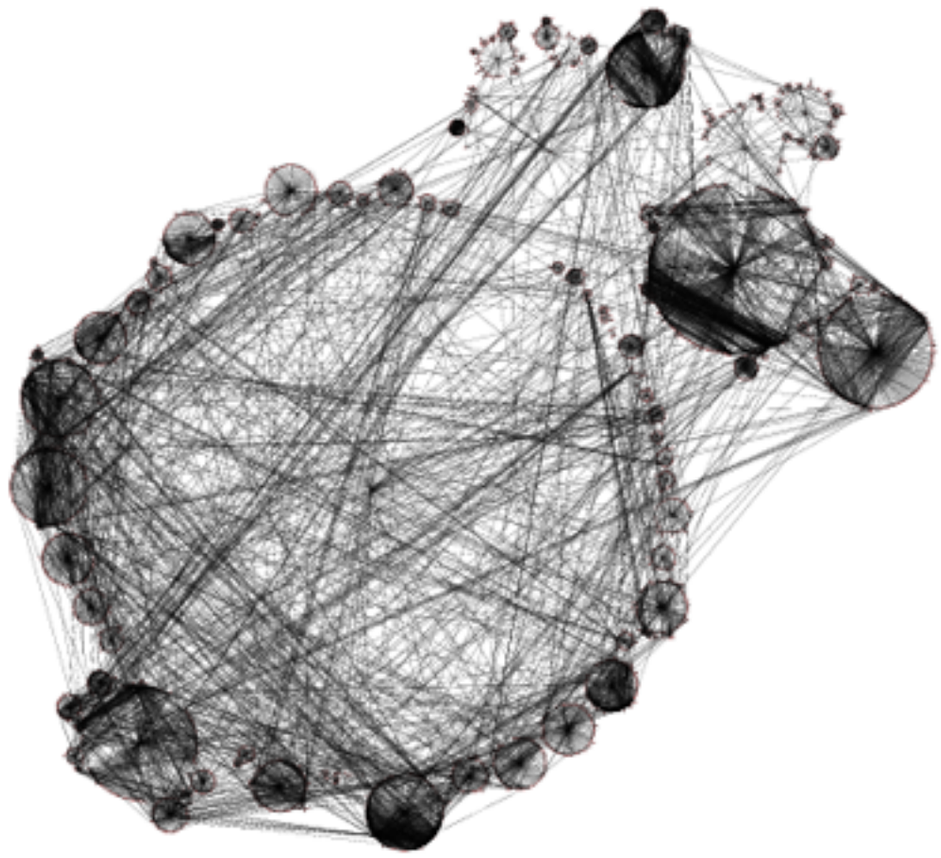


제6장

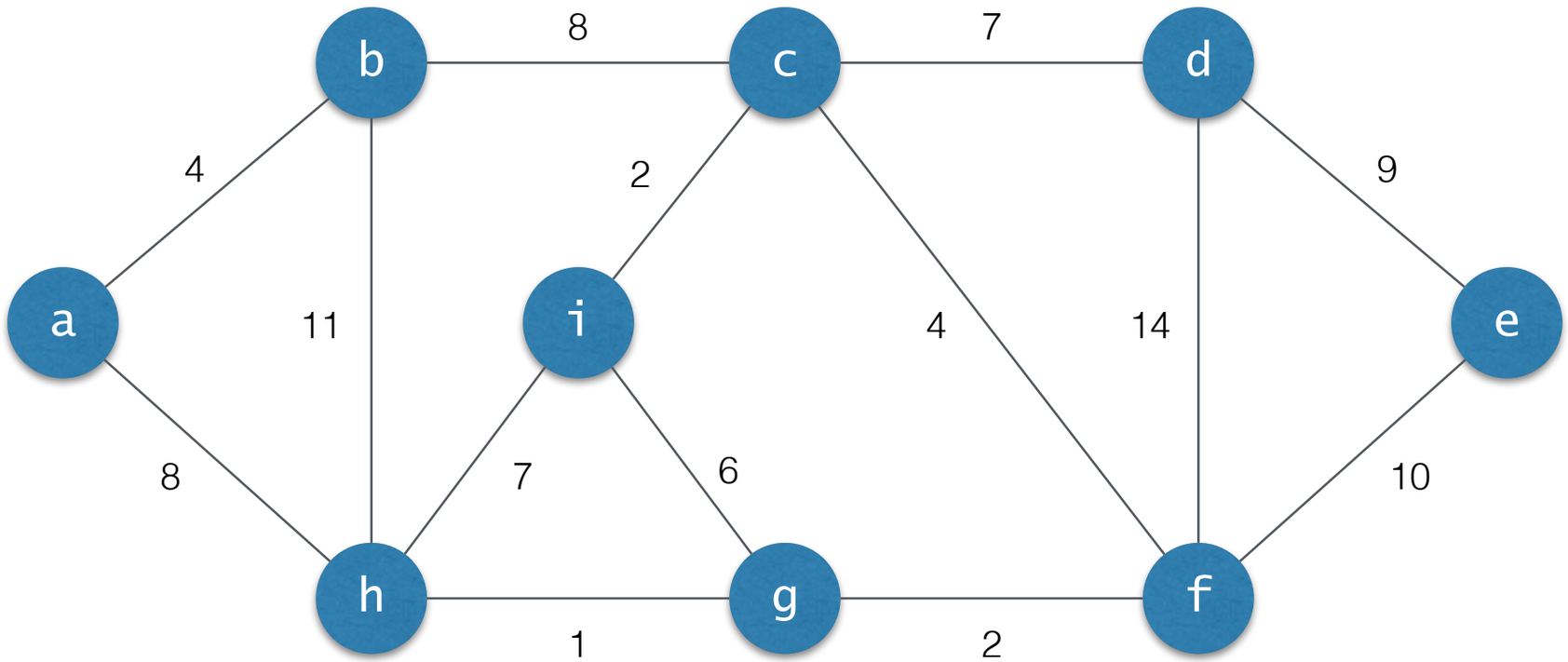
그래프 알고리즘 II

최소신장트리

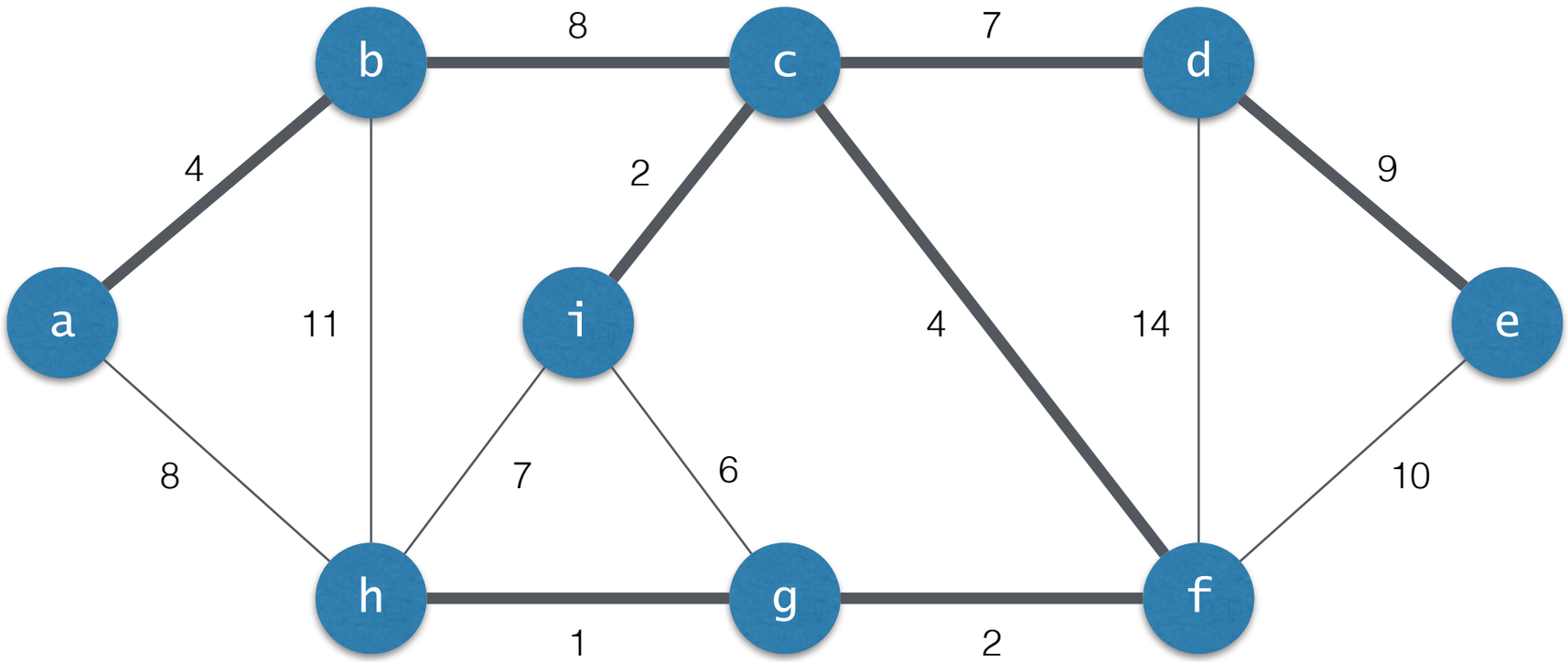
Minimum Spanning Tree

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

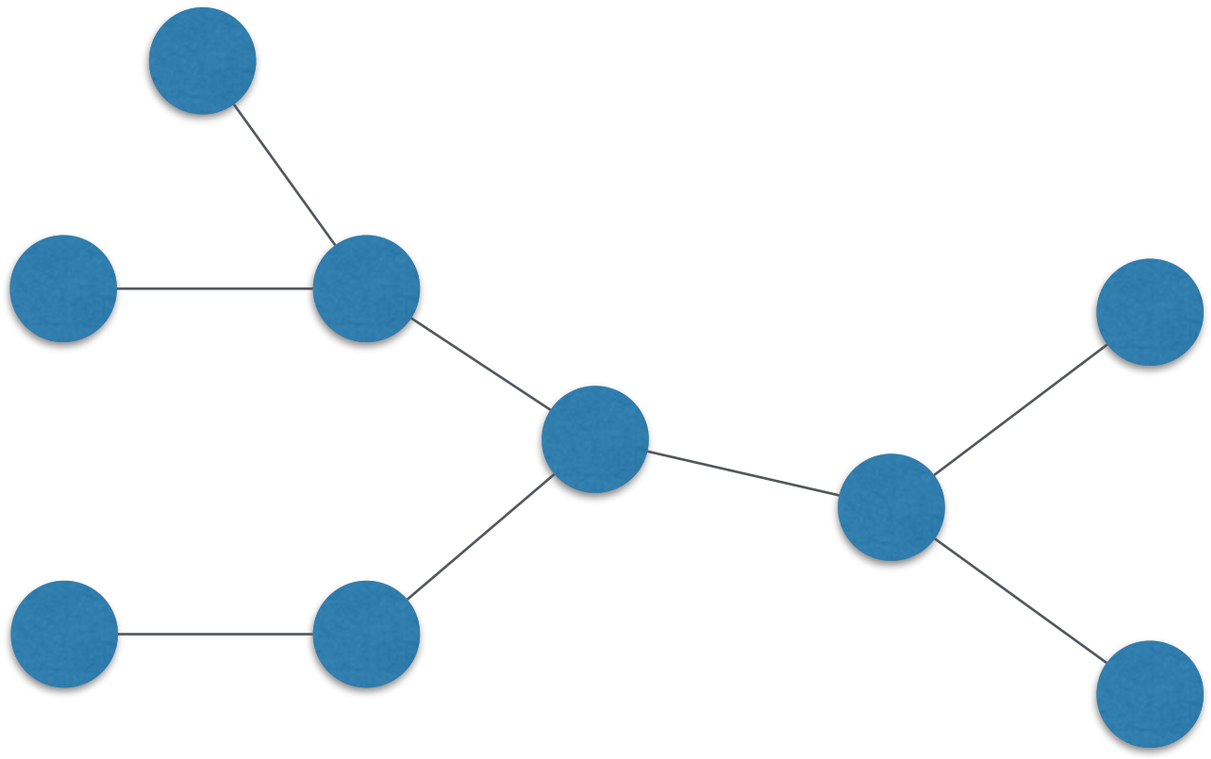
예: (b,c) 대신 (a,h)

최소비용 신장 트리(MST)

- 무방향 가중치 그래프 G=(V,E)
- 각 에지 (u,v)∈E 에 대해서 가중치 w(u,v)
- 문제: 다음과 같은 조건을 만족하는 에지들의 부분집합 T⊆E를 찾아라.
  1. T에 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 n개인 트리는 항상 n-1개의 에지를 가짐

Generic MST 알고리즘

- 어떤 MST의 부분집합 A에 대해서 Au{(u,v)}도 역시 어떤 MST의 부분집합이 될 경우 에지 (u,v)는 A에 대해서 안전하다(safe)고 한다.

