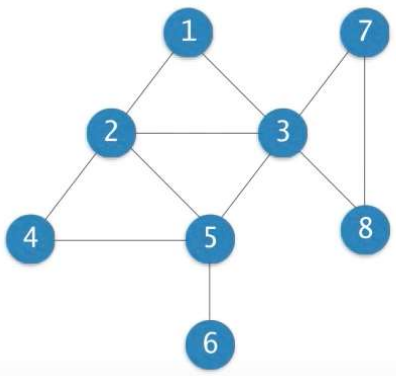
# 알고리즘10주차(그래프)



# 그래프 개념과 표현

## Graph

- (무방향) 그래프 G = (V(정점의 집합), E(간선의 집합))
  - V : 노드 혹은 정점(vertex)
  - E : 노드쌍을 연결하는 Edge 혹은 Link
  - Object들 간의 이진관계를 표현
  - n = |V|, m = |E|

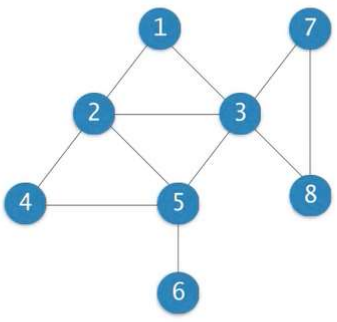



V={1,2,3,4,5,6,7,8}
E={(1,2),(1,3),(2,3),...,(7,8)}
n=8
m=11

- 방향 그래프와 가중치 그래프
  - 방향그래프(Directed Graph) G = (V, E)
  - Edge (u, v)는 u로부터 v로의 방향을 가짐
- 가중치 그래프
  - Edge마다 가중치(weight)가 존재

## 그래프의 표현

- 인접행렬(adjacency matrix)
  - n×n 행렬 $A = (a_{ij}),\ \text{where } a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E, \\ 0 & \text{otherwise.} \end{cases}$

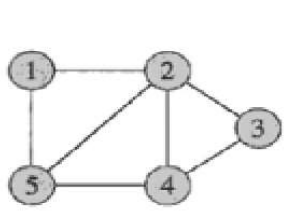


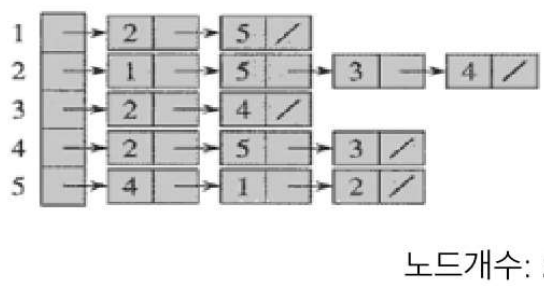

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 2 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| 3 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 |
| 4 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 5 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 8 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |

대칭행렬

저장 공간: O(n²)
어떤 노드 v에 인접한 모든 노드 찾기: O(n) 시간
어떤 에지 (u,v)가 존재하는지 검사: O(1) 시간

- 인접리스트(adjacency list)
  - 정점 집합을 표현하는 하나의 배열과
  - 각 정점마다 인접한 정점들의 연결 리스트

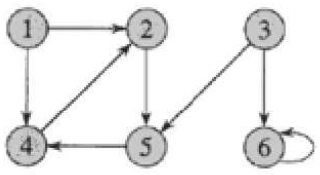


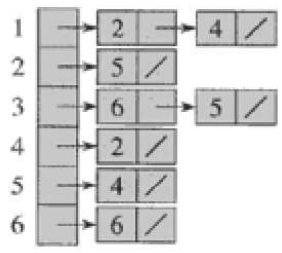




노드개수: 2m

  - 저장 공간 : O(n + 2m)
  - 어떤 노드 v에 인접한 모든 노드 찾기 : O(degree(v)연결리스트의 길이) 시간
  - 어떤 Edge (u, v)가 존재하는지 검사 : O(degree(u)) 시간

