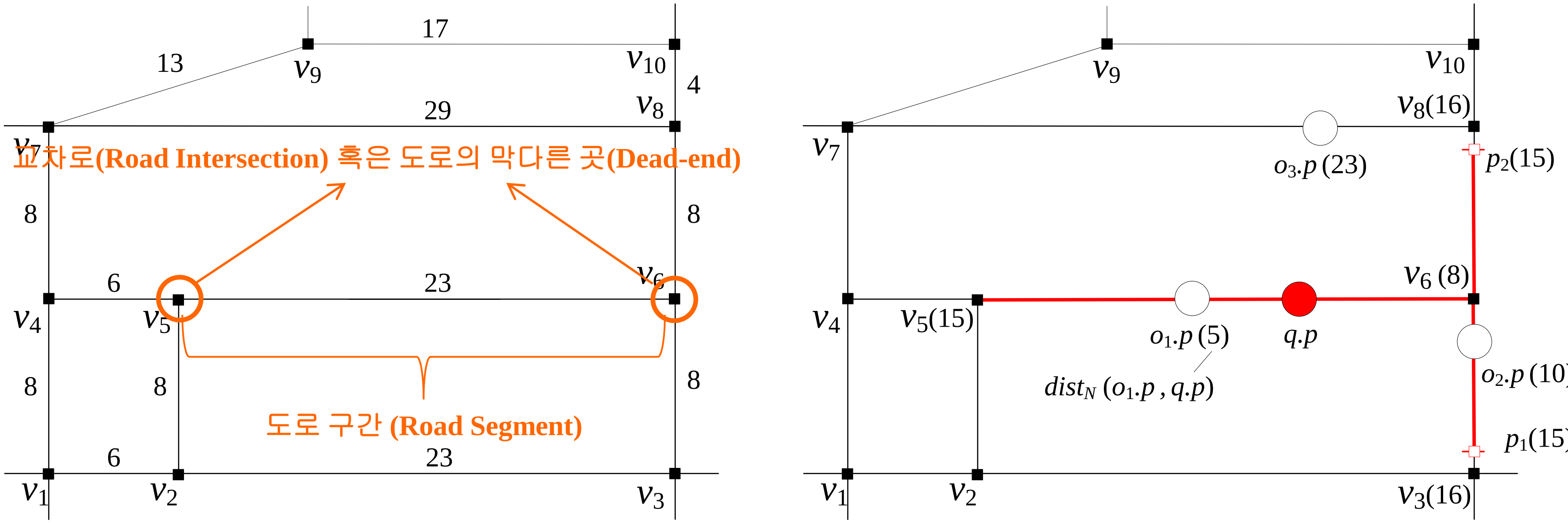
그래프 - 1

HaRim Jung, Ph.D.
Visiting Professor / Senior Researcher
SKKU Institute for Convergence / Convergence Research Institute
Sungkyunkwan University, Korea

그래프 (1/5)

❑ 그래프(Graph)의 개요

- 그래프는 연결되어 있는 원소 간의 관계를 표현할 수 있는 자료구조로서, 인터넷, 도로망, 운송, 전력, 상하수도망, 신경망, 화학성분 결합, 단백질 네트워크, 금융 네트워크, 소셜 네트워크 분석 등의 광범위한 분야에서 활용



도로망(Road Network) 표현의 예

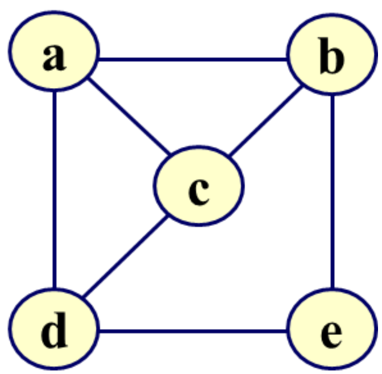
❑ 그래프의 용어(Terminology)

- 그래프는 연결할 원소를 나타내는 **정점(Vertex)**과 정점을 연결하는 **간선(Edge)**의 집합으로 하나의 간선은 한 개 혹은 두 개의 정점을 연결
- 그래프는 **G = (V, E)**로 표현, 여기서 V=정점의 집합, E=간선의 집합

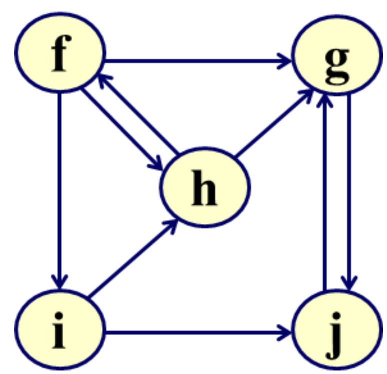
그래프 (2/5)

❑ 그래프의 용어(Terminology) contd.

- **무방향 그래프(Undirected Graph)**: 간선에 방향이 없는 그래프
  - 그림 (a): **V** = {a, b, c, d, e}, **E** = { (a, b), (a, c), (a, d), (b, c), (b, e), (c, d), (d, e) }
  - 정점 a와 b를 연결하는 간선을 (a, b)로 표현
- **방향 그래프(Directed Graph)**: 간선에 방향이 있는 그래프 **정점들 간의 순서가 중요(< f, h>과 < h, f>는 서로 다른 간선을 의미)**
  - 그림 (b): **V** = {f, g, h, i, j}, **E** = { <f, g>, <f, h>, <f, i>, <g, j>, <h, f>, <h, g>, <i, h>, <i, j>, <j, g> }
  - 정점 a에서 b로 간선의 방향이 있는 경우 <a, b>로 표현