- Generic MST 알고리즘:

  1. 처음에는 A=∅이다.
  2. 집합 A에 대해서 안전한 에지를 하나 찾은 후 이것을 A에 더한다.
  3. 에지의 개수가 n-1개가 될 때까지 2번을 반복한다.

Generic MST 알고리즘

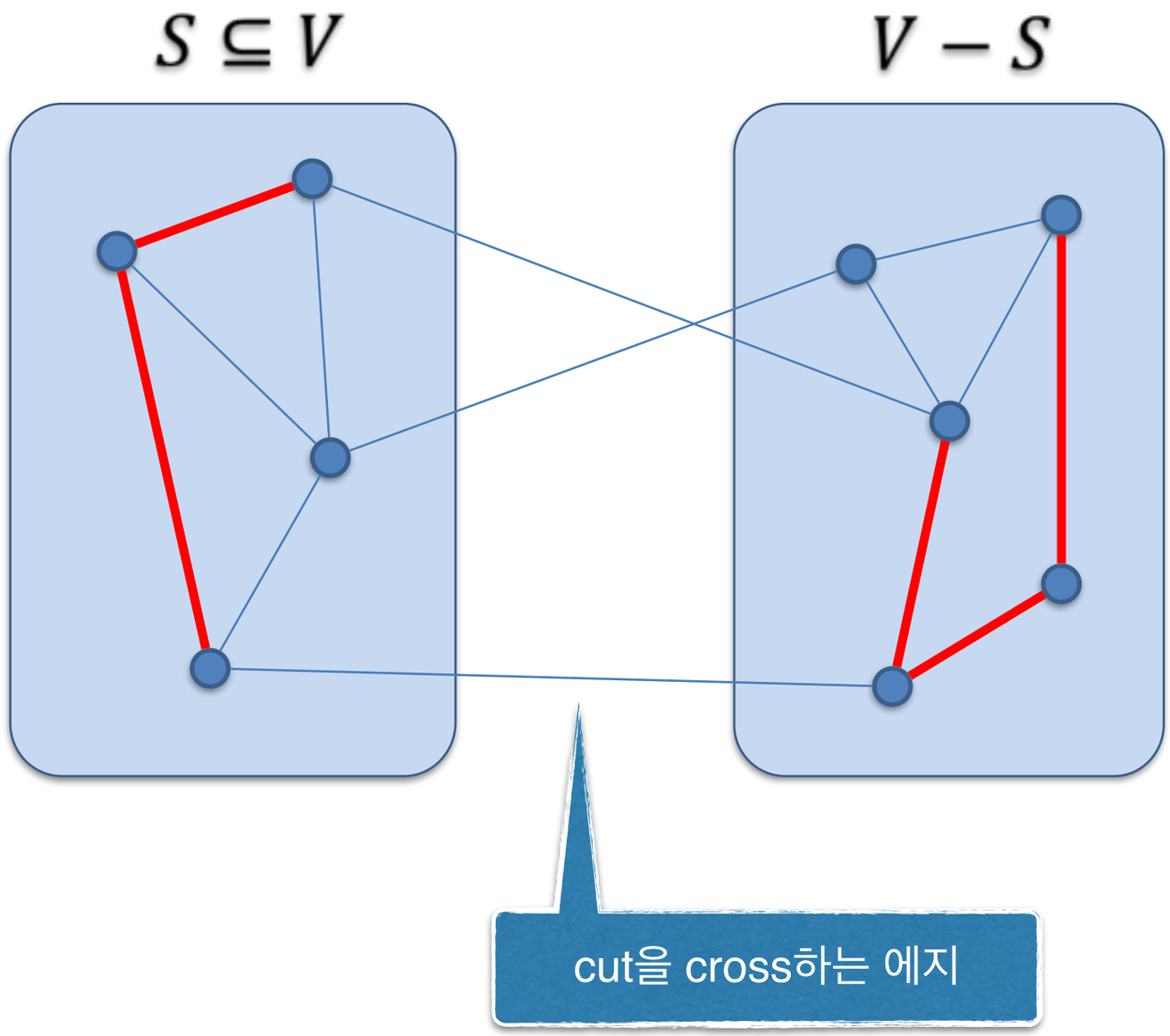
```
GENERIC-MST(G, w)
1   A ← ∅
2   while A does not form a spanning tree
3       do find an edge (u, v) that is safe for A
4          A ← A ∪ {(u, v)}
5   return A
```

안전한 에지 찾기

- 그래프의 정점들을 두 개의 집합 S와 V-S로 분할한 것을 컷(cut) (S,V-S)라고 부른다.
- 에지 (u,v)에 대해서 u∈S이고 v∈V-S일 때 에지 (u,v)는 컷 (S,V-S)를 cross한다고 말한다.
- 에지들의 부분집합 A에 속한 어떤 에지고 컷 (S,V-S)를 cross하지 않을 때 컷 (S,V-S)는 A를 존중한다(respect)고 말한다.