- 방향그래프(directed graph)
  - 인접행렬은 비대칭
  - 인접 리스트는 m개의 노드를 가진다



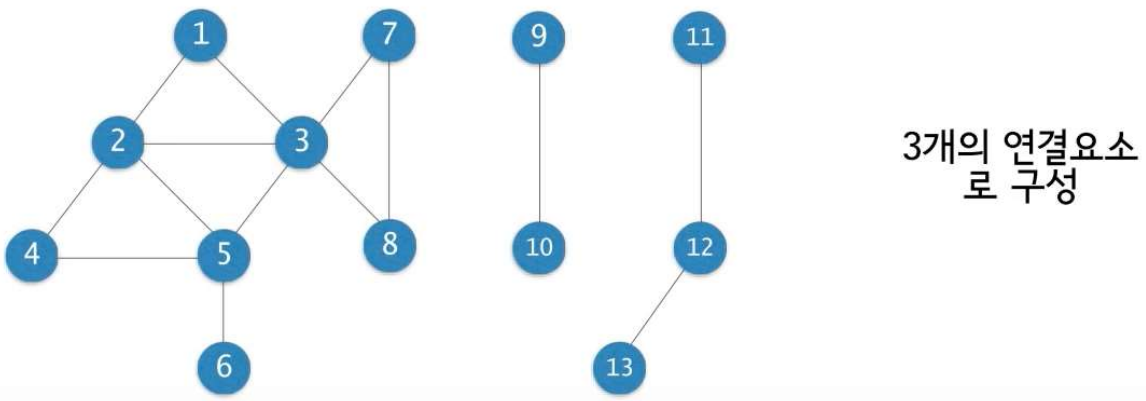

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 1 | 0 |
| 3 | 0 | 0 | 0 | 0 | 1 | 1 |
| 4 | 0 | 1 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 1 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 0 | 1 |

- 가중치 그래프의 인접행렬 표현
  - 엣지의 존재를 나타내는 값으로 1대신 엣지의 가중치를 저장
  - 엣지가 없을 때 혹은 대각선(가중치가 0일때를 고려할때)
    - 그때마다 상황에 따라 정의 가능
    - 특별히 정해진 규칙은 없으며, 그래프와 가중치가 의마하는 바에 따라서
    - Ex) 가중치가 거리 혹은 비용을 의미하는 경우라면 엣지가 없으면 oo(무한대), 대각선은 0
    - Ex) 만약 가중치가 용량을 의미한다면 엣지가 없을때 0, 대각선은 oo(무한대)

### 경로와 연결성

- 인접하다라는 것은 해당 경로를 거쳐서 그 노드에 도달할 수 있다라는 것이고, 연결되어 있다라는 것은 노드와 노드를 연결하는 경로가 존재할 때를 말한다.
- 무방향 그래프 G = (V, E)에서 노드 u와 노드 v를 연결하는 경로(path)가 존재할 때 v와 u는 서로 연결되어 있다고 말함
- 모든 노드 쌍들이 서로 연결된 그래프를 연결된(connected) 그래프라고 한다.

- 연결요소(connected component)



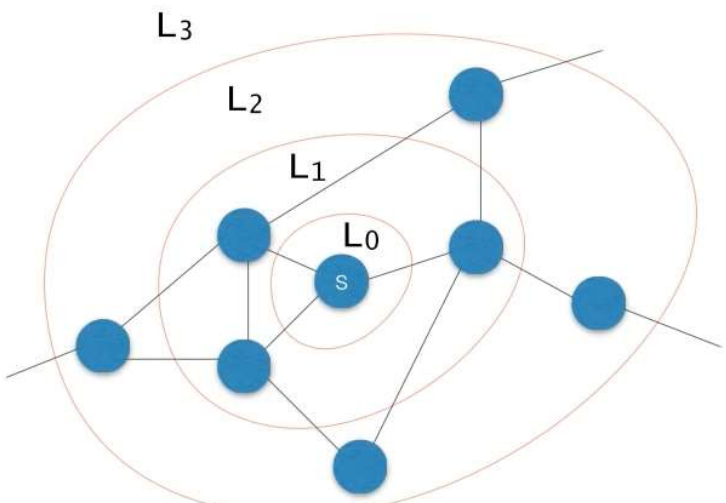

# BFS(Breadth-First Search, 너비우선탐색)

## 그래프 순회

- 순회(traversal)
  - 그래프의 모든 노드들을 방문하는 일
- 대표적 두 가지 방법
  - BFS (Breadth-First Search, 너비우선순회)
  - DFS (Depth-First Search, 깊이우선순회)

