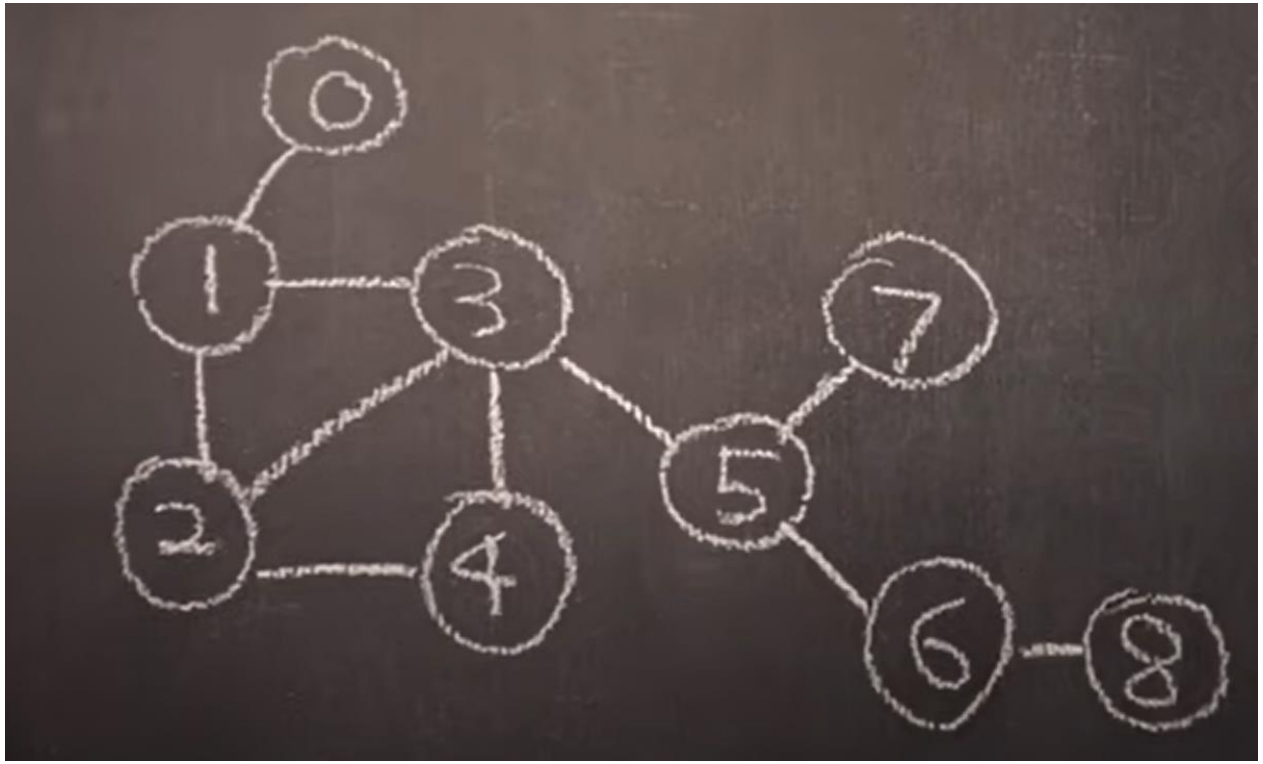
DFS, BFS



DFS, BFS



DFS : Stack

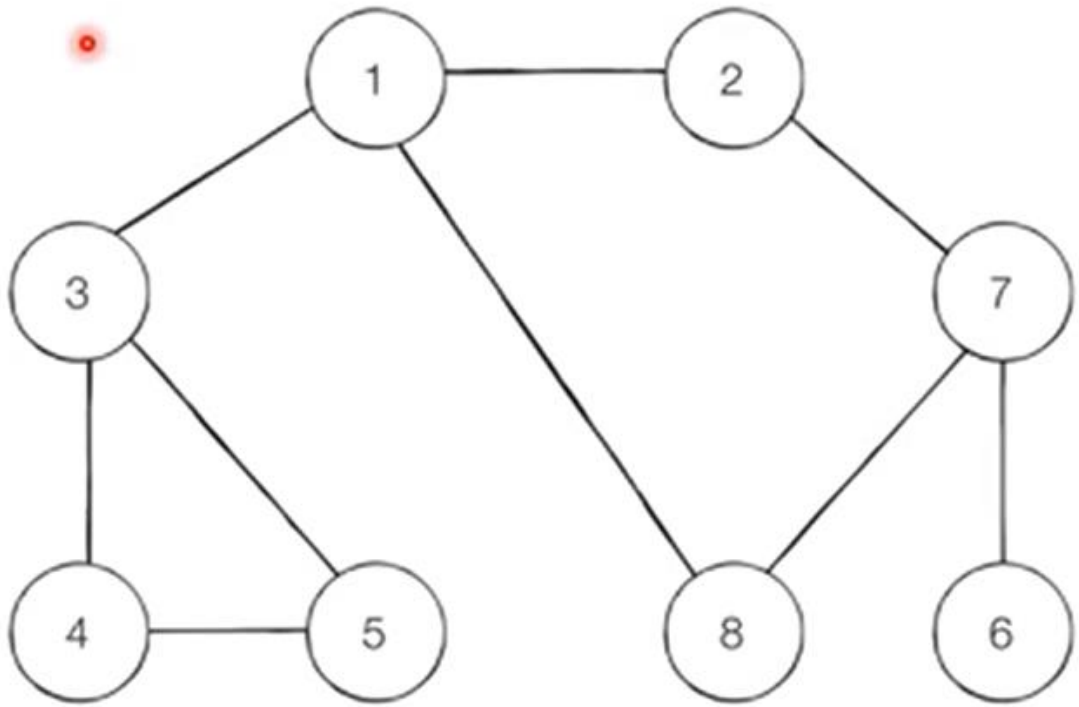


DFS

- DFS는 깊이 우선 탐색이라고도 부르며 그래프에서 깊은 부분을 우선적으로 탐색하는 알고리즘입니다
- DFS는 스택 자료구조(혹은 재귀 함수)를 이용하며, 구체적인 동작 과정은 다음과 같습니다
- 1. 탐색 시작 노드를 스택에 삽입하고 방문 처리를 합니다.
- 2. 스택의 최상단 노드에 방문하지 않은 인접한 노드가 하나라도 있으면 그 노드를 스택에 넣고 방문 처리합니다. 방문하지 않은 인접 노드가 없으면 스택에서 최상단 노드를 꺼냅니다.
- 3. 더 이상 2번의 과정을 수행할 수 없을 때까지 반복합니다.