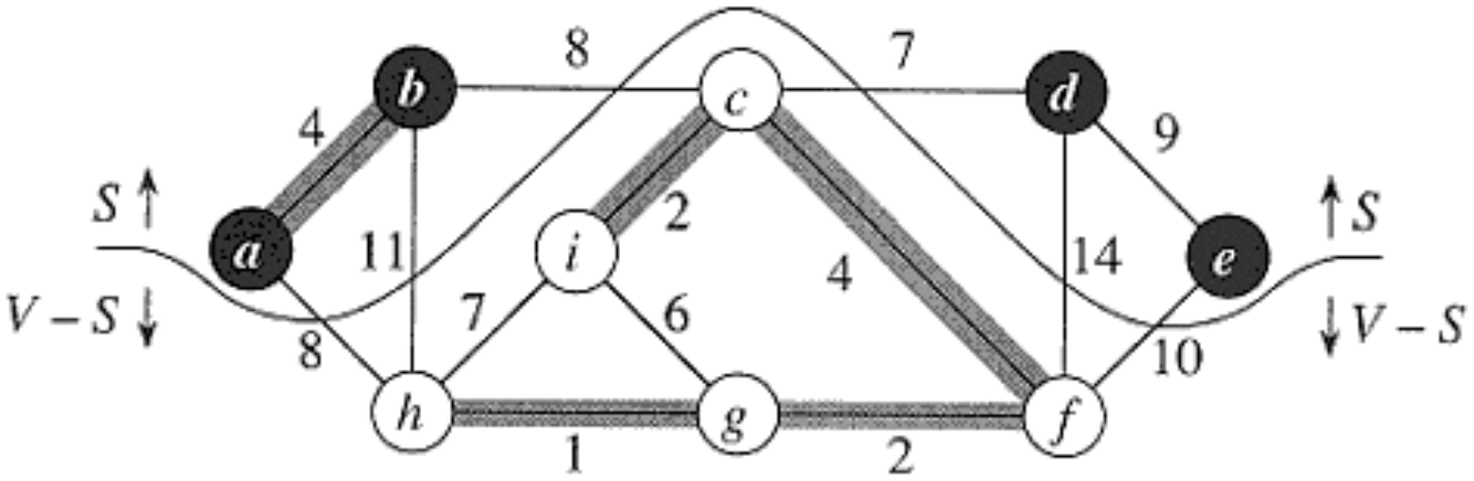
안전한 에지 찾기

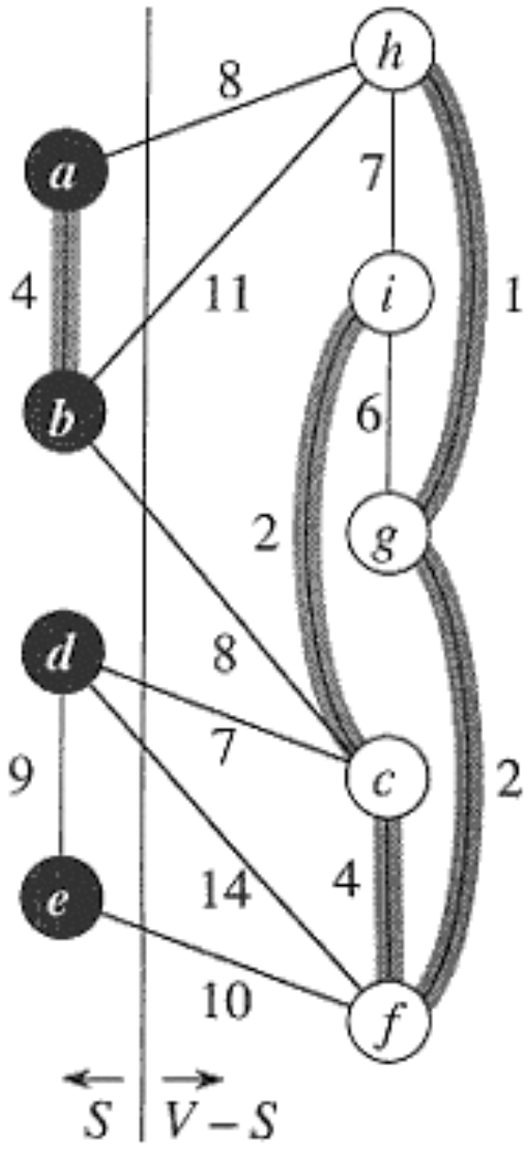
노드들을 두 그룹으로 분할한 것을 cut이라고 부름



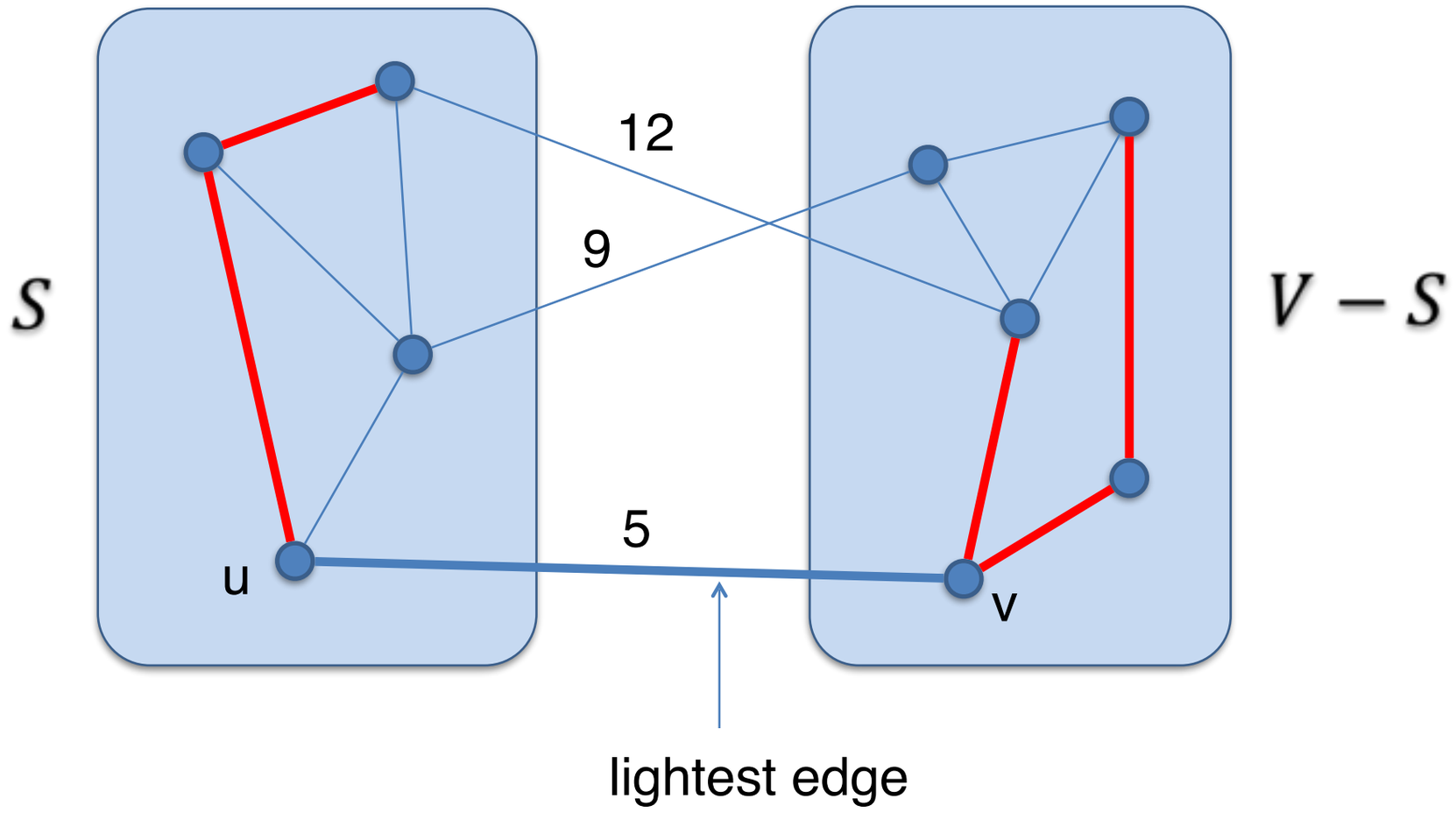
에지 집합 A에 속한 어떤 에지도 cut을 cross하지 않을 때 이 컷은 집합 A를 존중한다고 말함.
(예: 빨간 에지들을 A)

안전한 에지 찾기


S
V − S
a
b
c
d
e
f
g
h
i
4
8
7
9
11
14
10
2
4
7
6
8
1
2


h
a
i
b
g
d
c
e
f
S
V − S
8
7
4
11
1
6
2
8
9
7
2
14
4
10

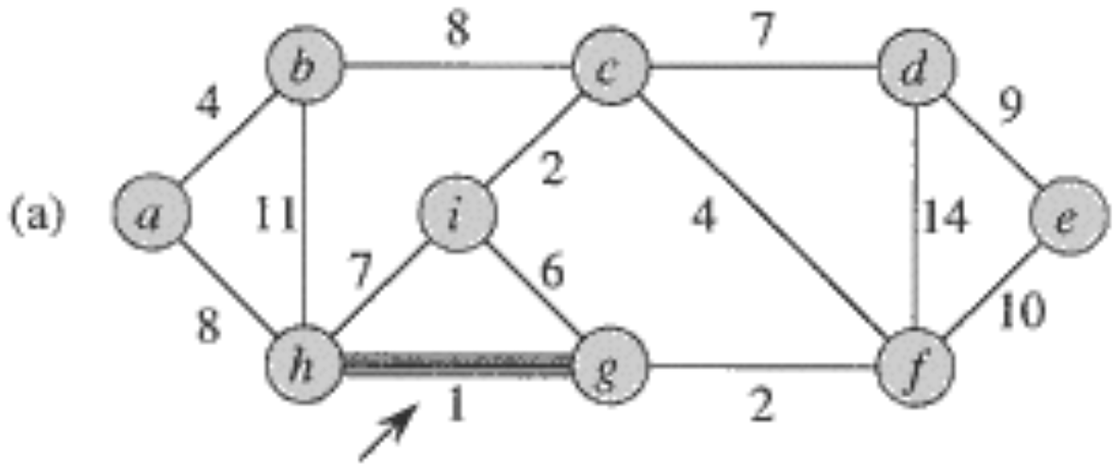
- A가 어떤 MST의 부분집합이고, (S,V-S)는 A를 존중하는 컷이라고 하자. 이 컷을 cross하는 에지들 중 가장 가중치가 작은 에지 (u,v)는 A에 대해서 안전하다.