### 너비우선탐색(BFS)

- BFS 알고리즘은 다음 순서로 노드들을 방문
  - L0 = {s}, 여기서 s는 출발 노드
  - L1 = L0의 모든 이웃 노드들
  - L2 = L1의 이웃들 중 L0에 속하지 않는 노드들
  - ...
  - Li = Li-1의 이웃들 중 Li-2에 속하지 않는 노드들
  - 한마디로 그래프에서 노드들을 동심원의 형태로 순회하는 방법

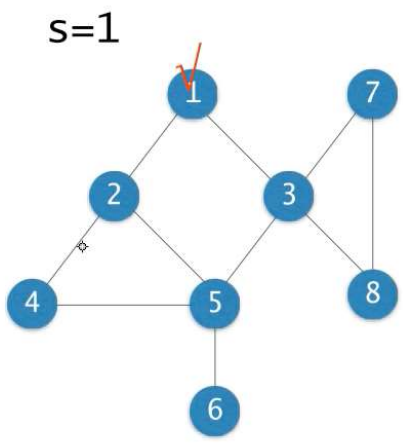



동심원 형태

## 큐를 이용한 너비우선탐색

- 출발노드를 check하고 시작한다.
- 큐에 스타트 노드(1번노드)를 삽입한다.

체크는 이미 방문된 노드라는 표시

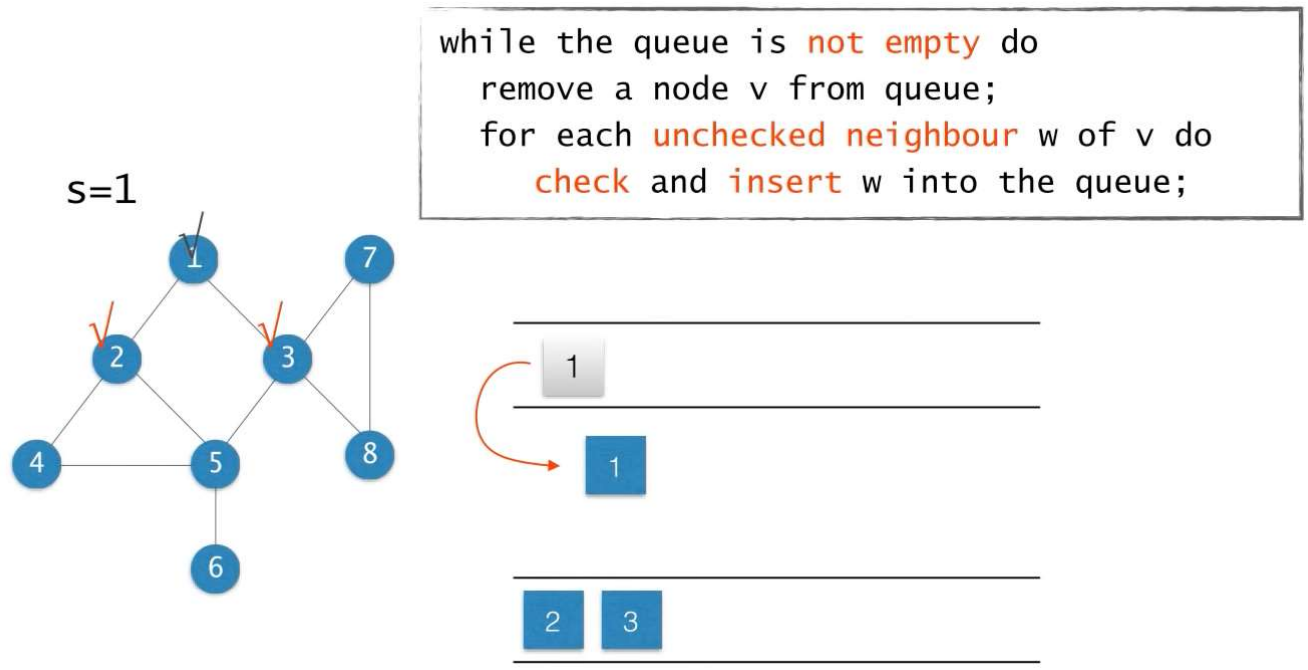



```
1. check the start node;
2. insert the start node into the queue;
```

