



제6장
그래프 알고리즘
(Graph Algorithms)

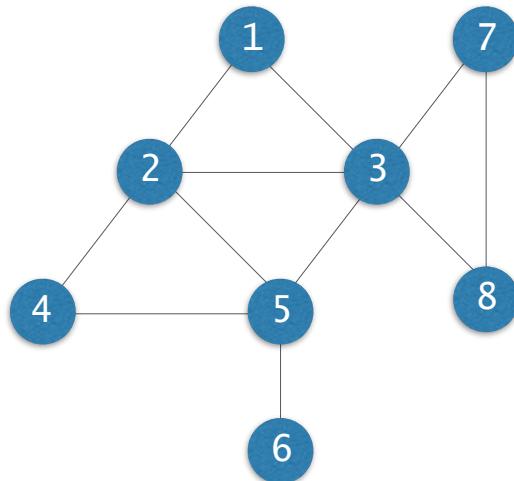
개념과 표현

Concepts and Representations

그래프 (Graph)

- (무방향) 그래프 $G=(V, E)$

- V : 노드(node) 혹은 정점(vertex)
- E : 노드쌍을 연결하는 에지(edge) 혹은 링크(link)
- 개체(object)들 간의 이진관계를 표현
- $n=|V|$, $m=|E|$



$$\begin{aligned}V &= \{1, 2, 3, 4, 5, 6, 7, 8\} \\E &= \{(1, 2), (1, 3), (2, 3), (2, 4), (2, 5), (3, 5), (3, 7), (4, 5), (5, 6), (7, 8)\} \\n &= 8 \\m &= 11\end{aligned}$$

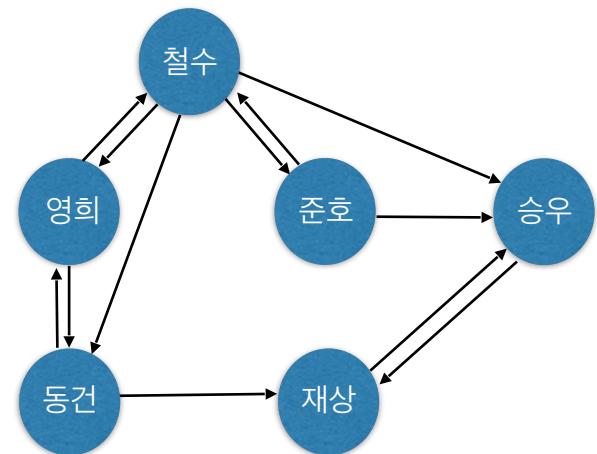
방향 그래프와 가중치 그래프

- 방향그래프(Directed Graph) $G=(V, E)$

- 예지 (u, v) 는 u 로부터 v 로의 방향을 가짐

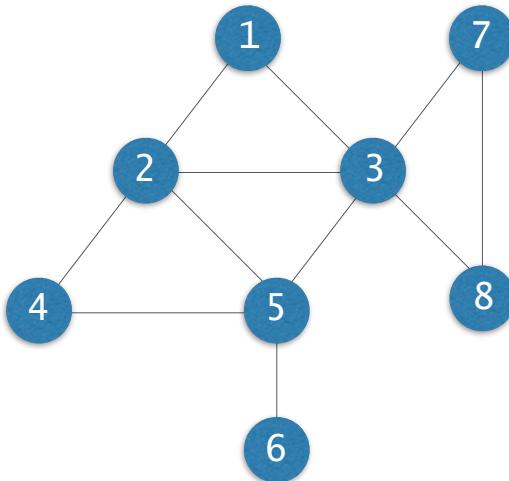
- 가중치(weighted) 그래프

- 예지마다 가중치(weight)가 지정



- 인접행렬 (adjacency matrix)

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



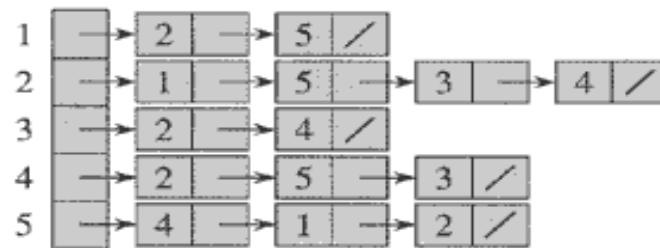
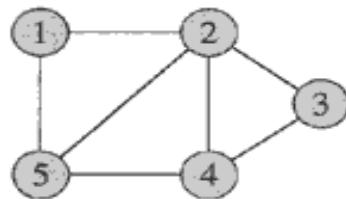
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 2 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| 3 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 |
| 4 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 5 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |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

대칭행렬

저장 공간: $O(n^2)$
 어떤 노드 v 에 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



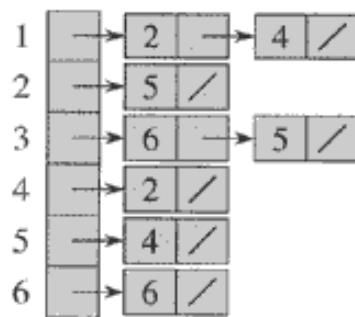
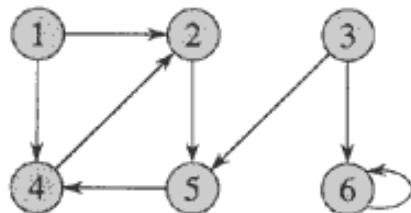
노드개수: $2m$

저장 공간: $O(n+m)$

어떤 노드 v 에 인접한 모든 노드 찾기: $O(\text{degree}(v))$ 시간
 어떤 에지 (u, v) 가 존재하는지 검사: $O(\text{degree}(u))$ 시간

방향그래프

- 인접행렬은 비대칭
- 인접 리스트는 m개의 노드를 가짐



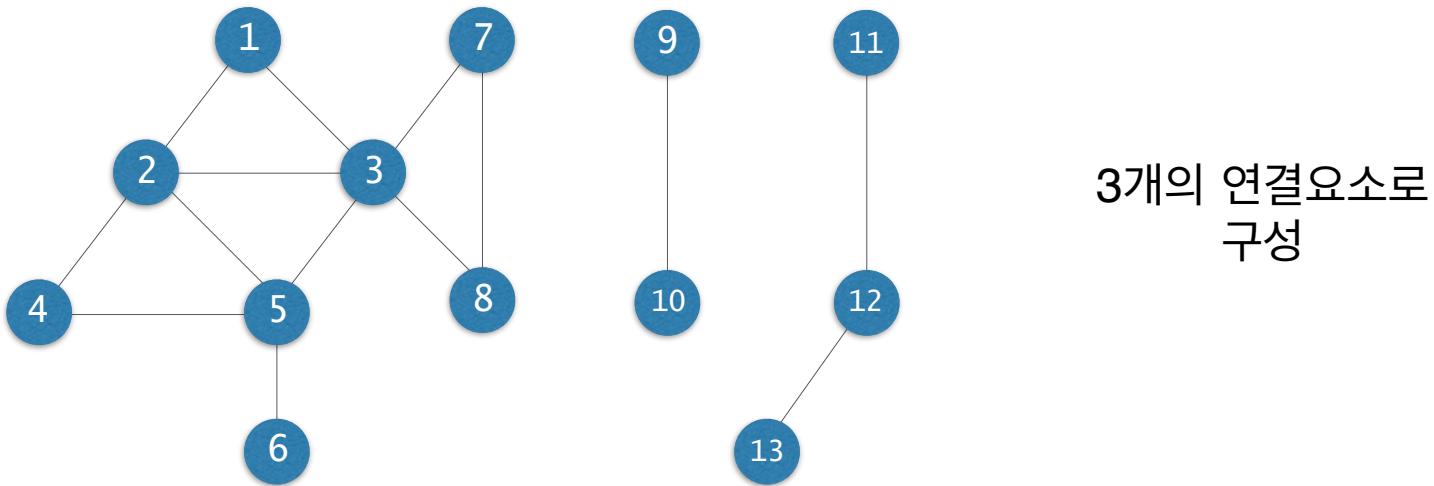
| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 1 | 0 |
| 3 | 0 | 0 | 0 | 0 | 1 | 1 |
| 4 | 0 | 1 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 1 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 0 | 1 |

가중치 그래프의 인접행렬 표현

- 에지의 존재를 나타내는 값으로 1대신 에지의 가중치를 저장
- 에지가 없을 때 혹은 대각선:
 - 특별히 정해진 규칙은 없으며, 그래프와 가중치가 의미하는 바에 따라서
 - 예: 가중치가 거리 혹은 비용을 의미하는 경우라면 에지가 없으면 ∞ , 대각선은 0.
 - 예: 만약 가중치가 용량을 의미한다면 에지가 없을때 0, 대각선은 ∞

경로와 연결성

- 무방향 그래프 $G=(V, E)$ 에서 노드 u 와 노드 v 를 연결하는 경로(path)가 존재할 때 v 와 u 는 서로 **연결되어** 있다고 말함
- 노든 노드 쌍들이 서로 연결된 그래프를 **연결된(connected)** 그래프라고 한다.
- 연결요소 (connected component)**

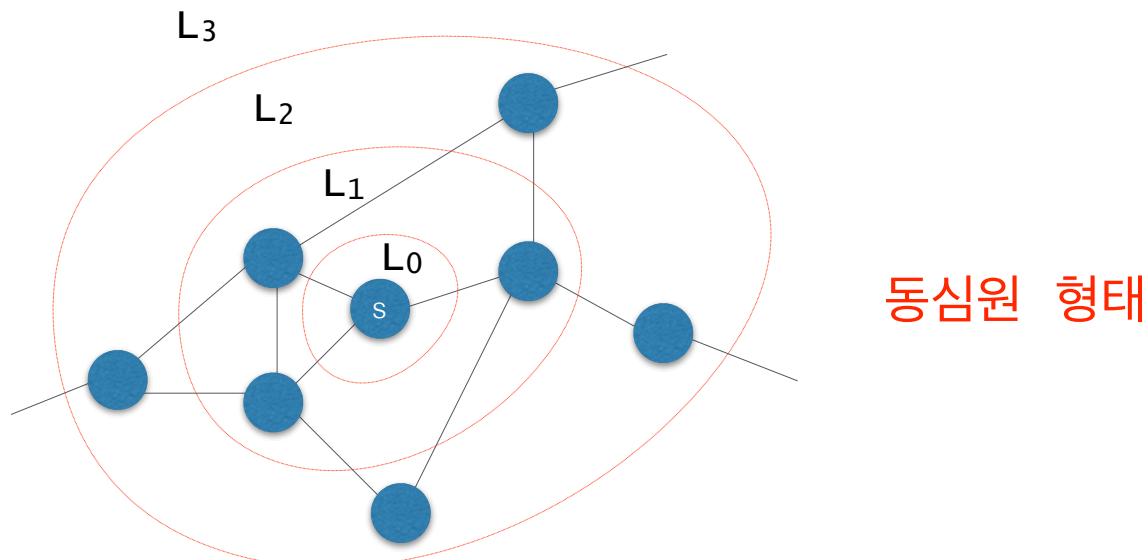


그래프 순회

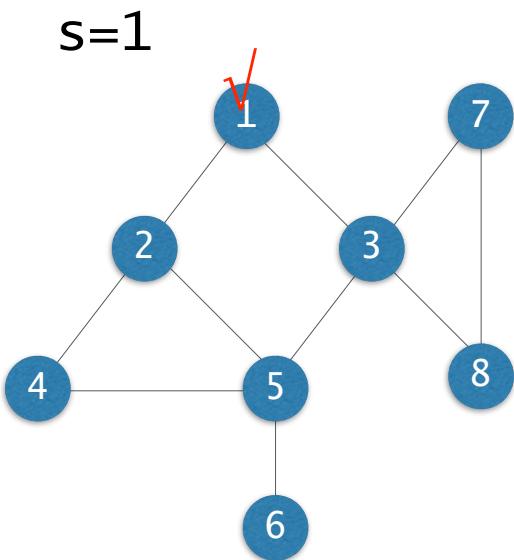
Graph Traversal

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

- BFS 알고리즘은 다음 순서로 노드들을 방문
 - $L_0 = \{s\}$, 여기서 s 는 출발 노드
 - $L_1 = L_0$ 의 모든 이웃 노드들
 - $L_2 = L_1$ 의 이웃들 중 L_0 에 속하지 않는 노드들
 - ...
 - $L_i = L_{i-1}$ 의 이웃들 중 L_{i-2} 에 속하지 않는 노드들



큐를 이용한 너비우선순회



체크는 이미 방문된 노드라는 표시

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

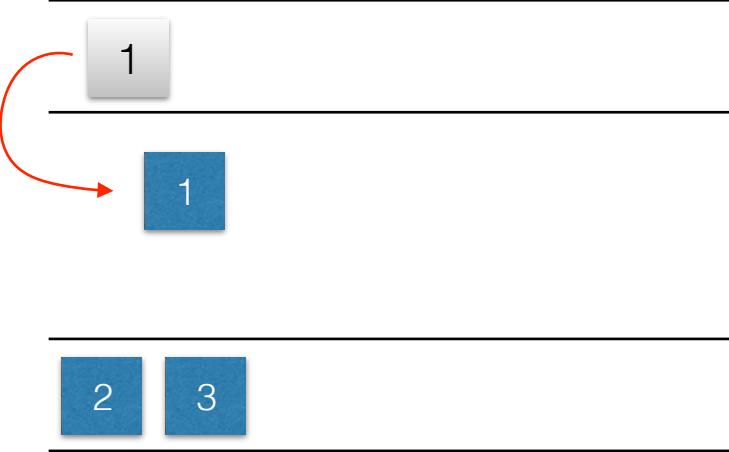
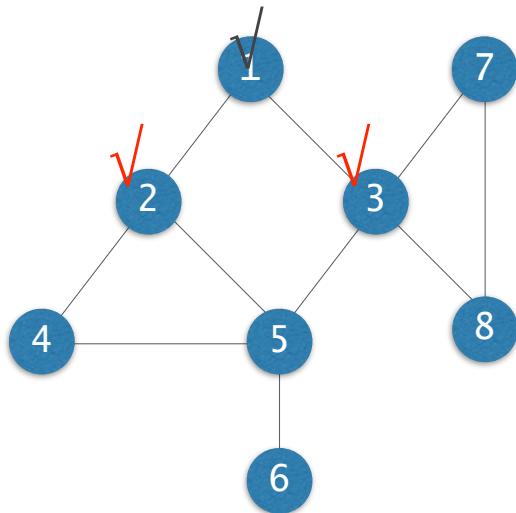
queue

1

큐를 이용한 너비우선순회

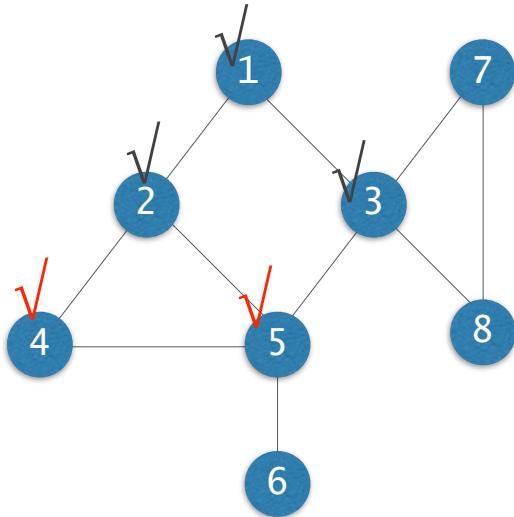
```
while the queue is not empty do  
    remove a node v from queue;  
    for each unchecked neighbour w of v do  
        check and insert w into the queue;
```

$s=1$

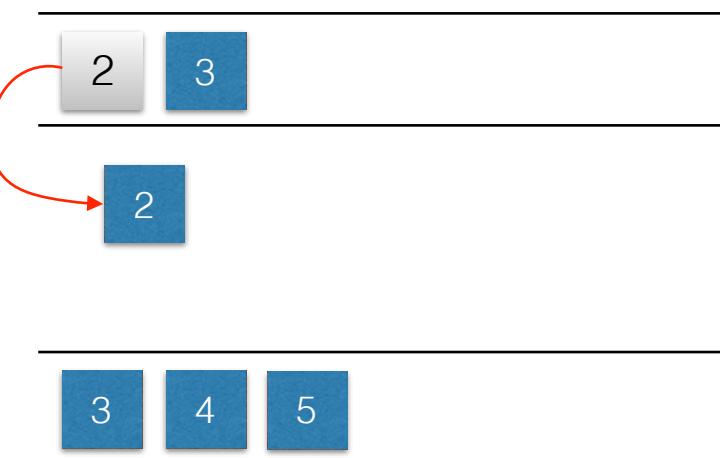


큐를 이용한 너비우선순회

$s=1$

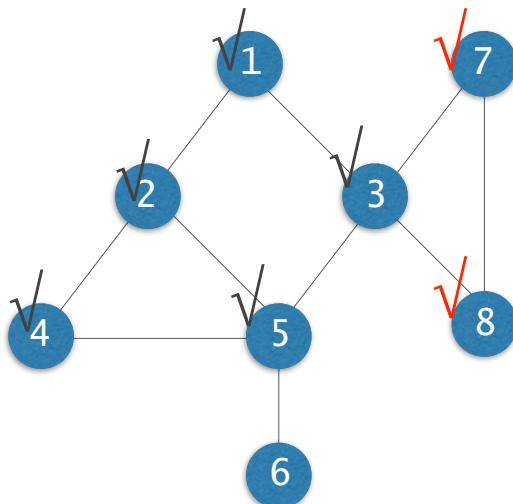


```
while the queue is not empty do  
    remove a node v from queue;  
    for each unchecked neighbour w of v do  
        check and insert w into the queue;
```

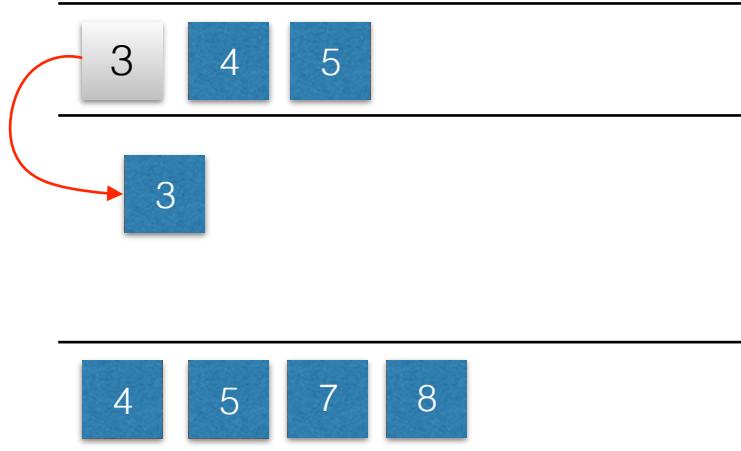


큐를 이용한 너비우선순회

$s=1$

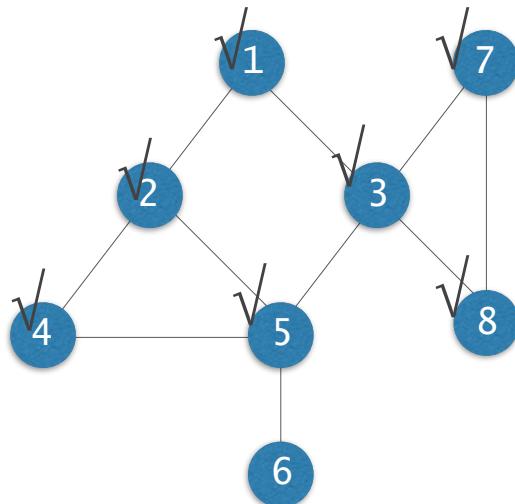


```
while the queue is not empty do  
    remove a node v from queue;  
    for each unchecked neighbour w of v do  
        check and insert w into the queue;
```

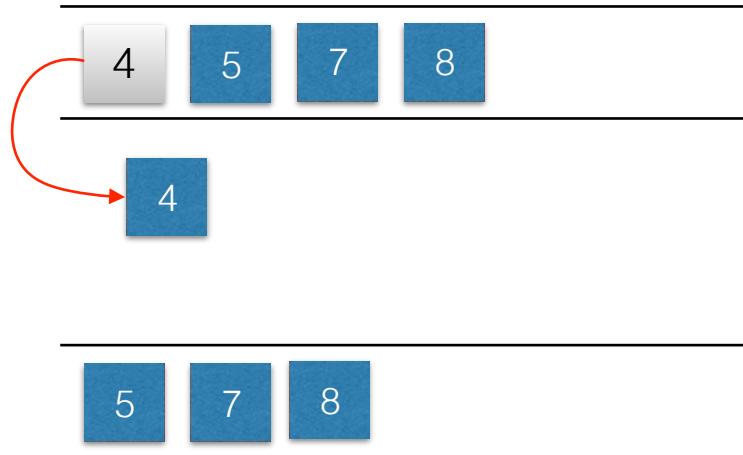


큐를 이용한 너비우선순회

$s=1$

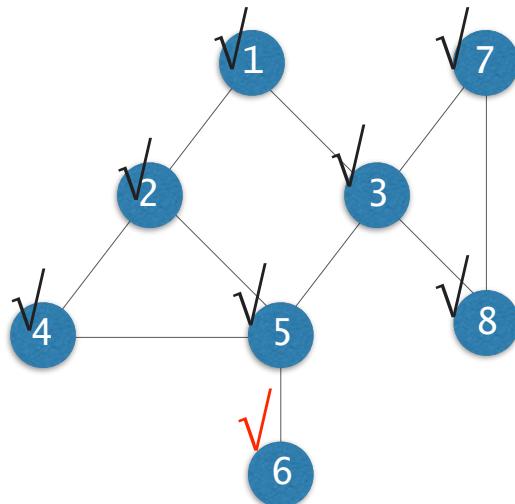


```
while the queue is not empty do  
    remove a node v from queue;  
    for each unchecked neighbour w of v do  
        check and insert w into the queue;
```

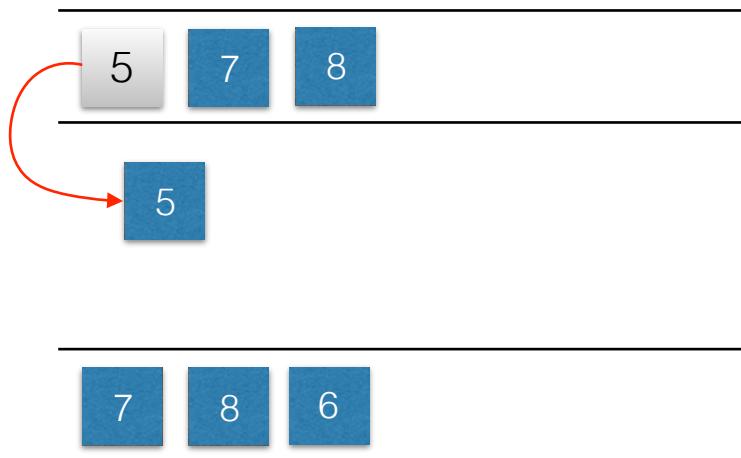


큐를 이용한 너비우선순회

$s=1$

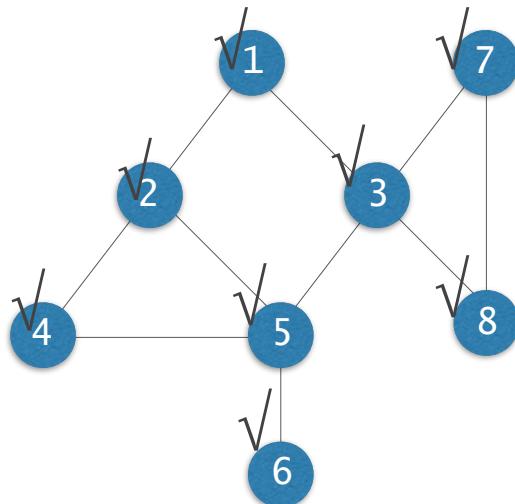


```
while the queue is not empty do  
    remove a node v from queue;  
    for each unchecked neighbour w of v do  
        check and insert w into the queue;
```

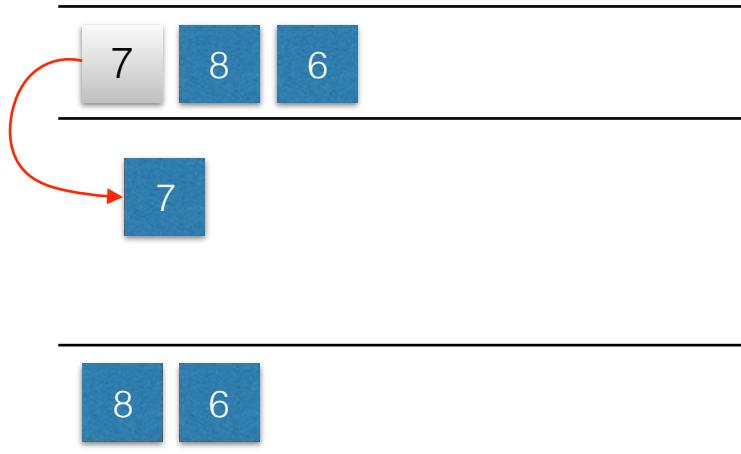


큐를 이용한 너비우선순회

$s=1$

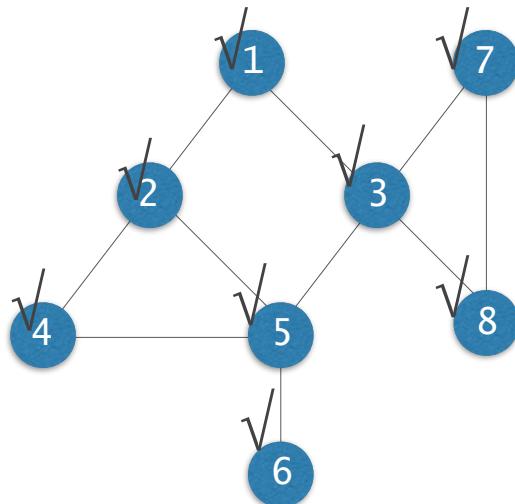


```
while the queue is not empty do  
    remove a node v from queue;  
    for each unchecked neighbour w of v do  
        check and insert w into the queue;
```

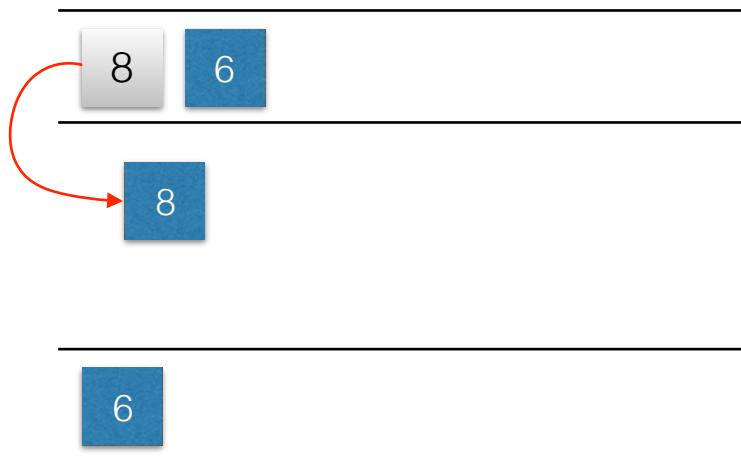


큐를 이용한 너비우선순회

$s=1$

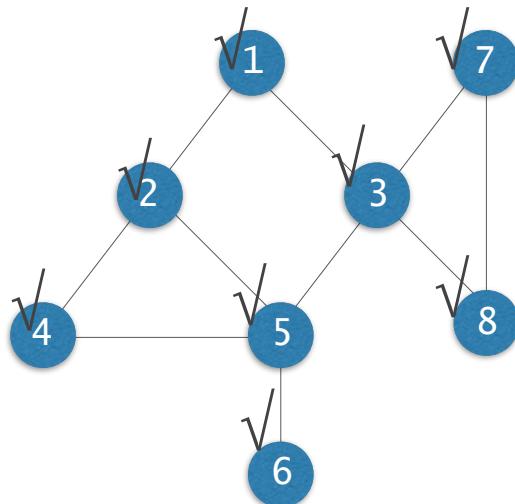


```
while the queue is not empty do  
    remove a node v from queue;  
    for each unchecked neighbour w of v do  
        check and insert w into the queue;
```