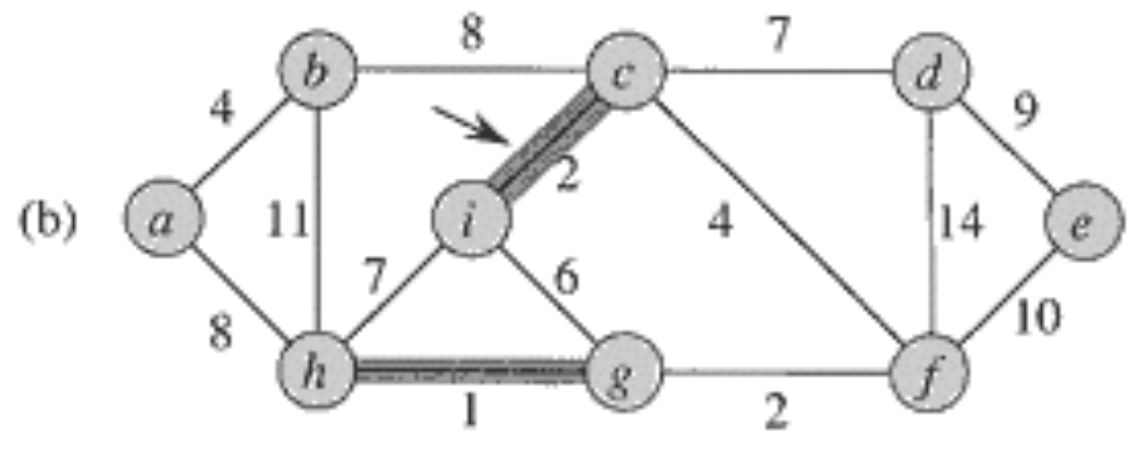


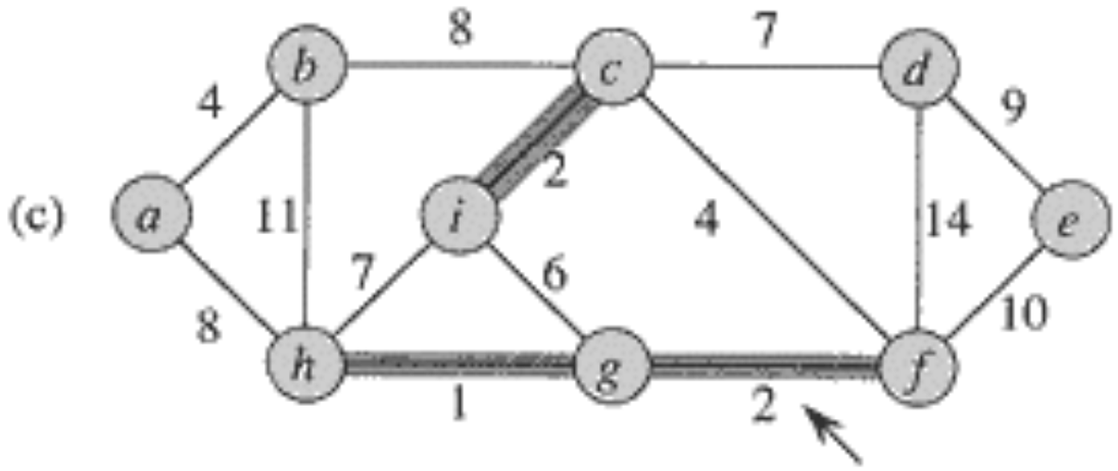
Kruskal의 알고리즘

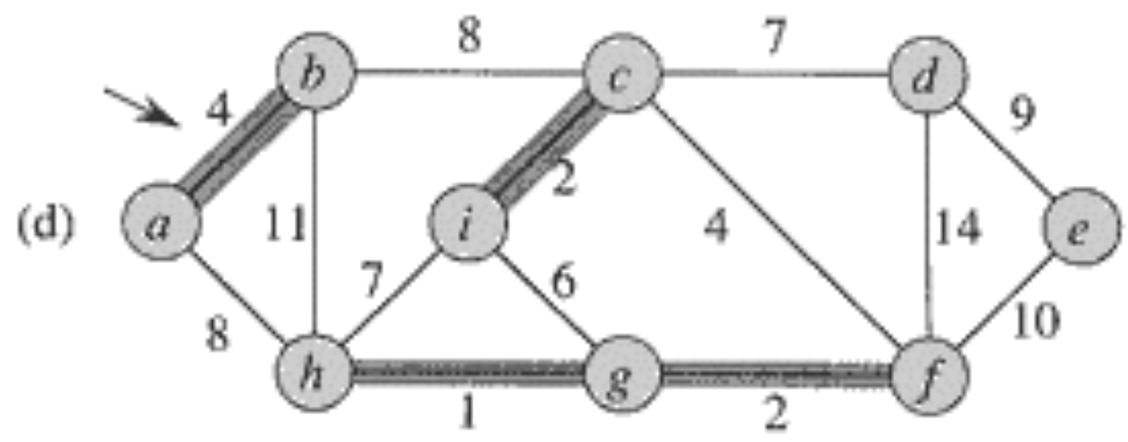
- 에지들을 가중치의 오름차순으로 정렬한다.
- 에지들을 그 순서대로 하나씩 선택해간다. 단, 이미 선택된 에지들과 사이클(cycle)을 형성하면 선택하지 않는다.
- n-1개의 에지가 선택되면 종료한다.

Kruskal의 알고리즘



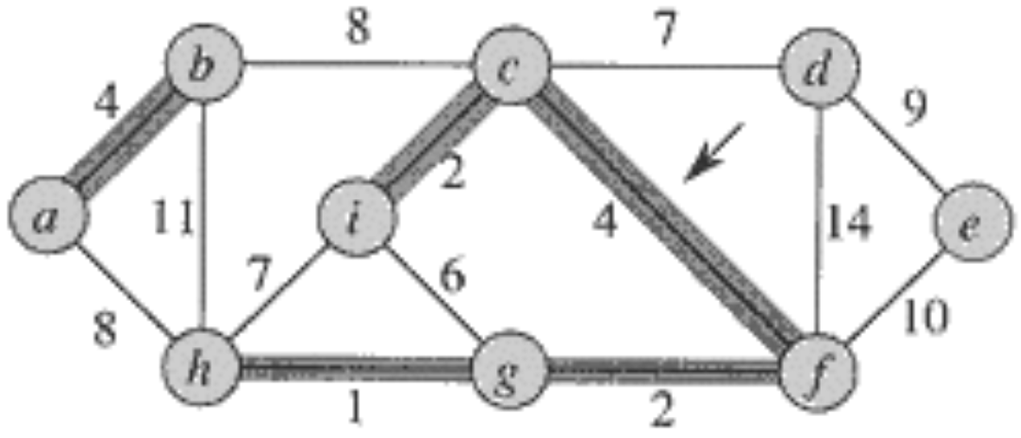




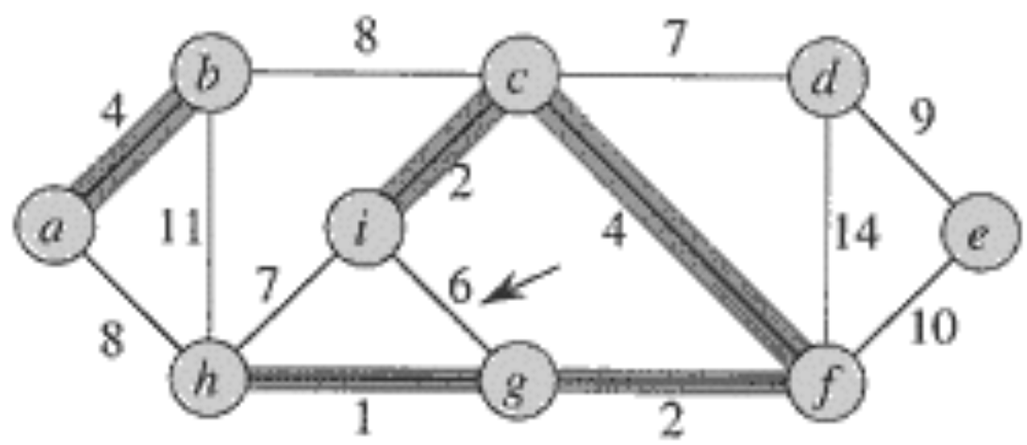


Kruskal의 알고리즘 (계속)

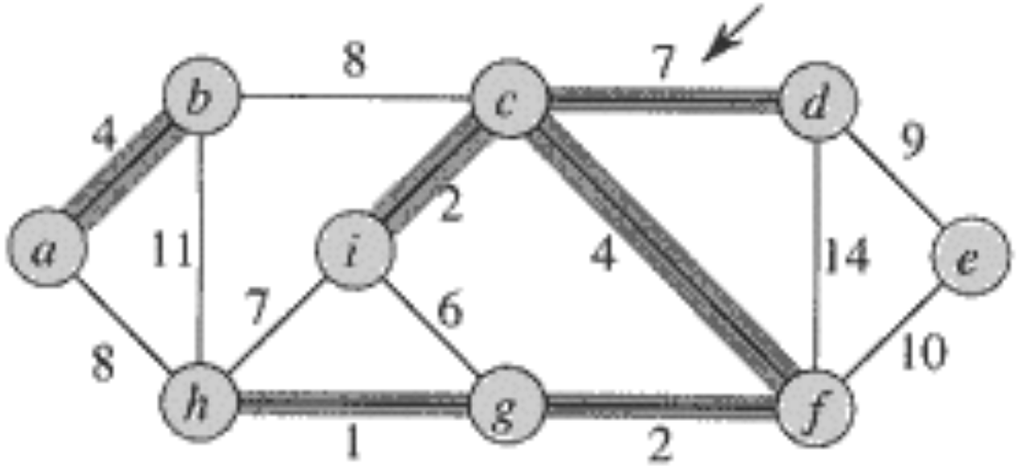
(e)



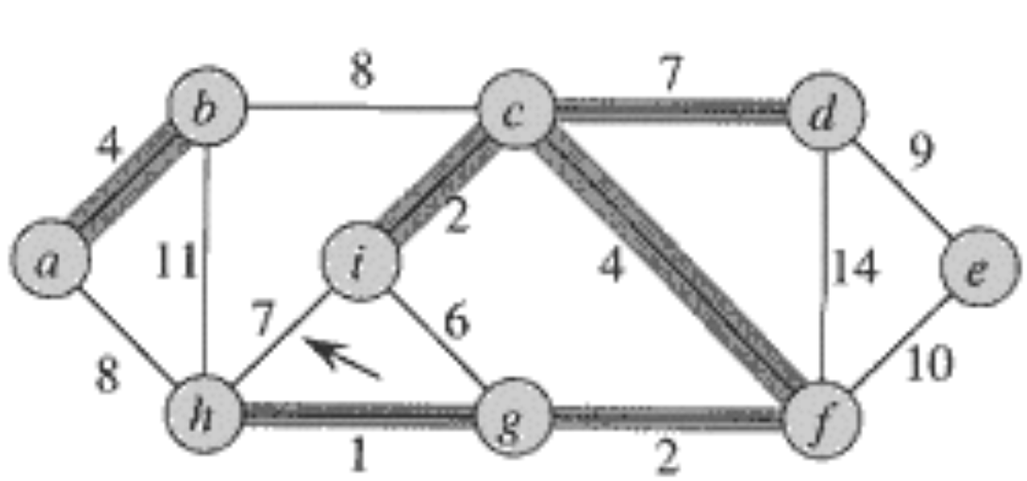
(f)



(g)

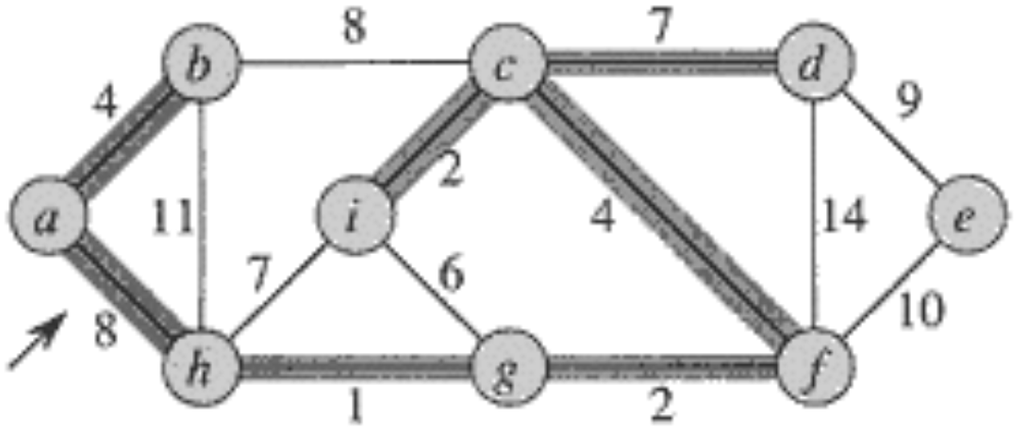


(h)