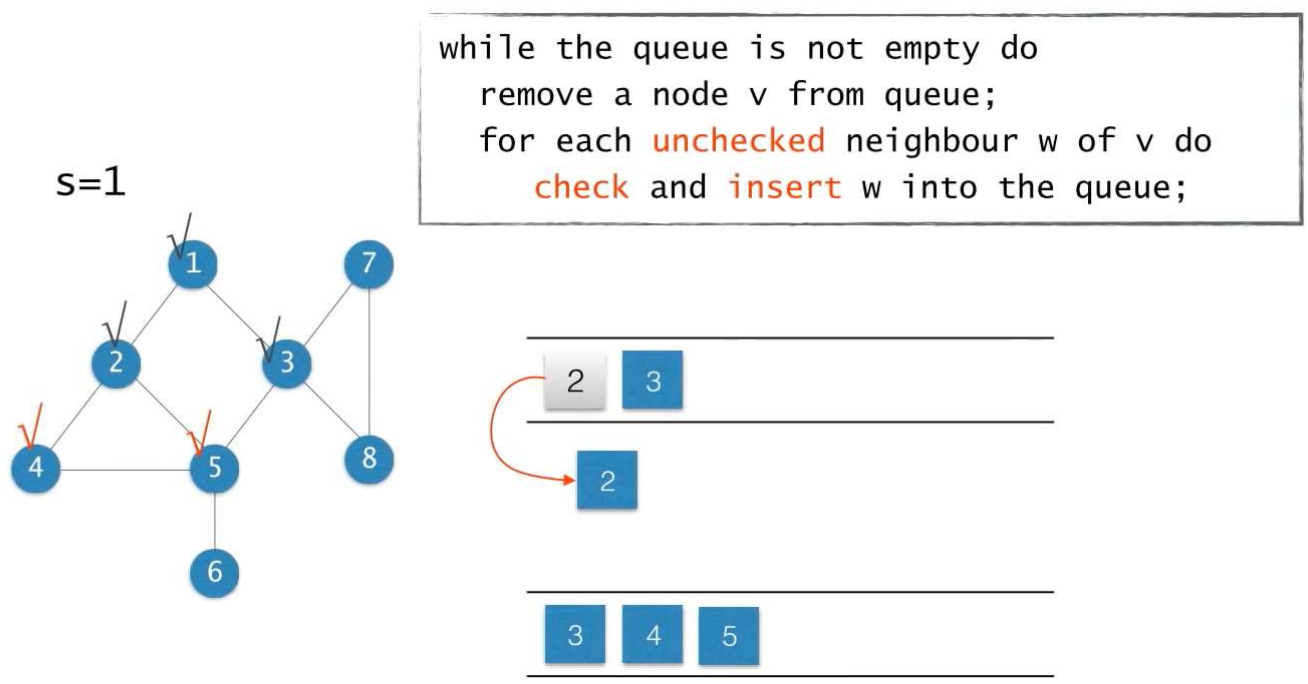
queue

| 1 |
|---|

- whil문을 돌면서 큐가 비어있을 때까지 반복한다.
  - 큐에서 노드(v)를 하나 꺼내고
  - 꺼낸 노드의 인접노드 중, 아직 방문되지 않은(unchecked) 노드들(w)을 체크하고 큐에 넣는다.
  - 이 때, 큐에 넣는 순서는 중요하지 않다.




- 다시 큐에서 노드를 하나 꺼내고(2번 노드)
- 2번 노드의 체크되지 않은 인접 노드들(4, 5번 노드)을 체크상태로 변경하고 큐에 넣는다.




- 이런 방법으로 큐가 비어있는 상태가 될때까지 반복한다.
- 최종적으로 노드를 방문한 순서는 1, 2, 3, 4, 5, 7, 8, 6 이 된다. 하지만, 이 방문 순서는 유일하지 않다. 큐에 인접노드를 삽입하는 순서에 따라 달라지기 때문이다.