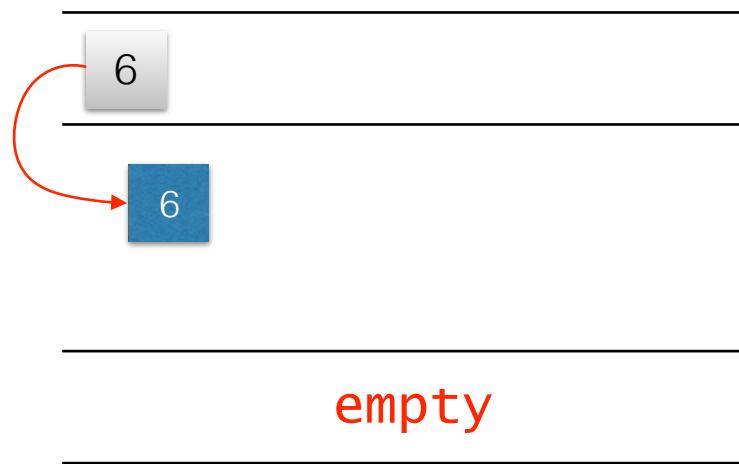


큐를 이용한 너비우선순회

$s=1$



```
while the queue is not empty do  
    remove a node v from queue;  
    for each unchecked neighbour w of v do  
        check and insert w into the queue;
```



노드 방문 순서: 1, 2, 3, 4, 5, 7, 8, 6

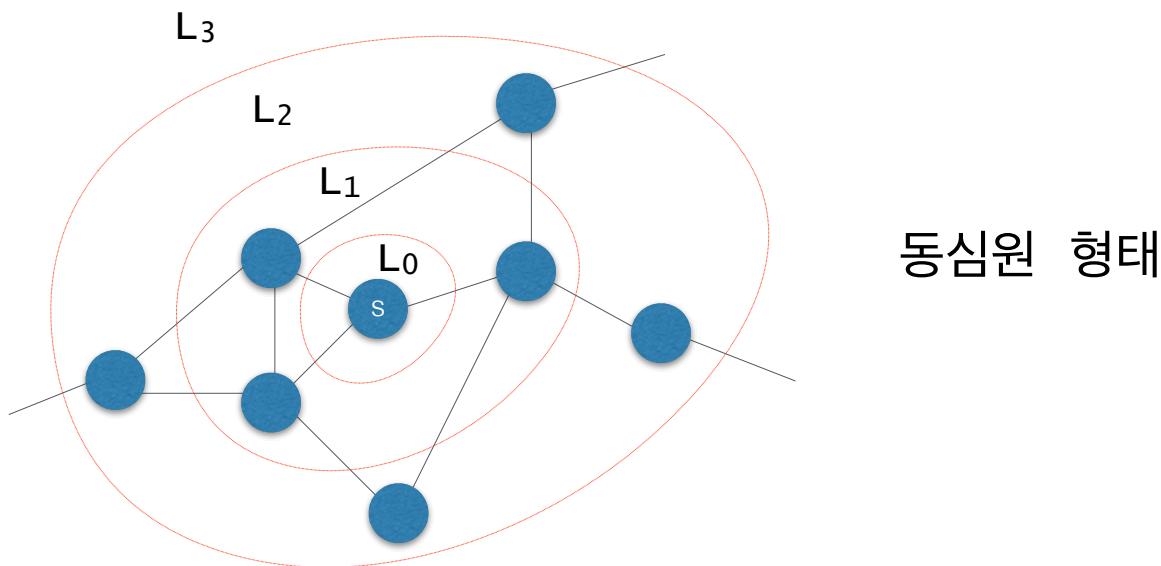
그래프 G 와 출발 노드 s

$BFS(G, s)$

```
Q $\leftarrow \emptyset$ ;  
Enqueue(Q, s);  
while Q $\neq \emptyset$  do  
    u  $\leftarrow$  Dequeue(Q)  
    for each v adjacent to u do  
        if v is unvisited then  
            mark v as visited;  
            Enqueue(Q, v);  
    end.  
end.  
end.
```

BFS와 최단경로

- s 에서 L_i 에 속한 노드까지의 최단 경로의 길이는 i 이다. 여기서 경로에 속한 에지의 개수를 의미한다.
- BFS를 하면서 각 노드에 대해서 최단 경로의 길이를 구할 수 있다.



BFS와 최단경로

- **입력:** 방향 혹은 무방향 그래프 $G=(V, E)$, 그리고 출발노드 $s \in V$
- **출력:** 모든 노드 v 에 대해서
 - $d[v] = s$ 로부터 v 까지의 최단 경로의 길이(에지의 개수)
 - $\pi[v] = s$ 로부터 v 까지의 최단경로상에서 v 의 직전 노드(predecessor)

```
BFS(G, s)
```

```

Q←∅;
d[s]←0;           /* distance from s to s is 0 */
π[s]←null;        /* no predecessor of s */

Enqueue(Q, s);
while Q≠∅ do
    u ← Dequeue(Q)
    for each v adjacent to u do
        if v is unvisited then
            mark v as visited;
            d[v]←d[u]+1;      /* distance to v */
            π[v]←u;          /* u is the predecessor of v */
            Enqueue(Q, v);
    end.
end.
```

보통 모든 노드들에 대해서 $d[v]$ 를 -1로 초기화해두고, -1이면 unvisited, 아니면 visited로 판단한다.

$O(n+m)$ with adjacent list

시간복잡도

BFS(G, s)

```
Q $\leftarrow \emptyset$ ;  
for each node  $u$  do  
   $d[u] \leftarrow -1$ ;  
   $\pi[u] \leftarrow \text{null}$ ;  
end.
```

```
 $d[s] \leftarrow 0$ ;  $\pi[s] \leftarrow \text{null}$ ;
```

```
Enqueue(Q,  $s$ );  
while  $Q \neq \emptyset$  do
```

```
   $u \leftarrow \text{Dequeue}(Q)$   
  for each  $v$  adjacent to  $u$  do  
    if ( $d[v] == -1$ ) then
```

```
       $d[v] \leftarrow d[u] + 1$ ;
```

```
       $\pi[v] \leftarrow u$ ;
```

```
      Enqueue(Q,  $v$ );
```

```
  end.
```

```
end.
```

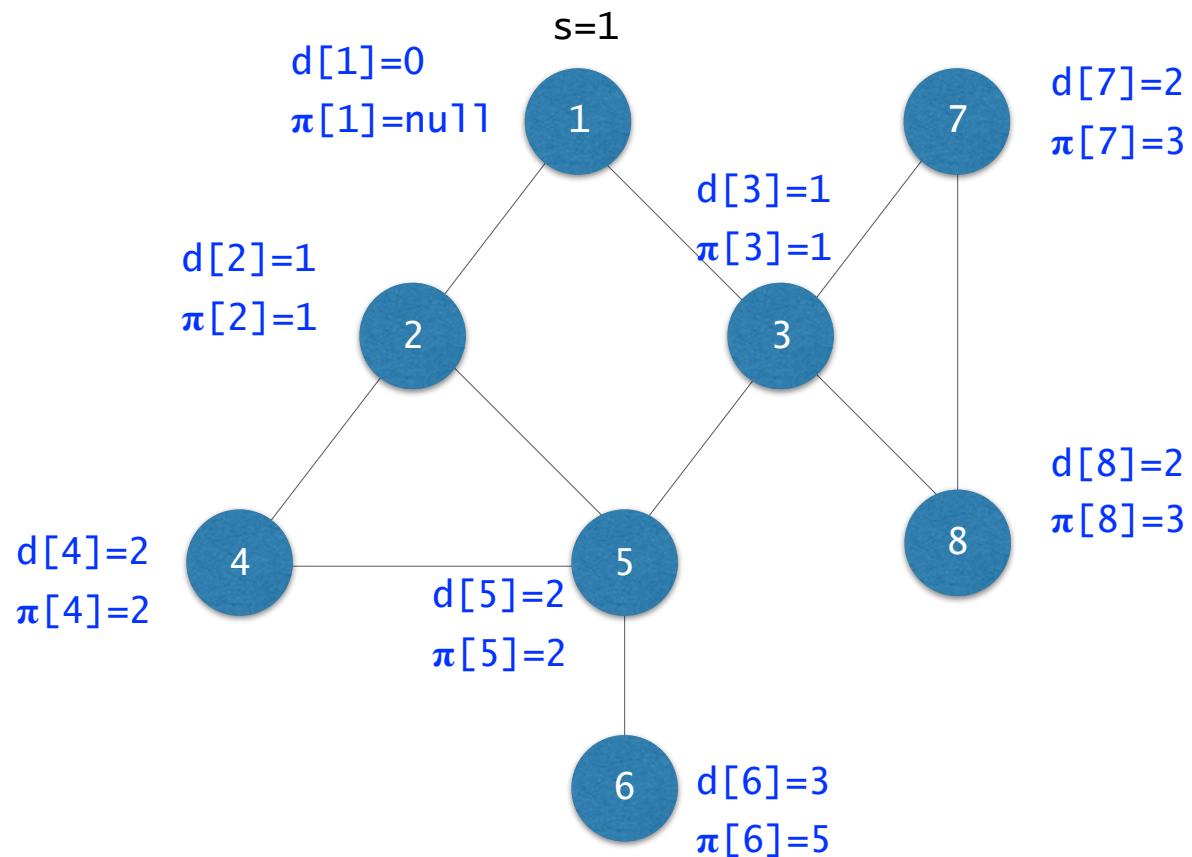
인접리스트로 구현할 경우 시간복잡도는

$$\sum_v \text{degree}(v) = 2m \text{ 이므로 } O(n+m)$$

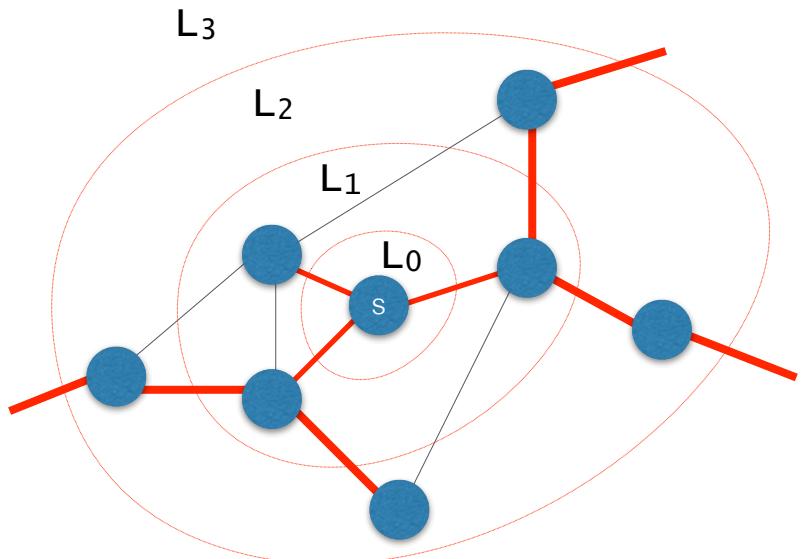
while문을 한 번 돌 때마다 큐에서 하나를 꺼내므로
while문은 최대 n 번 돈다.

인접리스트로 구현할 경우 for문은 각 노드 v 에 대해서 $\text{degree}(v)$ 번 돈다.

unchecked 노드만 queue에 들어갈 수 있으므로 어떤 노드도 큐에 두번 들어가지는 않는다.



- 각 노드 v 와 $\pi[v]$ 를 연결하는 에지들로 구성된 트리



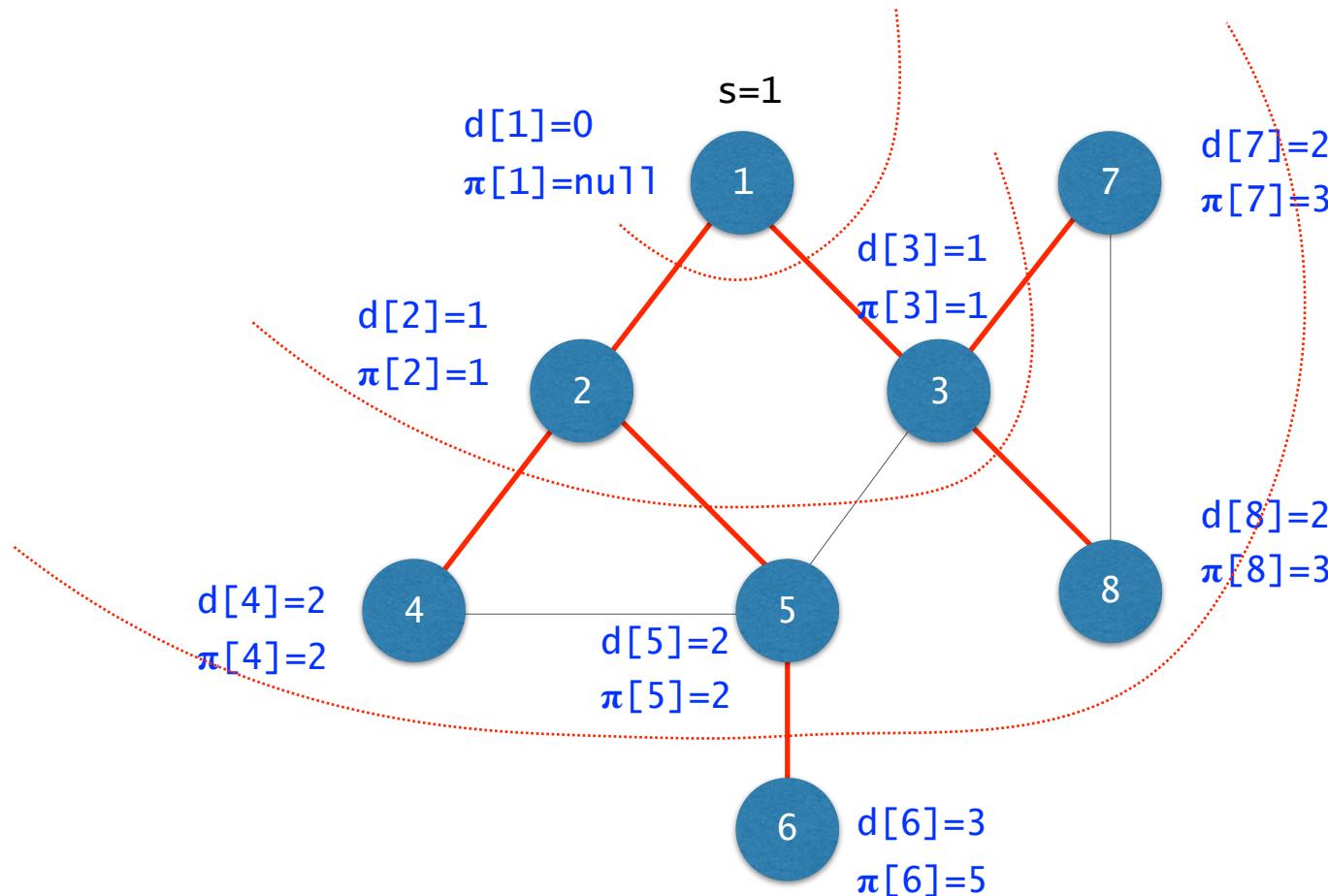
BFS 트리에서 s 에서 v 까지의 경로는 s 에서 v 까지 가는 최단경로

어떤 에지도 2개의 layer를 건너가지 않는다.

(동일 레이어의 노드를 연결하거나, 혹은 1 개의 layer를 건너간다.)

BFS Tree

- 각 노드 v 와 $\pi[v]$ 를 연결하는 에지들로 구성된 트리



너비우선순회: 최단 경로 출력하기

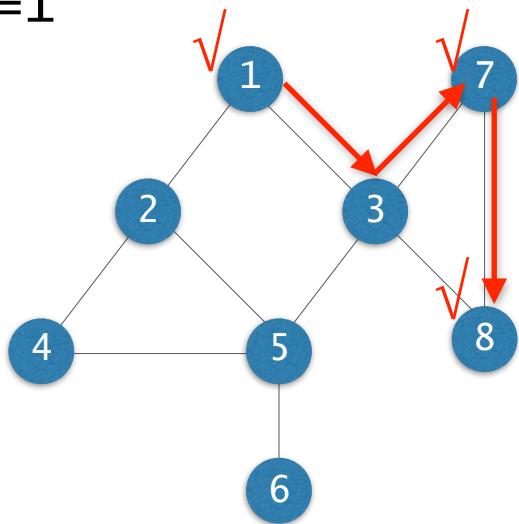
```
PRINT-PATH(G,s,v)      /* 출발점 s에서 노드 v까지의 경로 출력하기 */
if v=s then
    print s;
else if  $\pi[v]=\text{null}$  then
    print "no path from s to v exists";
else
    PRINT-PATH(G,s, $\pi[v]$ );
    print v;
end.
```

- 그래프가 `disconnected`이거나 혹은 방향 그래프라면 BFS에 의해서 모든 노드가 방문되지 않을 수도 있음
- BFS를 반복하여 모든 노드 방문

```
BFS-ALL( G )
{
    while there exists unvisited node v
        BFS(G, v);
}
```

깊이우선순회 (DFS)

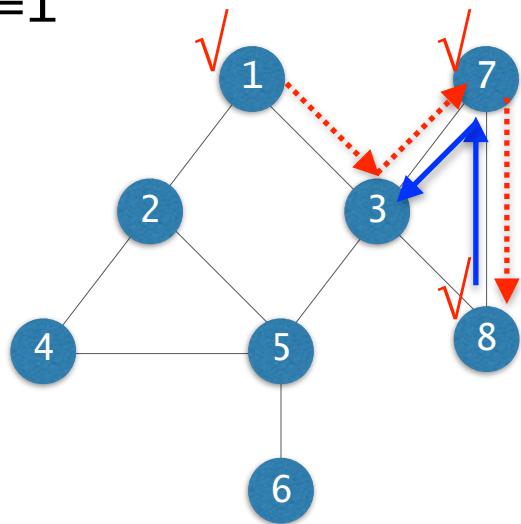
$s=1$



1. 출발점 s 에서 시작한다.
2. 현재 노드를 `visited`로 mark하고 인접한 노드들 중 `unvisited` 노드가 존재하면 그 노드로 간다.
3. 2번을 계속 반복한다.

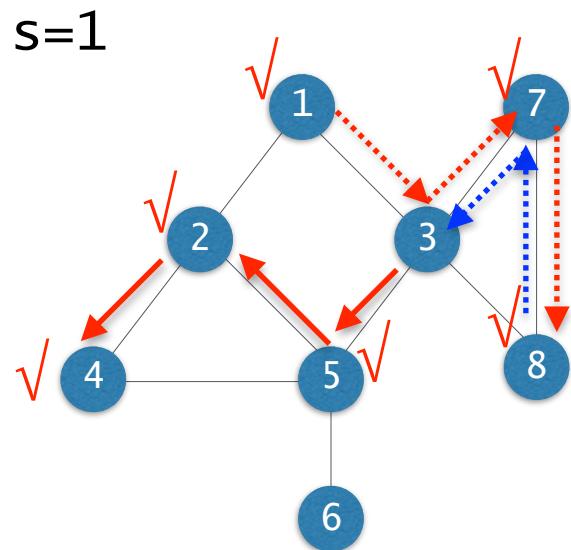
깊이우선순회 (DFS)

$s=1$



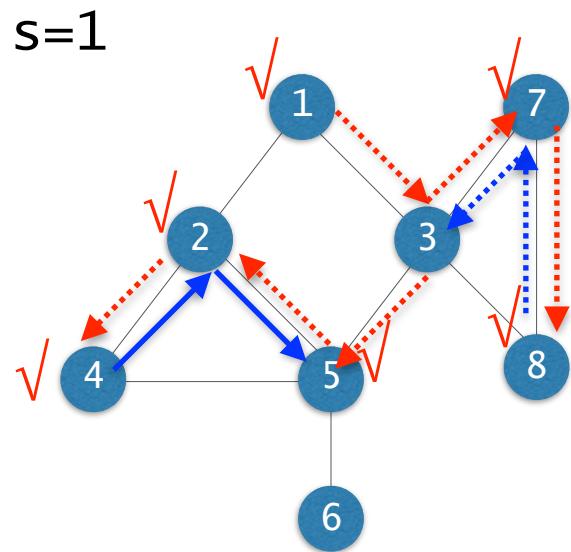
1. 출발점 s 에서 시작한다.
2. 현재 노드를 `visited`로 mark하고 인접한 노드들 중 `unvisited` 노드가 존재하면 그 노드로 간다.
3. 2번을 계속 반복한다.
4. `unvisited`인 이웃 노드가 존재하지 않는 동안 계속해서 직전 노드로 되돌아간다.

깊이우선순회 (DFS)



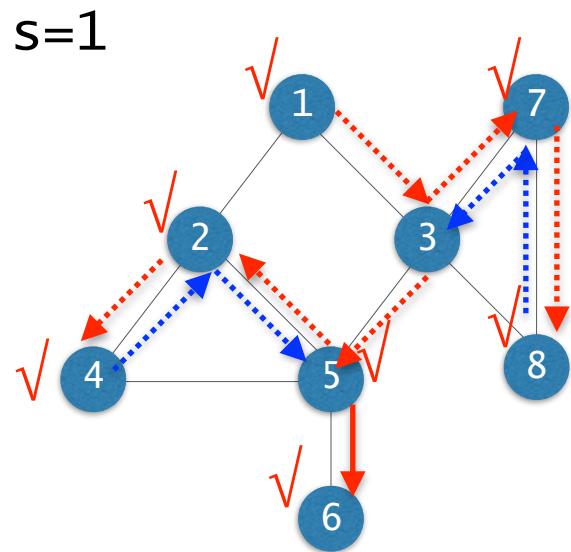
1. 출발점 s 에서 시작한다.
2. 현재 노드를 `visited`로 mark하고 인접한 노드들 중 `unvisited` 노드가 존재하면 그 노드로 간다.
3. 2번을 계속 반복한다.
4. `unvisited`인 이웃 노드가 존재하지 않는 동안 계속해서 직전 노드로 되돌아간다.
5. 다시 2번을 반복한다.

깊이우선순회 (DFS)



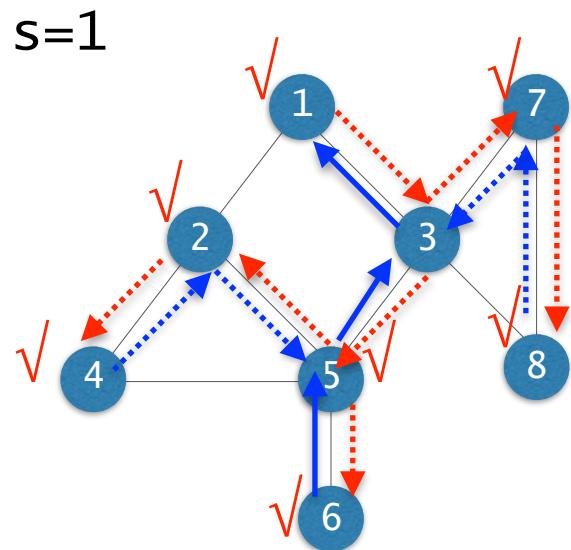
1. 출발점 s 에서 시작한다.
2. 현재 노드를 `visited`로 mark하고 인접한 노드들 중 `unvisited` 노드가 존재하면 그 노드로 간다.
3. 2번을 계속 반복한다.
4. `unvisited`인 이웃 노드가 존재하지 않는 동안 계속해서 직전 노드로 되돌아간다.
5. 다시 2번을 반복한다.

깊이우선순회 (DFS)



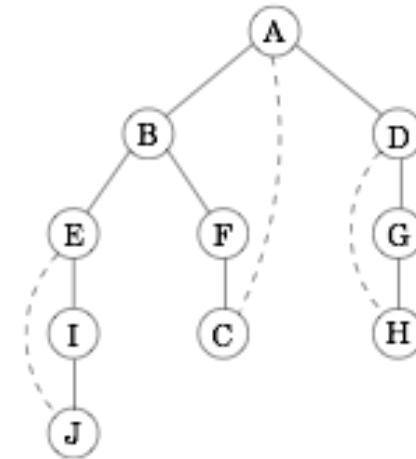
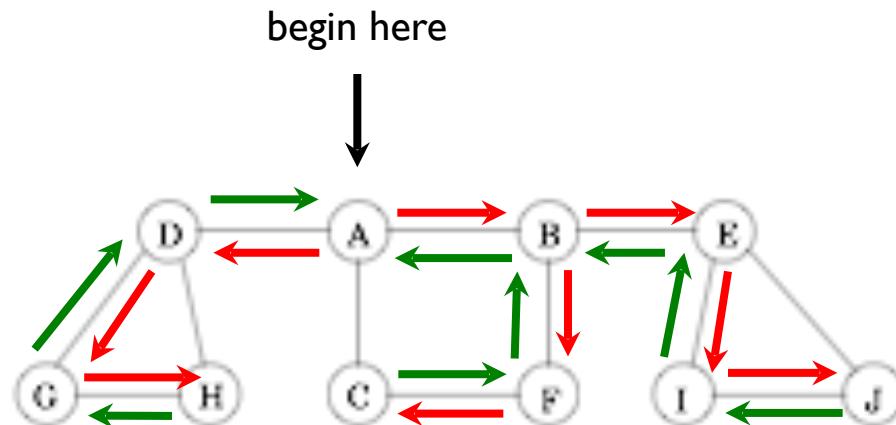
1. 출발점 s 에서 시작한다.
2. 현재 노드를 `visited`로 mark하고 인접한 노드들 중 `unvisited` 노드가 존재하면 그 노드로 간다.
3. 2번을 계속 반복한다.
4. `unvisited`인 이웃 노드가 존재하지 않는 동안 계속해서 직전 노드로 되돌아간다.
5. 다시 2번을 반복한다.

깊이우선순회 (DFS)



1. 출발점 s 에서 시작한다.
2. 현재 노드를 `visited`로 mark하고 인접한 노드들 중 `unvisited` 노드가 존재하면 그 노드로 간다.
3. 2번을 계속 반복한다.
4. `unvisited`인 이웃 노드가 존재하지 않는 동안 계속해서 직전 노드로 되돌아간다.
5. 다시 2번을 반복한다.
6. 시작노드 s 로 돌아오고 더 이상 곳이 없으면 종료한다.