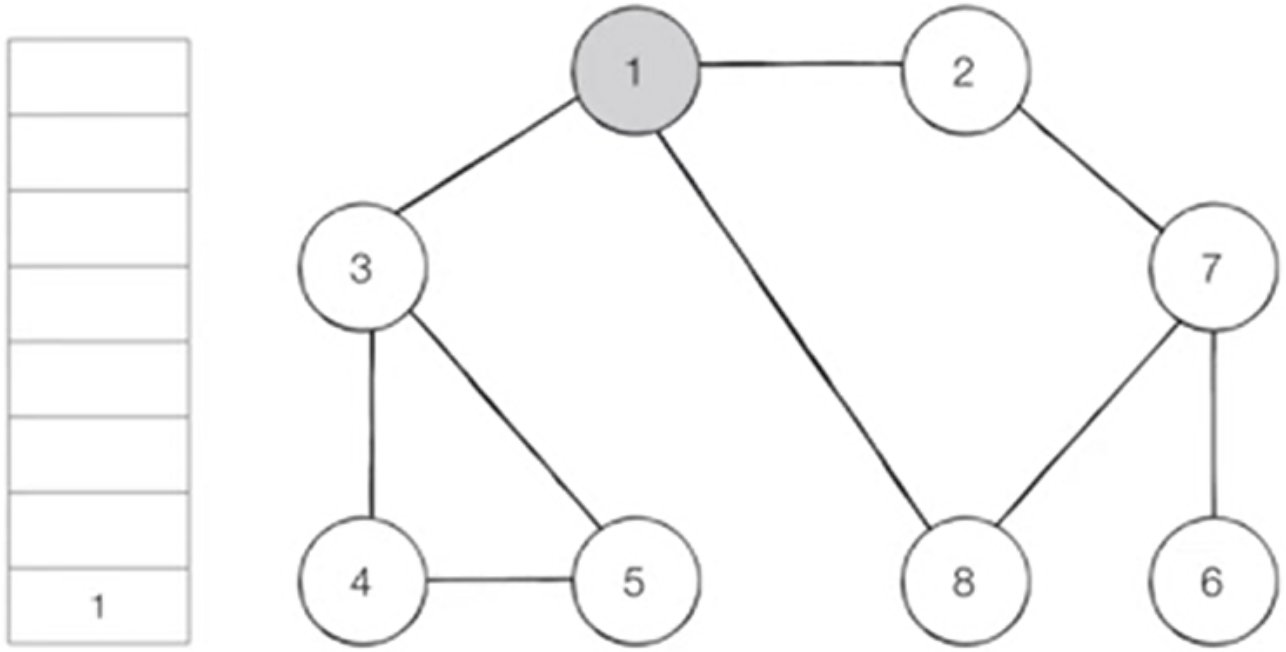
DFS

- **[Step 0]** 그래프를 준비합니다. (**방문 기준**: 번호가 낮은 인접 노드부터)
  - 시작 노드: 1



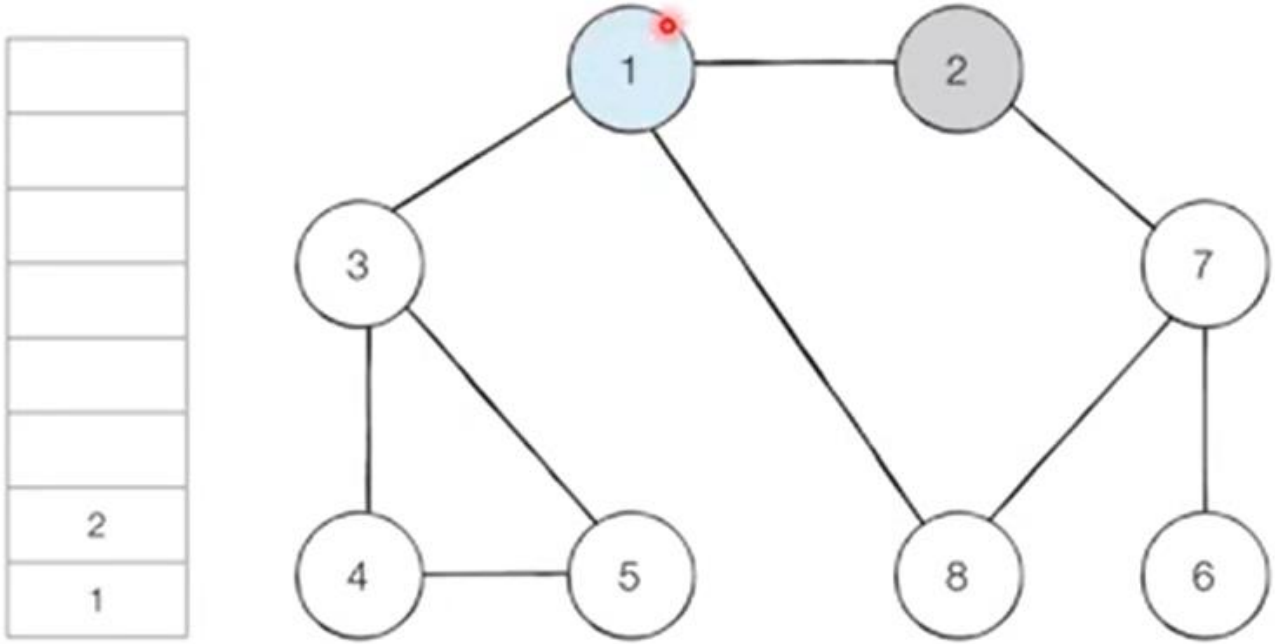
DFS

- **[Step 1]** 시작 노드인 '1'을 스택에 삽입하고 방문 처리를 합니다.