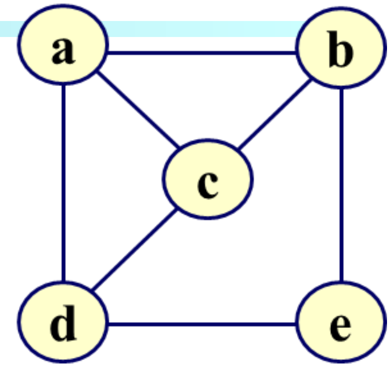
(a) 무방향 그래프



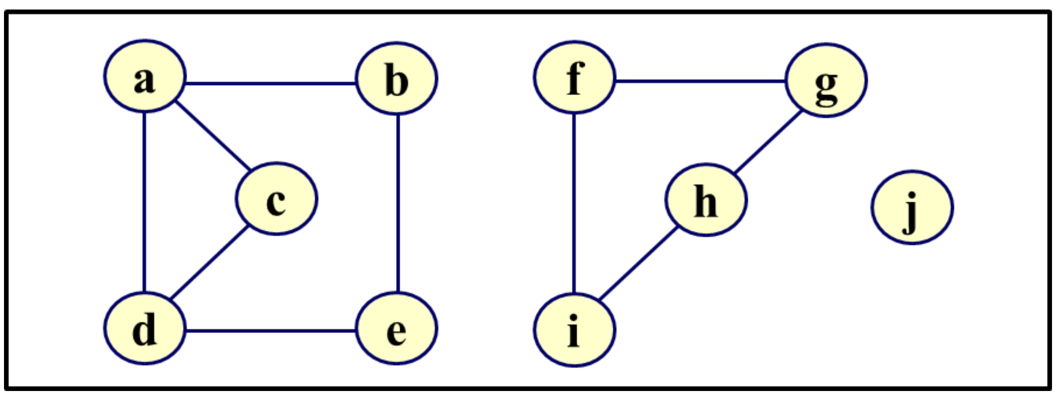
(b) 방향 그래프

- **차수(Degree)**: 특정 정점에 **인접한** 정점들의 수
  - 두 정점 a와 b가 연결되어 간선 (a, b), <a, b>/<b, a>가 있을 때 두 정점 a와 b는 **인접(adjacent)**되어 있다고 하고(a와 b는 서로 이웃), 간선 (a, b), <a, b>/<b, a>는 정점 a와 b에 부속(incident)되어 있다고 함
  - 방향 그래프에서는 차수를 진입 차수(In-degree)와 진출 차수(Out-degree)로 구분
    - 그림 (a): 정점 a의 차수 = 3, 정점 e의 차수 = 2 그림 (b): 정점 g의 진입 차수 = 3, 진출 차수 = 1

그래프 (3/5)



- **그래프의 용어(Terminology) contd.**
  - **경로(Path)**는 시작 정점 u로부터 도착 정점 v에 이르기까지의 정점들을 나열하여 표현
  - **단순 경로(Simple Path)**: 모두 다른 정점으로 구성된 경로
    - 즉, 경로 상의 정점들이 모두 다른 경우
    - 일반적인 경로는 동일한 정점을 중복하는 경우를 포함, e.g., [a, **b**, c, **b**, e]: 시작 정점 a로부터 도착 정점 e까지의 경로
  - **싸이클(Cycle)**: 시작 정점과 도착 정점이 동일한 단순 경로, e.g., [a, b, e, d, c, a]
  - **연결성(Connectivity)**: 그래프에서 서로 다른 모든 쌍의 정점들 사이에 경로가 존재하는 그래프를 연결 그래프 (Connected Graph)라고 하며, 아래의 그래프 G와 같이 연결되지 않은 정점이 존재하는 그래프를 단절 그래프 (Disconnected Graph)라고 함
  - **연결성분(Connected Component)**: 그래프에서 정점들이 서로 연결되어 있는 부분
    - 아래의 단절 그래프 G는 3개의 연결성분인 [a, b, c, d, e], [f, g, h, i], [j]로 구성



단절 그래프 G

그래프 (4/5)

- 그래프의 용어(Terminology) contd.

참조: 무방향 그래프 및 경로 정의

Jung, H.; Kim, U.-M. The SSP-Tree: A Method for Distributed Processing of Range Monitoring Queries in Road Networks. ISPRS Int. J. Geo-Inf. 2017, 6, 322.

In this paper, we address the problem of processing range monitoring queries in the road network. The road network is modeled as an undirected graph $G = (V, E)$, where $V = \{v_1, v_2 \ldots, v_{|V|}\}$ is a set of vertices and $E(\subseteq V \times V) = \{v_i v_j | 1 \leq i \leq |E|, 1 \leq j \leq |E|, i \neq j\}$ is a set of edges. A vertex $v \in V$ corresponds to a road intersection or dead-end. On the other hand, an edge $v_i v_j \in E$ corresponds to a road segment, which connects two vertices v_i and v_j. For convenience of notation, we sometimes indicate an edge $v_i v_j$ as e. We can assume that each road segment of the road network is a straight line because a curved road segment can be transformed into a set of straight lines by adding extra vertices and edges to G. Therefore, the length of an edge $v_i v_j$ can be the Euclidean distance between its two endpoints v_i and v_j. Hereafter, we use $dist_E(\cdot, \cdot)$ to denote the Euclidean distance between any two points (including the vertices) in the road network G.

Definition 1. *Given two vertices v_a and v_b in the road network $G = (V, E)$, where $v_a v_b \notin E$, a* ***path*** *from v_a to v_b, denoted by $P(v_a, v_b)$, is a sequence of vertices $(v_{p_1}, v_{p_2} \ldots, v_{p_k})$ such that $v_{p_1} = v_a$, $v_{p_k} = v_b$, and for each consecutive pair of vertices $(v_{p_i}, v_{p_{i+1}})$ for all $1 \leq i < k$, the condition: $v_{p_i} v_{p_{i+1}} \in E$ holds. Then, the* ***path length*** *of $P(v_a, v_b)$ is calculated as:*

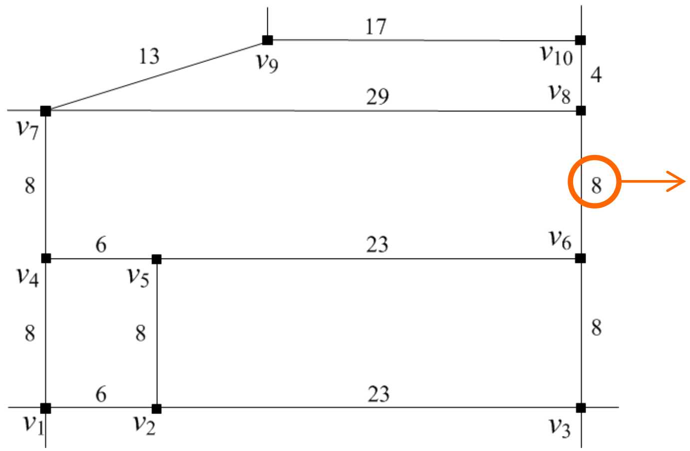
$$L(P(v_a, v_b)) = \sum_{i=1}^{k-1} dist_E(v_{p_i}, v_{p_{i+1}}), \tag{1}$$

where $v_{p_1} = v_a$ and $v_{p_k} = v_b$.