깊이우선순회 (DFS)



DFS 깊이우선탐색

```
DFS(G, v)
    visited[v] ← YES;
    for each node u adjacent to v do
        if visited[u] = NO then
            DFS(G, u);
    end.
end.
```

DFS 깊이우선탐색

- 그래프가 `disconnected`이거나 혹은 방향 그래프라면 DFS에 의해서 모든 노드가 방문되지 않을 수도 있음
- DFS를 반복하여 모든 노드 방문

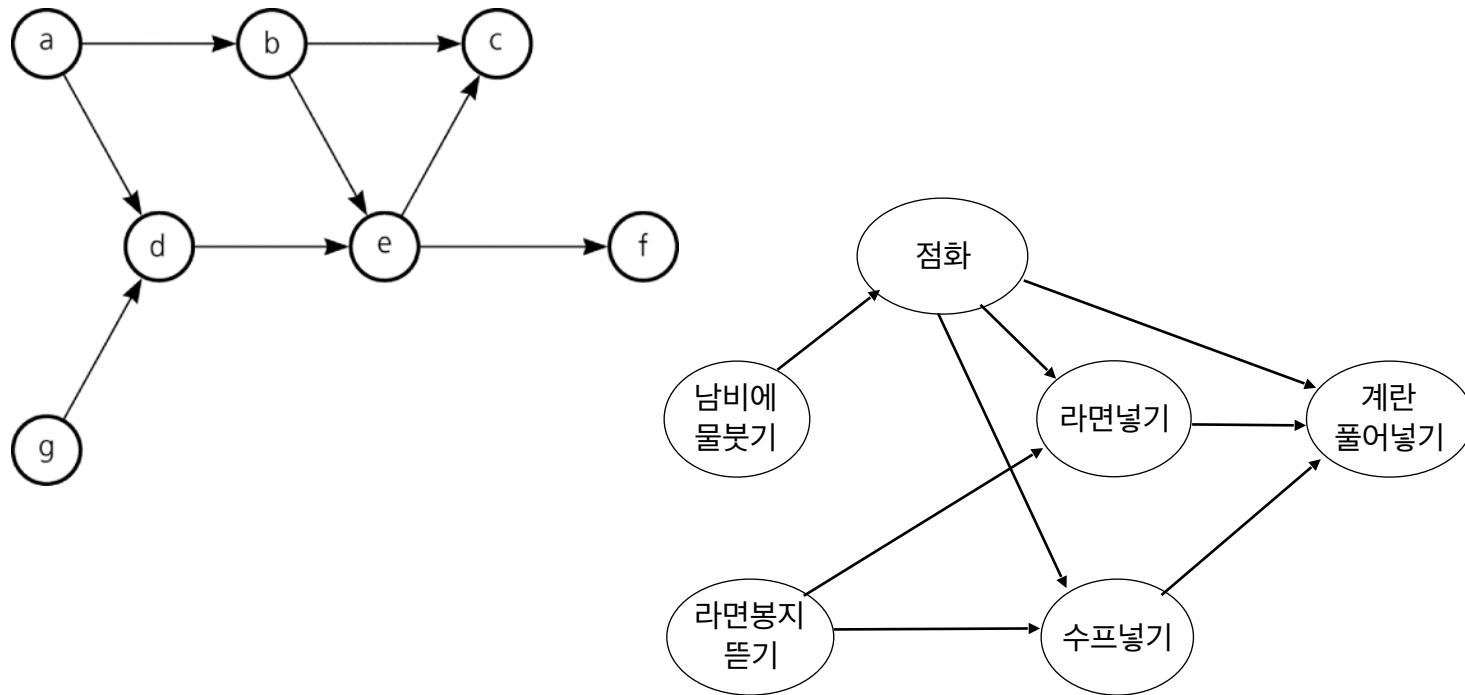
```
DFS-ALL(G)
{
    for each v ∈ V
        visited[v] ← NO;
    for each v ∈ V
        if (visited[v] = NO)  then
            DFS(G, v);
}
```

시간복잡도: $O(n+m)$

DAG
(Directed Acyclic Graph)

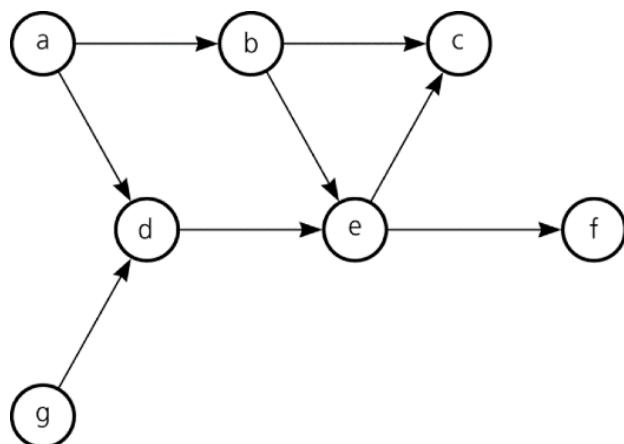
Directed Acyclic Graph

- DAG는 방향 사이클(directed cycle)이 없는 방향 그래프.
- 예: 작업들의 우선순위

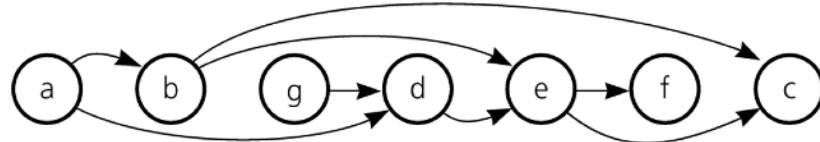
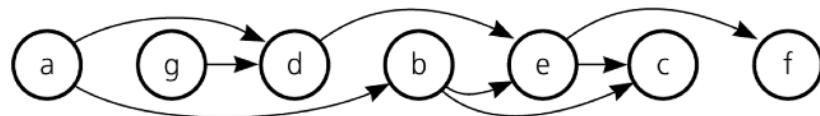


위상정렬(topological ordering)

- DAG에서 노드들의 순서화 v_1, v_2, \dots, v_n , 단, 모든 에지 (v_i, v_j) 에 대해서 $i < j$ 가 되도록.



이 그래프에 대한
위상정렬의 예 2개



```
topologicalSort1(G)
```

```
{
```

```
    for i  $\leftarrow$  1 to n {
```

진입간선이 없는 임의의 정점 *u*를 선택한다;

```
        A[i]  $\leftarrow$  u;
```

정점 *u*와 *u*의 진출간선을 모두 제거한다;

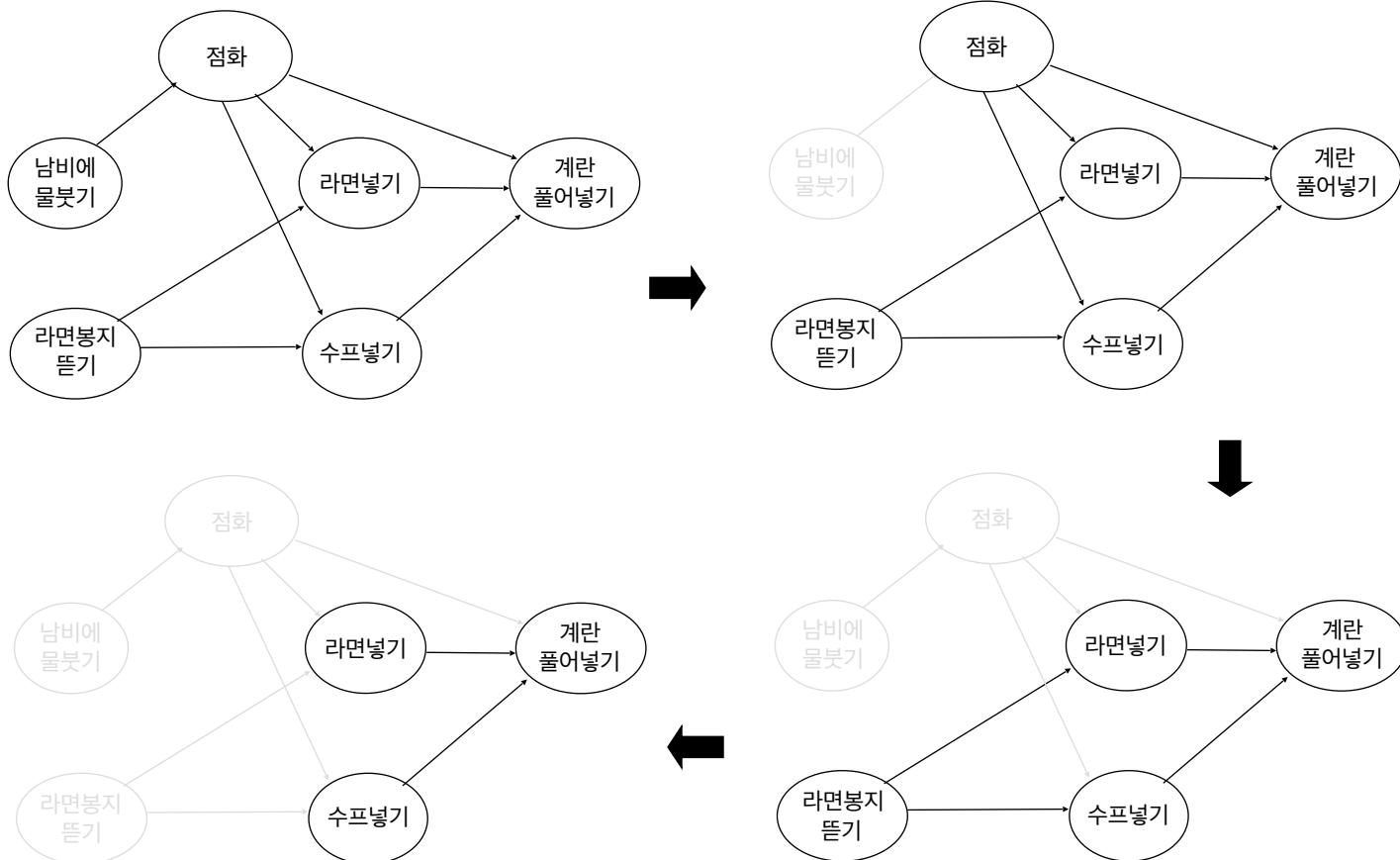
```
}
```

▷ 배열 *A*[1...*n*]에는 정점들을 위상정렬되어 있다

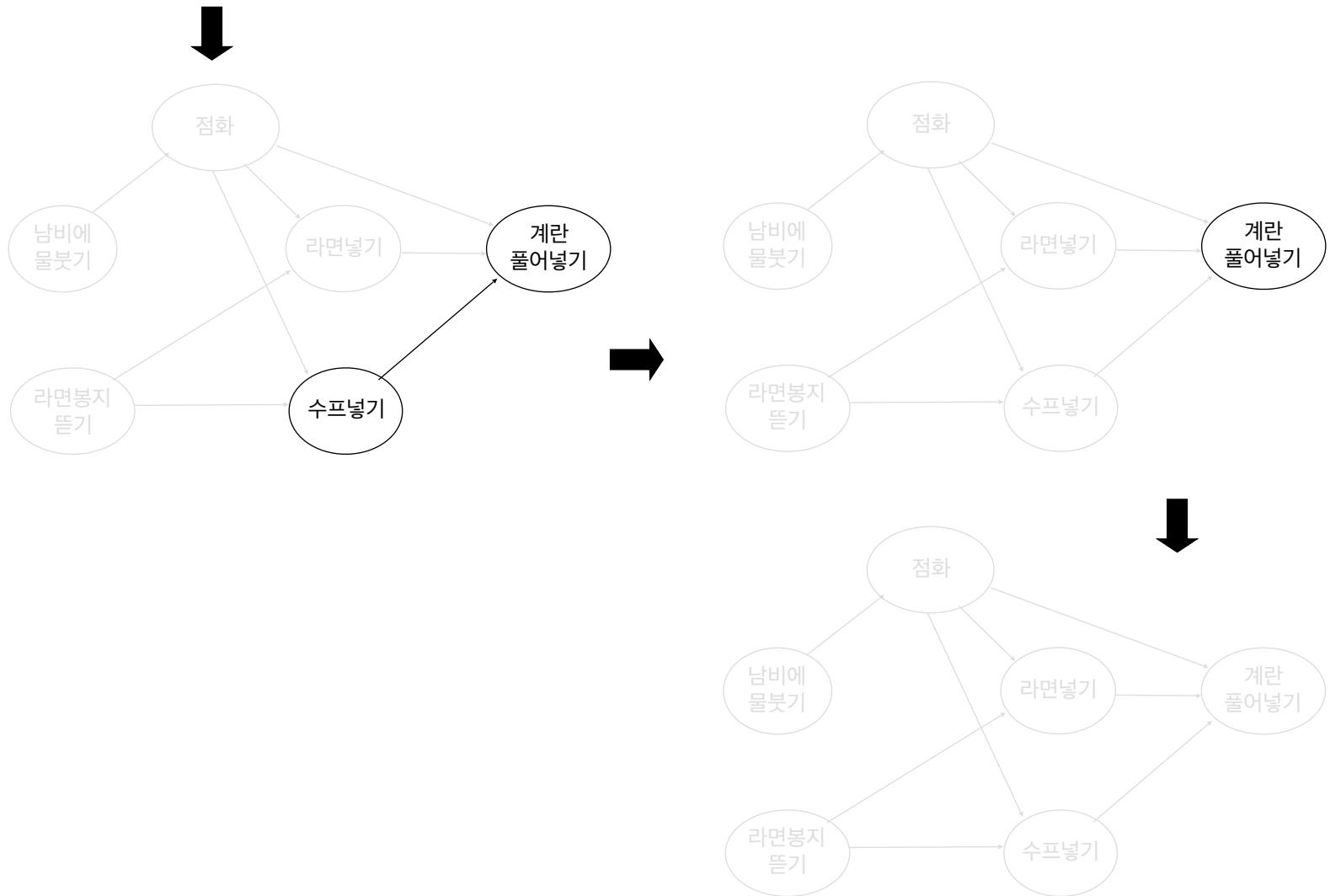
```
}
```

수행시간: $\Theta(n+m)$

위상정렬 알고리즘 1의 예



위상정렬 알고리즘 1의 예 (계속)



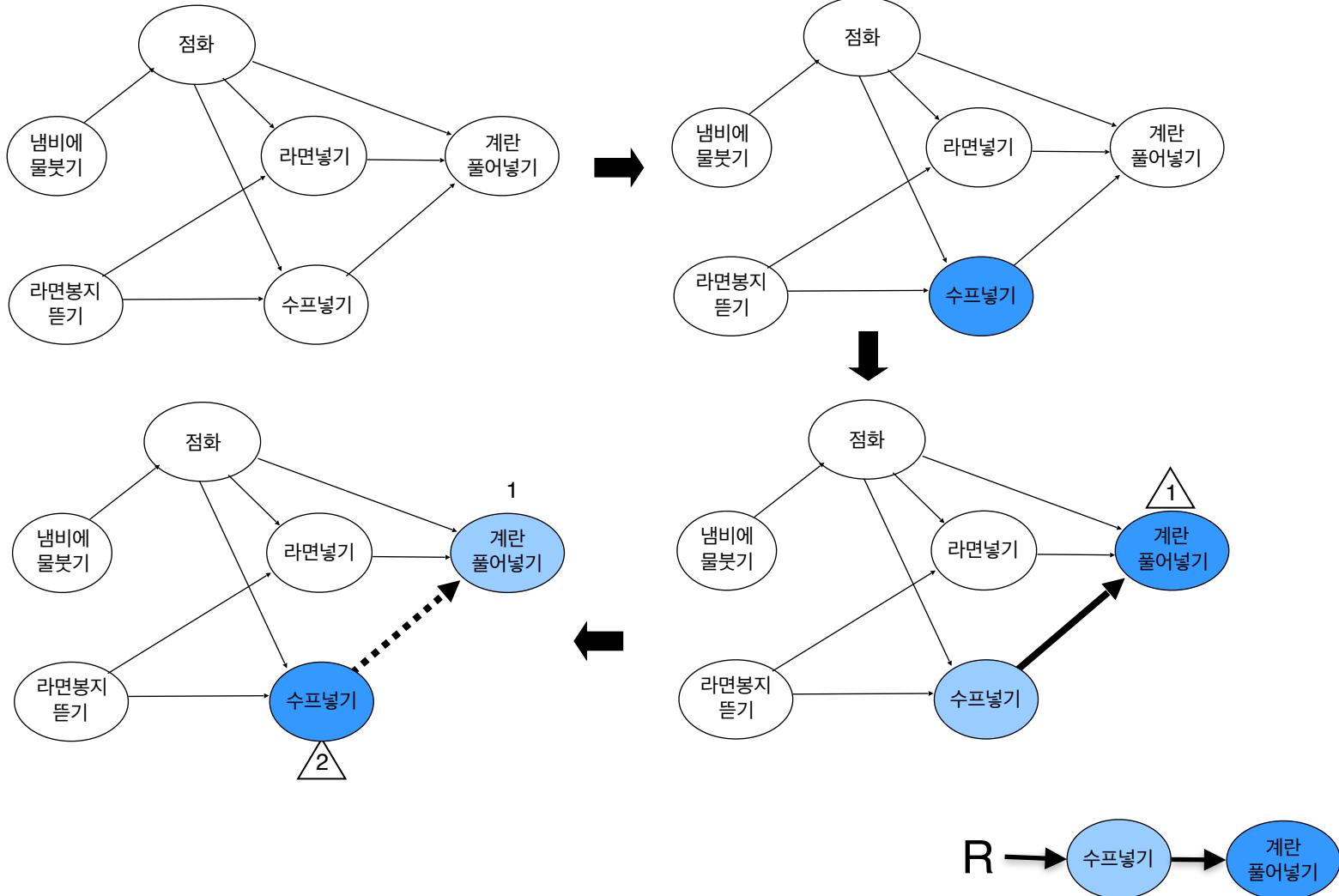
```
topologicalSort2(G)
{
    for each v∈V
        visited[v] ← NO;
    make an empty linked list R;
    for each v∈V    ▷ 정점의 순서는 상관없음
        if (visited[v] = NO) then
            DFS-TS(v, R);
}
```

```
DFS-TS(v, R)
{
    visited[v] ← YES;
    for each x adjacent to v do
        if (visited[x] = NO) then
            DFS-TS(x, R) ;
    add v at the front of the linked list R;
}
```

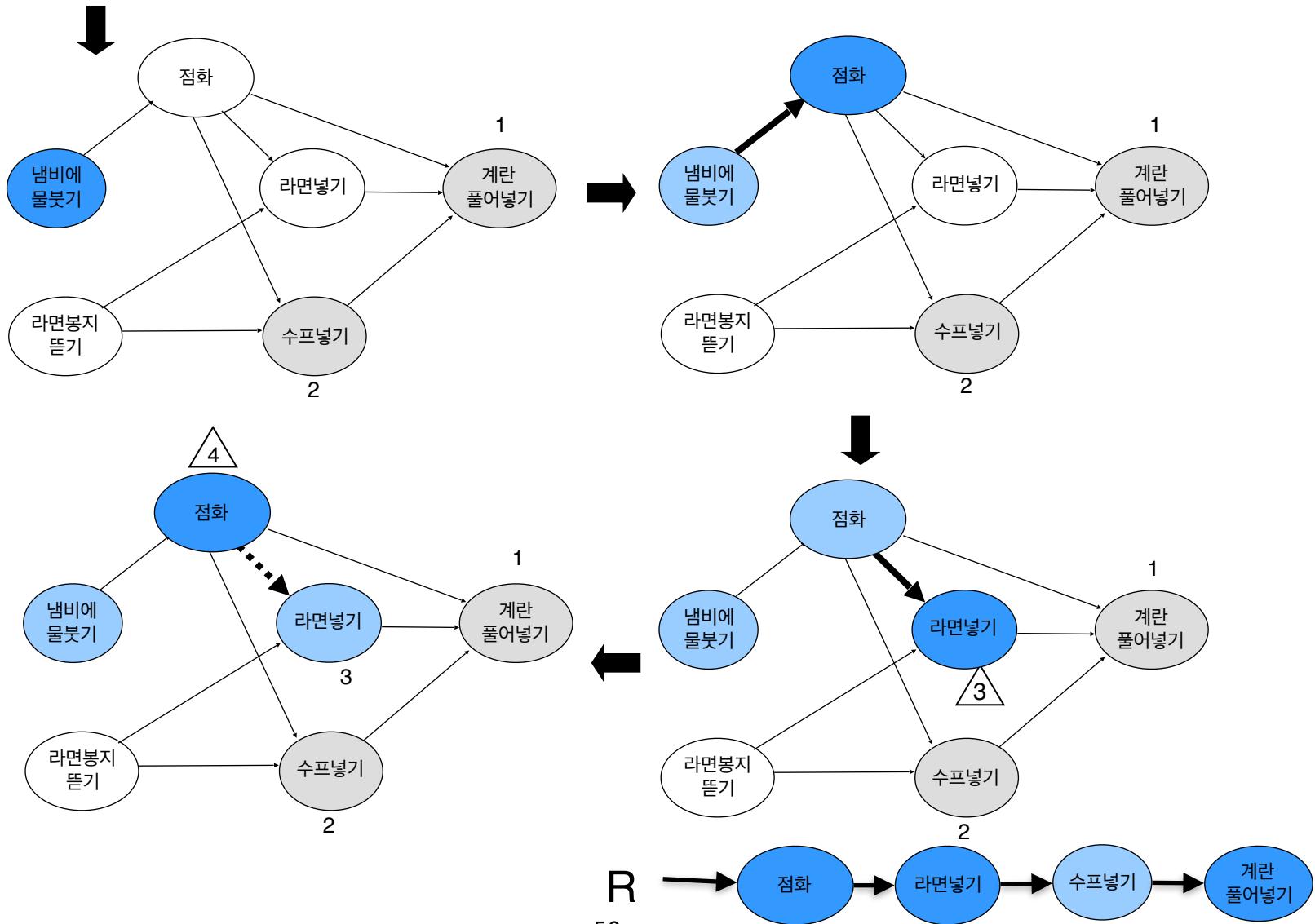
알고리즘이 끝나면 연결 리스트 R에는 정점들이 위상정렬된 순서로 매달려 있다.

수행시간: $\Theta(n+m)$

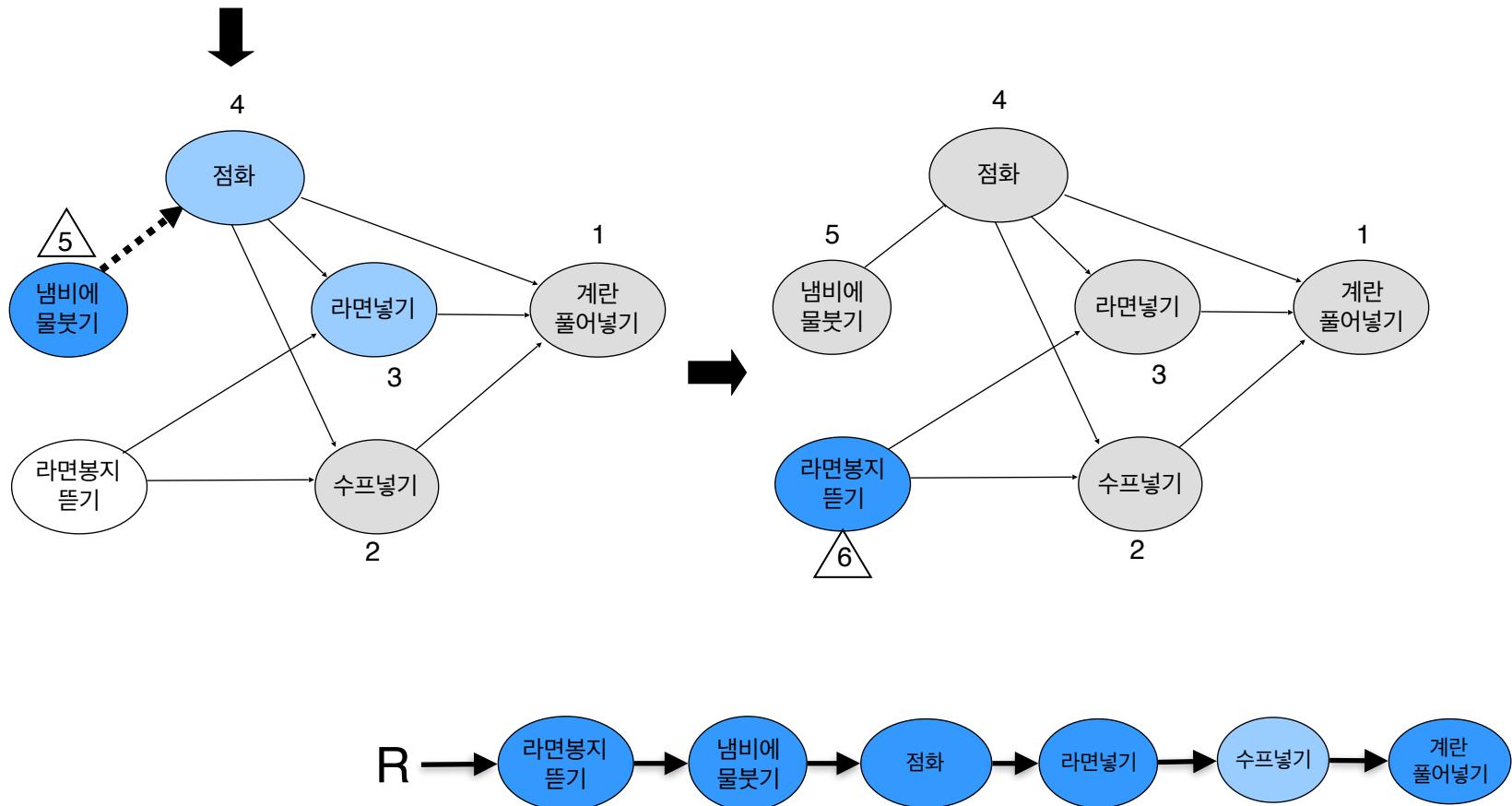
위상정렬 알고리즘 2의 작동 예



위상정렬 알고리즘 2의 작동 예 (계속)



위상정렬 알고리즘 2의 작동 예 (계속)

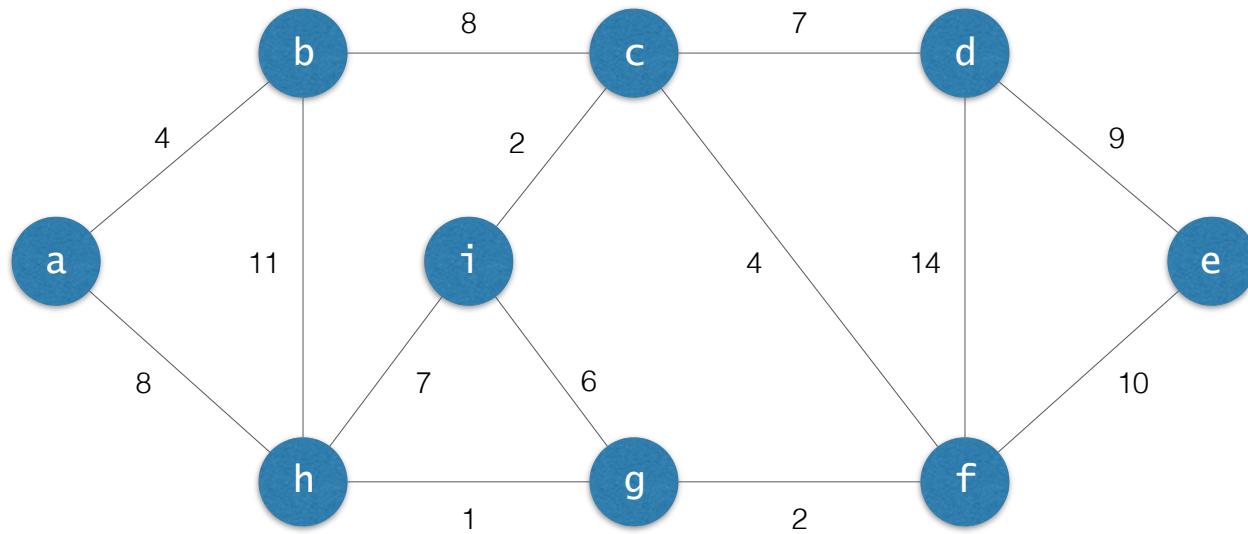


최소신장트리

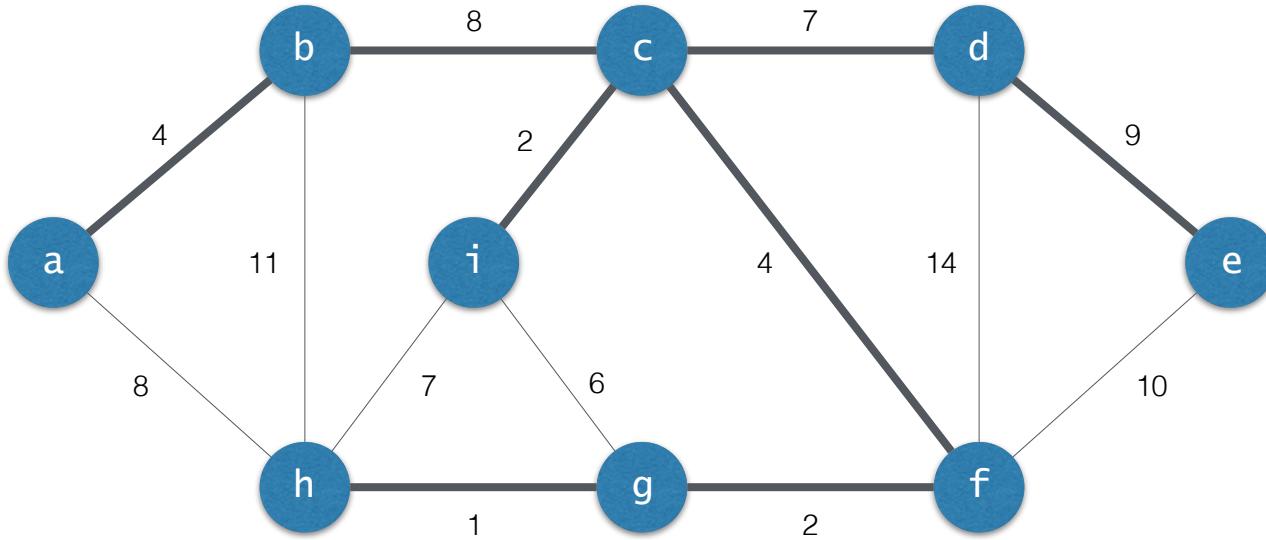
Minimum Spanning Tree

최소비용 신장 트리(MST)

- 입력: n개의 도시, 도시와 도시를 연결하는 비용
- 문제: 최소의 비용으로 모든 도시들이 서로 연결되게 한다.