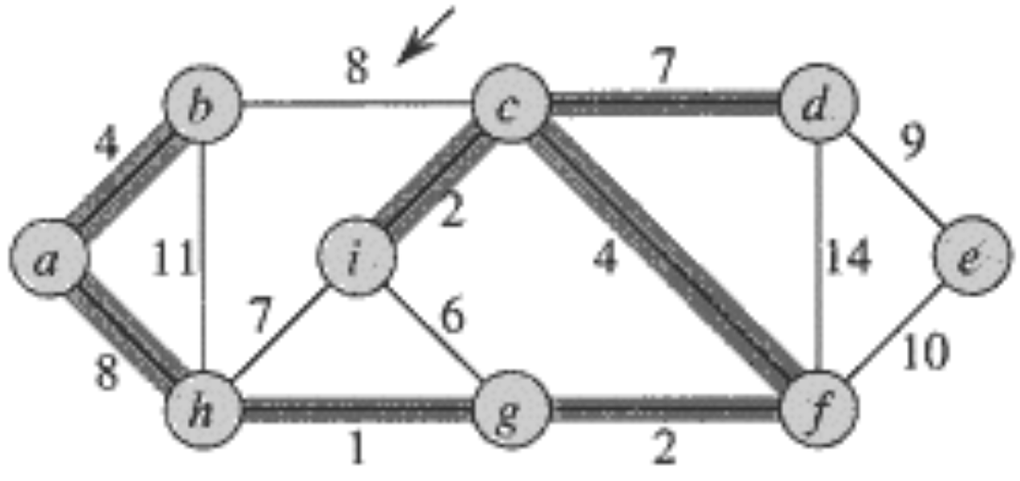


Kruskal의 알고리즘 (계속)

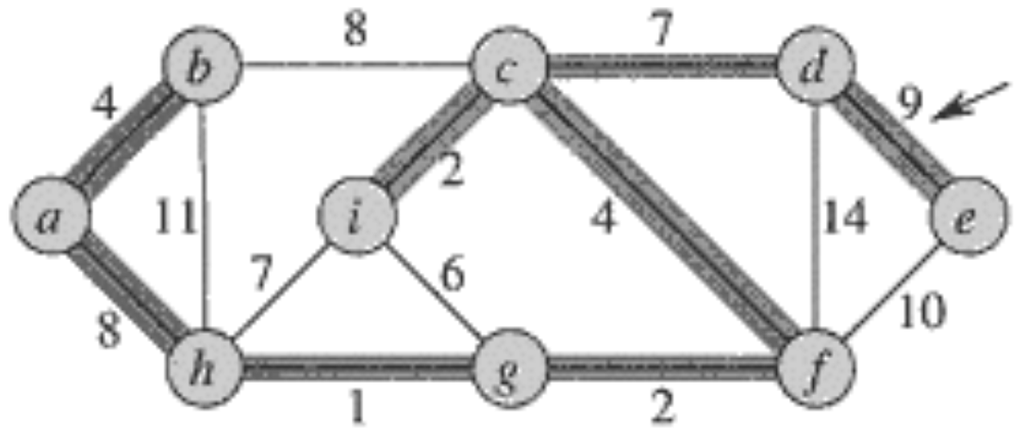
(i)



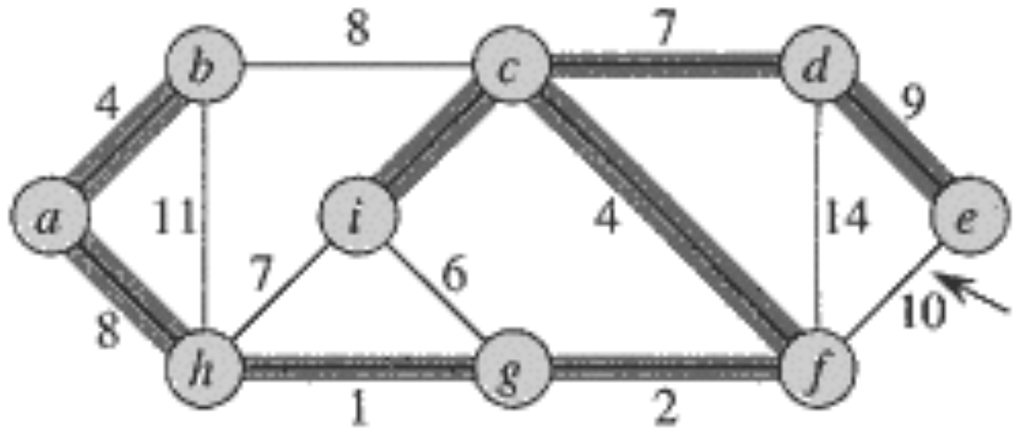
(j)



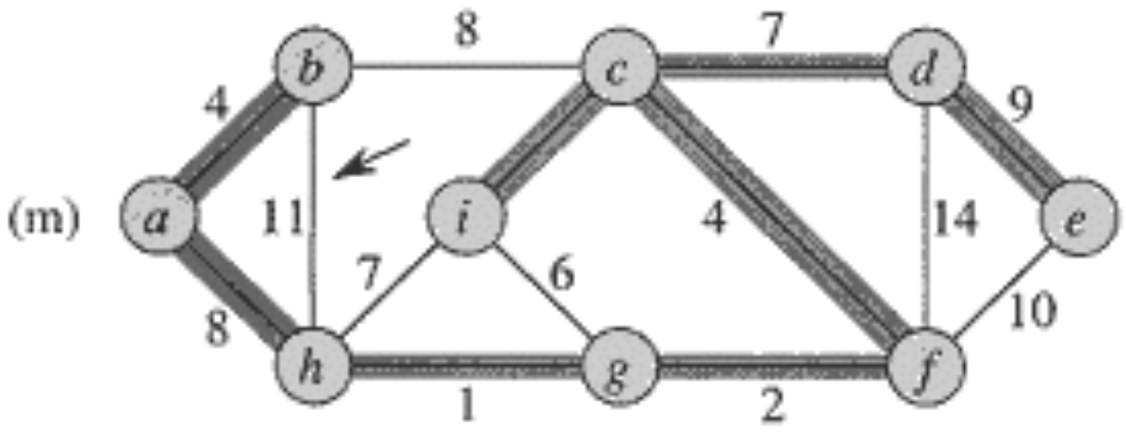
(k)

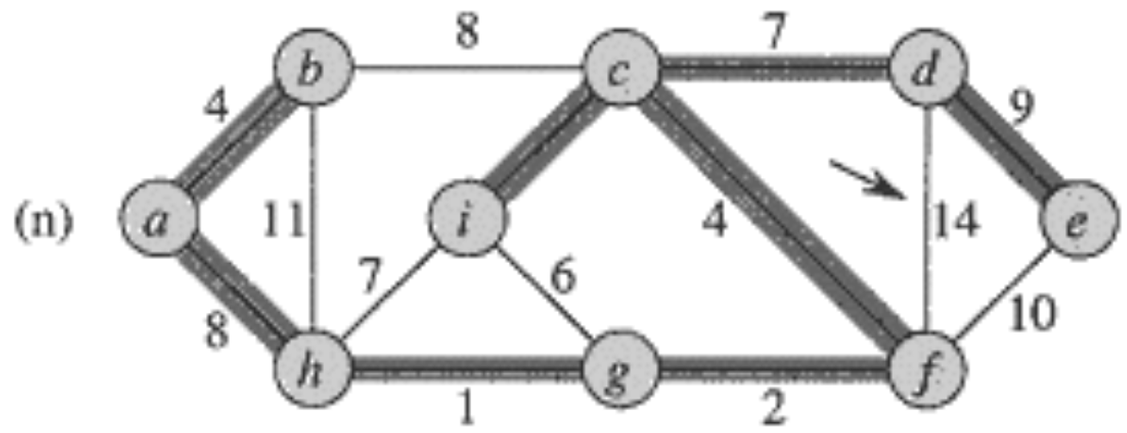


(l)



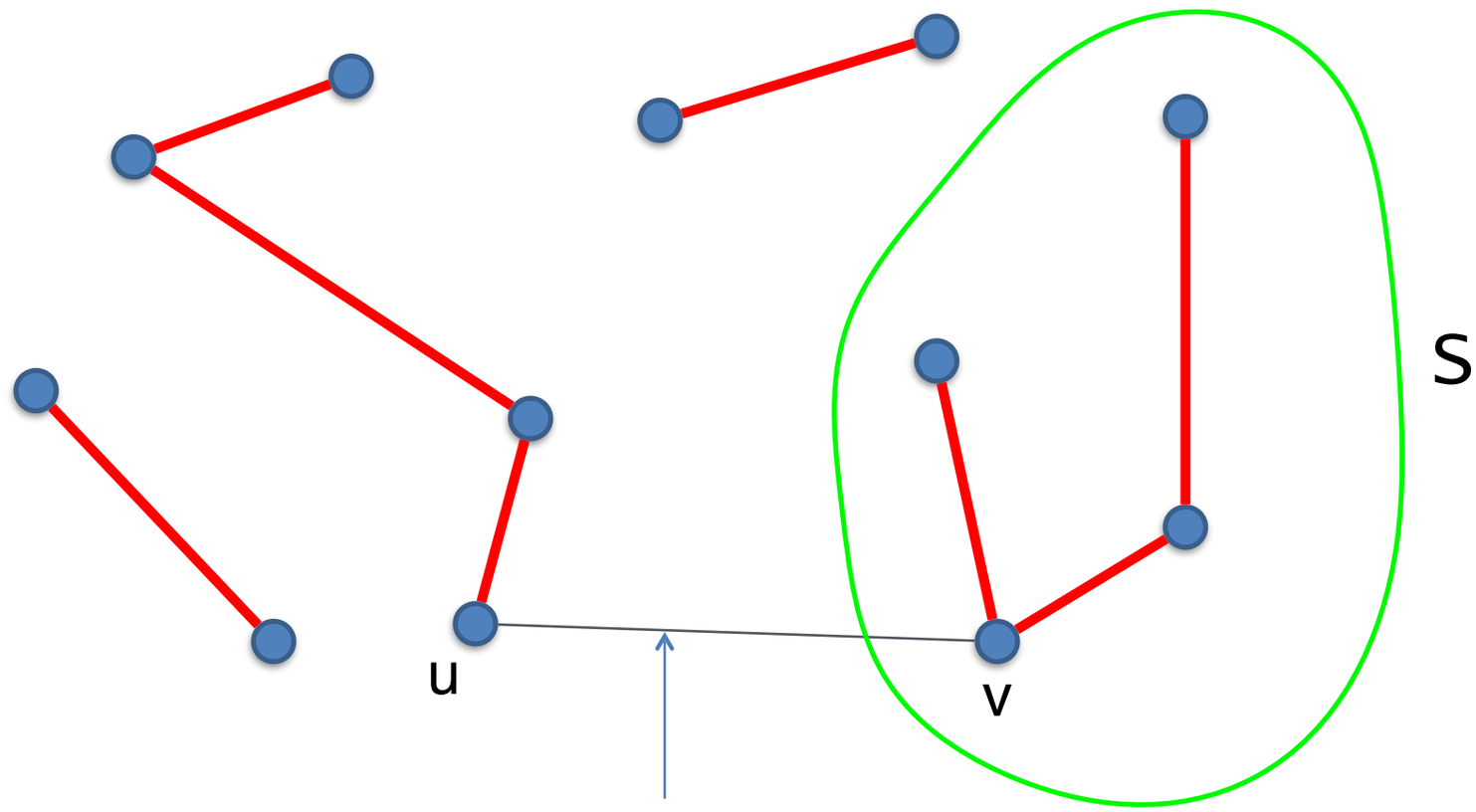
Kruskal의 알고리즘 (계속)