그래프 (5/5)

❑ 그래프의 용어(Terminology) contd.

- **가중치(Weighted) 그래프**: 간선에 가중치가 부여된 그래프
  - 가중치는 두 정점 a와 b 사이의 공간적 거리, 특정 객체가 정점 a에서 출발하여 정점 b에 도착하기 까지 걸린 시간 혹은 비용도 될 수 있으며, 음수인 경우도 존재



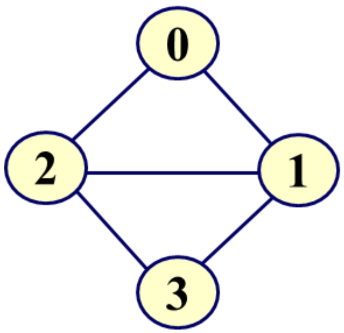
가중치의 예: 두 정점 v6과 v8 사이의 유클리드 거리

- **부분그래프(Subgraph)**: 주어진 그래프의 정점과 간선의 일부분(집합)으로 이루어진 그래프
  - 그래프 G = (V, E)가 주어졌을 때, V’ ⊆ V이고 E’ ⊆ E인 그래프 G’ = (V’, E’)를 G의 부분그래프라고 부름
  - 따라서 부분그래프는 원래의 그래프에 없는 정점이나 간선을 포함하지 않음
- **트리(Tree)**: 싸이클이 없는 그래프
- **신장트리(Spanning Tree)**: 주어진 그래프 G가 하나의 연결성분으로 구성되어 있을 때, 그래프의 모든 정점들을 싸이클 없이 연결하는 부분그래프
  - 하나의 연결성분으로 구성된 그래프 G = (V, E)가 주어졌을 때, 사이클이 존재하지 않고 V’ = V 및 E’ ⊆ E인 그래프 G’ = (V’, E’)를 신장트리라고 부름

그래프의 구현 (1/2)

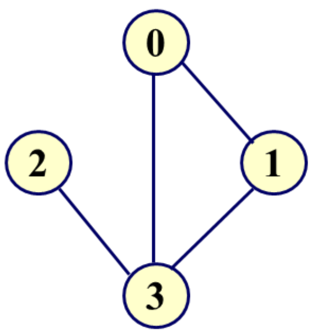
❑ 그래프의 (물리적) 구현

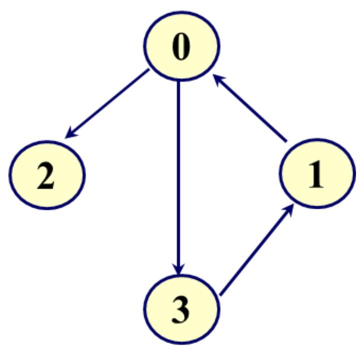
- 인접 행렬(Adjacency Matrix)과 **인접 리스트(Adjacency List)**로 표현
- **인접 행렬**
  - N 개의 정점을 가진 그래프의 인접 행렬은 2차원 N × N (Python) 리스트에 저장
  - 리스트가 G라면, 정점들을 0, 1, 2,…, N-1로 하여, 정점 i와 j 사이에 간선이 없으면 **G[i][j] = 0**, 간선이 있으면 **G[i][j] = 1**로 표현(가중치 그래프는 1 대신 가중치 저장)



$$\begin{array}{c c} & \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array} & \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix} \end{array}$$

[연습]다음 그래프를 인접 행렬로 표현하시오.

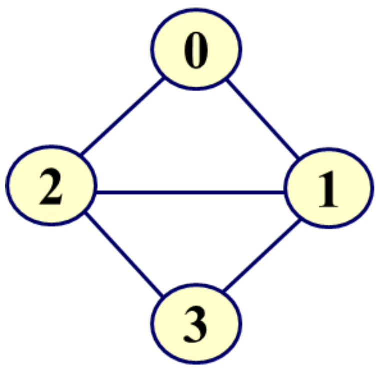


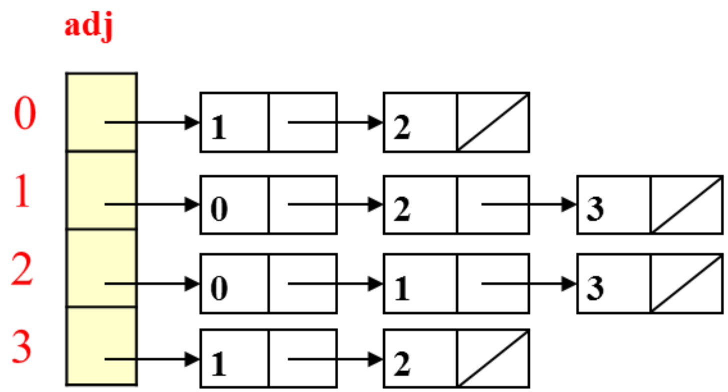


그래프의 구현 (2/2)

❑ 그래프의 (물리적) 구현 contd.