최소비용 신장 트리(MST)

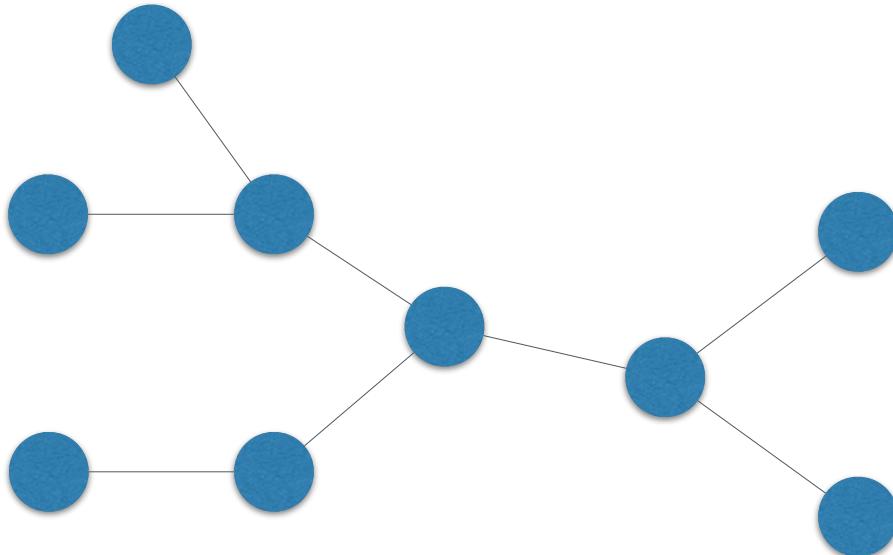


해가 유일하지는 않음
예: (b, c) 대신 (a, h)

- **무방향** 가중치 그래프 $G=(V, E)$
- 각 에지 $(u, v) \in E$ 에 대해서 가중치 $w(u, v)$
- 문제: 다음과 같은 조건을 만족하는 에지들의 부분집합 $T \subseteq E$ 를 찾아라.
 1. T 에 속한 에지들에 의해 그래프의 모든 정점들이 서로 연결된다.
 2. 가중치의 합 $\sum_{(u,v) \in T} w(u, v)$ 이 최소가 된다.

왜 트리라고 부르나?

- 싸이클이 없는 연결된(**connected**) 무방향 그래프를 트리라고 부른다.
- MST 문제의 답은 항상 트리가 됨. 왜?



노드가 n 개인 트리는 항상 $n-1$ 개의 에지를 가짐

Generic MST 알고리즘

- 어떤 MST의 부분집합 A 에 대해서 $A \cup \{(u, v)\}$ 도 역시 어떤 MST의 부분집합이 될 경우 에지 (u, v) 는 A 에 대해서 안전하다(safe)고 한다.
- Generic MST 알고리즘:
 1. 처음에는 $A = \emptyset$ 이다.
 2. 집합 A 에 대해서 안전한 에지를 하나 찾은 후 이것을 A 에 더한다.
 3. 에지의 개수가 $n-1$ 개가 될 때까지 2번을 반복한다.

GENERIC-MST(G, w)

- 1 $A \leftarrow \emptyset$
- 2 **while** A does not form a spanning tree
- 3 **do** find an edge (u, v) that is safe for A
- 4 $A \leftarrow A \cup \{(u, v)\}$
- 5 **return** A

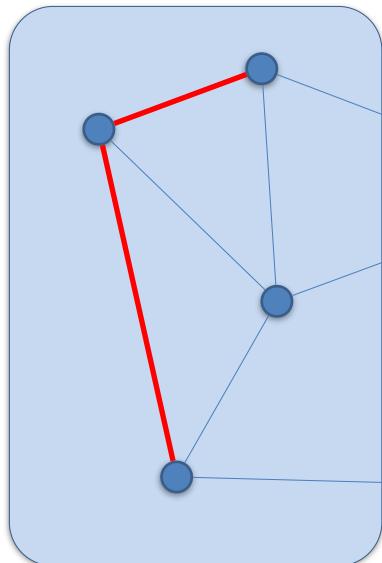
- 그리프의 정점들을 두 개의 집합 S 와 $V-S$ 로 분할한 것을 컷(cut) $(S, V-S)$ 라고 부른다.
- 에지 (u, v) 에 대해서 $u \in S$ 이고 $v \in V-S$ 일 때 에지 (u, v) 는 컷 $(S, V-S)$ 를 cross한다고 말한다.
- 에지들의 부분집합 A 에 속한 어떤 에지고 컷 $(S, V-S)$ 를 cross하지 않을 때 컷 $(S, V-S)$ 는 A 를 존중한다(respect)고 말한다.

안전한 에지 찾기

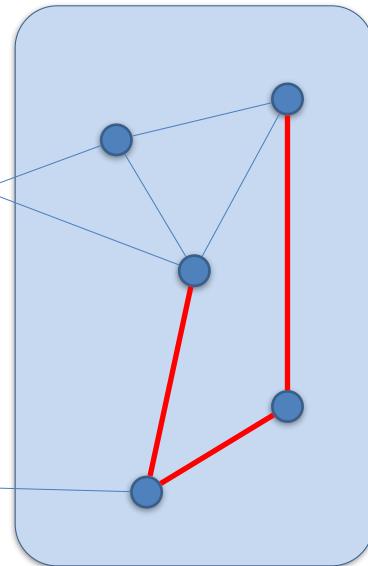
노드들을 두 그룹으로 분할한 것을 cut이라고 부름

에지 집합 A에 속한 어떤 에지도 cut을 cross하지 않을 때 이 컷은 집합 A를 존중한다고 말함.
(예: 빨간 에지들을 A)

$$S \subseteq V$$

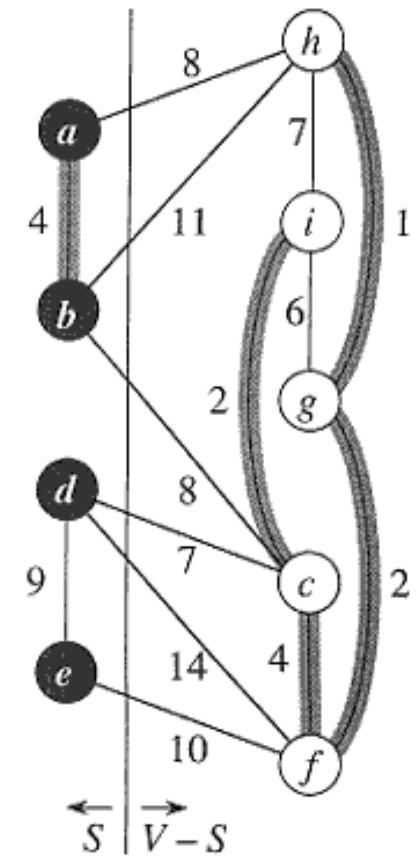
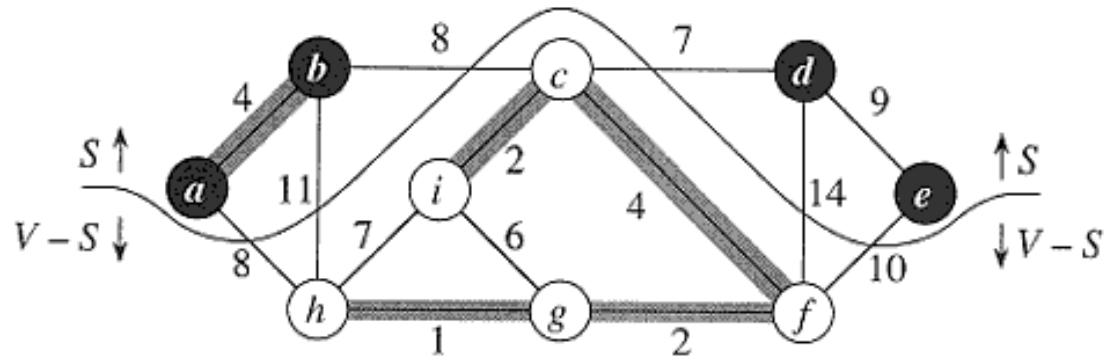


$$V - S$$

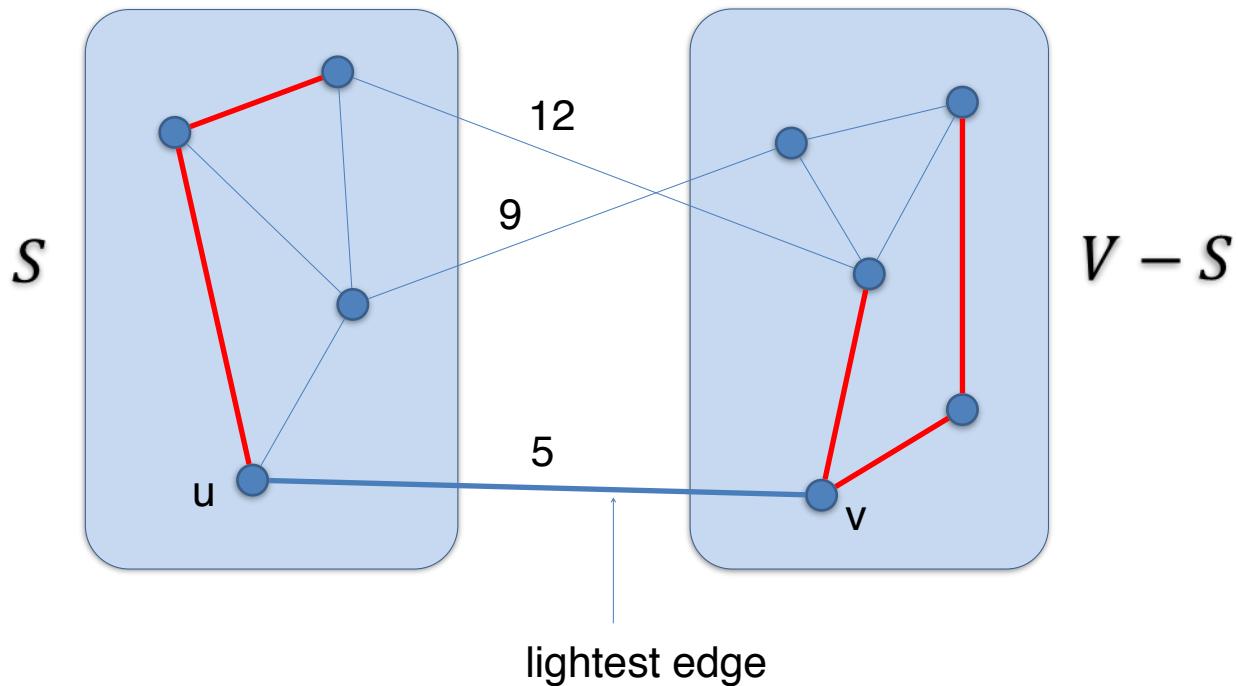


cut을 cross하는 에지

안전한 에지 찾기



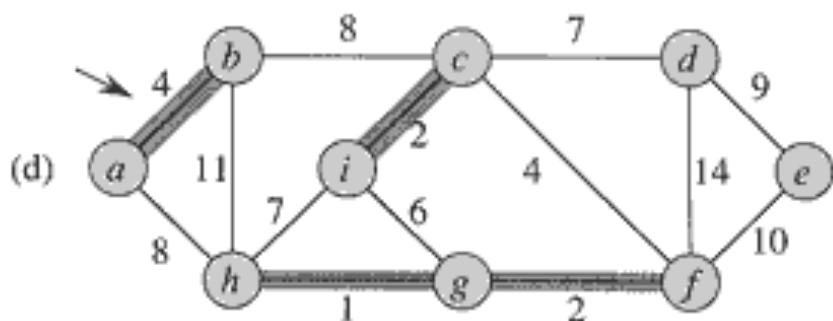
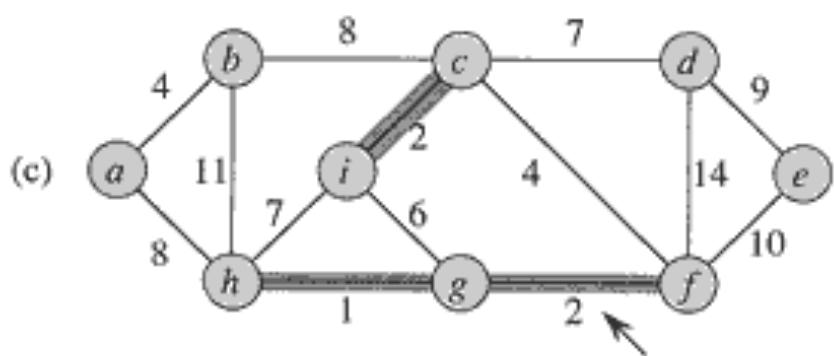
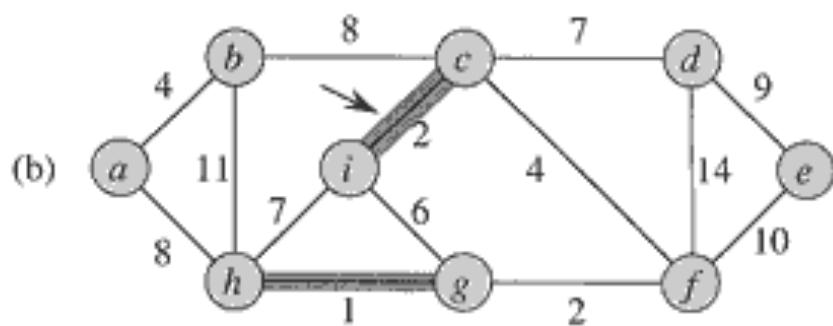
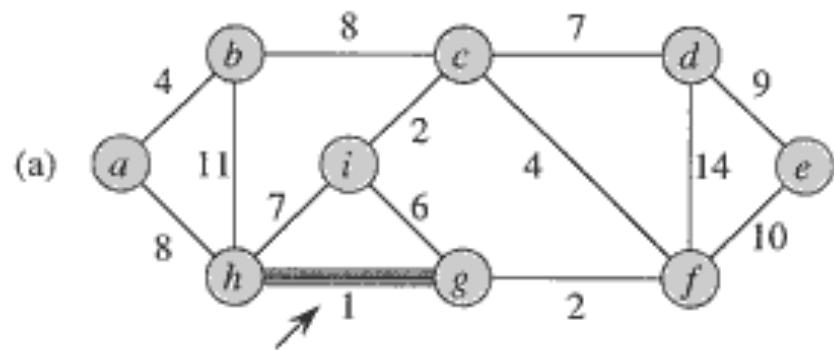
- A가 어떤 MST의 부분집합이고, $(S, V-S)$ 는 A를 존중하는 컷이라고 하자. 이 컷을 cross하는 에지들 중 가장 가중치가 작은 에지 (u, v) 는 A에 대해서 안전하다.



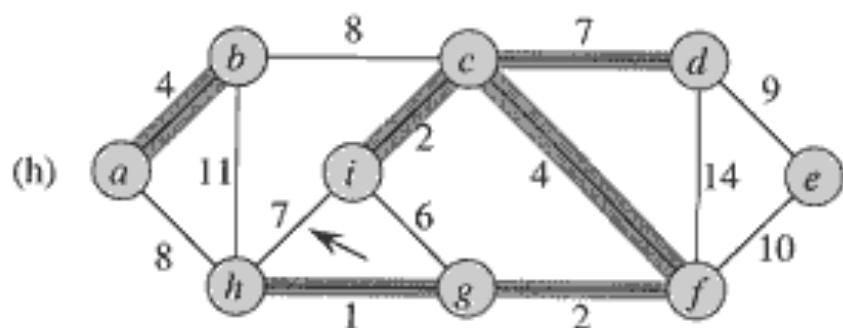
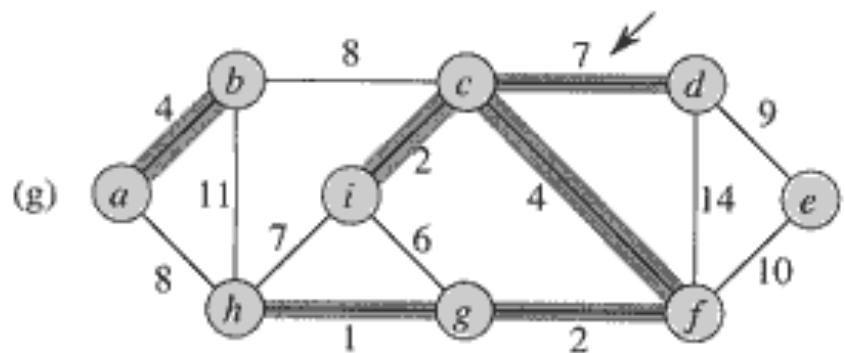
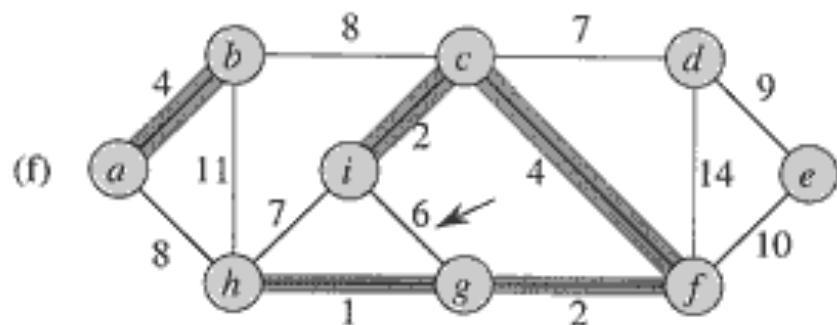
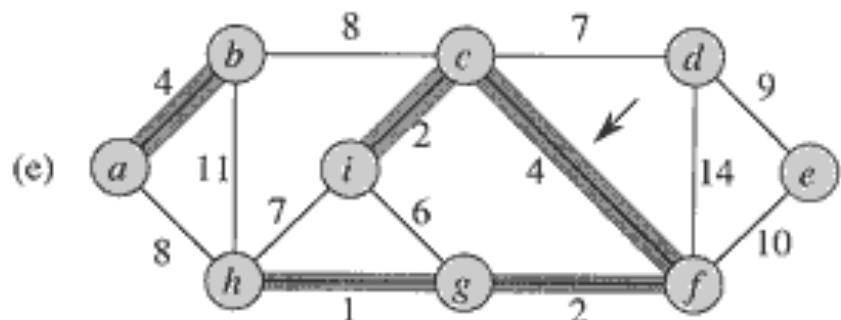
Kruskal의 알고리즘

- 에지들을 가중치의 오름차순으로 정렬한다.
- 에지들을 그 순서대로 하나씩 선택해간다. 단, 이미 선택된 에지들과 사이클 (cycle)을 형성하면 선택하지 않는다.
- $n-1$ 개의 에지가 선택되면 종료한다.

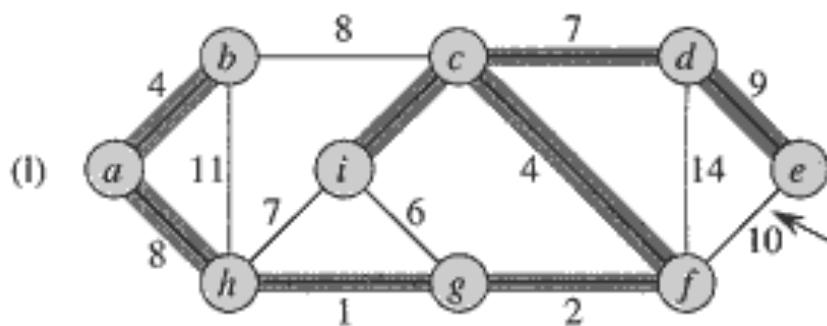
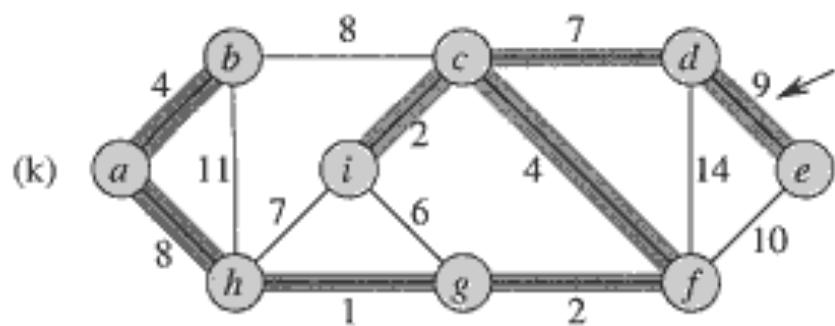
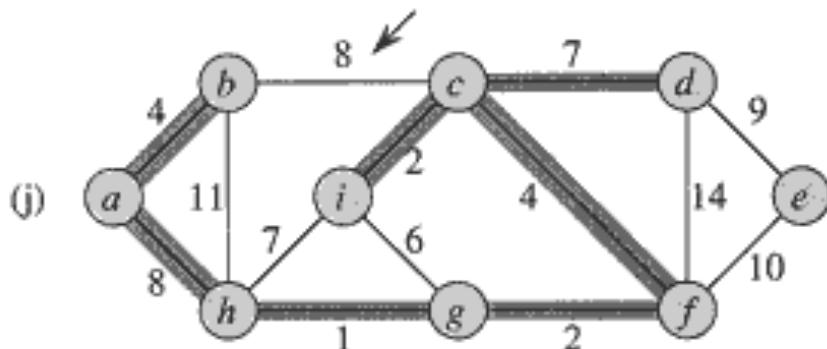
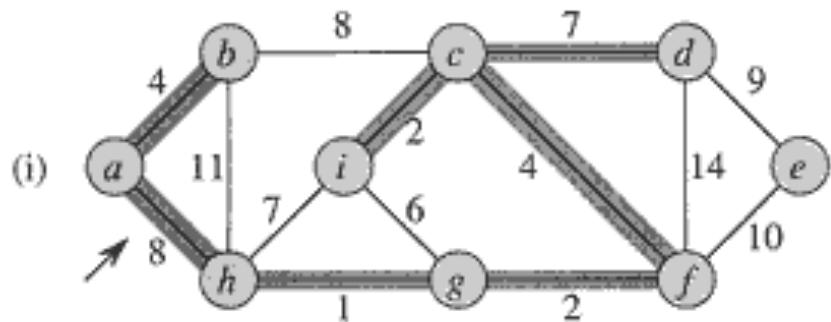
Kruskal의 알고리즘



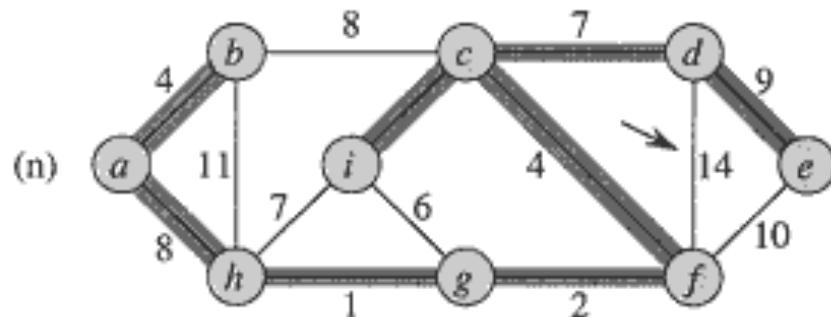
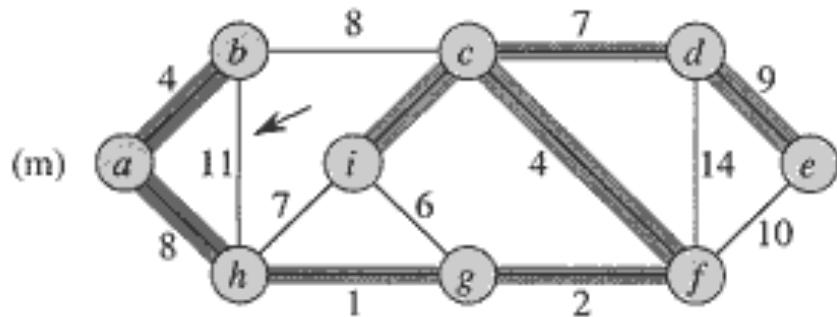
Kruska1의 알고리즘 (계속)



Kruska1의 알고리즘 (계속)

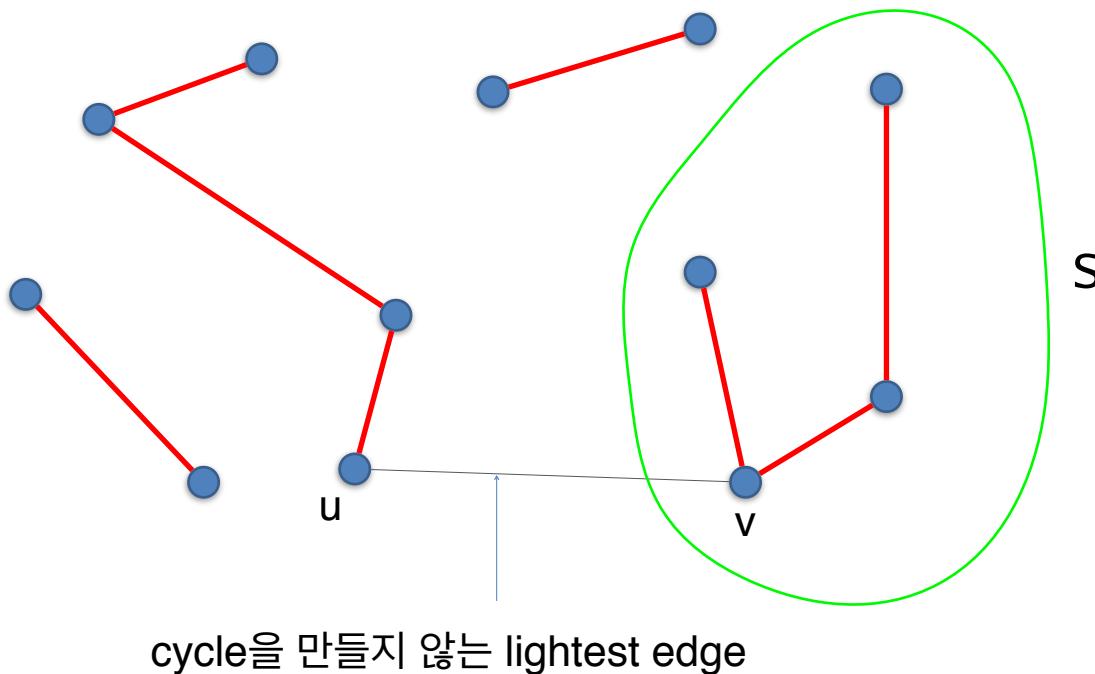


Kruska1의 알고리즘 (계속)

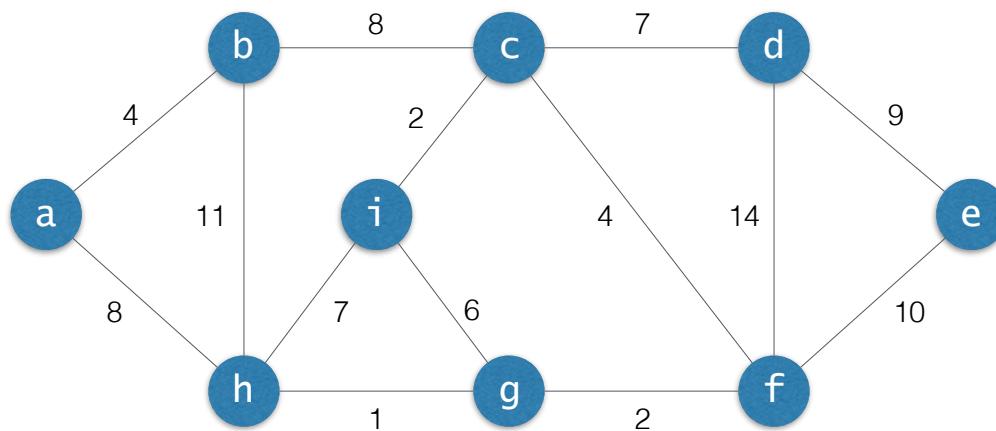


왜 MST가 찾아지는가?