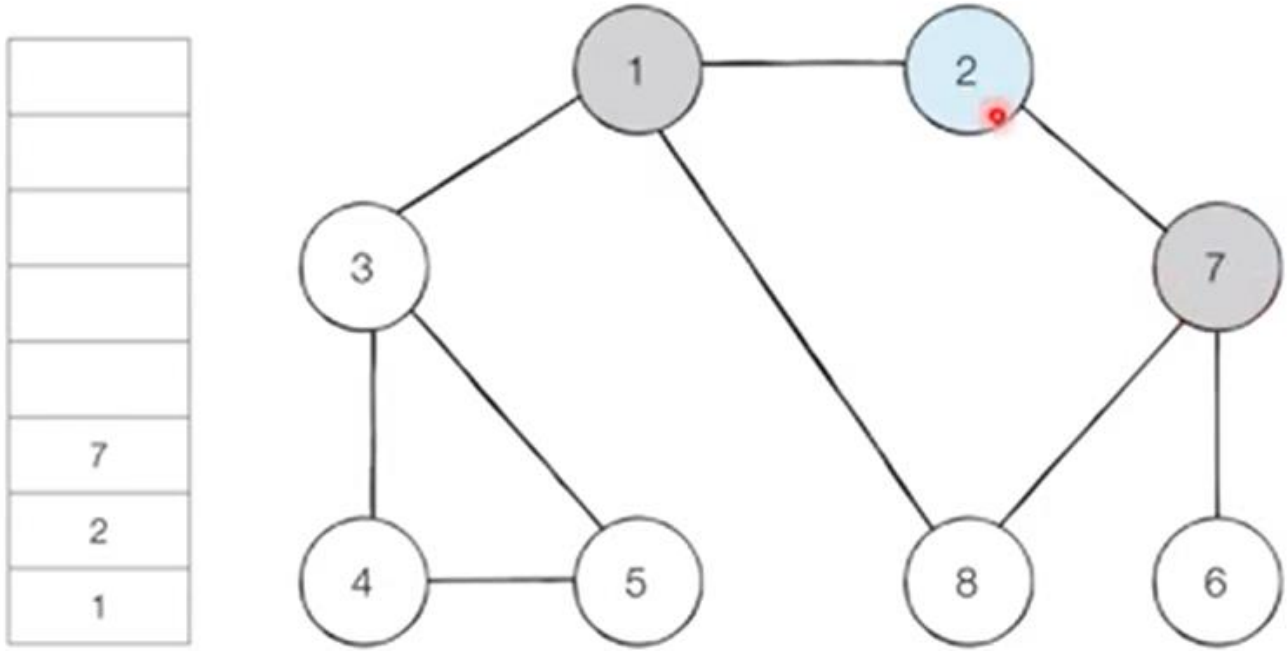
DFS

- [Step 2] 스택의 최상단 노드인 '1'에 방문하지 않은 인접 노드 '2', '3', '8'이 있습니다.
  - 이 중에서 가장 작은 노드인 '2'를 스택에 넣고 방문 처리를 합니다.



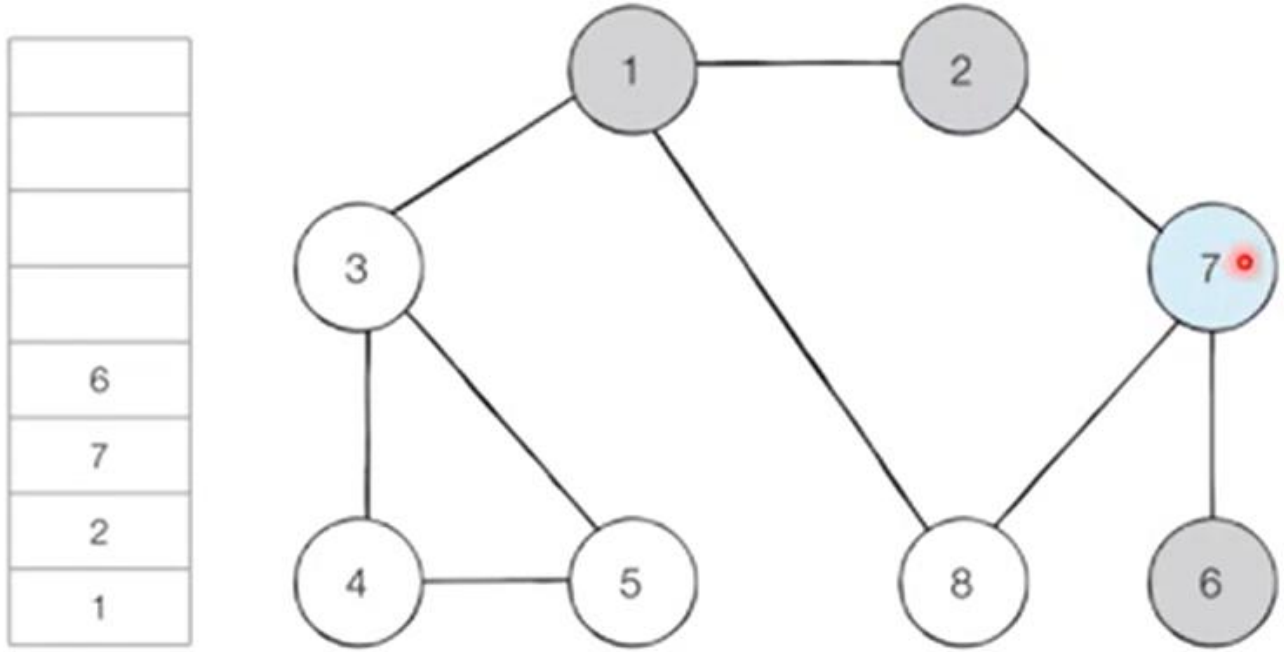
DFS

- [Step 3] 스택의 최상단 노드인 '2'에 방문하지 않은 인접 노드 '7'이 있습니다.
  - 따라서 '7'번 노드를 스택에 넣고 방문 처리를 합니다.