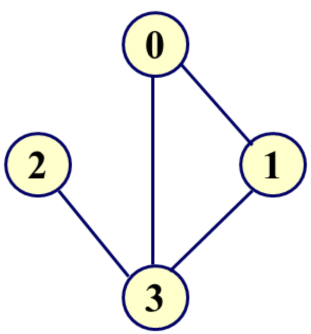
- **인접 리스트**
  - 각 정점마다 1 개의 **(Python) 리스트** and/or 단순 연결 리스트를 이용하여 인접한 각 정점을 저장

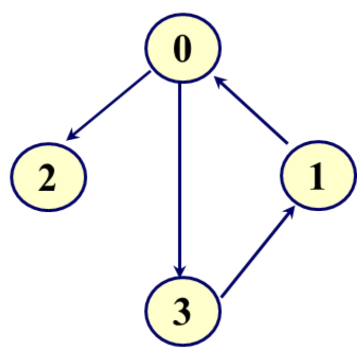




  - adj = [[1, 2], [0, 2, 3], [0, 1 ,3], [1, 2]]
  - 실세계의 그래프는 대부분 정점의 평균 차수가 작은 **희소 그래프(Sparse Graph)**이므로 그래프를 표현할 때 주로 인접 리스트를 사용

[연습]다음 그래프를 인접 리스트로 표현하시오.





그래프 탐색 (1/8)

❑ 그래프 탐색

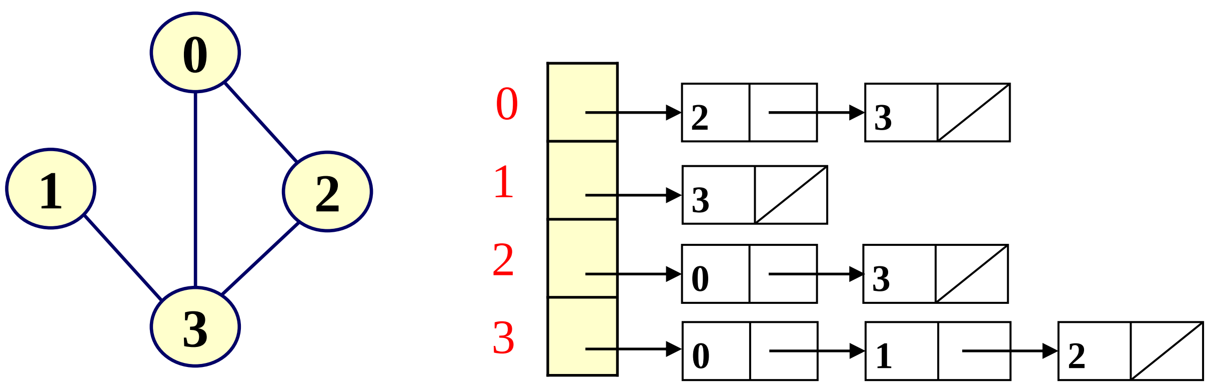
- 그래프에서는 **깊이 우선 탐색(DFS: Depth First Search)**과 **너비 우선 탐색(BFS: Breadth First Search)** 방식으로 모든 정점을 방문

❑ 깊이 우선 탐색(DFS)

-DFS 과정-

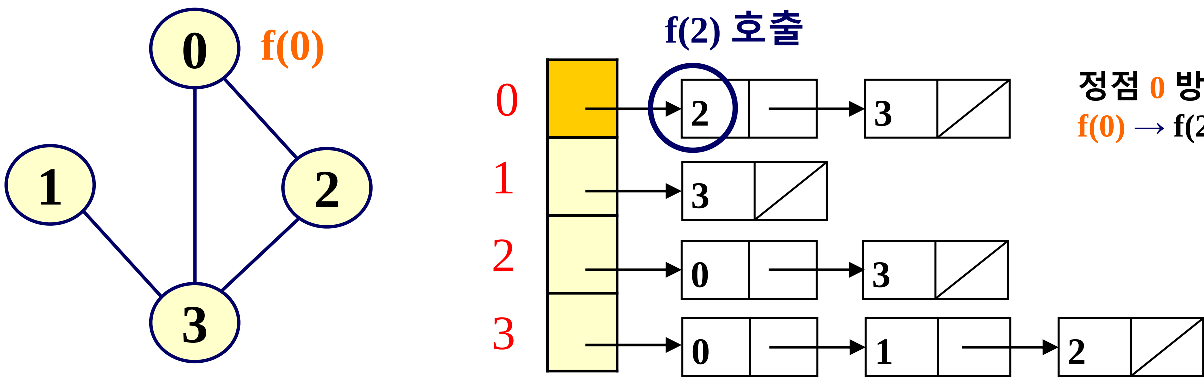
1. 그래프 G에 대하여 DFS는 임의의 정점 a에서 시작하여 이웃하는 하나의 정점 b를 방문하고, 방금 방문한 정점 b의 이웃 정점 c를 방문하는 방식으로 진행하며,
2. 특정 정점(예: c)의 이웃하는 정점들을 모두 방문한 경우에는 이전 정점(예: b)으로 되돌아 가서(Backtrack) 탐색을 수행하는 방식으로 진행

[예제] 아래의 그래프에서 DFS 수행 과정 (재귀호출 이용)



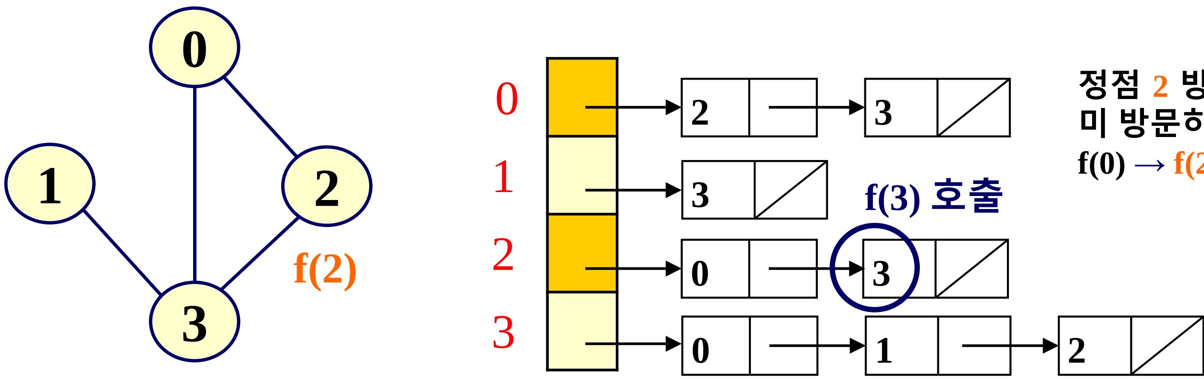
그래프 탐색 (2/8)