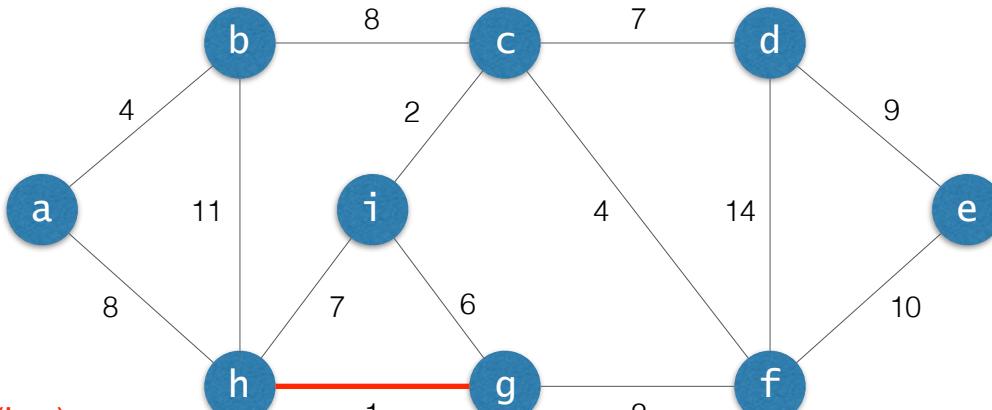
- Kruskal의 알고리즘의 임의의 한 단계를 생각해보자.
- A 를 현재까지 알고리즘이 선택한 에지의 집합이라고 하고, A 를 포함하는 MST가 존재한다고 가정하자.



- 초기 상태: 선택된 에지 없음
- 각각의 연결요소를 하나의 집합으로 표현



{a} {b} {c} {d} {e} {f} {g} {h} {i}



- 가중치가 최소인 에지 (h,g)를 고려한다.

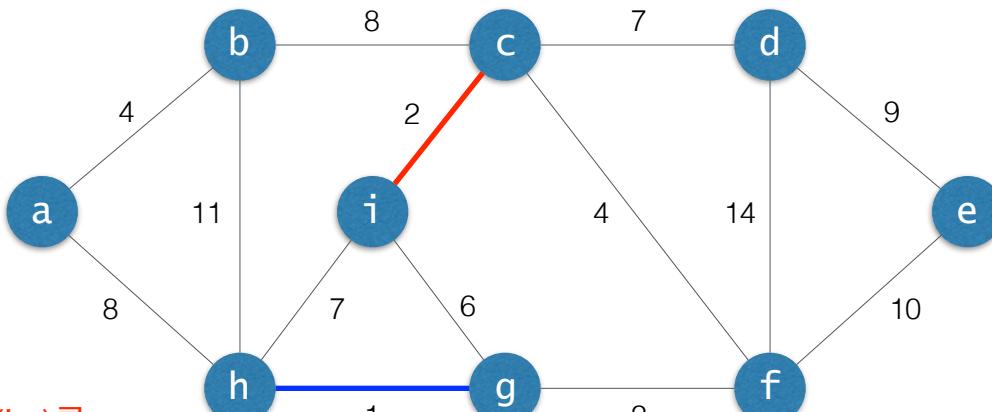
$\{a\} \ \{b\} \ \{c\} \ \{d\} \ \{e\} \ \{f\} \ \{g\} \ \underline{\{h\}} \ \underline{\{i\}}$

- 에지 (g,h)를 선택하고, g 와 h 가 속한 집합을 합집합하여 하나의 집합으로 만듬

- g 와 h 가 서로 다른 집합에 속함



$\{a\} \ \{b\} \ \{c\} \ \{d\} \ \{e\} \ \{f\} \ \{g,h\} \ \{i\}$



1. 가중치가 최소인 에지 (i,c) 를 고려한다.

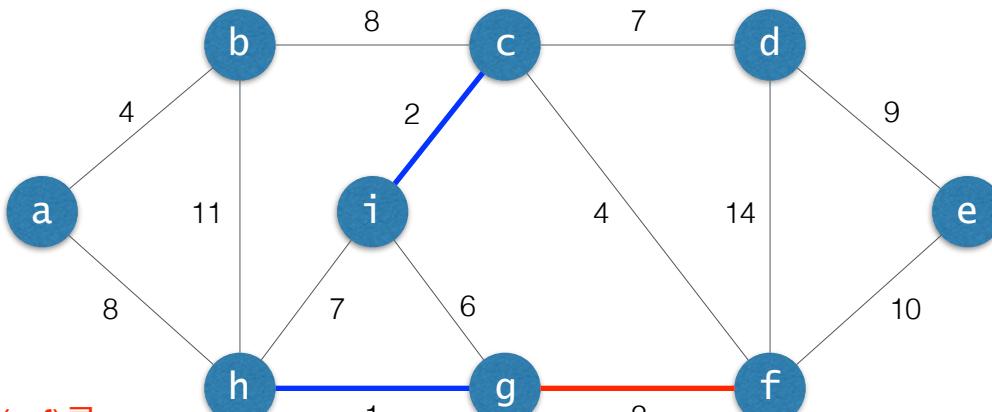
$\{a\}$ $\{b\}$ $\underline{\{c\}}$ $\{d\}$ $\{e\}$ $\{f\}$ $\{g, h\}$ $\underline{\{i\}}$

3. 에지 (i,c) 를 선택하고, i 와 c 가 속한 집합을 합집합하여 하나의 집합으로 만듬



2. i 와 c 가 서로 다른 집합에 속함

$\{a\}$ $\{b\}$ $\{c, i\}$ $\{d\}$ $\{e\}$ $\{f\}$ $\{g, h\}$



1. 가중치가 최소인 에지 (g,f)를 고려한다.

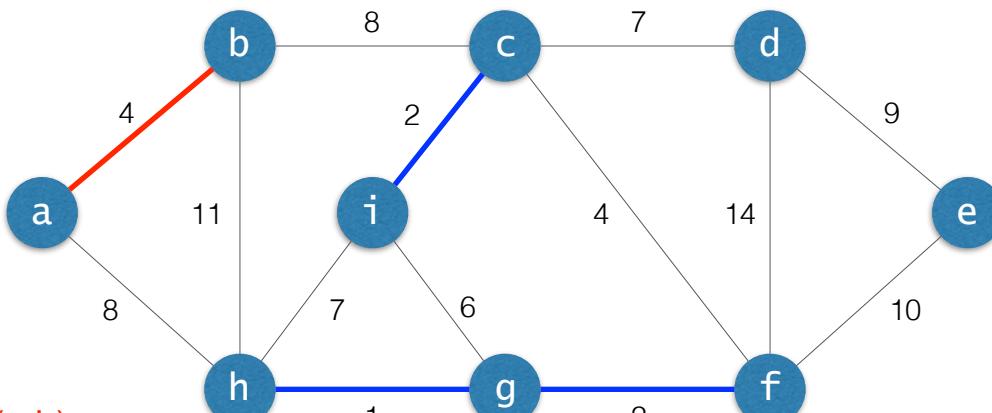
$\{a\} \ \{b\} \ \{c, i\} \ \{d\} \ \{e\} \ \underline{\{f\}} \ \underline{\{g, h\}}$

3. 에지 (g,f)를 선택하고, g 와 f 가 속한 집합을 합집합하여 하나의 집합으로 만듬

2. g 와 f 가 서로 다른 집합에 속함



$\{a\} \ \{b\} \ \{c, i\} \ \{d\} \ \{e\} \ \{f, g, h\}$



1. 가중치가 최소인 에지 (a,b)
를 고려한다.

{a} {b} {c, i} {d} {e} {f, g, h}

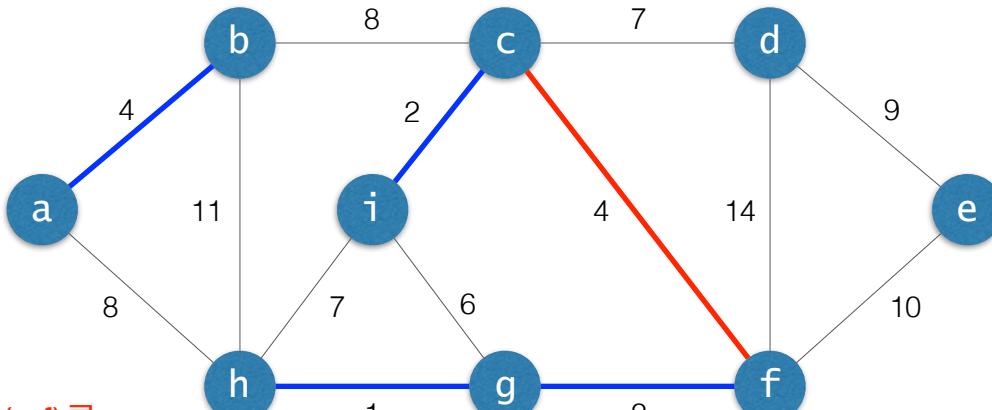
3. 에지 (a,b)를 선택하고, a와 b가 속한
집합을 합집합하여 하나의 집합으로 만
듬

2. a와 b가 서로 다른 집합에 속함



{a, b} {c, i} {d} {e} {f, g, h}

사이클 검사



1. 가중치가 최소인 에지 (c,f)를 고려한다.

$\{a, b\}$ $\underline{\{c, i\}}$ $\{d\}$ $\{e\}$ $\underline{\{f, g, h\}}$

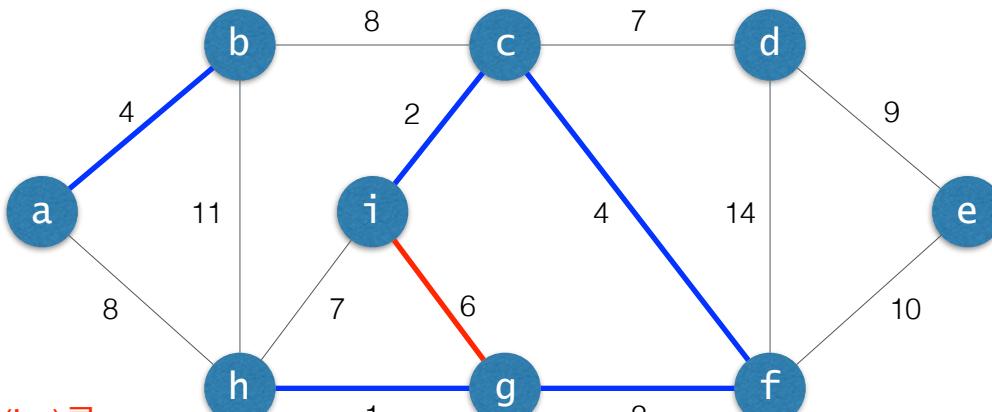
3. 에지 (c,f)를 선택하고, c와 f가 속한 집합을 합집합하여 하나의 집합으로 만듬

2. c와 f가 서로 다른 집합에 속함



$\{a, b\}$ $\{c, f, g, h, i\}$ $\{d\}$ $\{e\}$

사이클 검사



1. 가중치가 최소인 에지 (i,g)를 고려한다.

{a, b} {c, f, g, h, i} {d} {e}

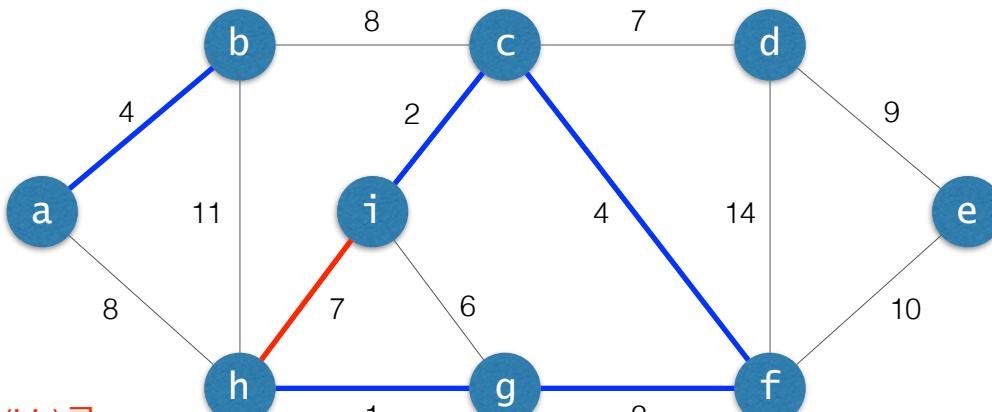
3. 에지 (i,g)를 선택하지 않는다.

2. i와 g는 이미 같은 집합에 속함. 즉 i와 g를 연결하면 사이클이 생김



{a, b} {c, f, g, h, i} {d} {e}

사이클 검사



1. 가중치가 최소인 에지 (i,h) 를 고려한다.

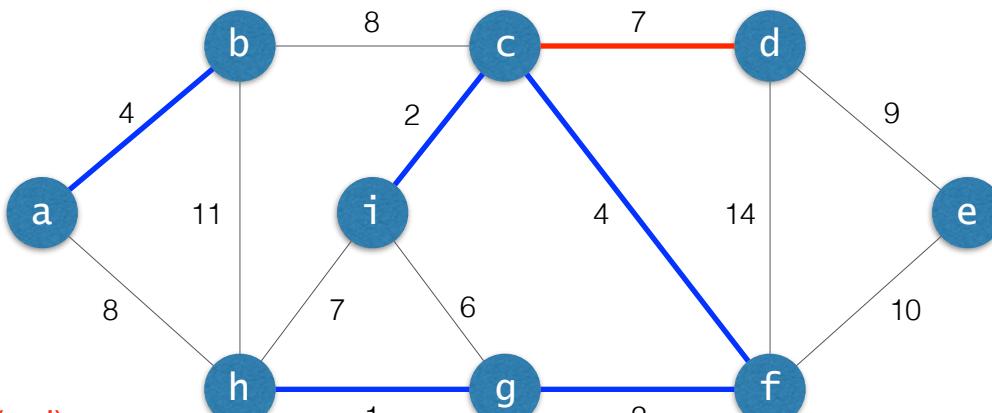
$\{a, b\}$ $\{c, f, g, \underline{h}, \underline{i}\}$ $\{d\}$ $\{e\}$

3. 에지 (i,h) 를 선택하지 않는다.

2. i 와 h 는 이미 같은 집합에 속함. 즉 i 와 h 를 연결하면 사이클이 생김



$\{a, b\}$ $\{c, f, g, h, i\}$ $\{d\}$ $\{e\}$



- 가중치가 최소인 에지 (c,d)를 고려한다.

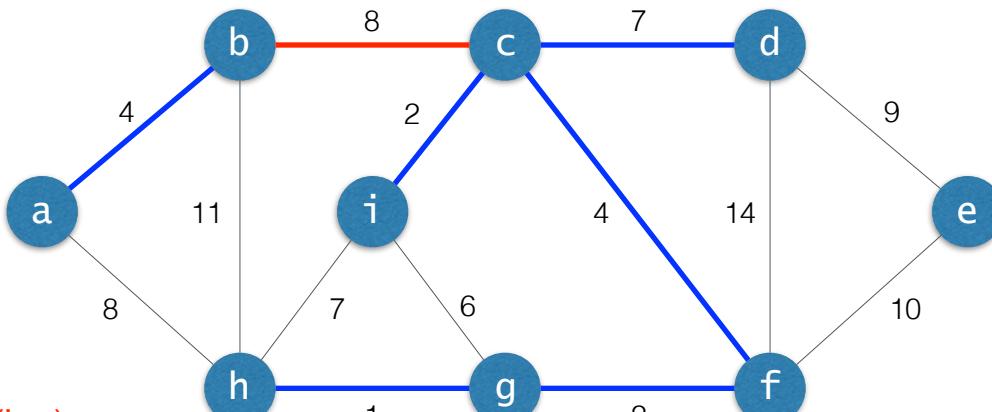
$\{a, b\}$ $\{c, f, g, h, i\}$ $\{d\}$ $\{e\}$

- 에지 (c,d)를 선택하고, c와 d가 속한 집합을 합집합하여 하나의 집합으로 만듬

- c와 d가 서로 다른 집합에 속함



$\{a, b\}$ $\{c, f, g, h, i, d\}$ $\{e\}$



1. 가중치가 최소인 에지 (b,c)
를 고려한다.

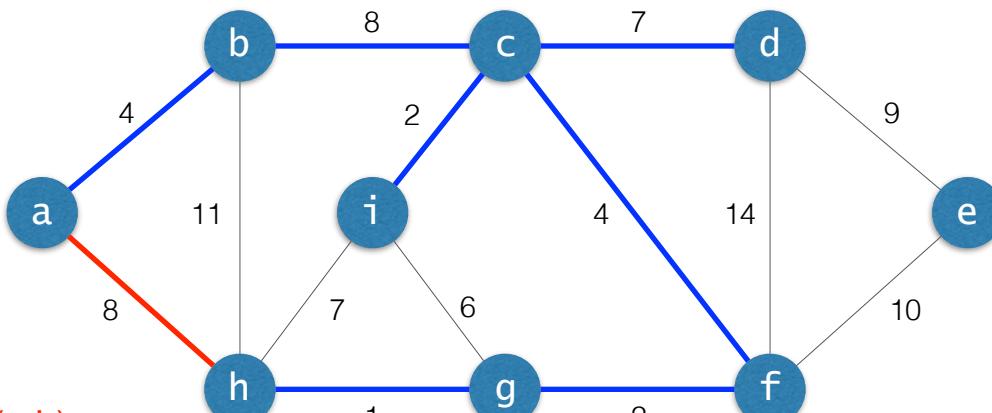
$\{\underline{a}, \underline{b}\}$ $\{\underline{c}, f, g, h, i, d\}$ $\{e\}$

3. 에지 (b,c)를 선택하고, b와 c가 속한
집합을 합집합하여 하나의 집합으로 만
듬

2. b와 c가 서로 다른 집합에 속함



$\{a, b, c, f, g, h, i, d\}$ $\{e\}$



- 가중치가 최소인 에지 (a,h) 를 고려한다.

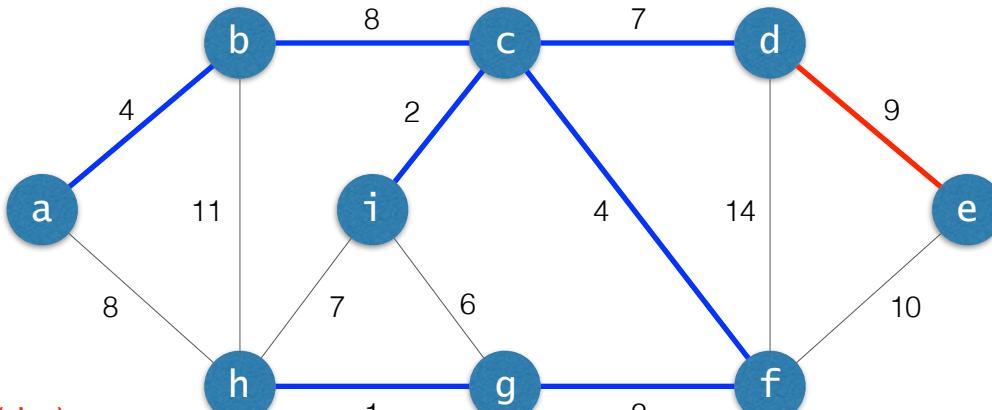
{a, b, c, f, g, h, i, d} {e}

3. 에지 (a,h) 를 선택하지 않는다.

2. a와 h는 이미 동일한 집합에 속함



{a, b, c, f, g, h, i, d} {e}



1. 가중치가 최소인 에지 (d,e)를 고려한다.

$\{a, b, c, f, g, h, i, d\}$ {e}

3. 에지 (d,e)를 선택하고, d와 e가 속한 집합을 합집합한다.



2. d와 e는 서로 다른 집합에 속함

$\{a, b, c, f, g, h, i, d, e\}$

4. n-1개의 에지가 선택되었으므로 종료한다.

MST-KRUSKAL(G, w)

```

1    $A \leftarrow \emptyset$ 
2   for each vertex  $v \in V[G]$ 
3       do MAKE-SET( $v$ )
4   sort the edges of  $E$  into nondecreasing order by weight  $w$ 
5   for each edge  $(u, v) \in E$ , taken in nondecreasing order by weight
6       do if FIND-SET( $u$ )  $\neq$  FIND-SET( $v$ )
7           then  $A \leftarrow A \cup \{(u, v)\}$ 
8           UNION( $u, v$ )
9   return  $A$ 
```

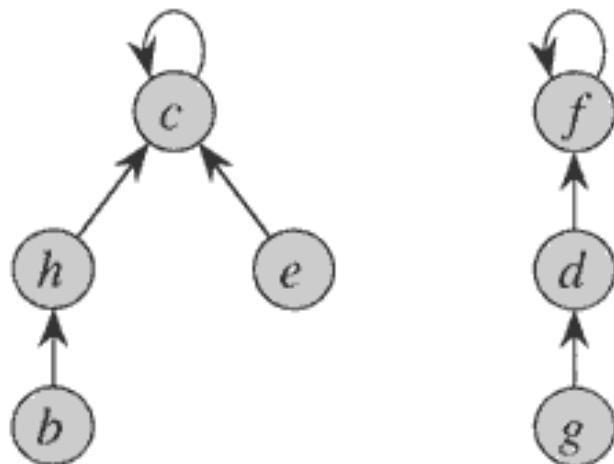
각각의 노드들을 유일한 원소로
가지는 집합들을 만들어라.

노드 v 가 속한 집합을
찾아라

u와 v가 속한
두 집합을 하나로 합친다.

서로소인 집합들의 표현

- 각 집합을 하나의 **트리**로 표현
- 예: 2개의 집합

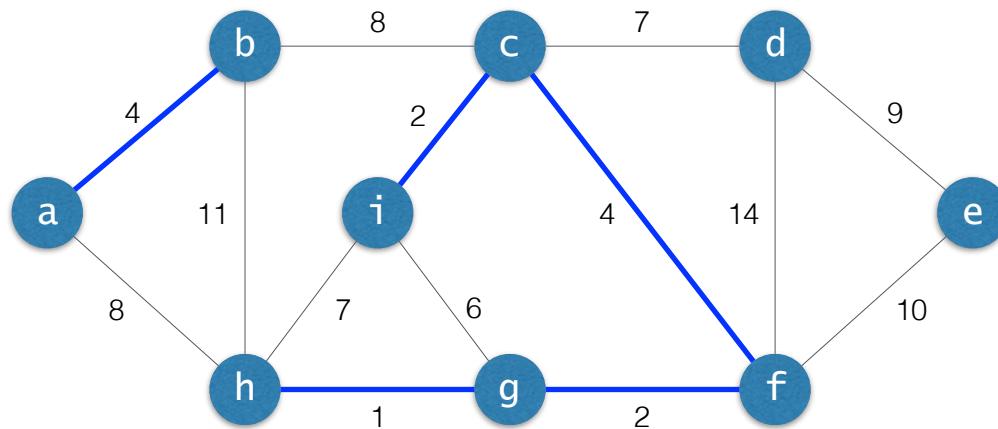


$\{b, c, e, h\}$, $\{d, f, g\}$

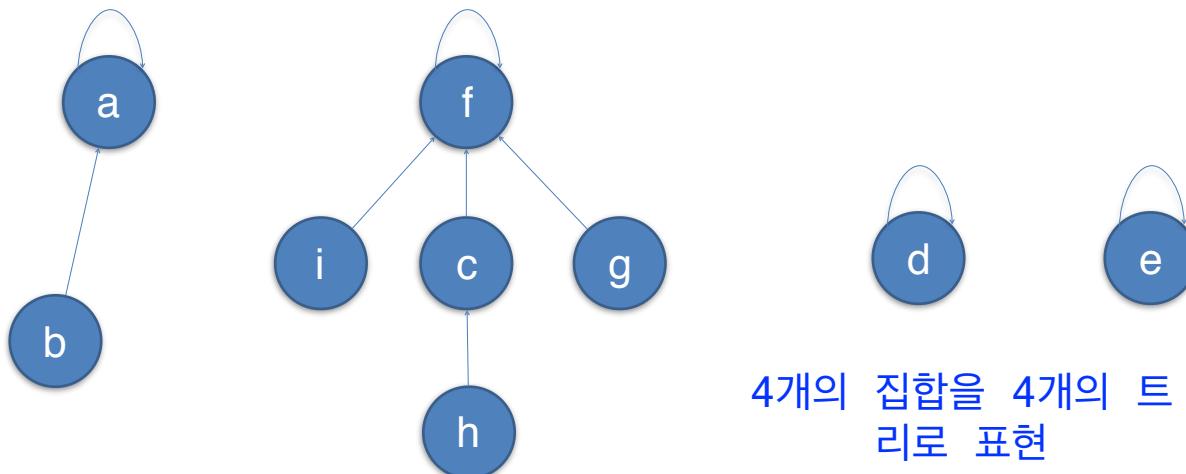
집합의 각 원소들이 트리의 노드가 됨. 누가 루트이고 누가 누구의 부모이든 상관없음.