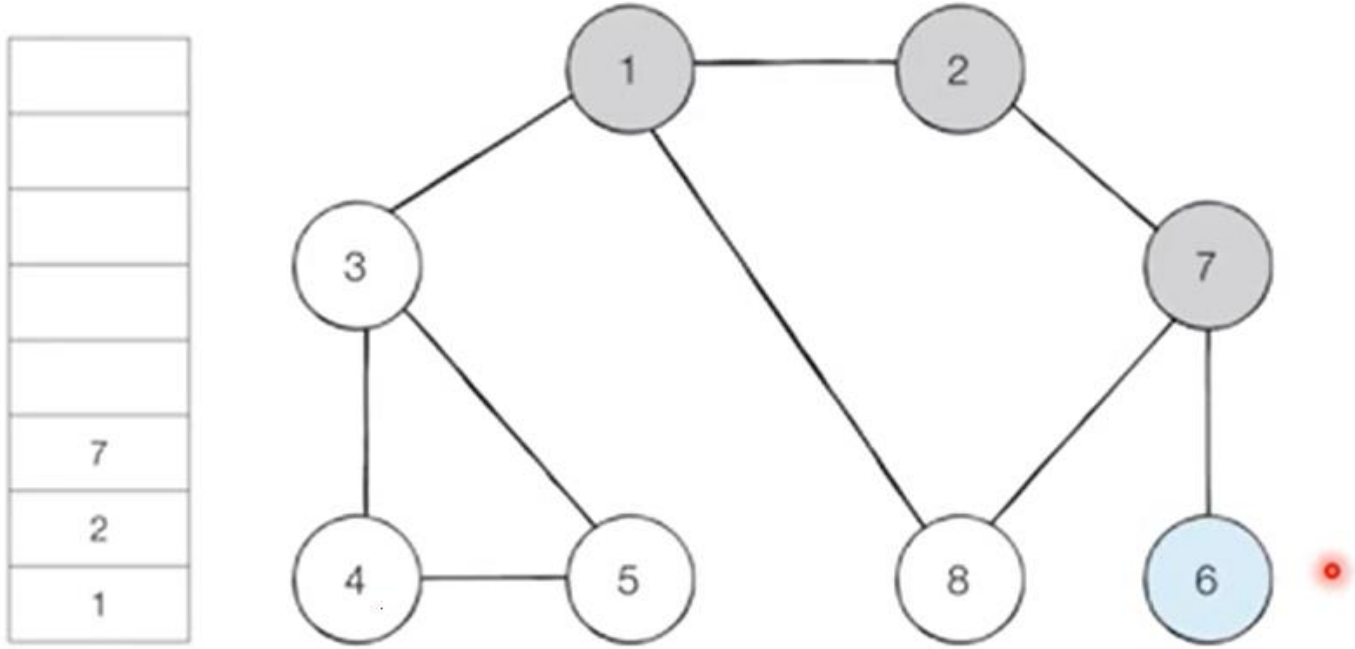
DFS

- **[Step 4]** 스택의 최상단 노드인 '7'에 방문하지 않은 인접 노드 '6', '8'이 있습니다.
  - 이 중에서 가장 작은 노드인 '6'을 스택에 넣고 방문 처리를 합니다.



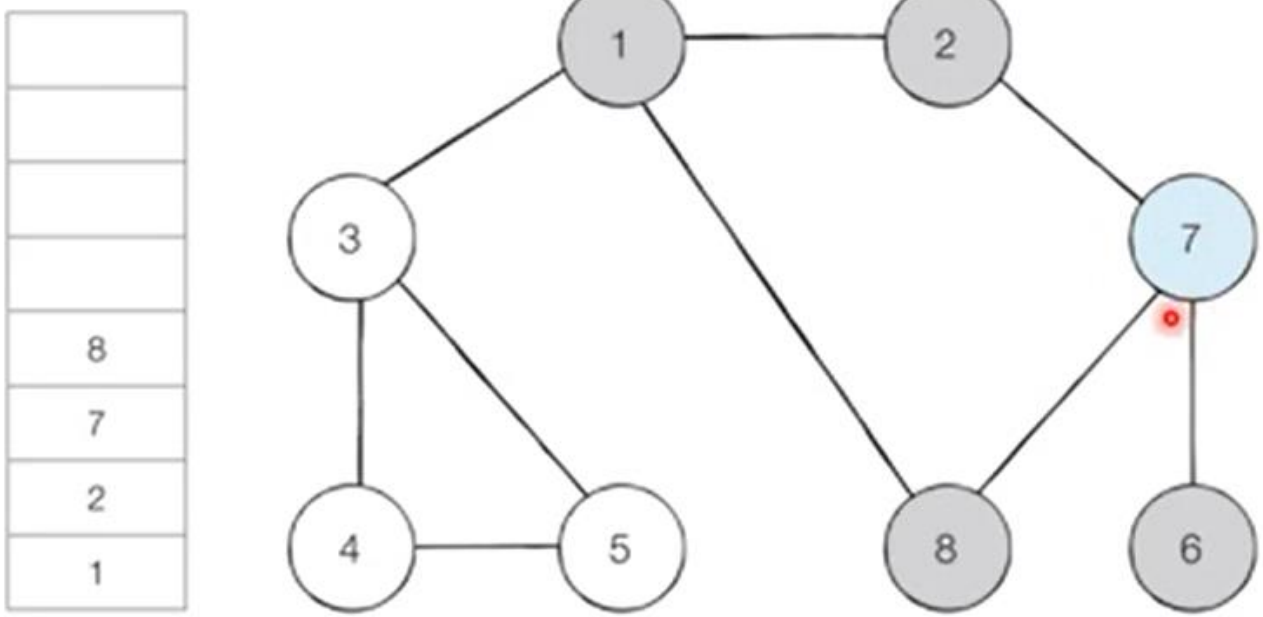
DFS

- [Step 5] 스택의 최상단 노드인 '6'에 방문하지 않은 인접 노드가 없습니다.
  - 따라서 스택에서 '6'번 노드를 꺼냅니다.