## BFS pesudo code

- 그래프 G, 출발 노드 S

```
00 BFS(G, s) 01   Q <- null; 02   Enqueue(Q, s); 03   while Q != null do 04     u <- Deq
ueue(Q); 05     for each v adjacent to u do 06       if v is unvisited then 07         m
ark v as visited; 08         Enqueue(Q, v);
```

## BFS와 최단경로

- BFS는 단순히 그래프의 모든 노드를 방문하는 것 이상의 추가적인 중요한 일을 할 수 있다. 최단경로를 구하는 일이다.
- s에서 Li에 속한 노드까지 최댄 경로의 길이는 i이다.
  - 여기서 경로의 길이는 경로에 속한 엣지의 수를 의미한다.
- BFS를 하면서 각 노드에 대해서 최단 경로의 길이를 구할 수 있다.
- 입력
  - 방향 혹은 무방향 그래프 G = (V, E), 그리고 출발노드 s

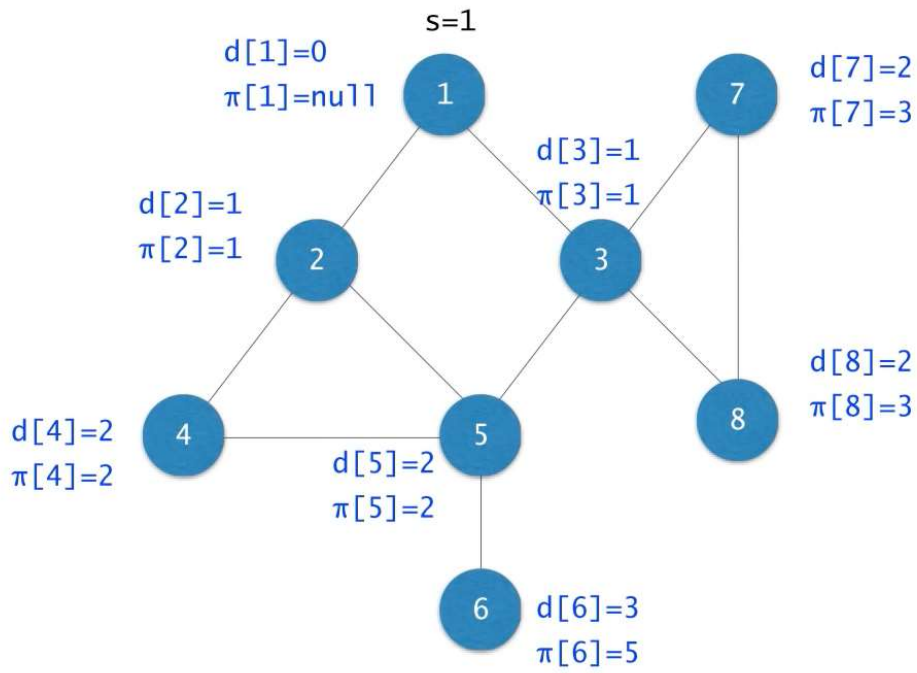
- 출력
  - 모든 노드 v에 대해서
  - d[v] = s로 부터 v까지의 최단 경로의 길이(엣지의 개수)
  - π[V] = s로 부터 v까지의 최단경로상에서 v의 직전 노드(predecessor)
- Pseudo code
  - 02 - 04 : 모든 노드 u에 대해서 d[], π[]를 초기화
  - 05 - 06 : 스타트 노드의 d[], π[]를 설정
  - 11 : d[v] 가 -1인가를 체크하여 unvisited 체크를 구현
  - 12 - 13 : unvisited 노드에 대하여 d[v], π[v]를 저장
    - 최단경로 길이 d[v]는 u까지의 최단경로길이 d[u]를 지나오는 것이므로 d[u] + 1이 될 것이고,
    - v노드의 최단경로상에서 v의 직전노드는 u가 된다.
  - 14 : unchecked 노드만 큐에 들어갈 수 있으므로 어떤 노드도 큐에 두번 들어가지는 않는다.

```
00 BFS(G, s) 01   Q <- null; 02   for each node u do 03     d[u] <- -1; 04     π[u] <- n
ull; 05   d[s] <- 0; //distance from s to s is 0 06   π[s] <- null; //no predecessor of
s 07   Enqueue(Q, s); 08   while Q != null do 09     u <- Dequeue(Q); 10     for each v
adjacent to u do 11       if (d[v] == -1) then 12         d[v] <- d[u] + 1; //distance t
o v 13         π[v] <- u; //u is the predecessor of v 14         Enqueue(Q, v);
```