트리의 각 노드는 자식노드가 아닌 부모 노드의 주소를 가짐
(상향식 트리)

서로소인 집합들의 표현

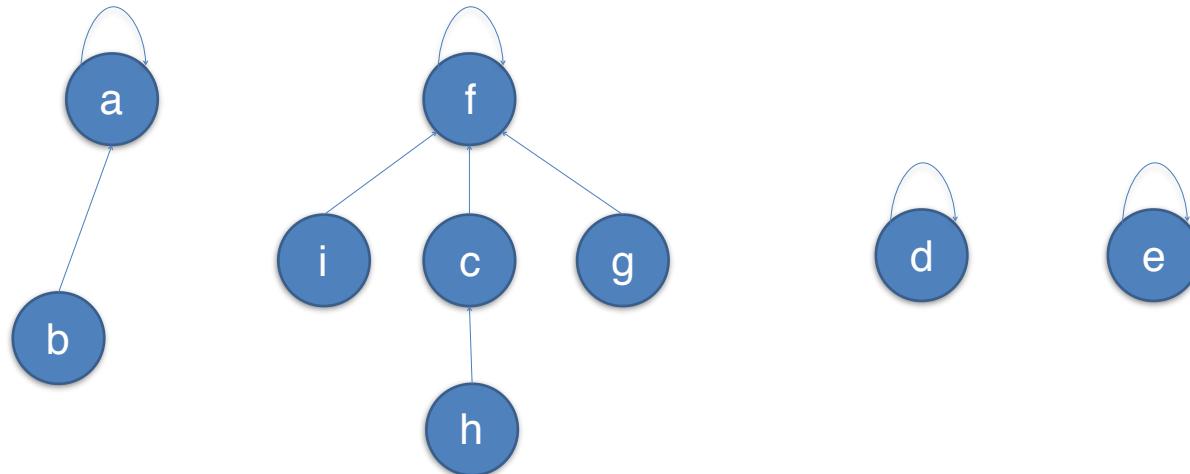


$\{a, b\}$ $\{c, i, f, g, h\}$ $\{d\}$ $\{e\}$



4개의 집합을 4개의 트리로 표현

{a,b} {c,i,f,g,h} {d} {e}

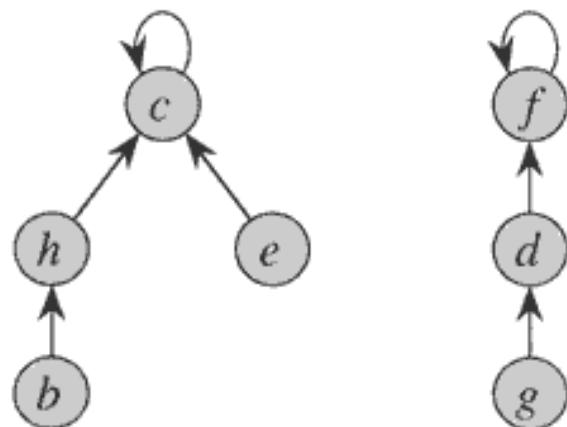


모든 트리를 하나의
배열로 표현

| | a | b | c | d | e | f | g | h | i |
|------|---|---|---|---|---|---|---|---|---|
| 배열 p | a | a | f | d | e | f | f | c | f |

Find-Set(v)

- 자신이 속한 트리의 루트를 찾음



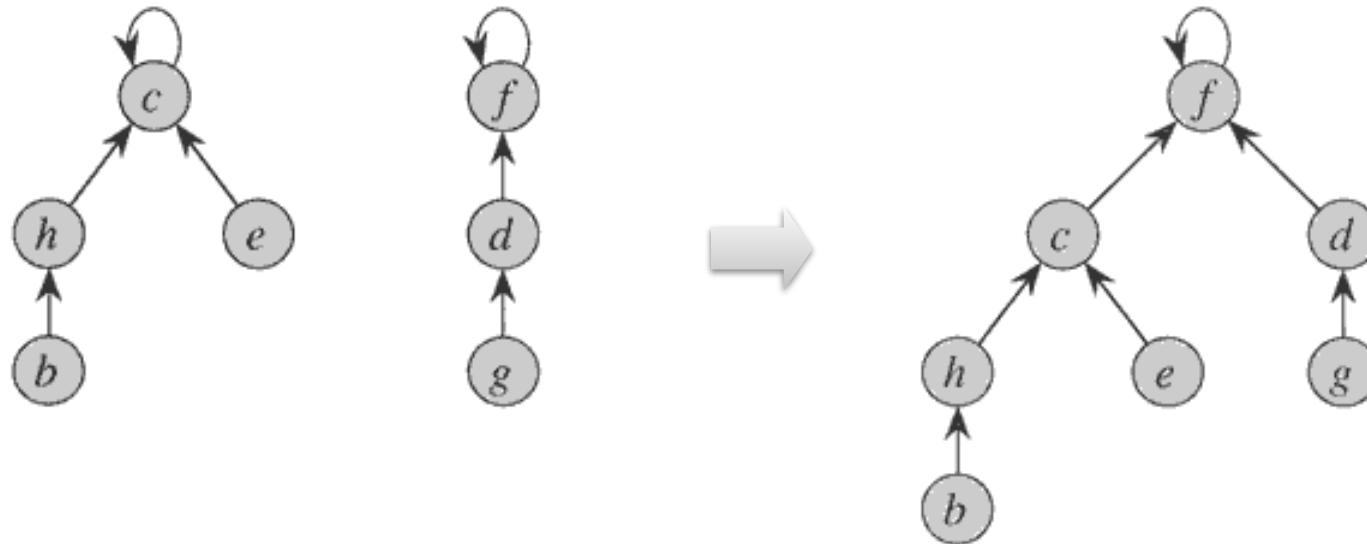
FIND-SET(x)

```
1 if  $x \neq p[x]$ 
2   then  $p[x] \leftarrow \text{FIND-SET}(p[x])$ 
3 return  $p[x]$ 
```

$O(h)$, h 는 트리의 높이
 h 는 최악의 경우 $O(n)$

Union(u, v)

- 한 트리의 루트를 다른 트리의 루트의 자식 노드로 만듬



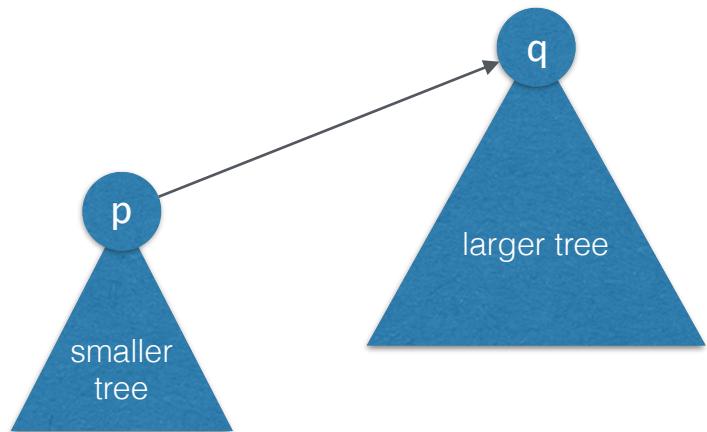
UNION(u, v)

1. $x \leftarrow \text{FIND-SET}(u);$
2. $y \leftarrow \text{FIND-SET}(v);$
3. $p[x] \leftarrow y;$

루트 노드를 찾은 이후에는 $O(1)$
하지만 루트를 찾는데 $O(h)$

weighted Union

- 두 집합을 union할 때 작은 트리의 루트를 큰 트리의 루트의 자식으로 만듬 (여기서 크기란 노드의 개수)
- 각 트리의 크기(노드의 개수)를 카운트하고 있어야



WEIGHTED-UNION(u, v)

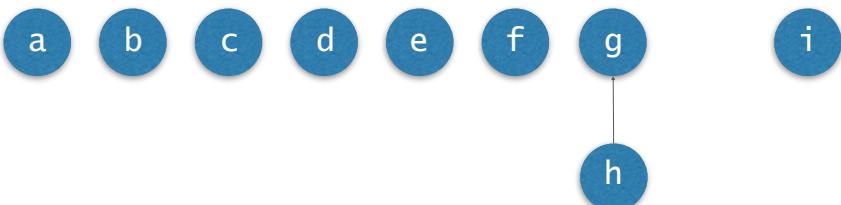
1. $x \leftarrow \text{FIND-SET}(u);$
2. $y \leftarrow \text{FIND-SET}(v);$
3. if $\text{size}[x] > \text{size}[y]$ then
4. $p[x] \leftarrow y;$
5. $\text{size}[x] \leftarrow \text{size}[x] + \text{size}[y];$
6. else
7. $p[y] \leftarrow x;$
8. $\text{size}[y] \leftarrow \text{size}[y] + \text{size}[x];$

Worst vs. Weighted Union

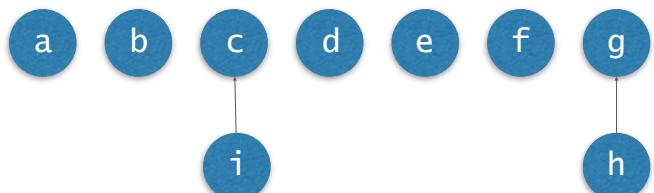
worst



$\text{union}(h, g)$



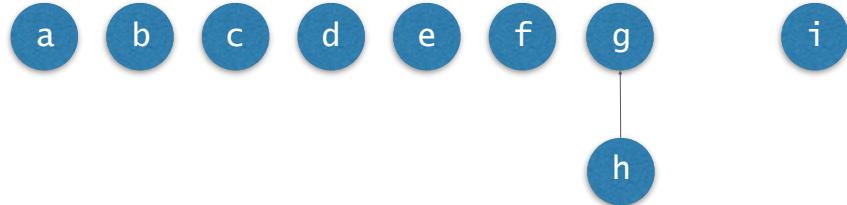
$\text{union}(c, i)$



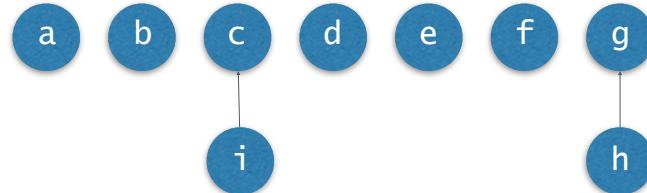
weighted



$\text{union}(h, g)$

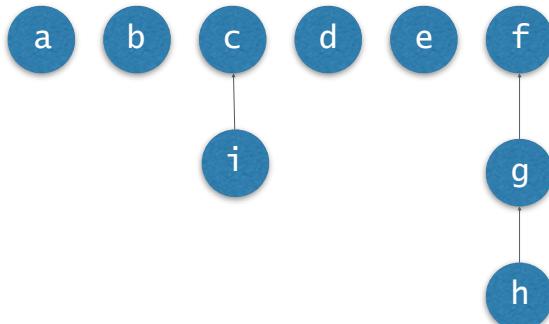


$\text{union}(c, i)$

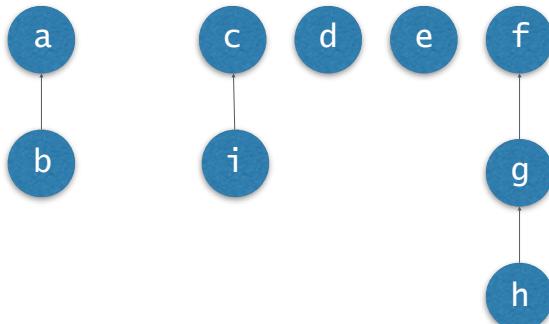


Worst vs. Weighted Union

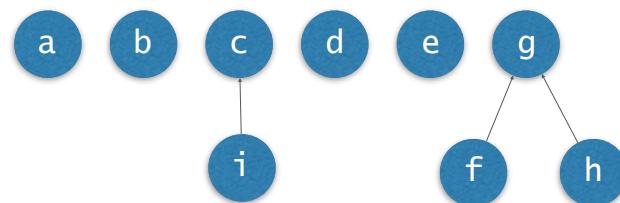
$\text{union}(g, f)$



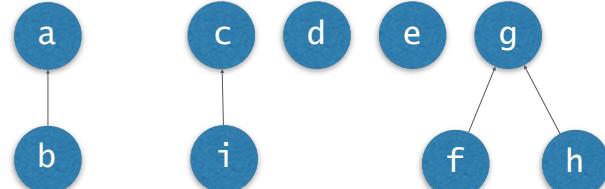
$\text{union}(a, b)$



$\text{union}(g, f)$

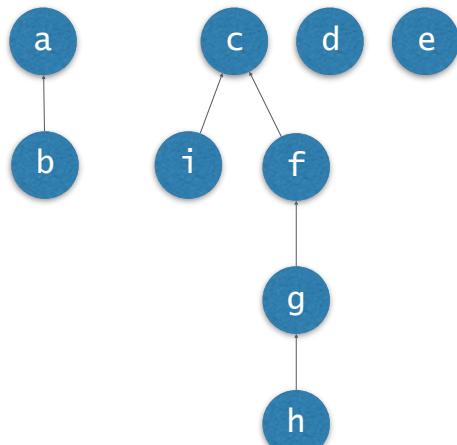


$\text{union}(a, b)$

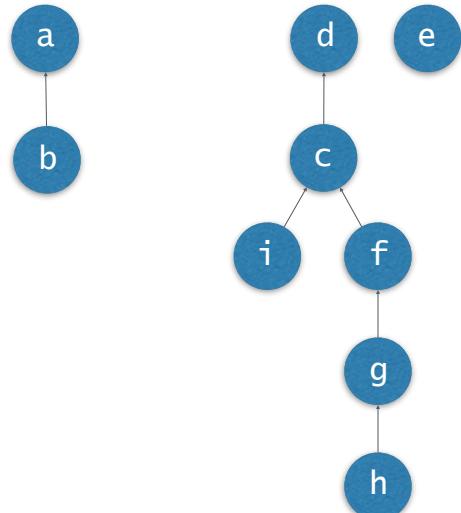


Worst vs. Weighted Union

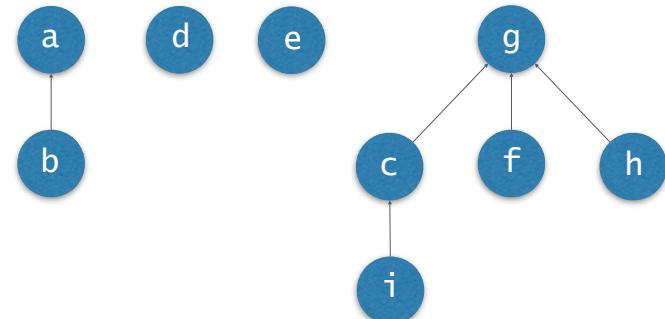
union(c, f)



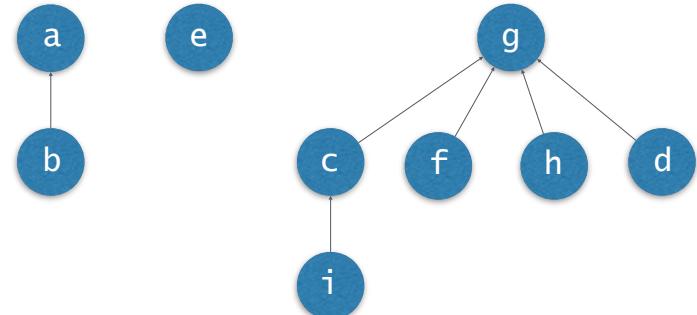
union(c, d)



union(c, f)

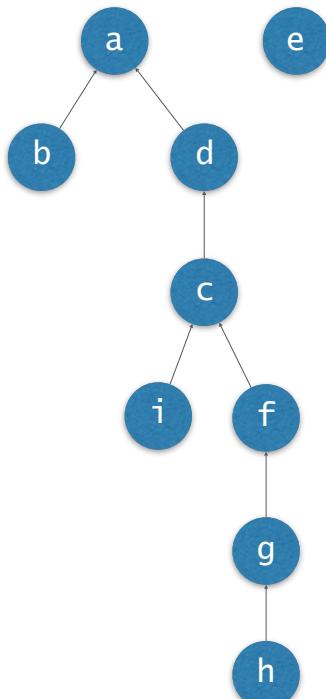


union(c, d)

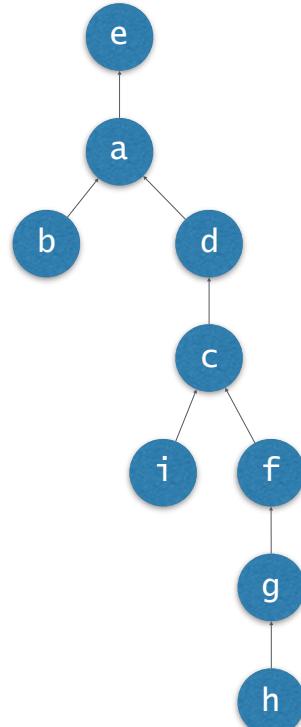


Worst vs. Weighted Union

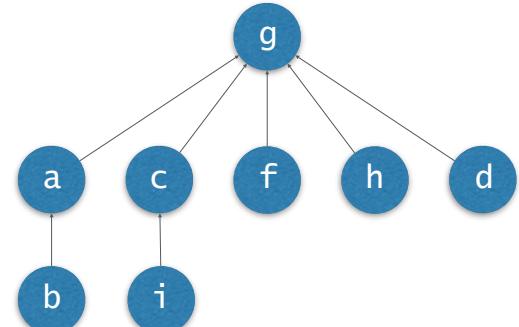
union(b,c)



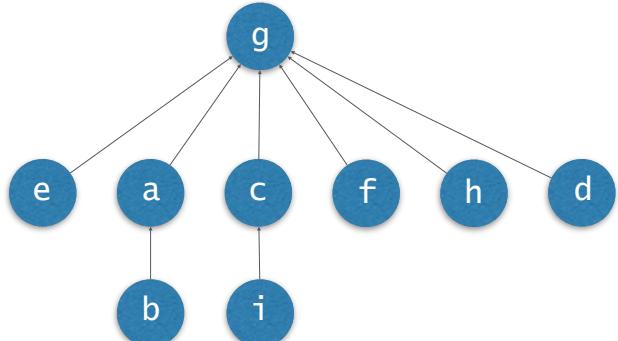
union(d,e)



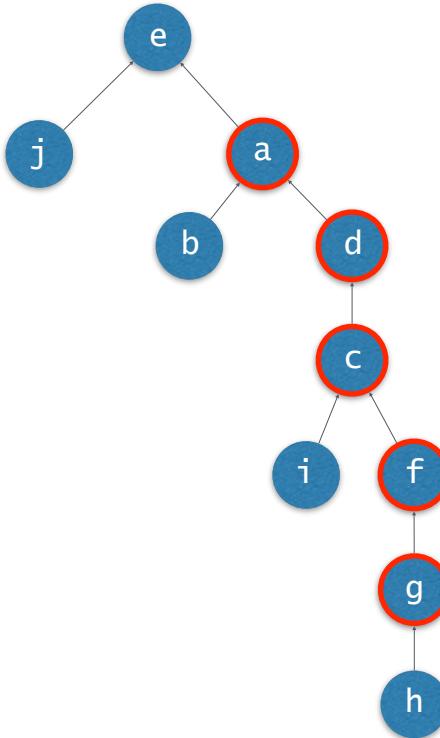
union(b,c)



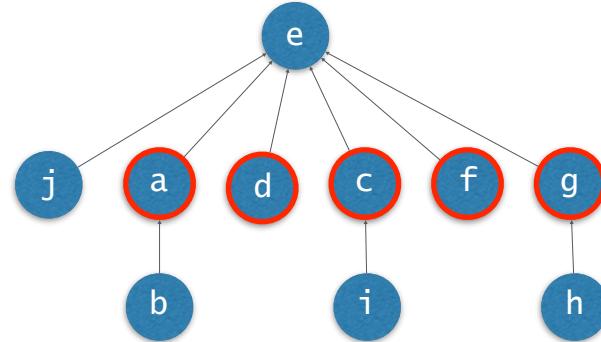
union(d,e)



Path Compression



Find(g)



FIND-SET-PC(x)

1. while $x \neq p[x]$ do
2. $p[x] \leftarrow p[p[x]]$;
3. $x \leftarrow p[x]$;
4. end.
5. return $p[x]$;

Weighted Union with Path Compression (WUPC)

- M번의 union-find 연산의 총 시간복잡도는 $O(N+M\lg^* N)$. 여기서 N은 원소의 개수
- 거의 선형시간 알고리즘, 즉 한 번의 Find 혹은 Union이 거의 $O(1)$ 시간

| N | $\lg^* N$ |
|--------|-----------|
| 1 | 0 |
| 2 | 1 |
| 4 | 2 |
| 16 | 3 |
| 65536 | 4 |
| 265536 | 5 |

\lg^* function

- 시간복잡도

Initialize A: $O(1)$

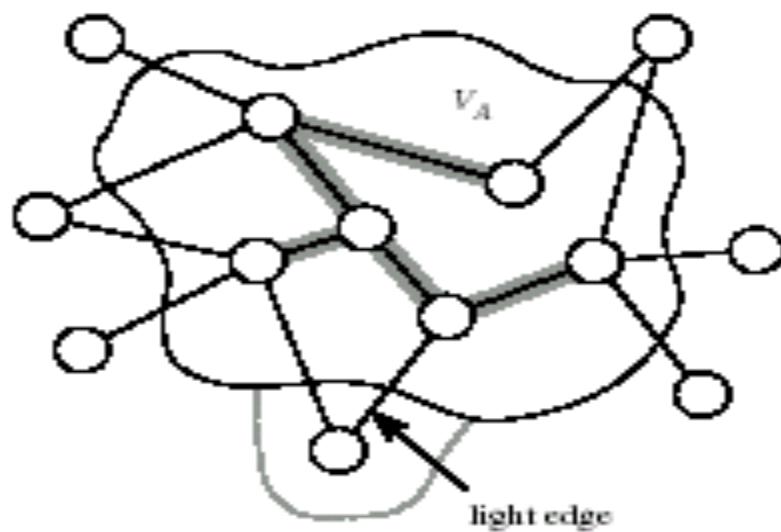
First for loop: $|V|$ MAKE-SETs

Sort E: $O(|E| \log_2 |E|)$

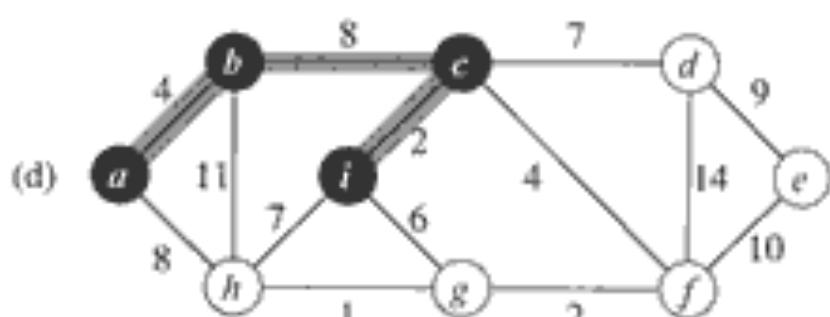
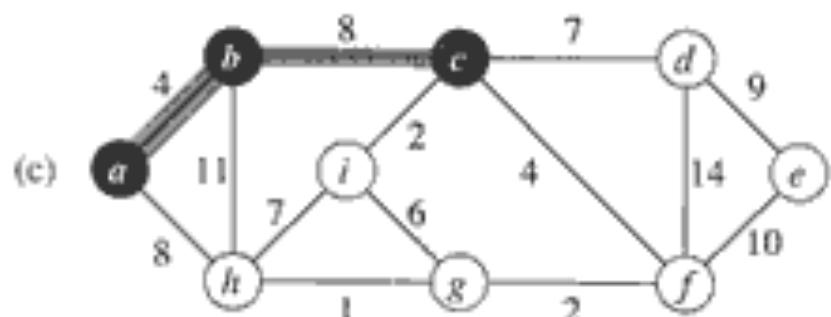
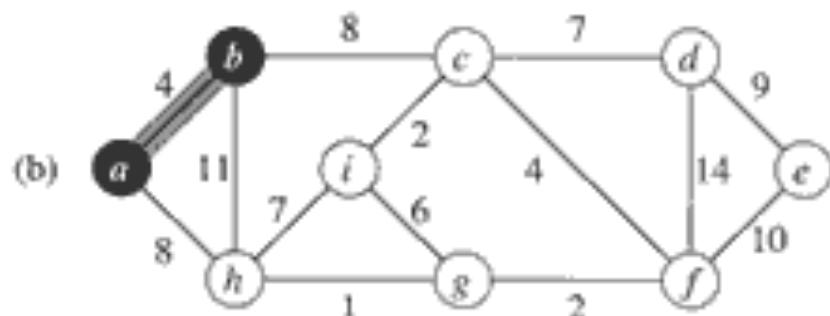
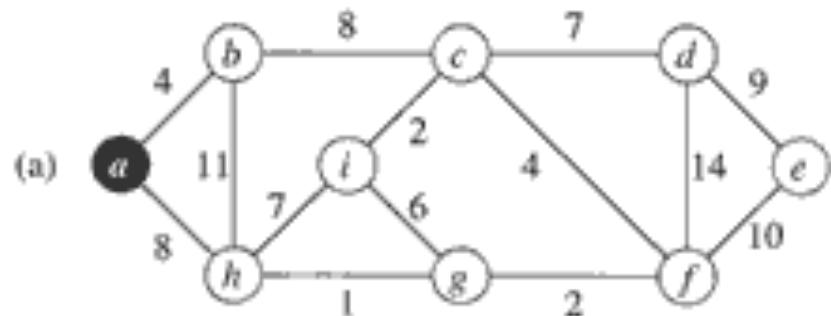
Second for loop: $O(|E|)$ FIND-SETs and UNIONs ?

$$O(|E| \log_2 |E|)$$

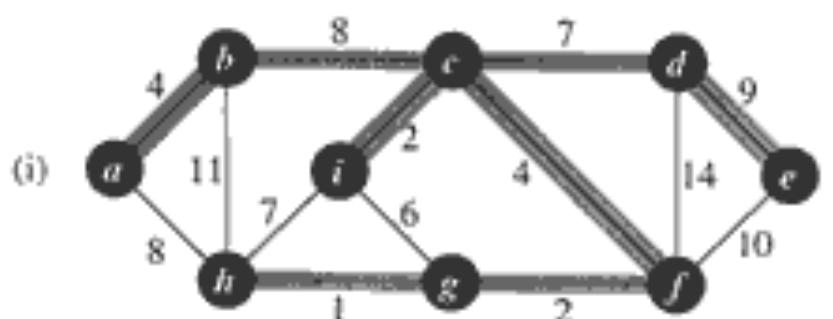
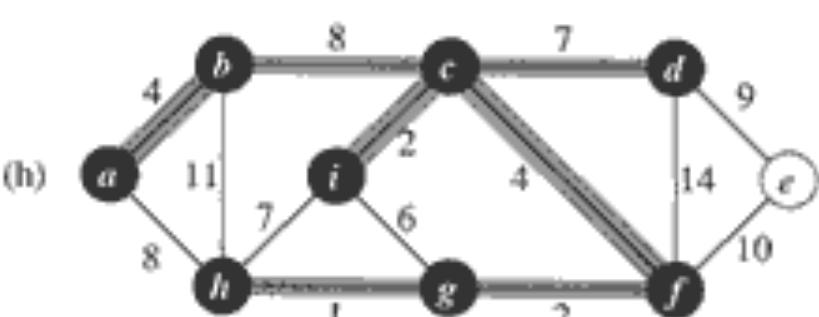
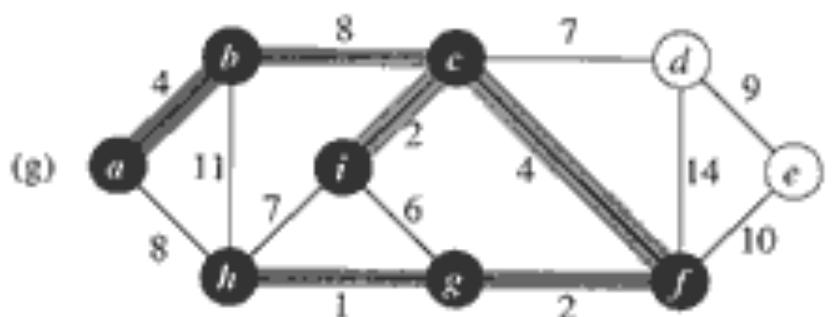
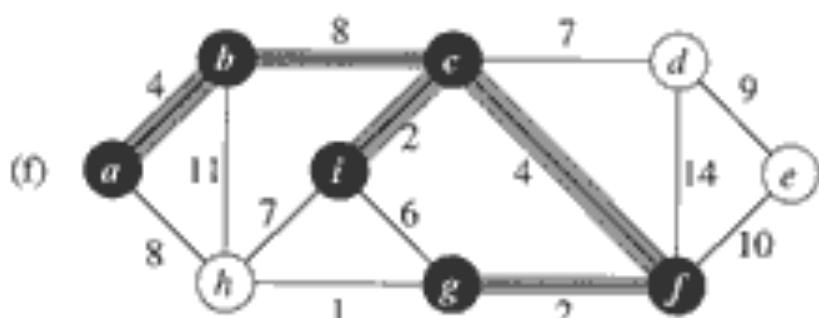
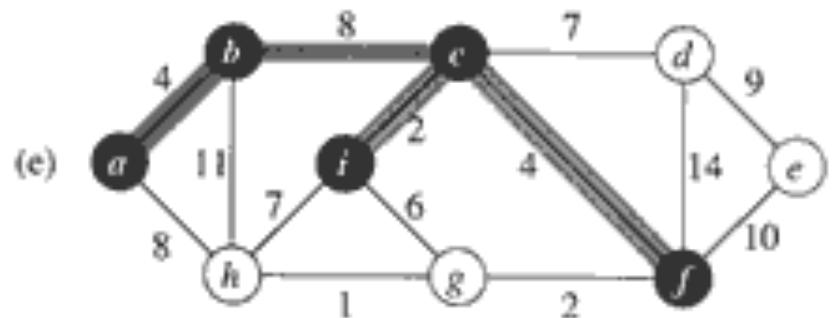
- 임의의 노드를 출발노드로 선택
- 출발 노드를 포함하는 트리를 점점 키워감.
- 매 단계에서 이미 트리에 포함된 노드와 포함되지 않은 노드를 연결하는 에지들 중 가장 가중치가 작은 에지를 선택



Prim의 알고리즘

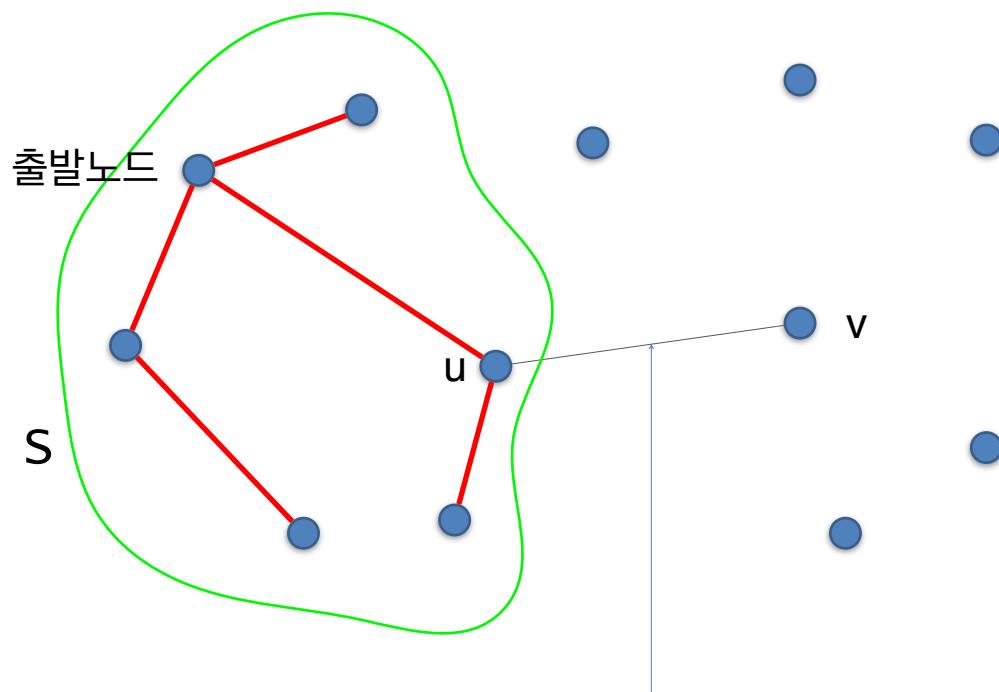


Prim의 알고리즘



왜 MST가 찾아지는가?