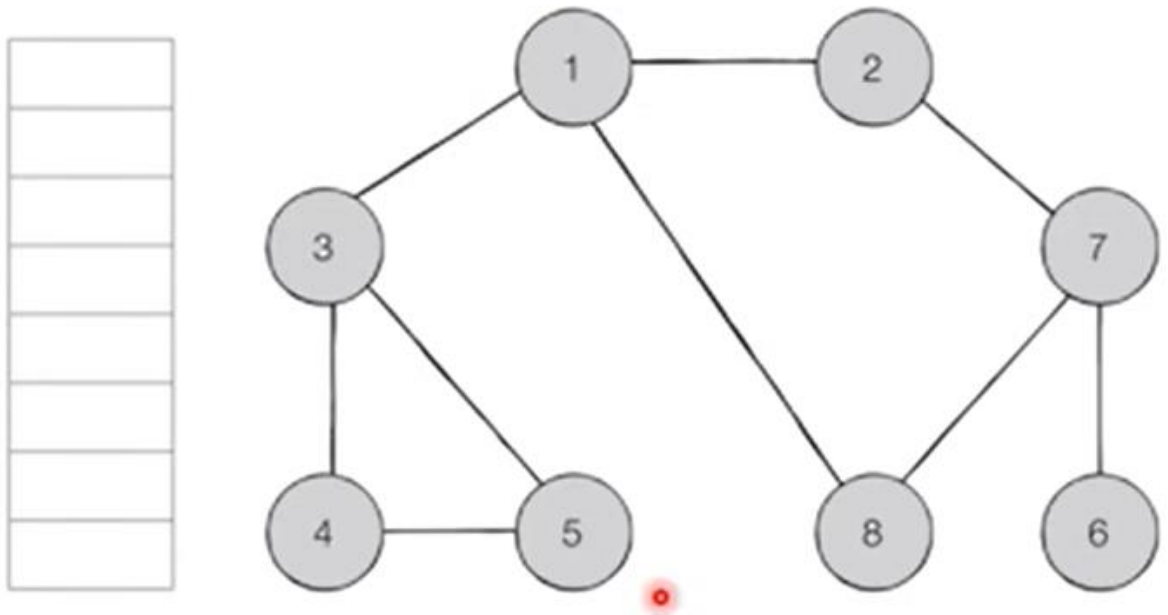
DFS

- [Step 6] 스택의 최상단 노드인 '7'에 방문하지 않은 인접 노드 '8'이 있습니다.
  - 따라서 '8'번 노드를 스택에 넣고 방문 처리를 합니다.



DFS

- 이러한 과정을 반복하였을 때 **전체 노드의 탐색 순서**(스택에 들어간 순서)는 다음과 같습니다.



탐색 순서: 1 → 2 → 7 → 6 → 8 → 3 → 4 → 5

DFS 소스코드 예제(Python)

```python
#DFS 메서드 정의
def dfs(graph, v, visited):
    #현재 노드를 방문 처리
    visited[v]=True
    print(v,end=' ')
    #현재 노드와 연결된 다른 노드를 재귀적으로 방문
    for i in graph[v]:
        if not visited[i]:
            dfs(graph, i, visited)
```

```python
#각 노드가 연결된 정보를 표현(2차원 리스트)
graph = [
    [],
    [2,3,8],
    [1,7],
    [1,4,5],
    [3,5],
    [3,4],
    [7],
    [2,6,8],
    [1,7]
]
#각 노드가 방문된 정보를 표현(1차원 리스트)
visited=[False]*9

#정의된 DFS 함수 호출
dfs(graph, 1, visited)
```

```
1 2 7 6 8 3 4 5
```

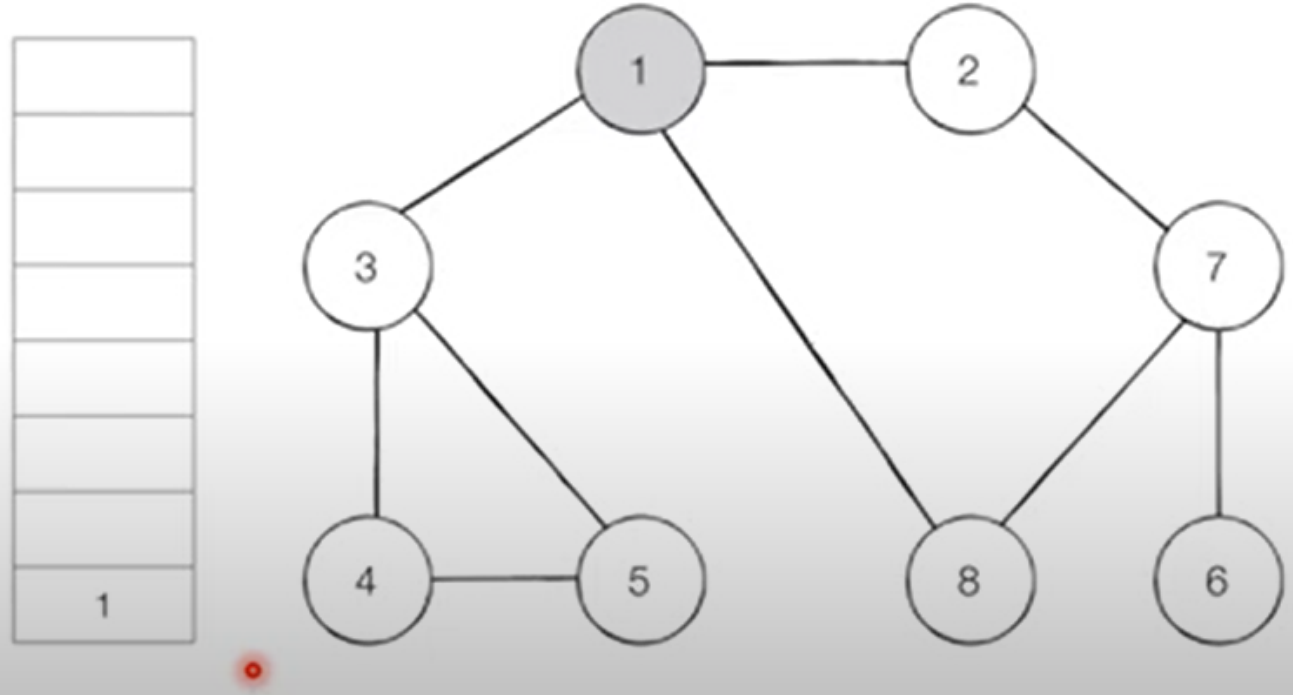
BFS

- BFS 는 너비 우선 탐색이라고 부르며, 그래프에서 가까운 노드부터 우선적으로 탐색하는 알고리즘이다.
- BFS는 큐 자료구조를 이용하며, 구체적인 동작 과정은 다음과 같다.

- 1. 탐색 시작 노드를 큐에 삽입하고 방문 처리를 한다.
- 2. 큐에서 노드를 꺼낸 뒤에 해당 노드의 인접 노드 중에서 방문하지 않은 노드를 모두 큐에 삽입하고 방문 처리한다.
- 3. 더 이상 2번의 과정을 수행할 수 없을 때까지 반복한다.