## 시간복잡도

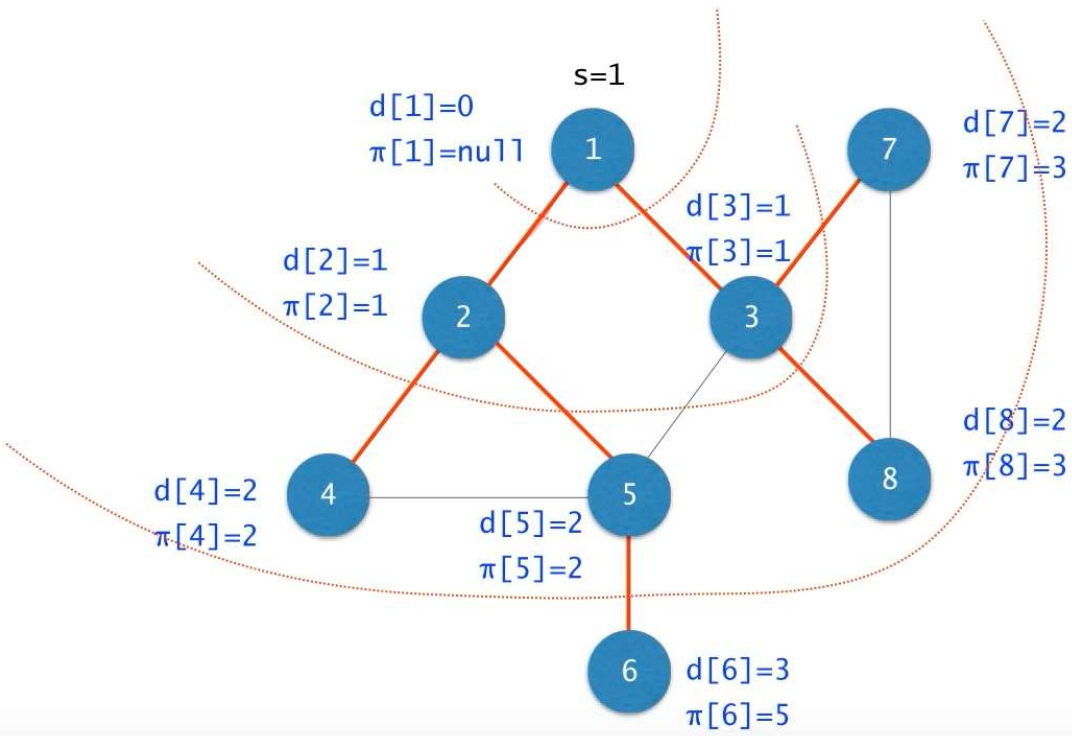
- 02라인 for 의해 기본적으로 O(n)이다.
- 실제로는 08라인의 while문이 알고리즘의 시간복잡도를 결정한다.
- 기본적으로 while문이 한번 돌 때마다 큐에서 노드 하나씩 꺼내므로 while문은 최대 n번 돈다.
- 10라인의 for문은 u의 degree() 만큼 돈다. 리스트를 인접 행렬로 구현하느냐, 인접리스트로 구현하느냐에 따라 for문의 시간복잡도가 달라진다.
  - degree(v)는 어떤 한 노드 v에 실제로 인접한 노드의 수
- 그래프를 **인접 행렬로 구현할 경우** degree(v)를 찾으려면 O(n)이 든다. 따라서 인접행렬로 구현했을 때의 while문의 시간복잡도는 O(n^2)이 된다.
- **인접 리스트로 구현할 경우** 전체 그래프에서 보면, for 문은 결국 모든 노드들의 degree() 만큼 돌 것이다. 인접리스트에서 그것은 2m이다.(무방향 그래프에서 총 엣지의 수) 시간복잡도는 O(m). 따라서, while문의 시간복잡도는 O(n + m)이 된다.
- 결과적으로 인접 리스트의 최악의경우 m이 n이 되므로 최악의 경우가 아닌 이상 인접 리스트로 구현하는 것이 좀 더 효율적이다.