- Prim의 알고리즘의 임의의 한 단계를 생각해보자.
- A를 현재까지 알고리즘이 선택한 에지의 집합이라고 하고, A를 포함하는 MST가 존재한다고 가정하자.

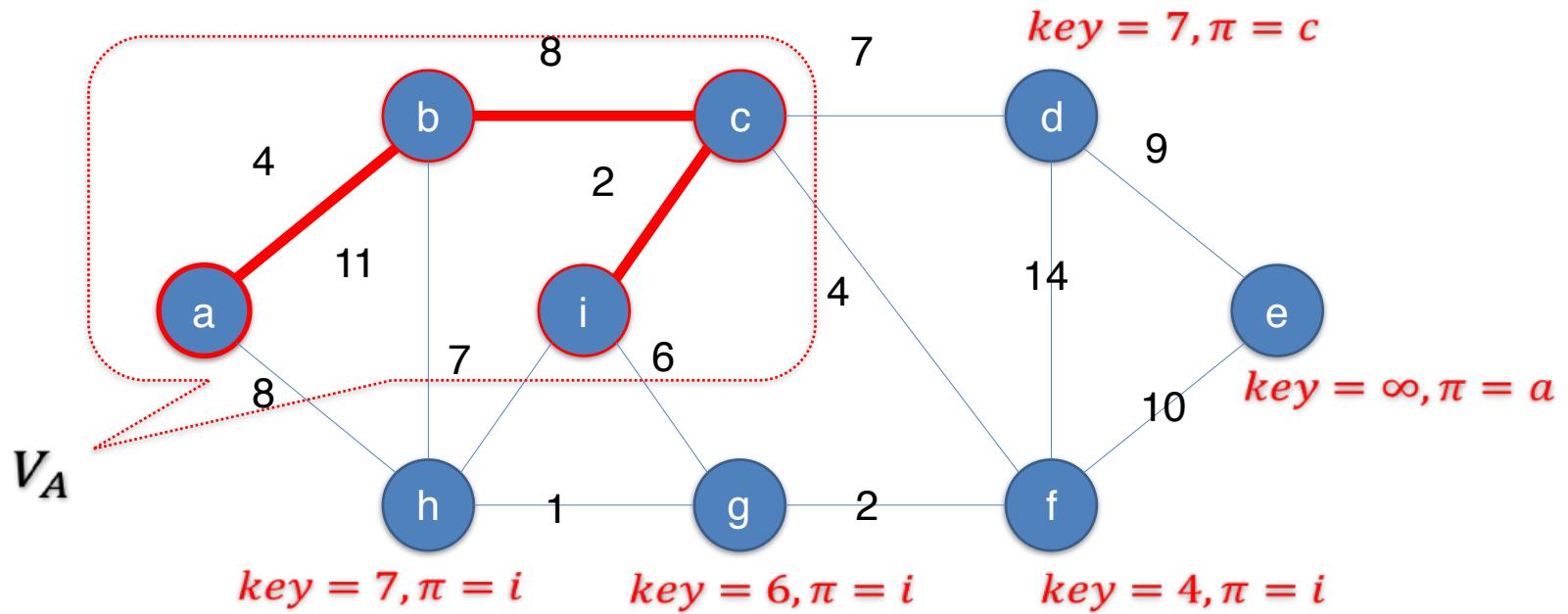


출발 노드에 이미 연결된 노드와 그렇지 않은 노드를 연결하는 에지들 중 **lightest edge**

가중치가 최소인 에지 찾기

- V_A : 이미 트리에 포함된 노드들
- V_A 에 아직 속하지 않은 각 노드 v 에 대해서 다음과 같은 값을 유지
 - $\text{key}(v)$: 이미 V_A 에 속한 노드와 자신을 연결하는 에지들 중 가중치가 최소인 에지 (u, v) 의 가중치
 - $\pi(v)$: 그 에지 (u, v) 의 끝점 u

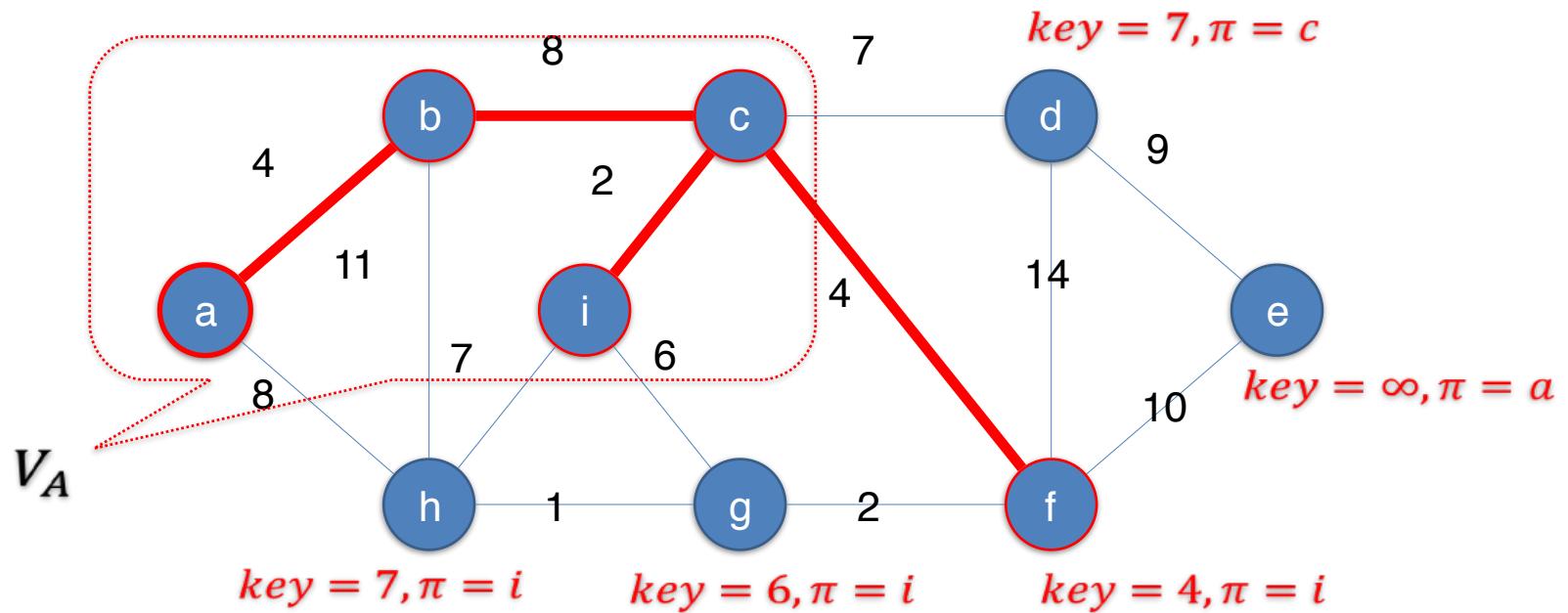
가중치가 최소인 에지 찾기



가중치가 최소인 에지를 찾는 대신

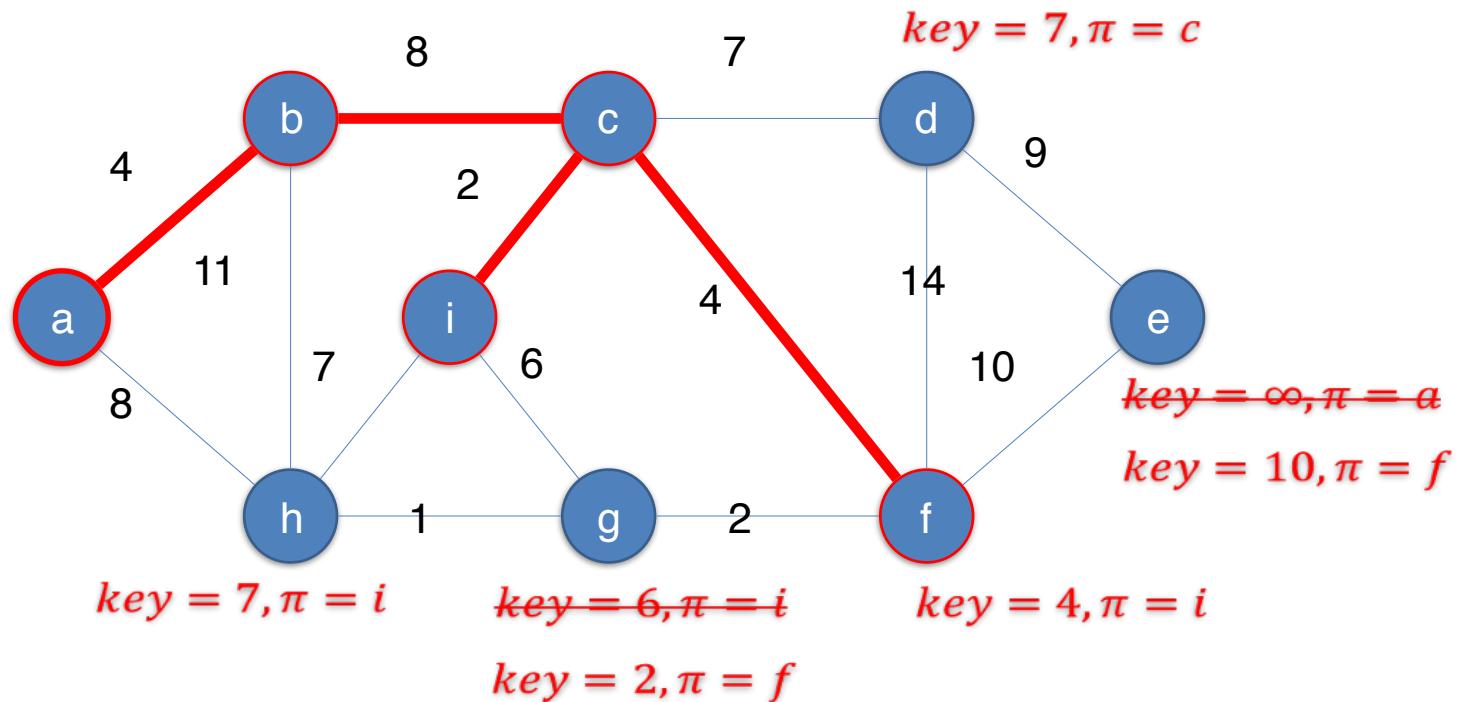
key값이 최소인 노드를 찾는다.

가중치가 최소인 에지 찾기



key 값이 최소인 노드 f를 찾고,
에지 $(f, \pi(f))$ 를 선택한다.

가중치가 최소인 에지 찾기



노드 d, g, e의 key 값과 π 값을 갱신한다.

Prim의 알고리즘

MST-Prim(G, w, r)

1. for each $u \in V$ do
2. $key[u] \leftarrow \infty$
3. $\pi[u] \leftarrow \text{NIL}$
4. end.
5. $V_A \leftarrow \{r\}$
6. $key[r] \leftarrow 0$
7. while $|V_A| < n$ do
8. find $u \notin V_A$ with the **minimum key value**;
9. $V_A \leftarrow V_A \cup \{u\}$
10. for each $v \notin V_A$ adjacent to u do
11. if $key[v] > w(u, v)$ then
12. $key[v] \leftarrow w(u, v)$
13. $\pi[v] \leftarrow u$
14. end.
15. end.
16. end.

while문은 $n-1$ 번 반복

최소값 찾기 $O(n)$

$\text{degree}(u) = O(n)$

시간복잡도 $O(n^2)$

key값이 최소인 노드 찾기

- 최소 우선순위 큐를 사용
 - $V - V_A$ 에 속한 노드들을 저장
 - Extract-Min: key값이 최소인 노드를 삭제하고 반환

MST-PRIM(G, w, r)

```

1  for each  $u \in V[G]$ 
2      do  $key[u] \leftarrow \infty$ 
3           $\pi[u] \leftarrow \text{NIL}$ 
4   $key[r] \leftarrow 0$ 
5   $Q \leftarrow V[G]$ 
6  while  $Q \neq \emptyset$ 
7      do  $u \leftarrow \text{EXTRACT-MIN}(Q)$ 
8          for each  $v \in Adj[u]$ 
9              do if  $v \in Q$  and  $w(u, v) < key[v]$ 
10                 then  $\pi[v] \leftarrow u$ 
11                  $key[v] \leftarrow w(u, v)$ 

```

최소값 찾기 $O(\log n)$

우선순위큐에서 key값 decrease: $O(\log n)$

Prim의 알고리즘: 시간복잡도

- 이진 힙(binary heap)을 사용하여 우선순위 큐를 구현한 경우
- while loop:
 - n 번의 Extract-Min 연산: $O(n \log n)$
 - m 번의 Decrease-Key 연산: $O(m \log n)$
- 따라서 시간복잡도: $O(n \log n + m \log n) = O(m \log n)$
- 우선순위 큐를 사용하지 않고 단순하게 구현할 경우: $O(n^2)$
- Fibonacci 힙을 사용하여 $O(m+n \log n)$ 에 구현 가능 [Fredman-Tarjan 1984]

최단경로
Shortest Path

최단경로

- 가중치 (방향) 그래프 $G=(V, E)$, 즉 모든 에지에 가중치가 있음
- 경로 $p=(v_0, v_1, \dots, v_k)$ 의 길이는 경로상의 모든 에지의 가중치의 합
- 노드 u 에서 v 까지의 최단경로의 길이를 $\delta(u, v)$ 라고 표시하자.

최단경로문제의 유형

- **Single-source:**

- 하나의 출발 노드 s 로부터 다른 모든 노드까지의 최단 경로를 찾아라.
- 예: Dijkstra의 알고리즘

- **Single-destination:**

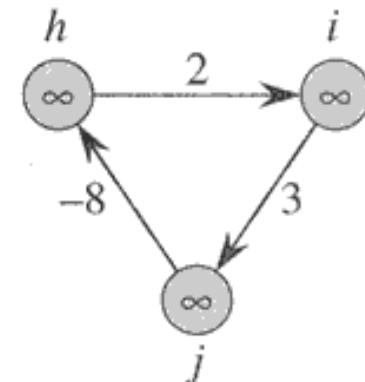
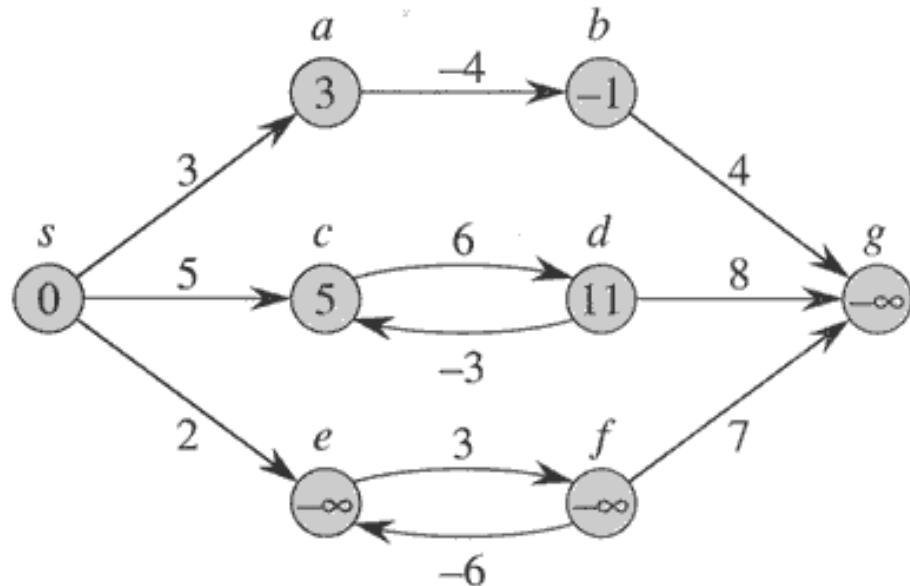
- 모든 노드로부터 하나의 목적지 노드까지의 최단 경로를 찾아라.
- Single-source 문제와 동일

- **Single-pair:**

- 주어진 하나의 출발 노드 s 로부터 하나의 목적지 노드 t 까지의 최단 경로를 찾아라
- 최악의 경우 시간복잡도에서 Single-source 문제보다 나은 알고리즘이 없음

- **All-pairs:**

- 모든 노드 쌍에 대해서 최단 경로를 찾아라.



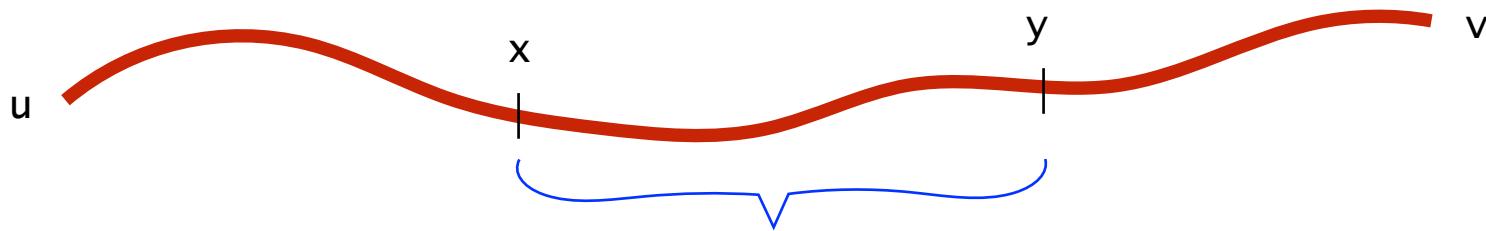
음수 사이클(negative cycle)이
있으면 최단 경로가 정의되지 않음

알고리즘에 따라 음수 가중치가 있어도 작동하는 경우도
있고 그렇지 않은 경우도 있음

최단경로의 기본 특성

- 최단 경로의 어떤 부분경로도 역시 최단 경로이다.

이 경로가 u 에서 v 까지의 최단경로라면



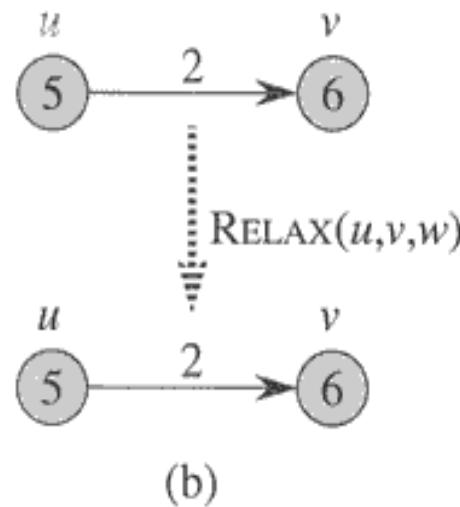
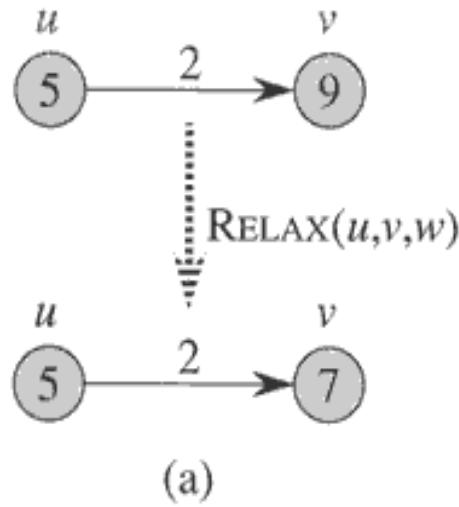
이 경로는 x 에서 y 까지의 최단경로이다.

- 최단 경로는 사이클을 포함하지 않는다. (음수 사이클이 없다는 가정하에서)

Single-source 최단경로문제

- **입력:** 음수 사이클이 없는 가중치 방향그래프 $G=(V, E)$ 와 출발 노드 $s \in V$
- **목적:** 각 노드 $v \in V$ 에 대해서 다음을 계산한다.
 - $d[v]$
 - 처음에는 $d[s]=0$, $d[v]=\infty$ 로 시작한다.
 - 알고리즘이 진행됨에 따라서 감소해간다. 하지만 항상 $d[v] \geq \delta(s, v)$ 를 유지한다
 - 최종적으로는 $d[v] = \delta(s, v)$ 가 된다.
 - $\pi[v]$: s 에서 v 까지의 최단경로상에서 v 의 직전 노드(predecessor)
 - 그런 노드가 없는 경우 $\pi[v]=NIL$.

기본 연산: Relaxation



$\text{RELAX}(u, v, w)$

- 1 **if** $d[v] > d[u] + w(u, v)$
- 2 **then** $d[v] \leftarrow d[u] + w(u, v)$
- 3 $\pi[v] \leftarrow u$

Single-source 최단경로

- 대부분의 single-source 최단경로 알고리즘의 기본 구조
 1. 초기화: $d[s]=0$, 노드 $v \neq s$ 에 대해서 $d[v]=\infty$, $\pi[v]=NIL$.
 2. 에지들에 대한 반복적인 relaxation
- 알고리즘들 간의 차이는 어떤 에지에 대해서, 어떤 순서로 relaxation을 하느냐에 있음

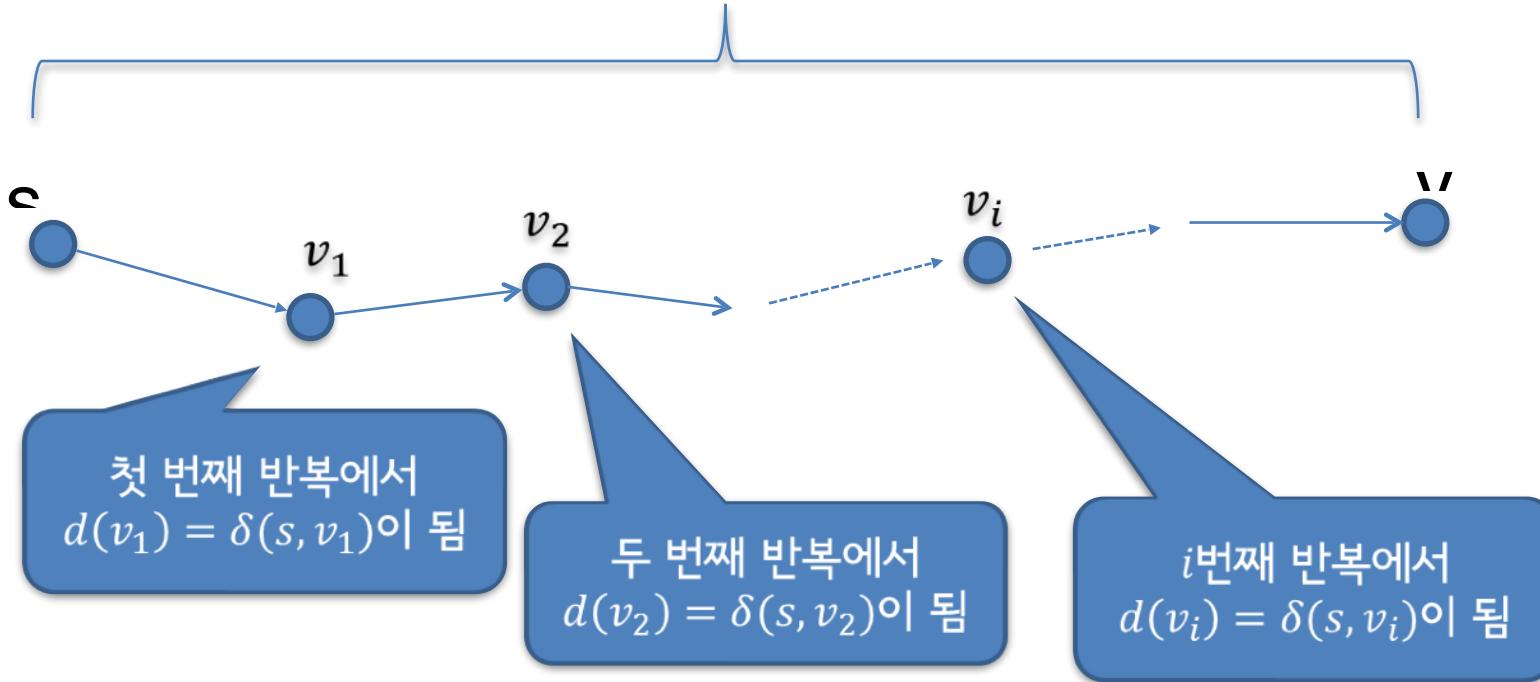
Generic-Single-Source(G, w, s)

1. INITIALISE-SINGLE-SOURCE(G, s)
2. repeat
3. for each edge $(u, v) \in E$
4. RELAX(u, v, w)
5. until there is no change.

질문 2: 몇 번 반복해야 ?

질문 1: 이렇게 계속 반복하면 최단 경로가 찾아지는가?

이것이 s 에서 v 까지의 최단 경로라면



즉, $n-1$ 번의 반복으로 충분하다.