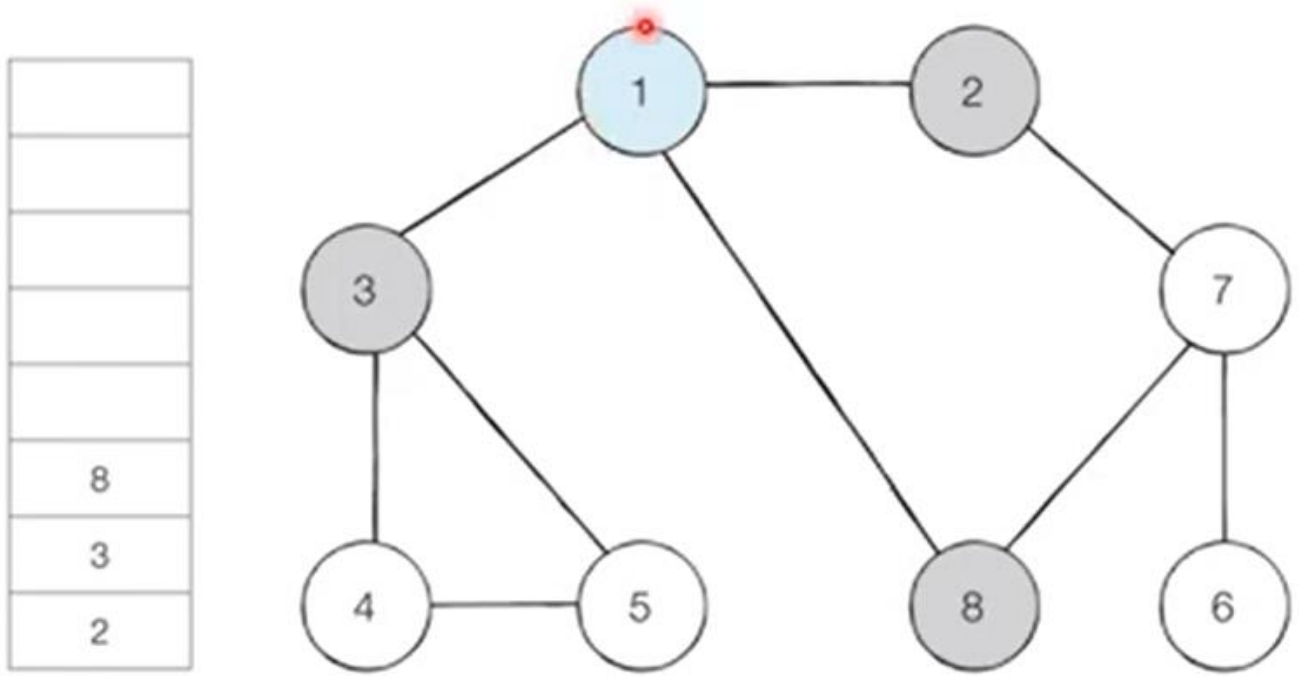
BFS : Queue

- **[Step 1]** 시작 노드인 '1'을 큐에 삽입하고 방문 처리를 합니다.



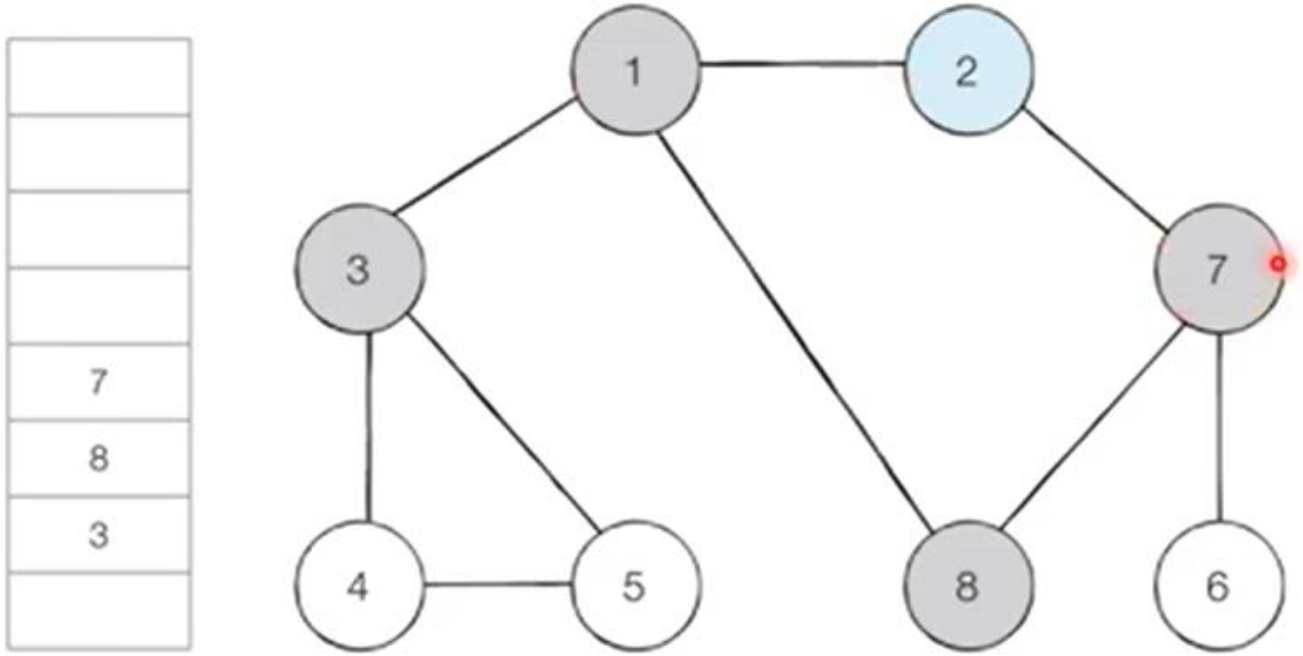
BFS : Queue

- **[Step 2]** 큐에서 노드 '1'을 꺼내 방문하지 않은 인접 노드 '2', '3', '8'을 큐에 삽입하고 방문 처리합니다.