왜 MST가 찾아지는가?

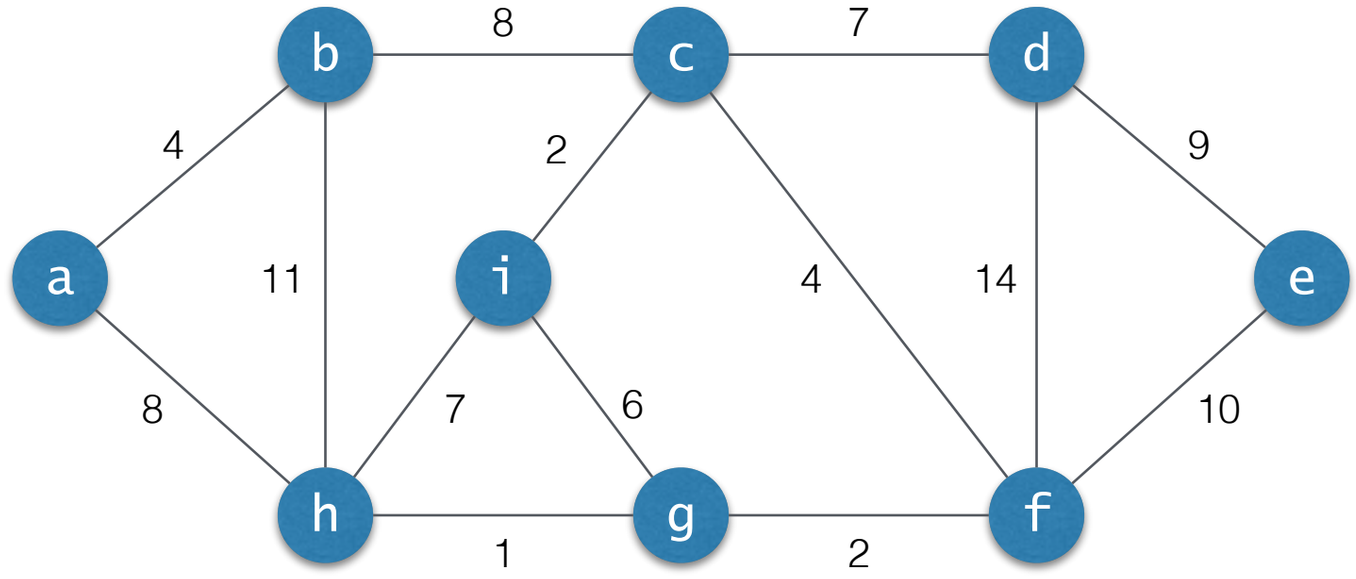
- Kruskal의 알고리즘의 임의의 한 단계를 생각해보자.
- A를 현재까지 알고리즘이 선택한 에지의 집합이라고 하고, A를 포함하는 MST가 존재한다고 가정하자.



cycle을 만들지 않는 lightest edge

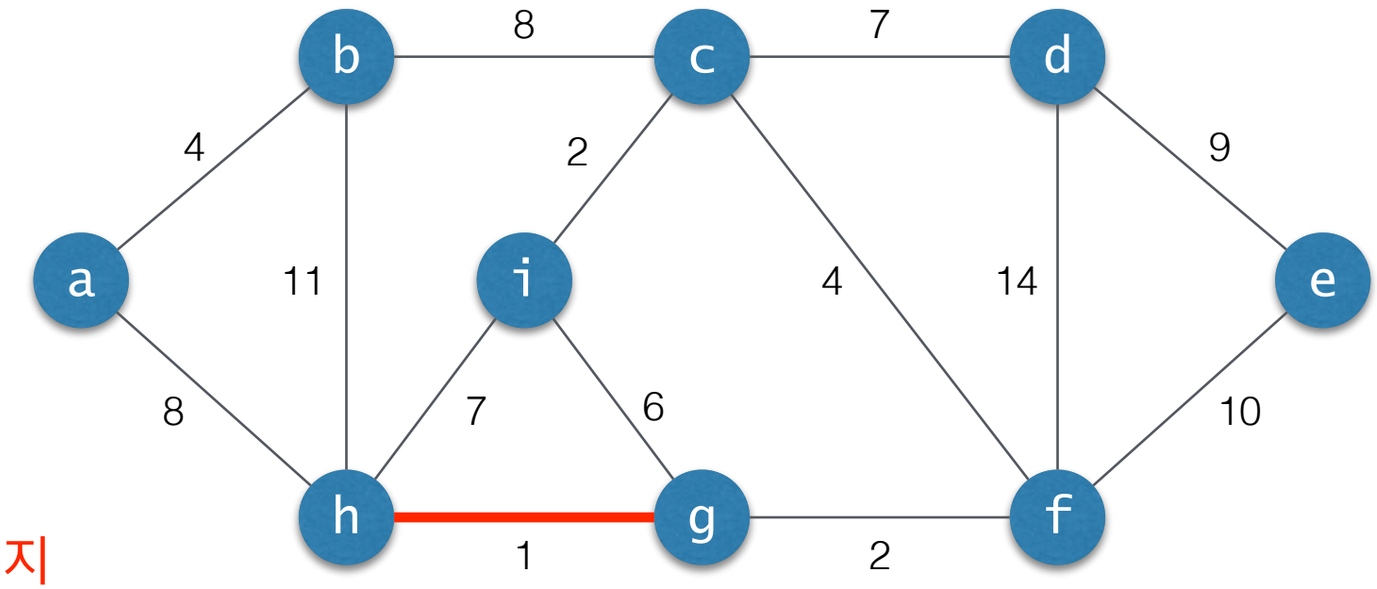
사이클 검사

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



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

사이클 검사



1. 가중치가 최소인 에지 (h,g)를 고려한다.

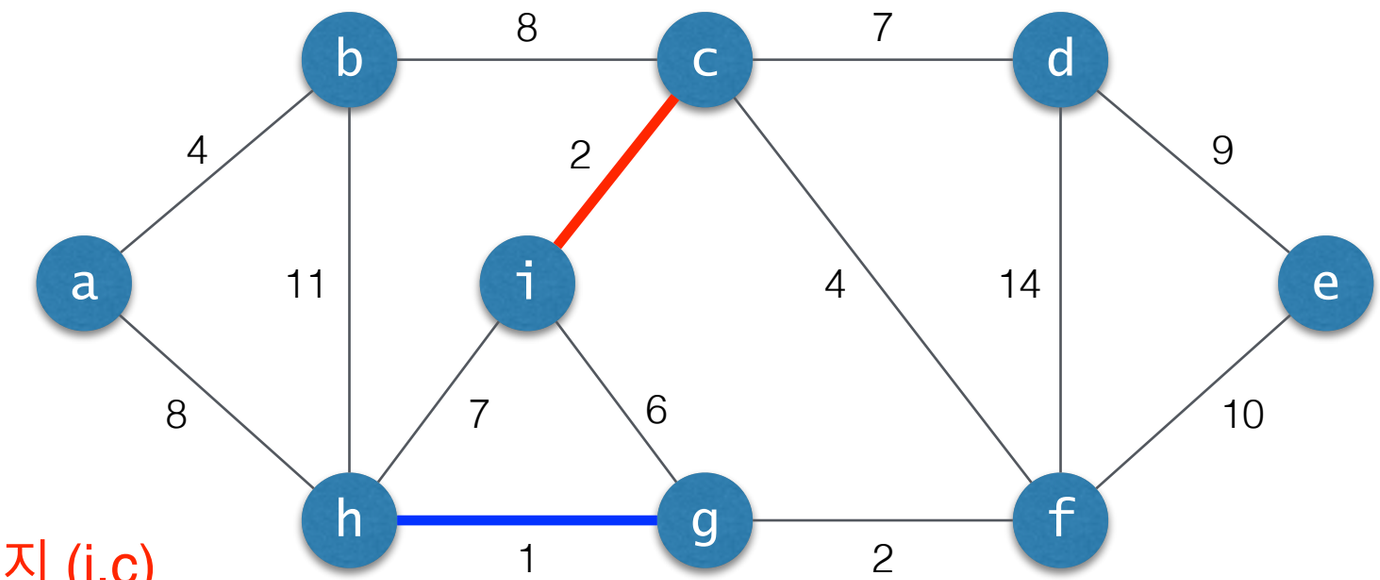
```
{a} {b} {c} {d} {e} {f} {g} {h} {i}
```

2. g와 h가 서로 다른 집합에 속함

3. 에지 (g,h)를 선택하고, g와 h가 속한 집합을 합집합하여 하나의 집합으로 만듬

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

사이클 검사



1. 가중치가 최소인 에지 (i,c)를 고려한다.