Bellman-Ford 알고리즘

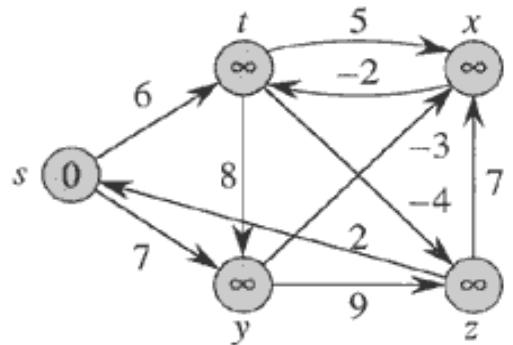
BELLMAN-FORD(G, w, s)

```
1  INITIALIZE-SINGLE-SOURCE( $G, s$ )
2  for  $i \leftarrow 1$  to  $|V[G]| - 1$ 
3      do for each edge  $(u, v) \in E[G]$ 
4          do RELAX( $u, v, w$ )
5  for each edge  $(u, v) \in E[G]$ 
6      do if  $d[v] > d[u] + w(u, v)$ 
7          then return FALSE
8  return TRUE
```

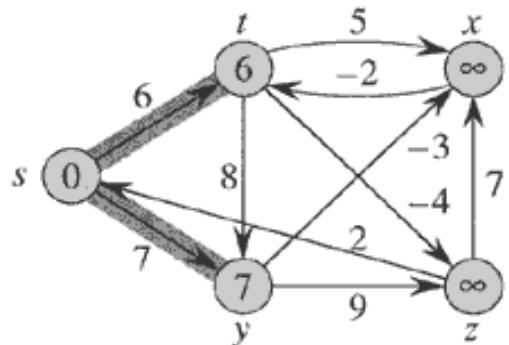
음수 사이클이 존재한다는 의미

시간복잡도 $O(nm)$

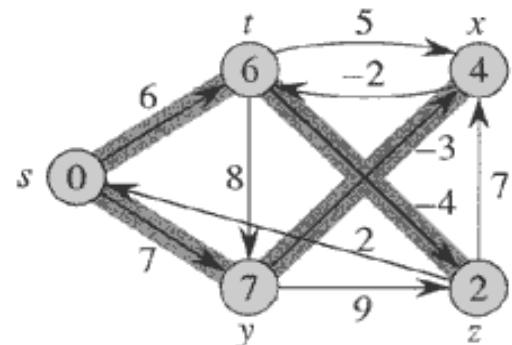
Bellman-Ford 알고리즘



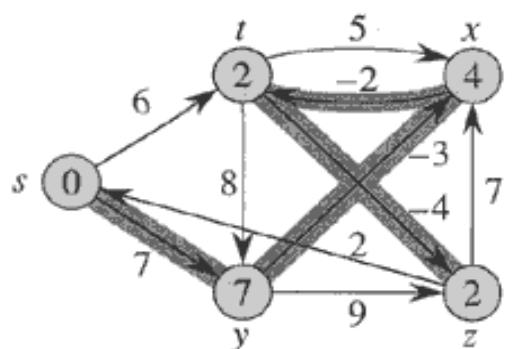
(a)



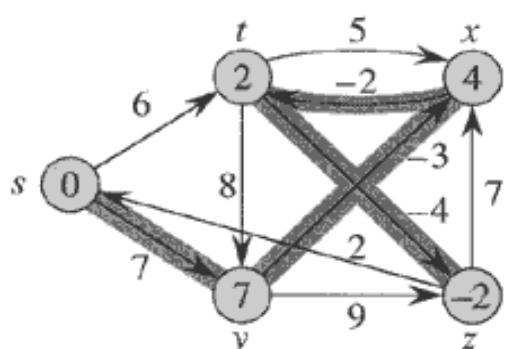
(b)



(c)



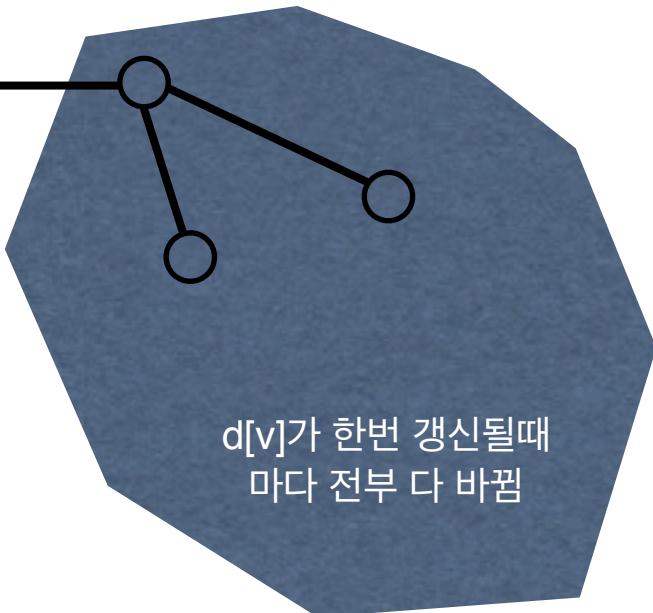
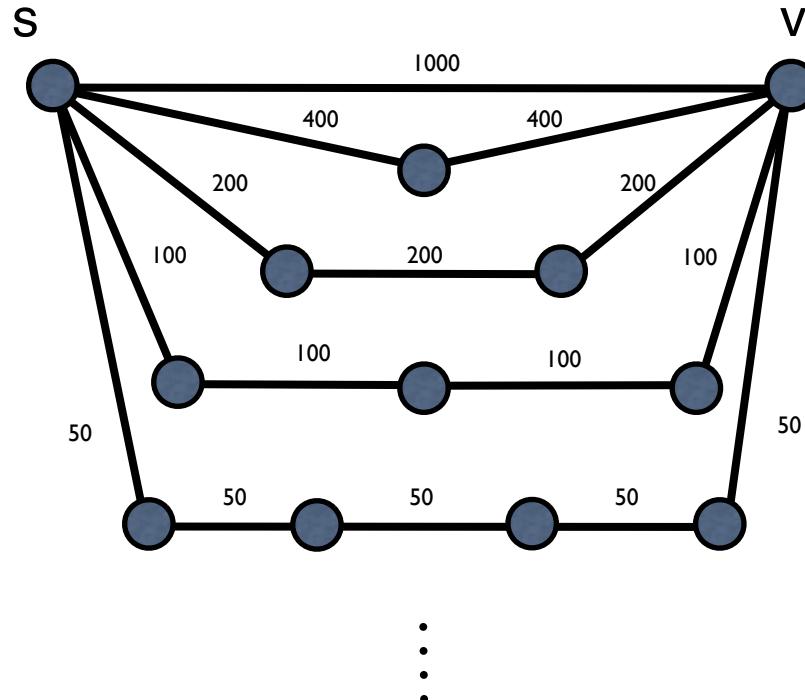
(d)



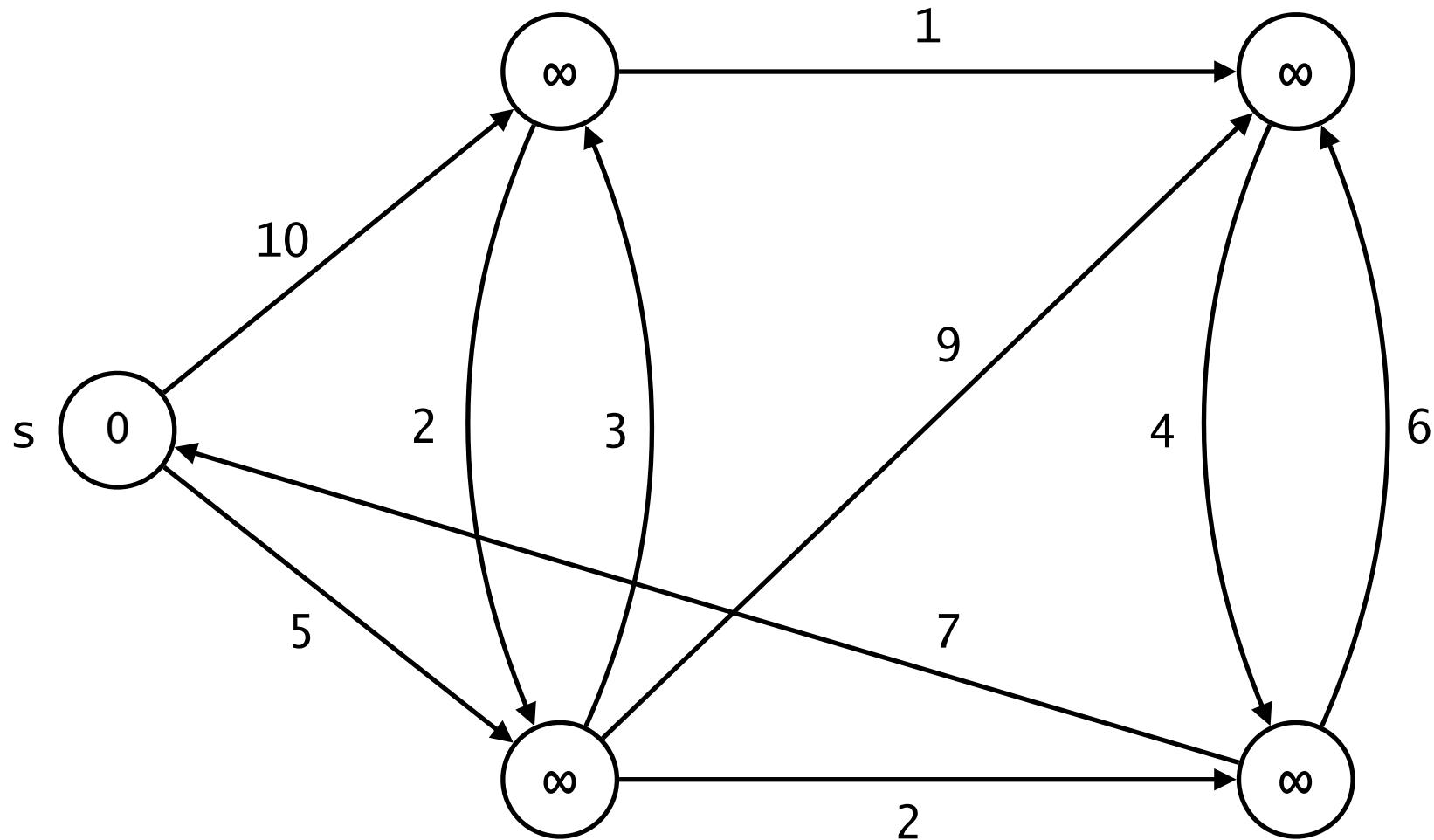
(e)

- 한 노드가 여러번 queue에 들어감

Worst Scenario

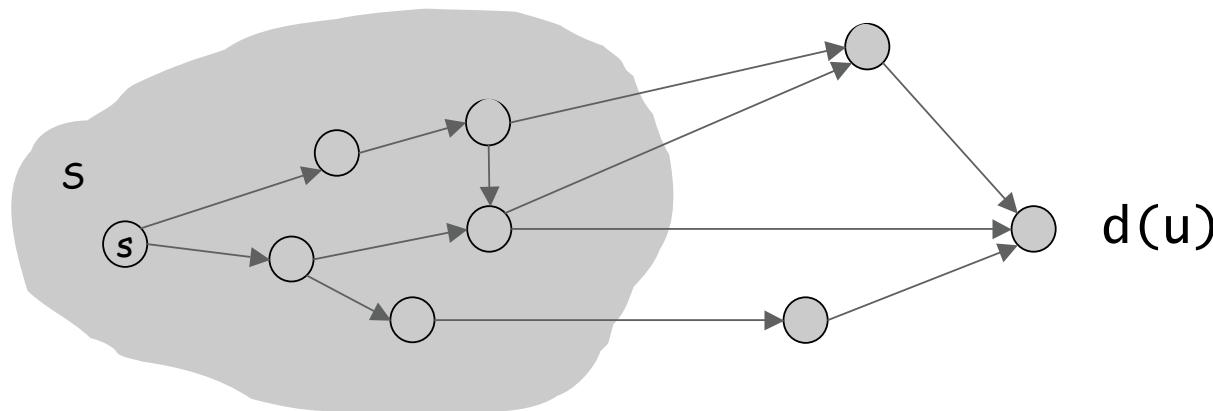


Dijkstra의 알고리즘



Dijkstra의 알고리즘

- 음수 가중치가 없다고 가정
- s 로부터의 최단경로의 길이를 이미 알아낸 노드들의 집합 S 를 유지. 맨 처음엔 $S=\{s\}$.
- Loop invariant:
 - $u \notin S$ 인 각 노드 u 에 대해서 $d(u)$ 는 이미 S 에 속한 노드들만 거쳐서 s 로부터 u 까지 가는 최단경로의 길이

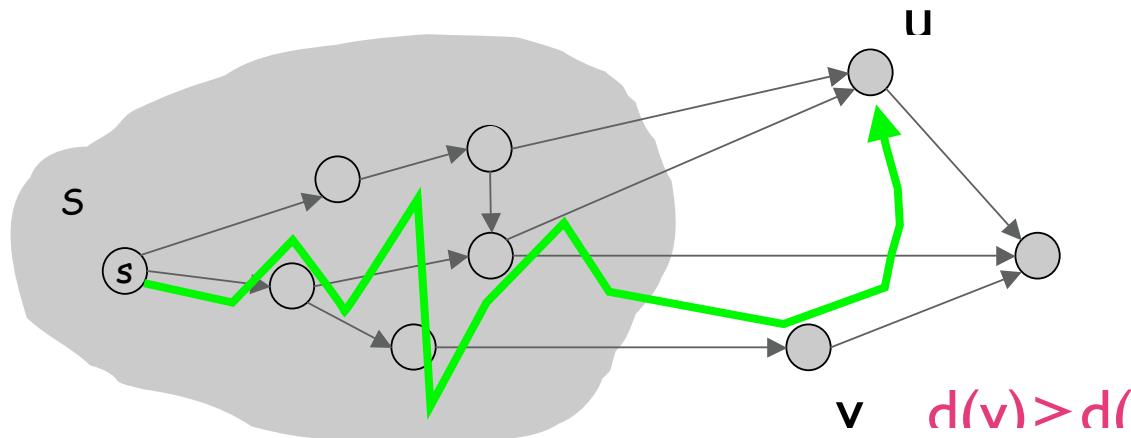


Dijkstra의 알고리즘

정리: $d(u) = \min_{v \notin S} d(v)$ 인 노드 u 에 대해서, $d(u)$ 는 s 에서 u 까지의 최단경로의 길이이다.

증명: (proof by contradiction)

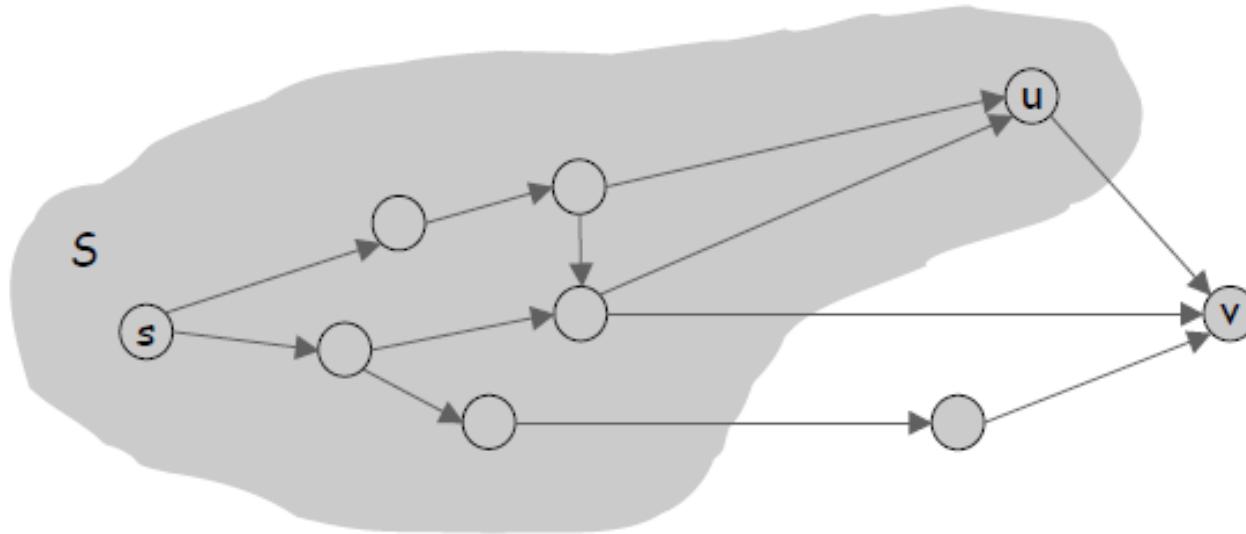
- 아니라고 하자. 그러면 s 에서 u 까지 다른 최단경로가 존재



- $d(v) \geq d(u)$ 이므로 모순

Dijkstra의 알고리즘

- $d(u)$ 가 최소인 노드 $u \notin S$ 를 찾고, S 에 u 를 추가
- S 가 변경되었으므로 다른 노드들의 $d(v)$ 값을 갱신



$$d(v) = \min\{d(v), d(u) + w(u, v)\}$$

즉, 에지 (u, v) 에 대해서 relaxation하면 Loop Invariant가 계속 유지됨

Dijkstra의 알고리즘

```
Gijkstra(G, w, s)
```

1. for each $u \in V$ do
2. $d[u] \leftarrow \infty$
3. $\pi[u] \leftarrow \text{NIL}$
4. end.
5. $S \leftarrow \{s\}$
6. $d[s] \leftarrow 0$
7. while $|S| < n$ do
8. find $u \notin S$ with the **minimum $d[u]$ value**;
9. $S \leftarrow S \cup \{u\}$
10. for each $v \notin S$ adjacent to u do
11. if $d[v] > d[u] + w(u, v)$ then
12. $d[v] \leftarrow d[u] + w(u, v)$
13. $\pi[v] \leftarrow u$
14. end.
15. end.
16. end.

while문은 $n-1$ 번 반복

최소값 찾기 $O(n)$

$\text{degree}(u) = O(n)$

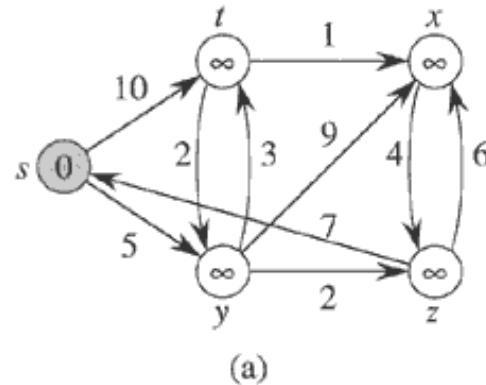
Dijkstra의 알고리즘

DIJKSTRA(G, w, s)

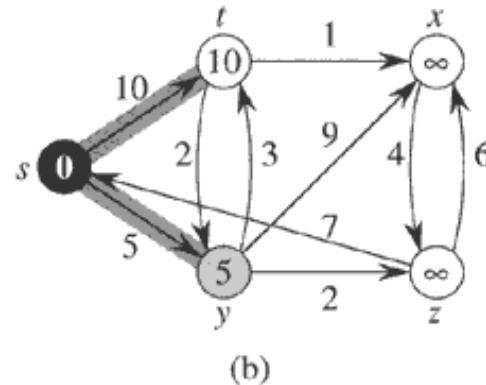
- 1 INITIALIZE-SINGLE-SOURCE(G, s)
- 2 $S \leftarrow \emptyset$
- 3 $Q \leftarrow V[G]$
- 4 **while** $Q \neq \emptyset$
 - 5 **do** $u \leftarrow \text{EXTRACT-MIN}(Q)$
 - 6 $S \leftarrow S \cup \{u\}$
 - 7 **for each vertex** $v \in Adj[u]$
 - 8 **do** RELAX(u, v, w)

Q는 최소우선순위 큐

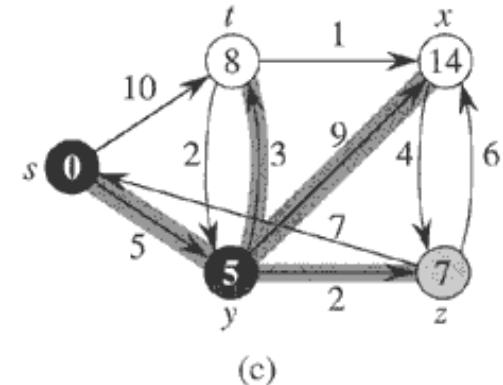
Dijkstra의 알고리즘



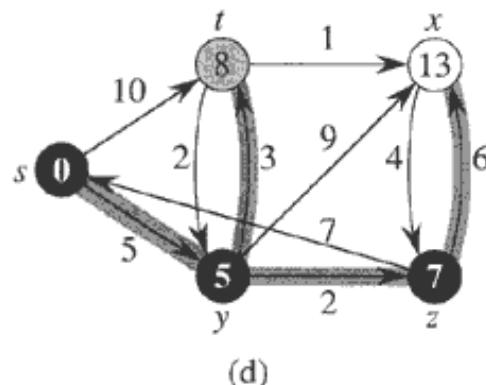
(a)



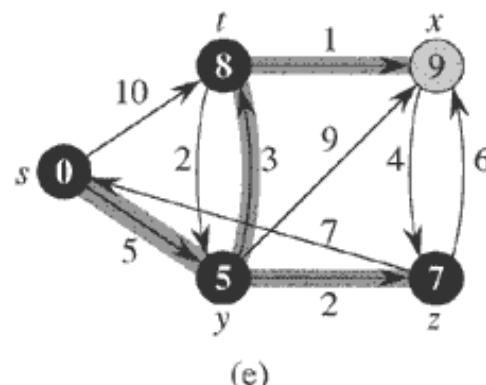
(b)



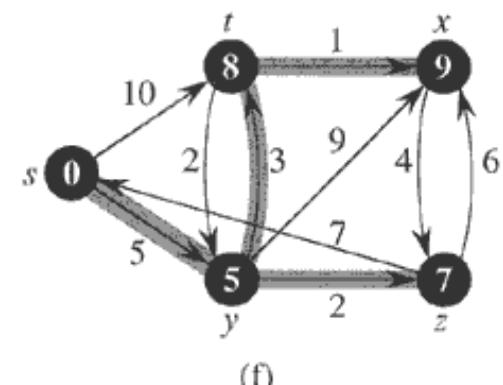
(c)



(d)



(e)



(f)

- Prim의 알고리즘과 동일함
- 우선순위 큐를 사용하지 않고 단순하게 구현할 경우 $O(n^2)$
- 이진힙을 우선순위 큐로 사용할 경우 $O(n \log_2 n + m \log_2 n)$
- Fibonacci Heap을 사용하면 $O(n \log_2 n + m)$ 에 구현가능

Floyd-Warshall Algorithm

- 가중치 방향 그래프 $G=(V, E)$, $V=\{1, 2, \dots, n\}$
- 모든 노드 쌍들간의 최단경로의 길이를 구함
- $d^k[i, j]$
 - 중간에 노드집합 $\{1, 2, \dots, k\}$ 에 속한 노드들만 거쳐서 노드 i 에서 j 까지 가는 최단경로의 길이

Floyd-Warshall Algorithm

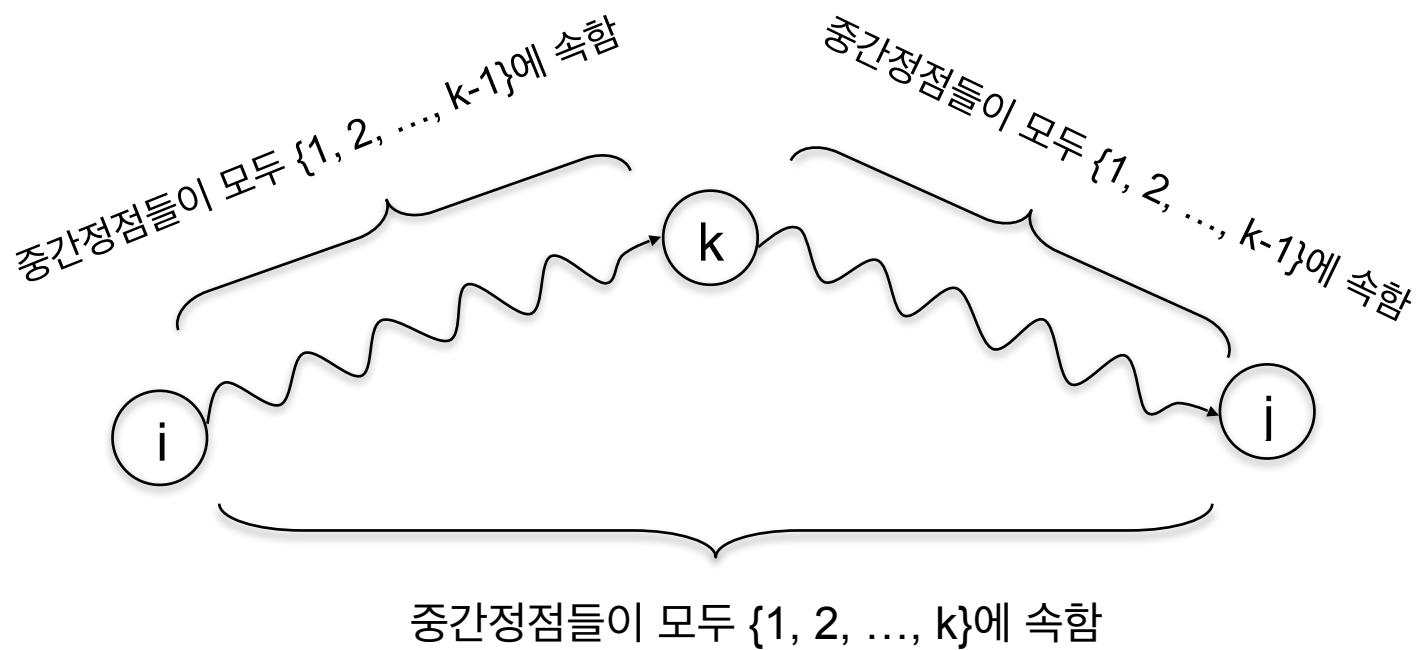
$$d^0[i, j] = \begin{cases} w_{ij}, & \text{if } (i, j) \in E \\ \infty, & \text{otherwise.} \end{cases}$$

$$d^k[i, j] = \min\{d^{k-1}[i, j], d^{k-1}[i, k] + d^{k-1}[k, j]\}$$

$$d^n[i, j] = \delta(i, j)$$

중간에 노드집합 $\{1, 2, \dots, k\}$ 에 속한 노드들만 거쳐서 노드 i 에서 j 까지 가는 최단경로는 두가지 경우가 있음: 노드 k 를 지나는 경우와 지나지 않는 경우

Floyd-Warshall Algorithm



Floyd-Warshall Algorithm

$$d^k[i, j] = \min\{d^{k-1}[i, j], d^{k-1}[i, k] + d^{k-1}[k, j]\}$$

```
FloydWarshall(G)
```

```
{
```

```
    for i ← 1 to n
```

```
        for j ← 1 to n
```

```
            d0[i, j] ← wij;
```

```
    for k ← 1 to n           ▷ 중간정점 집합 {1, 2, ..., k}
```

```
        for i ← 1 to n
```

```
            for j ← 1 to n
```

```
                dk[i, j] ← min{dk-1[i, j], dk-1[i, k]
```

```
+ dk-1[k, j]};
```

```
}
```

Floyd-Warshall Algorithm

```
FloydWarshall(G)
{
    for i ← 1 to n
        for j ← 1 to n
            d[i,j] ← wij;
    for k ← 1 to n           ▷ 중간정점 집합 {1,2,...,k}
        for i ← 1 to n
            for j ← 1 to n
                d[i,j] ← min{d[i,j], d[i,k]+d[k,j]};
}
```

```
Floydwarshall(G)
{
    for i ← 1 to n
        for j ← 1 to n
            d[i,j] ← wij;
             $\pi[i,j] \leftarrow \text{NIL};$ 

    for k ← 1 to n           ▷ 중간정점 집합 {1,2,...,k}
        for i ← 1 to n
            for j ← 1 to n
                if d[i,j] > d[i,k]+d[k,j] then
                    d[i,j] = d[i,k]+d[k,j];
                     $\pi[i,j] = k;$ 

}
```

경로 출력하기

s에서 t까지 가는 경로가 존재한다는 가정하에 최단경로상의 중간노드들(s와 t자신은 제외)을 출력함

```
Print-PATH(s, t,  $\pi$ )
```

```
{
```

```
    if  $\pi[s, t] = \text{NIL}$  then  
        return;  
    print-PATH(s,  $\pi[s, t]$ );  
    print( $\pi[s, t]$ );  
    print-PATH( $\pi[s, t]$ , t);
```

```
}
```