## BFS로 구현한 d[]와 π[] 예시



## BFS 트리

- 각 노드 v와 π[v]를 연결하는 엣지들로 구성되는 트리
- BFS 트리에서 s에서 v까지의 경로는 s에서 v까지 가는 최단 경로
- 어떤 엣지도 동심원의 2개의 layer(L0에서 L2로 가지 않는다)를 건너가지 않는다.(동일 레이어의 노드를 연결하거나, 혹은 1개의 layer를 건너간다.)




## 너비우선탐색: 최단 경로 출력하기

- 출발점 s에서 노드 v까지의 경로 출력하기
  - resursion으로 해결한다.
  - s에서 v까지 가는 최단 경로는 먼저 s에서 π[v]까지 가는 경로를 출력하고, v를 추가로 출력하면 된다.

```
00 PRINT-PATH(G, s, v) 01   if v = s then 02    print s; 03  else if π[v] = null then // 실제로 s에서 v까지 가는 경로가 없을 경우(최단경로도 없음) 04    print "no path from s to v exists"; 05  else 06    PRINT-PATH(G, s, π[v]); 07    print v;
```

## 너비우선탐색(BFS) 정리

- 그래프가 connected 라면 모든 노드를 방문하게 된다. 하지만, 그래프가 **disconnected** 이거나 혹은 방향 그래프라면 BFS에 의해서 모든 노드가 방문되지 않을 수도 있다.
- disconnected 그래프의 모든 노드를 방문하려면 BFS를 반복하여 모든 노드 방문
  - 전체 노드중 unvisited 노드가 없을 때까지 BFS를 반복한다.

```
BFS-ALL(G)  while there exists unvisited node v    BFS(G, V)
```

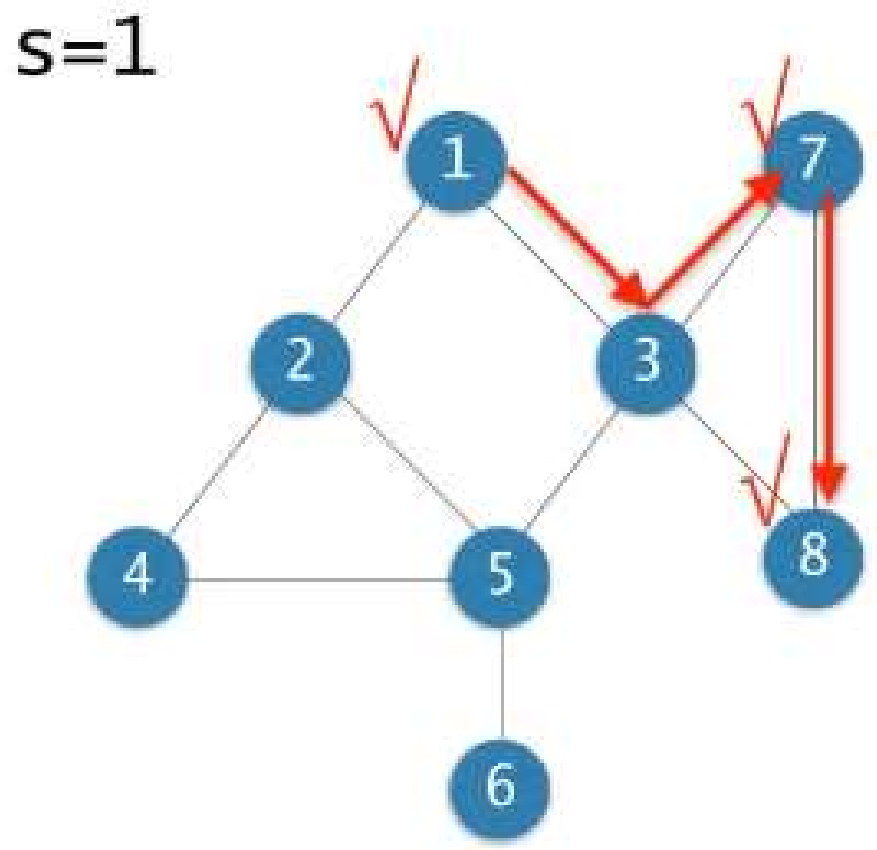
# DFS(Depth-First Search, 깊이우선탐색)

- 이진트리의 순회 방법인 inorder, preorder, postorder 순회방법이 DFS의 이진트리 버전에 해당한다.
- lever order는 BFS의 이진트리 순회 버전이다.