❑ 깊이 우선 탐색(DFS) contd. →: 호출 →: 반환



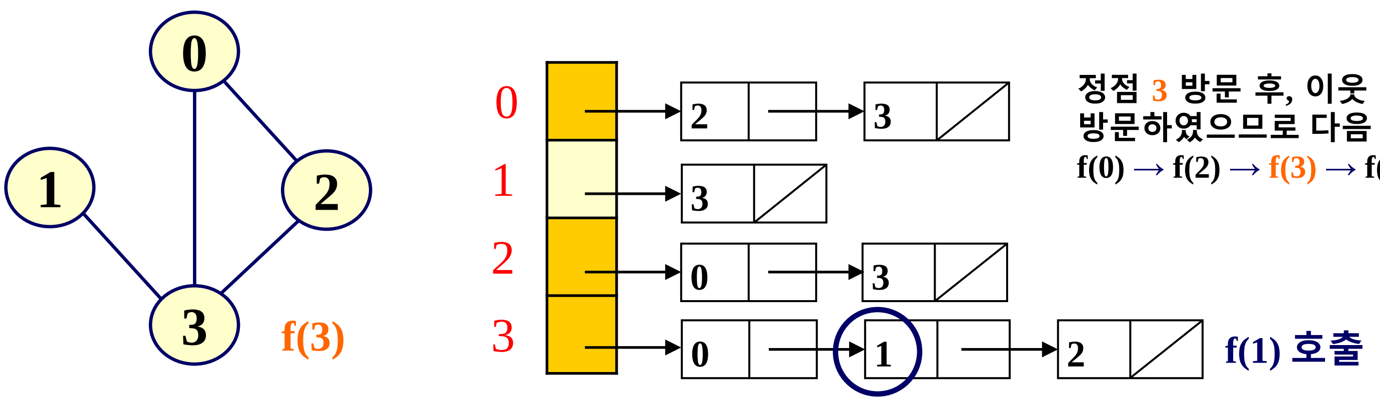
정점 0 방문 후, 이웃 정점 2 선택

f(0) → f(2)



정점 2 방문 후, 이웃 정점 0을 선택하지만 0은 이미 방문하였으므로 다음 이웃 정점 3 선택

f(0) → f(2) → f(3)

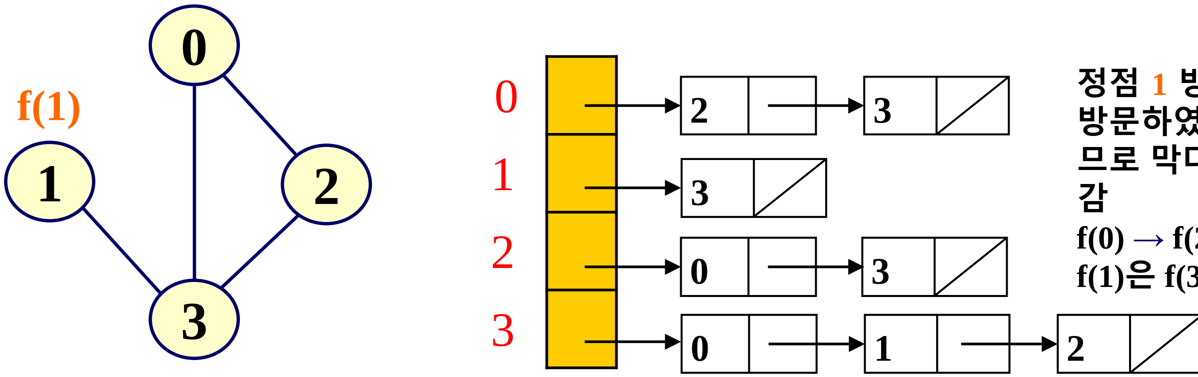


정점 3 방문 후, 이웃 정점 0 선택하지만 0은 이미 방문하였으므로 다음 이웃 정점 1 선택

f(0) → f(2) → f(3) → f(1)

그래프 탐색 (3/8)

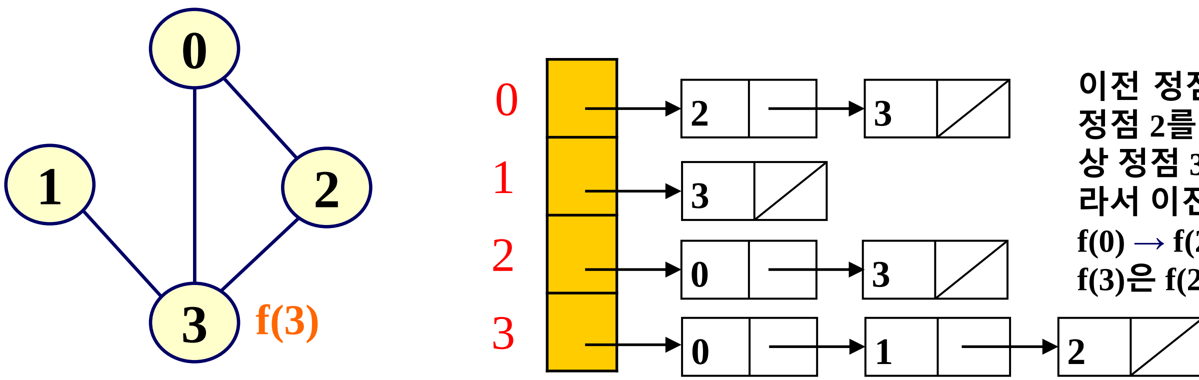
❑ **깊이 우선 탐색**(DFS) contd. →: 호출 →: 반환



정점 1 방문 후, 이웃 정점 3 선택하지만 3은 이미 방문하였으며 더 이상 정점 1의 이웃 정점이 없으므로 막다른 길임. 따라서 이전 정점 3으로 되돌아감

f(0) → f(2) → f(3) → f(1) → f(3)