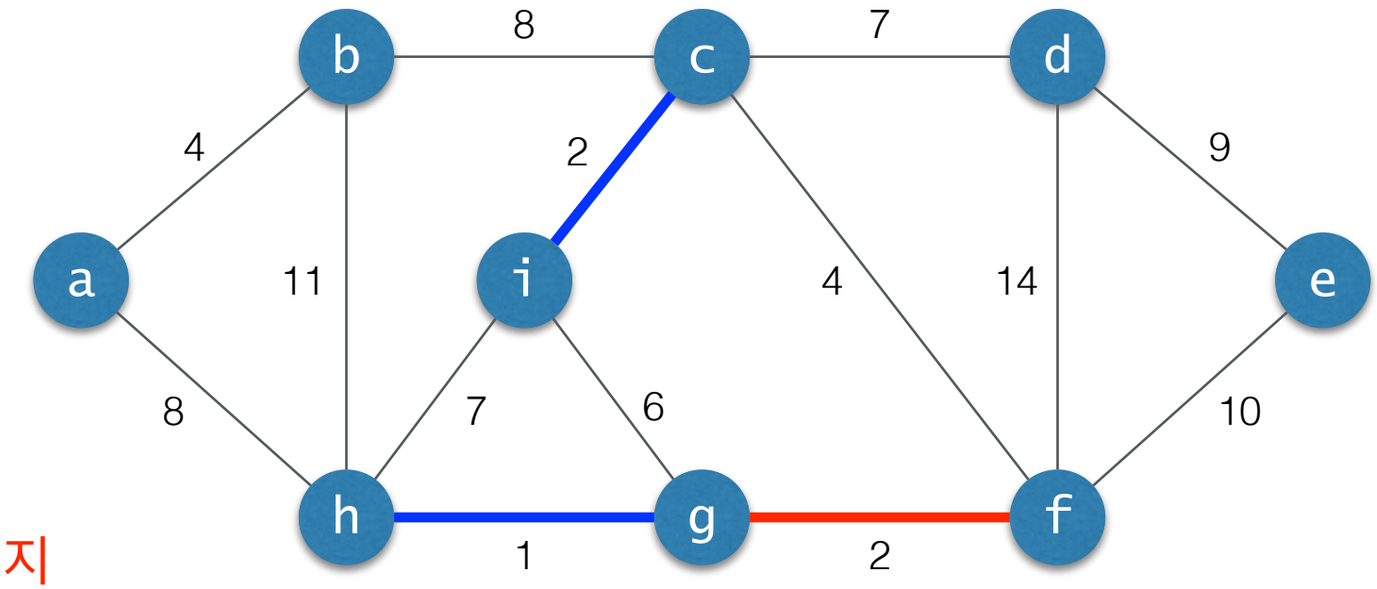
{a} {b} {c} {d} {e} {f} {g,h} {i}

2. i와 c가 서로 다른 집합에 속함

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

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

사이클 검사



1. 가중치가 최소인 에지 (g,f)를 고려한다.

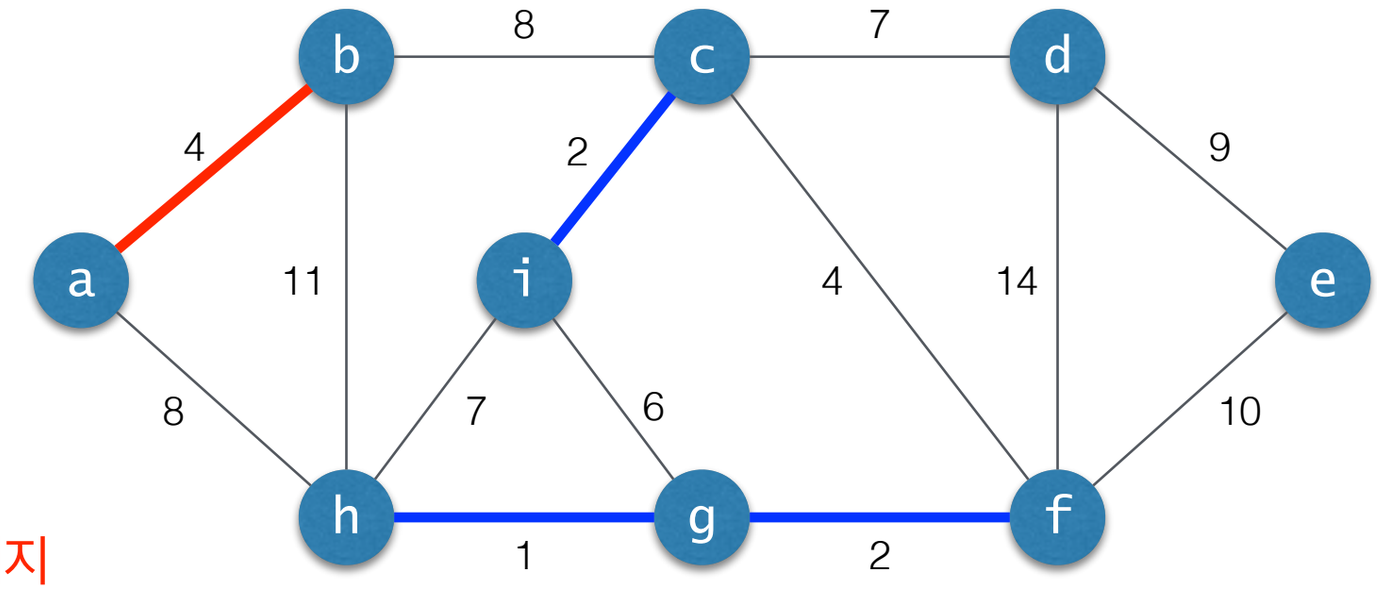
{a} {b} {c,i} {d} {e} {f} {g,h}

2. g와 f가 서로 다른 집합에 속함

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

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

사이클 검사



1. 가중치가 최소인 에지 (a,b)를 고려한다.

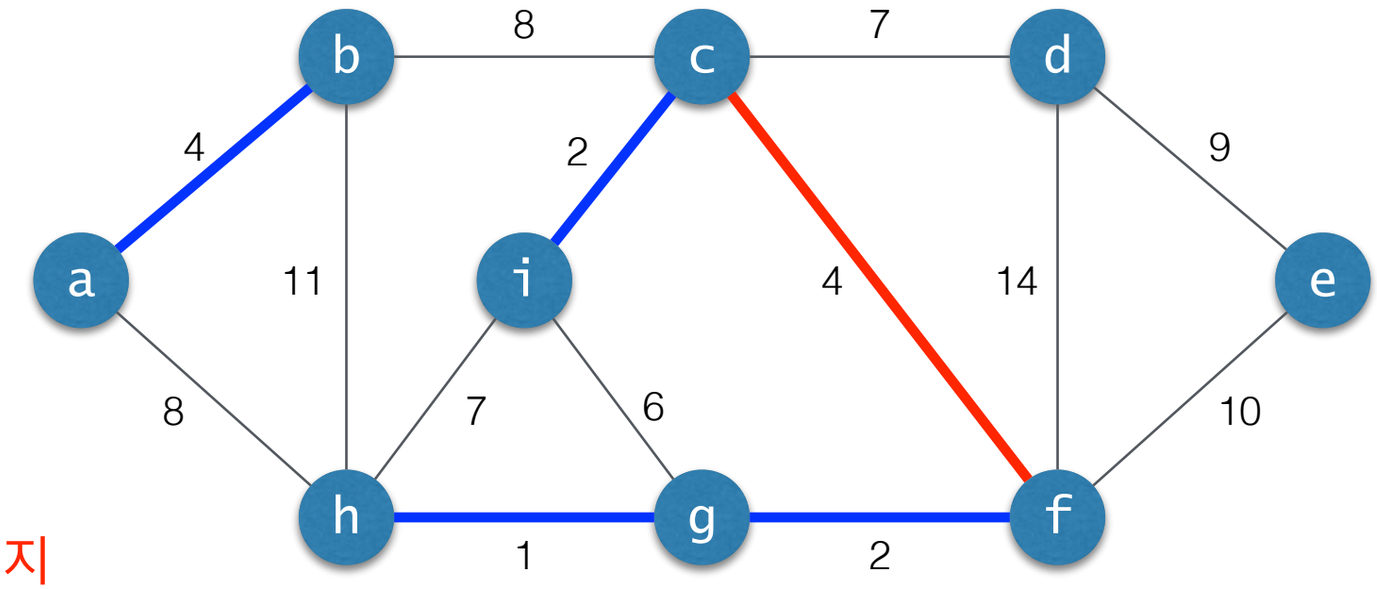
{a} {b} {c,i} {d} {e} {f,g,h}

2. a와 b가 서로 다른 집합에 속함

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

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

사이클 검사



1. 가중치가 최소인 에지 (c,f)를 고려한다.

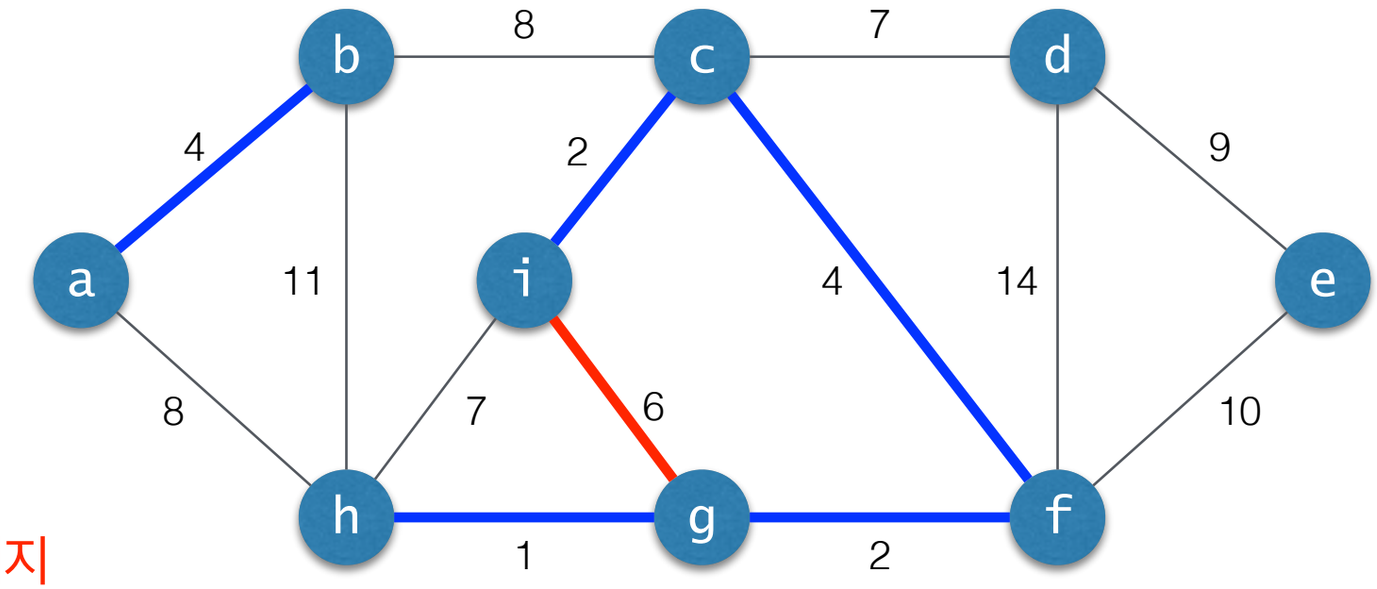
{a,b} {c,i} {d} {e} {f,g,h}

2. c와 f가 서로 다른 집합에 속함

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

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

사이클 검사



1. 가중치가 최소인 에지 (i,g)를 고려한다.