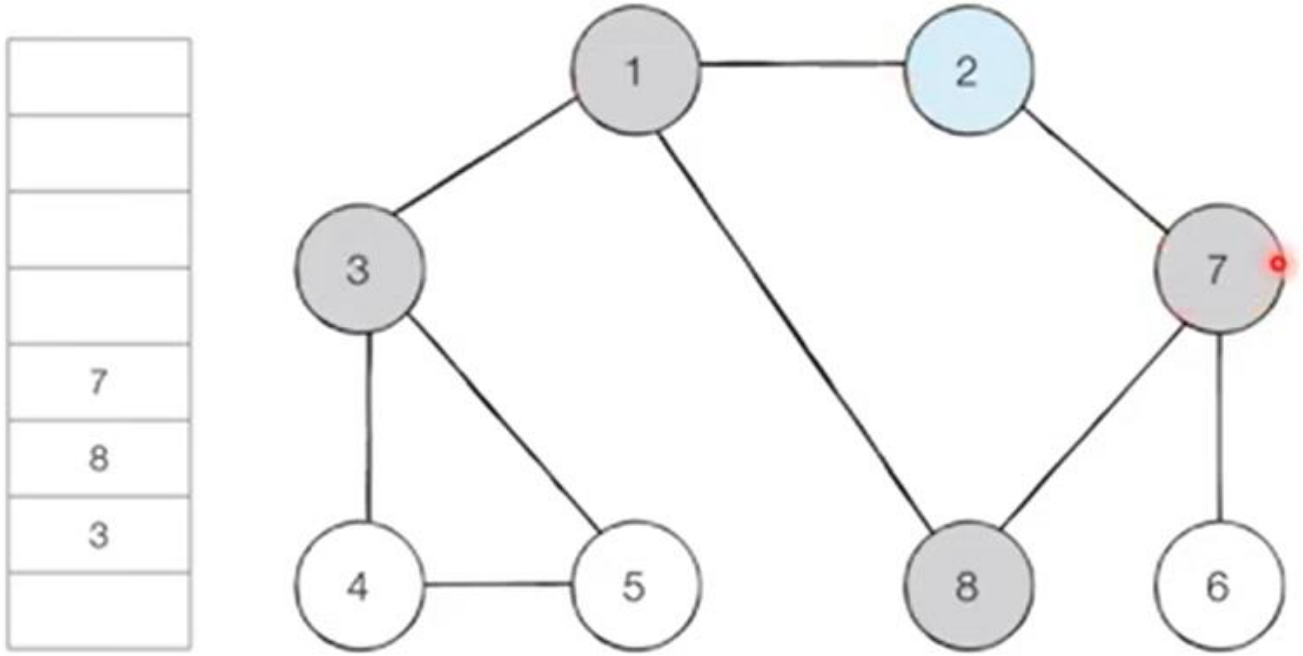
BFS : Queue

- **[Step 3]** 큐에서 노드 '2'를 꺼내 방문하지 않은 인접 노드 '7'을 큐에 삽입하고 방문 처리합니다.



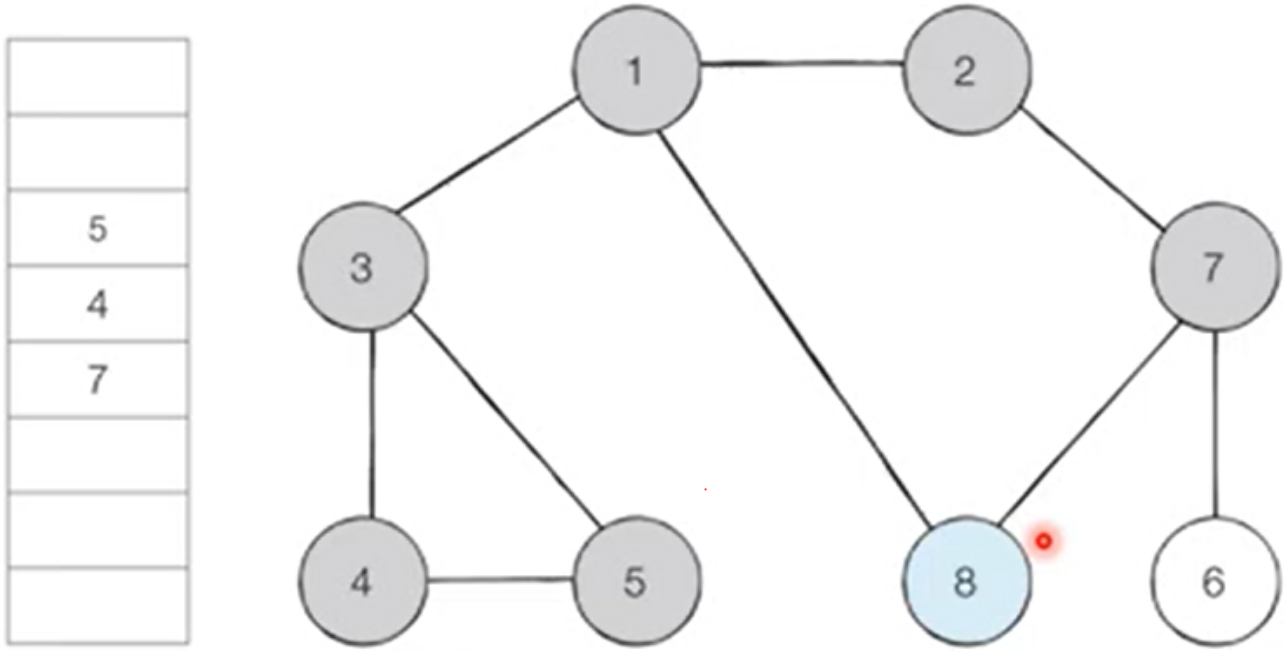
BFS : Queue

- **[Step 3]** 큐에서 노드 '2'를 꺼내 방문하지 않은 인접 노드 '7'을 큐에 삽입하고 방문 처리합니다.