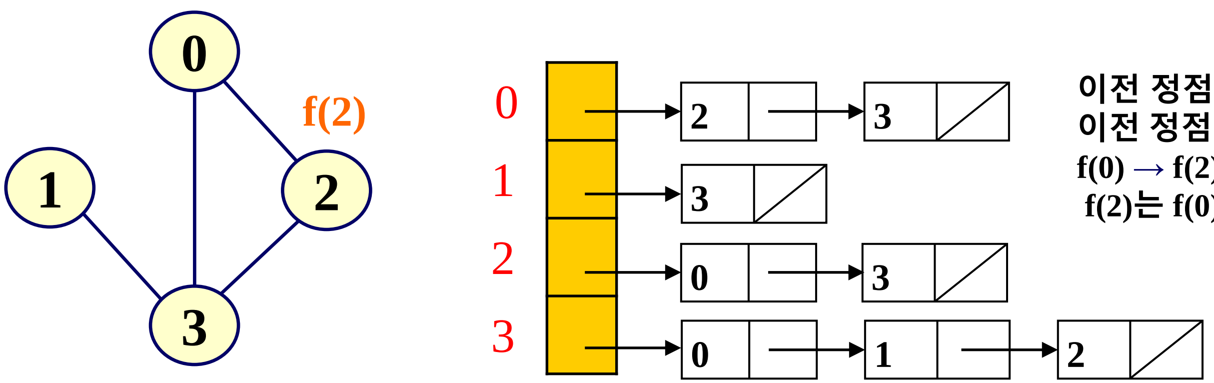
f(1)은 f(3)에게 return None



이전 정점 3으로 되돌아와, 정점 3의 다음 이웃인 정점 2를 선택하지만 2는 이미 방문하였으며 더 이상 정점 3의 이웃 정점이 없으므로 막다른 길임. 따라서 이전 정점 2로 되돌아감

f(0) → f(2) → f(3) → f(1) → f(3) → f(2)

f(3)은 f(2)에게 return None



이전 정점 2로 되돌아와 보니 막다른 길임. 따라서 이전 정점 0으로 되돌아감

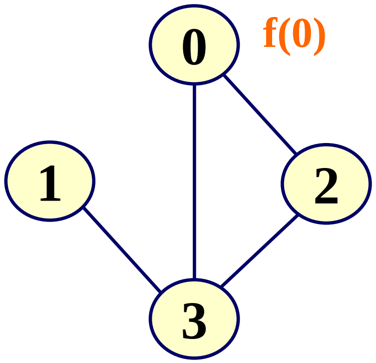
f(0) → f(2) → f(3) → f(1) → f(3) → f(2) → f(0)

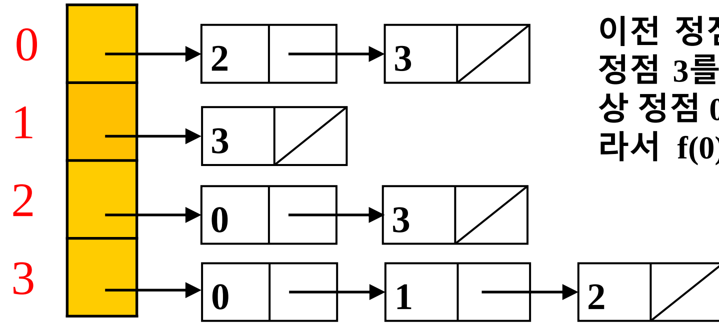
f(2)는 f(0)에게 return None

그래프 탐색 (4/8)

❑ 깊이 우선 탐색(DFS) contd.

→ : 호출 → : 반환





이전 정점 0으로 되돌아와 정점 0의 다음 이웃인 정점 3를 선택하지만 3는 이미 방문하였고, 더 이상 정점 0의 이웃 정점이 없으므로 막다른 길임. 따라서 f(0)은 return None 반환

- **DFS를 위한 함수 dfs 정의**

```
1  adj_list = [[2, 1], [3, 0], [3, 0], [9, 8, 2, 1],
2              [5], [7, 6, 4], [7, 5], [6, 5], [3], [3]]
3  N = len(adj_list)
4  visited = [False] * N
5
6  def dfs(v):
7      visited[v] = True
8      print(v, ' ', end='')
9      for i in adj_list[v]:
10         if not visited[i]:
11             dfs(i)
12
13 print('DFS 방문 순서:')
14 for i in range(N):
15     if not visited[i]:
16         dfs(i)
```

- **라인 1-2:** 인접 리스트로 그래프 표현
- **라인 3:** 지역 변수 N 선언 후 인접 리스트의 정점 수 할당
- **라인 4**: 각 노드 v의 방문 여부 판단을 위한 N 사이즈를 가진 python 리스트 생성

- dfs(v) -

- **라인 7:** v의 방문 여부를 True로 설정
- **라인 8:** v를 출력(방문)
- **라인 9-11**: 아직 방문하지 않은 v의 모든 이웃 정점에 대해 dfs 호출

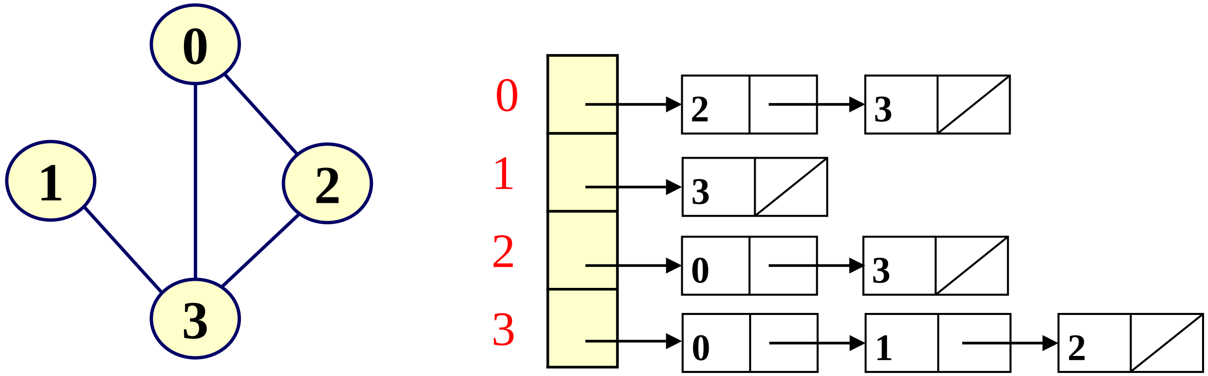
```
DFS 방문 순서:
0  2  3  9  8  1  4  5  7  6
```