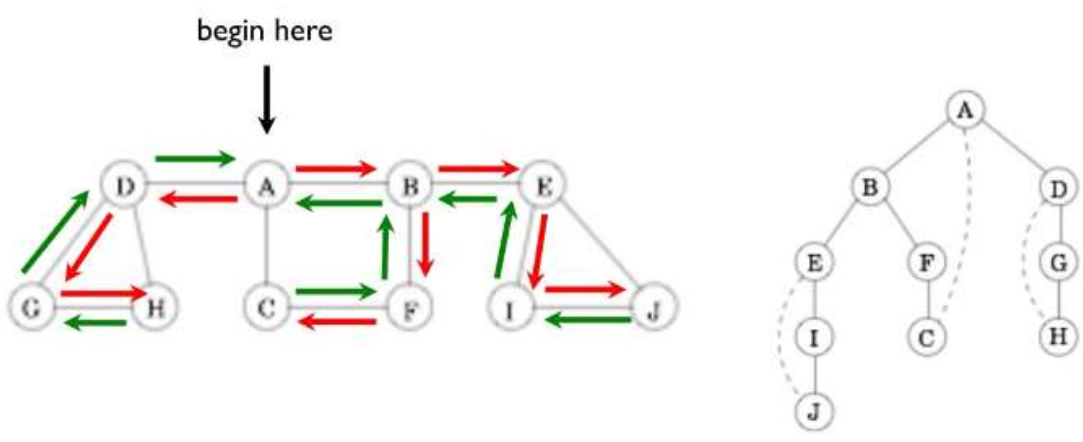
## 깊이우선탐색(DFS)

- 출발점 s에서 시작한다.
- 현재 노드를 visited로 mark하고 인접한 노드들 중 unvisited 노드가 존재하면 그 노드로 간다.
- 2번을 계속 반복한다.

- 노드 8에 도달했을 때처럼 인접한 노드들중 invisited 노드가 존재하지 않는다면, unvisited인 이웃 노드가 존재하지 않는 동안 계속해서 직전 노드로 되돌아간다.
- 다시 2번을 계속 반복한다.
- 시작노드 s로 돌아오고 더 이상 갈 곳이 없으면 종료한다.




- 다음과 같은 흐름으로 깊이우선순회가 이루어 진다.




## DFS, 깊이우선탐색

- 이진트리의 순회를 recursion으로 구현한 것처럼, 깊이우선탐색도 resursion으로 구현하는 것이 간명하다.
- 01 : 먼저 방문한 노드 v에 대해서 visited 체크를 하고
- 02 : v와 인접한 노드 x들에 대해서
- 03 : visited[x]가 No 이면,
- 04 : DFS(G, x)를 recursive하게 호출한다.
- 순회를 위해서 직전의 노드로 돌아가는 행동이 recursion으로 간명하게 구현된다.

```
00 DFS(G, v)01   visited[v] <- YES; 02   for each node x adjacent to v do 03     if visi
ted[x] = No then 04       DFS(G, x);
```

- 그래프가 **disconnected 그래프 이거나 혹은 방향 그래프**라면 DFS에 의해서 모든 노드가 방문되지 않을 수도 있음

- DFS를 반복하여 모든 노드 방문
  - 모든 노드의 visited를 NO로 설정하고,
  - 해당 노드들을 출발노드로 하여 DFS를 호출, 연결되지 않은 그래프에 해당하는 노드는 visited가 NO로 유지되어서 DFS를 호출하게됨
- 시간복잡도
  - 첫번째 for문에 의해서 시간복잡도 O(n)은 피할 수 없고,
  - 두번째 for문에 의해서 v노드와 엣지로 이어진 노드가 visited인지를 체크한다. 따라서, 인접리스트로 표현했다면 시간복잡도는 엣지의 갯수에 비례하게 된다. O(m)
  - 최종적으로 O(n + m)의 시간복잡도를 갖는다.
  - 만약, 인접행렬로 표현했다면 인접노드의 여부를 알기위해서 전체 노드의 수 만큼 검색해야하므로 O(n)이므로, O(n^2)의 시간복잡도를 갖는다.