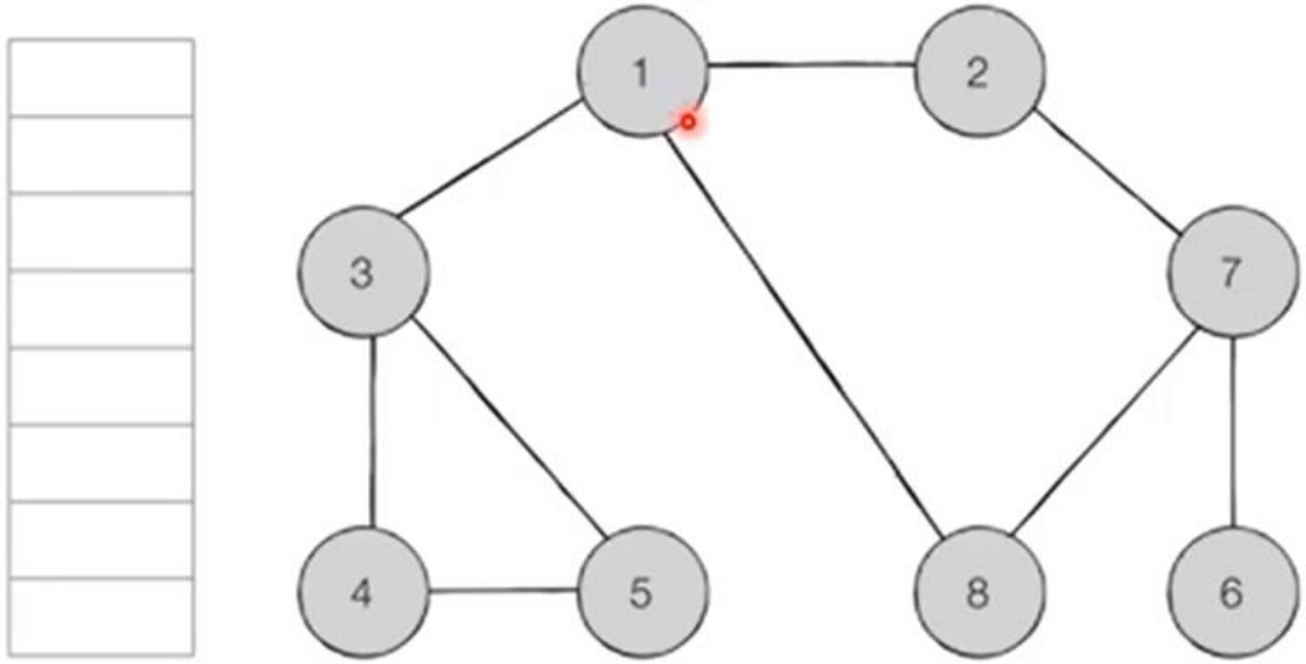
BFS : Queue

- **[Step 5]** 큐에서 노드 '8'을 꺼내고 방문하지 않은 인접 노드가 없으므로 무시합니다.



BFS : Queue

- 이러한 과정을 반복하여 **전체 노드의 탐색 순서**(큐에 들어간 순서)는 다음과 같습니다.



탐색 순서: 1 → 2 → 3 → 8 → 7 → 4 → 5 → 6

BFS 구현(Python)

```python
from collections import deque

#BFS 메서드 정의
def bfs(graph, start, visited):
    #큐 구현을 위해 deque 라이브러리 사용
    queue = deque([start])
    #현재 노드를 방문 처리
    visited[start]=True
    #큐가 빌 때까지 반복
    while queue:
        v=queue.popleft()
        print(v,end=' ')
        #아직 방문하지 않은 인접한 원소들을 큐에 삽입
        for i in graph[v]:
            if not visited[i]:
                queue.append(i)
                visited[i]=True

#각 노드가 연결된 정보를 표현(2차원 리스트)
graph = [
    [],
    [2,3,8],
    [1,7],
    [1,4,5],
    [3,5],
    [3,4],
    [7],
    [2,6,8],
    [1,7]
]

#각 노드가 방문된 정보를 표현(1차원 리스트)
visited=[False]*9

#정의된 BFS 함수 호출
bfs(graph, 1, visited)
```

문제 – 음료수 얼려 먹기

N × M 크기의 얼음 틀이 있습니다. 구멍이 뚫려 있는 부분은 0, 칸막이가 존재하는 부분은 1로 표시됩니다. 구멍이 뚫려 있는 부분끼리 상, 하, 좌, 우로 붙어 있는 경우 서로 연결되어 있는 것으로 간주합니다. 이때 얼음 틀의 모양이 주어졌을 때 생성되는 총 아이스크림의 개수를 구하는 프로그램을 작성하세요. 다음의 4 × 5 얼음 틀 예시에서는 아이스크림이 총 3개 생성됩니다.

```
00110
00011
11111
00000
```

0	0	1	1	0
0	0	0	1	1
1	1	1	1	1
0	0	0	0	0

입력 조건

- 첫 번째 줄에 얼음 틀의 세로 길이 N과 가로 길이 M이 주어집니다. (1 <= N, M <= 1,000)
- 두 번째 줄부터 N + 1번째 줄까지 얼음 틀의 형태가 주어집니다.
- 이때 구멍이 뚫려있는 부분은 0, 그렇지 않은 부분은 1입니다.

출력 조건

- 한 번에 만들 수 있는 아이스크림의 개수를 출력합니다.

입력 예시

```
4 5
00110
00011
11111
00000
```

출력 예시

```
3
```

해결 – 아이스크림 얼려먹기

- DFS를 활용하는 알고리즘은 다음과 같습니다.
  1. 특정한 지점의 주변 상, 하, 좌, 우를 살펴본 뒤에 주변 지점 중에서 값이 '0'이면서 아직 방문하지 않은 지점이 있다면 해당 지점을 방문합니다.
  2. 방문한 지점에서 다시 상, 하, 좌, 우를 살펴보면서 방문을 진행하는 과정을 반복하면, **연결된 모든 지점을 방문**할 수 있습니다.
  3. 모든 노드에 대하여 1 ~ 2번의 과정을 반복하며, 방문하지 않은 지점의 수를 카운트합니다.

코드

```python
# DFS로 특정 노드를 방문하고 연결된 모든 노드들도 방문
def dfs(x, y):
    # 주어진 범위를 벗어나는 경우에는 즉시 종료
    if x <= -1 or x >= n or y <= -1 or y >= m:
        return False
    # 현재 노드를 아직 방문하지 않았다면
    if graph[x][y] == 0:
        # 해당 노드 방문 처리
        graph[x][y] = 1
        # 상, 하, 좌, 우의 위치들도 모두 재귀적으로 호출
        dfs(x - 1, y)
        dfs(x, y - 1)
        dfs(x + 1, y)
        dfs(x, y + 1)
        return True
    return False
```

```python
# N, M을 공백을 기준으로 구분하여 입력 받기
n, m = map(int, input().split())

# 2차원 리스트의 맵 정보 입력 받기
graph = []
for i in range(n):
    graph.append(list(map(int, input())))

# 모든 노드(위치)에 대하여 음료수 채우기
result = 0
for i in range(n):
    for j in range(m):
        # 현재 위치에서 DFS 수행
        if dfs(i, j) == True:
            result += 1

print(result) # 정답 출력
```

백준

- 백준 18532
- 백준 18405
- 백준 18428