결과

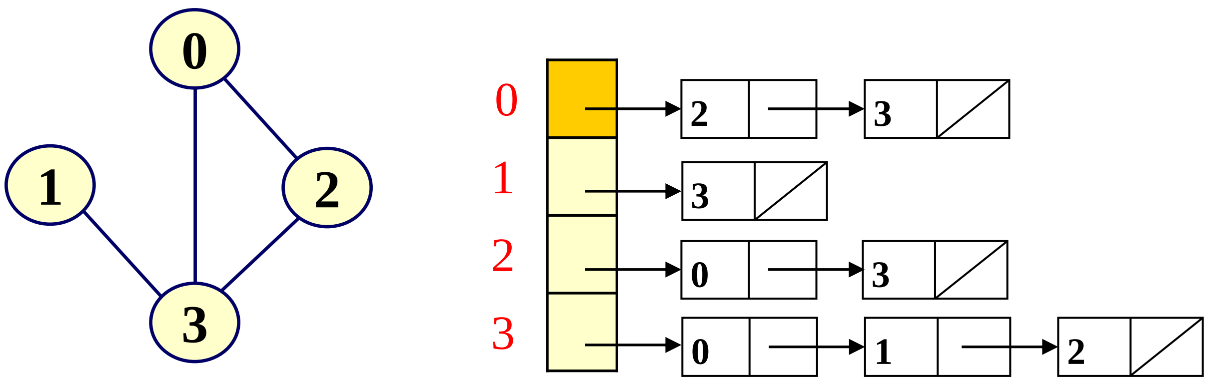
그래프 탐색 (5/8)

❑ 너비 우선 탐색(BFS) – 이진 트리에서의 레벨 순회와 유사

[예제] 아래의 그래프에서 BFS 수행 과정 (큐 사용)



empty 큐



정점 0에 대하여 visited[0]을 True로 설정 후, 0을 enqueue

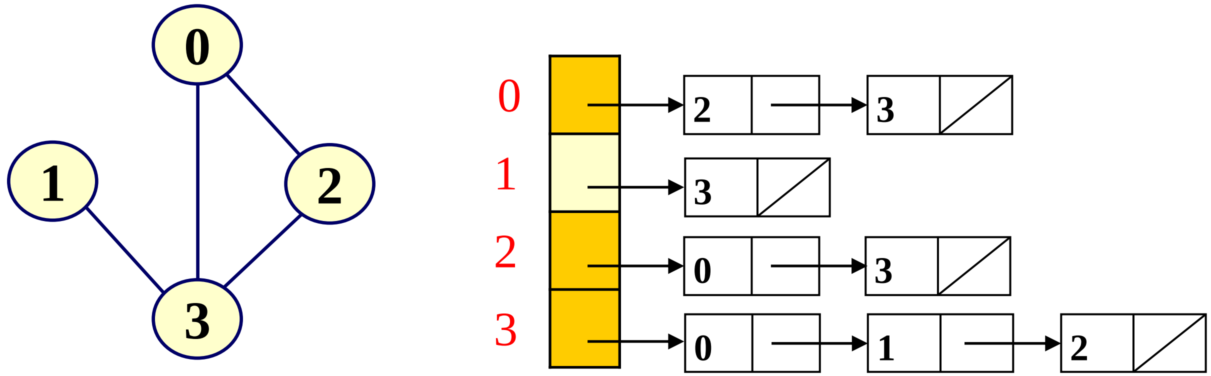
0

IN OUT

정점 0을 dequeue 후 방문(출력)

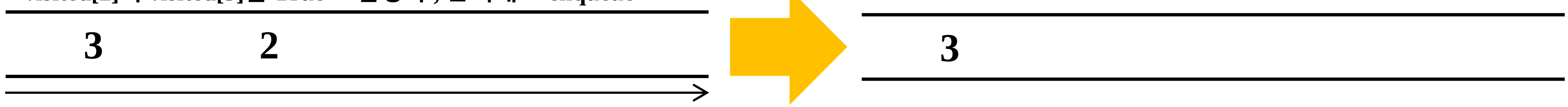
그래프 탐색 (6/8)

❑ **너비 우선 탐색**(BFS) – **이진 트리에서의 레벨 순회와 유사** contd.

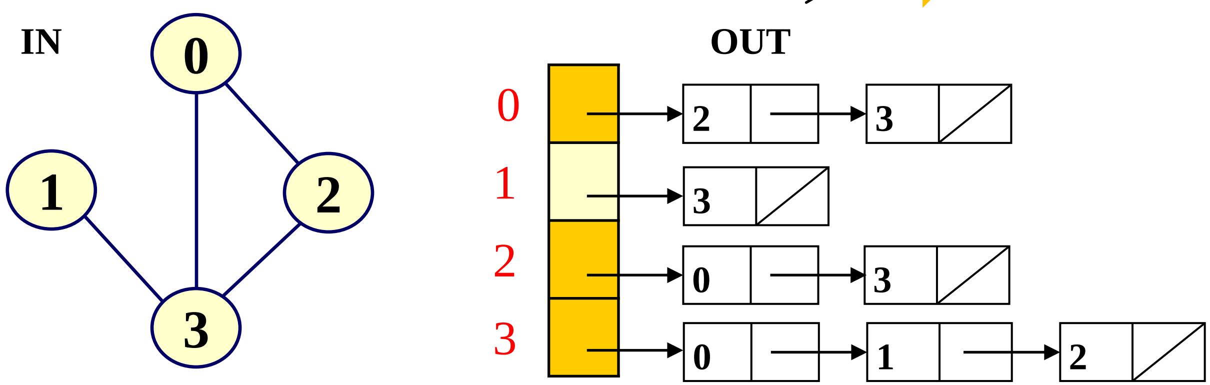


정점 0의 이웃 정점이면서 현재 visited = False인 2와 3에 대하여 visited[2]와 visited[3]을 True로 설정 후, 순서대로 enqueue

정점 2를 dequeue 후 방문(출력)

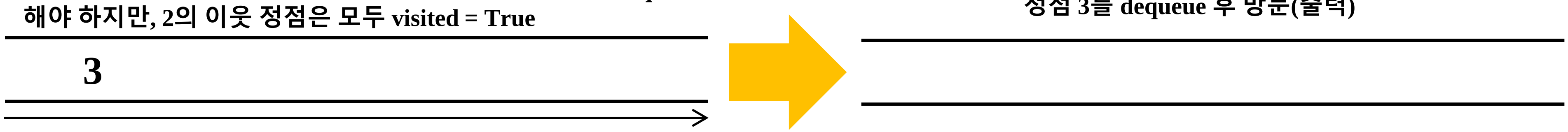


IN **OUT**



정점 2의 이웃 정점이면서 현재 visited = False인 정점을 enqueue 해야 하지만, 2의 이웃 정점은 모두 visited = True

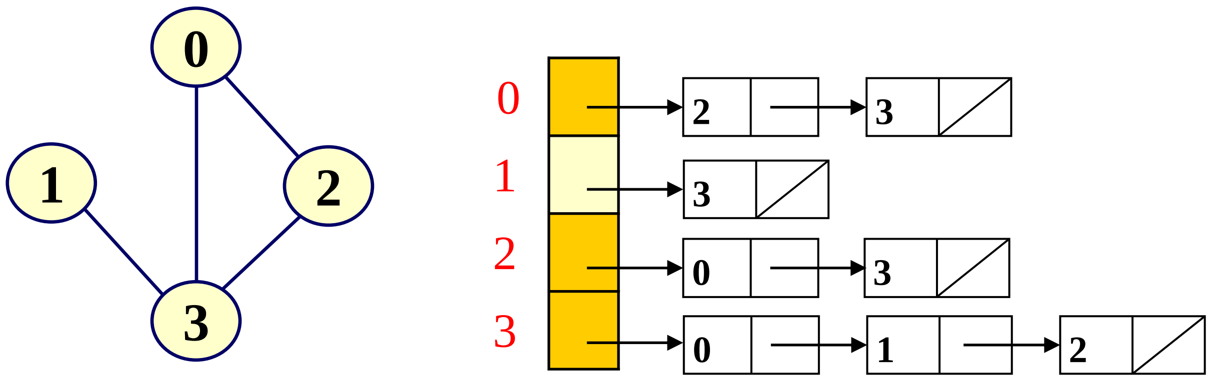
정점 3를 dequeue 후 방문(출력)



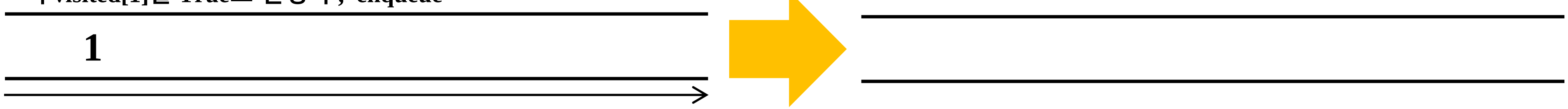
IN **OUT**

그래프 탐색 (7/8)

❑ **너비 우선 탐색**(BFS) – **이진 트리에서의 레벨 순회와 유사** contd.



정점 3의 이웃 정점이면서 현재 visited = False인 정점 1에 대하여 visited[1]을 True로 설정 후, enqueue



IN

OUT

정점 1를 dequeue 후 방문(출력)

정점 3의 이웃 정점이면서 현재 visited = False인 정점 1에 대하여 visited[1]을 True로 설정 후, enqueue

큐가 empty이므로 BFS 종료

-(DFS와) BFS의 수행 시간 분석-

- DFS와 BFS의 수행 시간은 탐색이 각 정점을 한번씩 방문하며, 각 간선을 한번씩만 사용하여 탐색하기 때문에 **O(N+M)**, 여기서 N은 그래프의 정점의 수이고, M은 간선의 수
- DFS와 BFS는 정점의 방문 순서나 간선을 사용하는 순서만 다를 뿐임

그래프 탐색 (8/8)

❑ 너비 우선 탐색(BFS) – 이진 트리에서의 레벨 순회와 유사 contd.

- **BFS를 위한 함수 bfs 정의**

```
1  from clqueue import Queue
2  adj_list = [[2, 1], [3, 0], [3, 0], [9, 8, 2, 1],
3              [5], [7, 6, 4], [7, 5], [6, 5], [3], [3]]
4  N = len(adj_list)
5  visited = [False] * N
6
7  def bfs(v):
8      queue = Queue()
9      visited[v] = True
10     queue.enqueue(v)
11     while not queue.is_empty():
12         v = queue.dequeue()
13         print(v, ' ', end='')
14         for i in adj_list[v]:
15             if not visited[i]:
16                 visited[i] = True
17                 queue.enqueue(i)
18
19 print('BFS 방문 순서:')
20 for i in range(N):
21     if not visited[i]:
22         bfs(i)
```

- **라인 2-3:** 인접 리스트로 그래프 표현
- **라인 4:** 지역 변수 N 선언 후 인접 리스트의 정점 수 할당
- **라인 5**: 각 노드 v의 방문 여부 판단을 위한 N 사이즈를 가진 python 리스트 생성

- bfs(v) -

- **라인 8:** empty 큐 queue 생성
- **라인 9:** queue에 enqueue하기 전에 미리 v의 방문 여부를 True로 설정
- **라인 10**: v를 queue에 enqueue
- **라인 11**: queue가 empty일 때까지 while-루프 수행
  - **라인 12-13**: 가장 먼저 enqueue된 정점 v를 dequeue 후 출력(방문)
  - **라인 14-17**: 아직 방문하지 않은 v의 모든 이웃 정점에 대해 방문 여부를 미리 True로 설정 후 queue에 enqueue

```
BFS 방문 순서:
0  2  1  3  9  8  4  5  7  6
```

결과