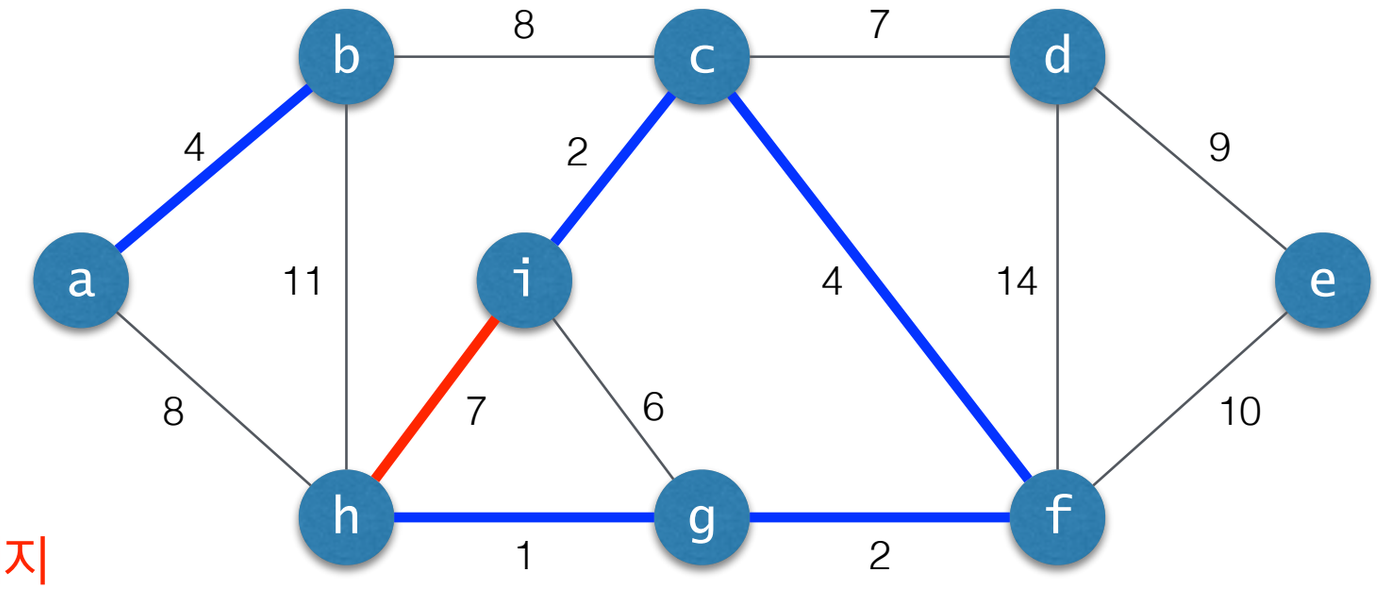
```
{a,b} {c,f,g,h,i} {d} {e}
```

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

3. 에지 (i,g)를 선택하지 않는다.

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

사이클 검사



1. 가중치가 최소인 에지 (i,h)를 고려한다.

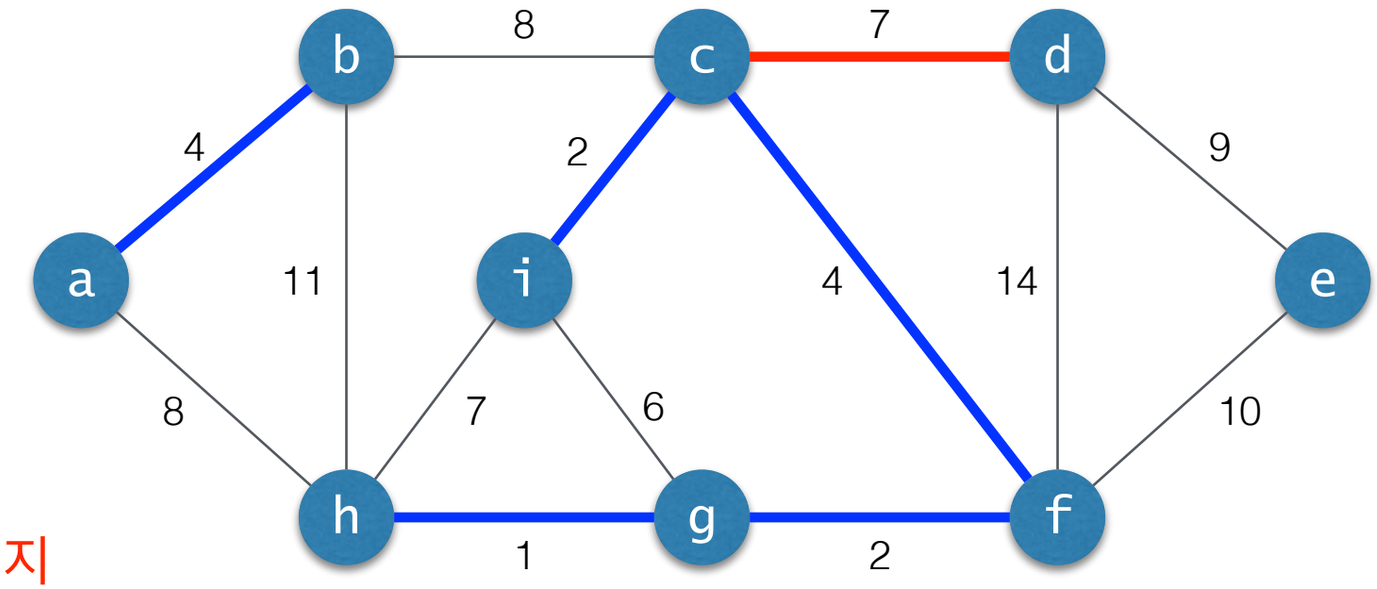
{a,b} {c,f,g,h,i} {d} {e}

2. i와 h는 이미 같은 집합에 속함. 즉 i와 h를 연결하면 사이클이 생김

3. 에지 (i,h)를 선택하지 않는다.

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

사이클 검사



1. 가중치가 최소인 에지 (c,d)를 고려한다.

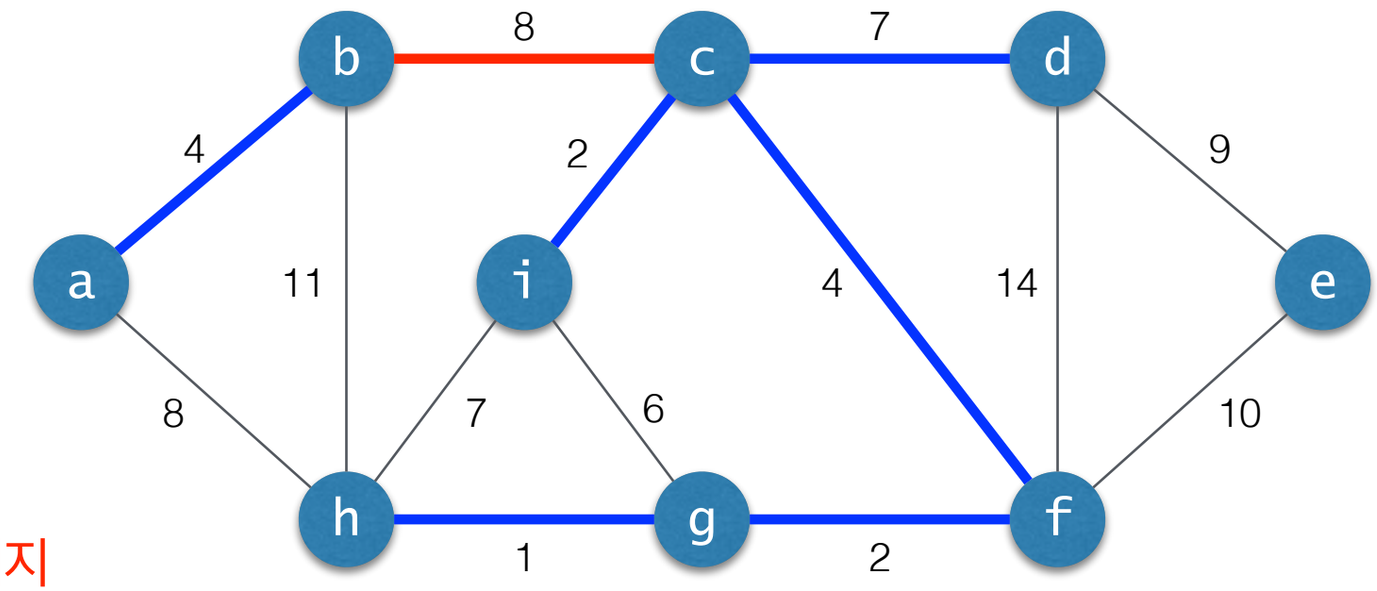
{a,b} {c,f,g,h,i} {d} {e}

2. c와 d가 서로 다른 집합에 속함

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

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

사이클 검사



1. 가중치가 최소인 에지 (b,c)를 고려한다.

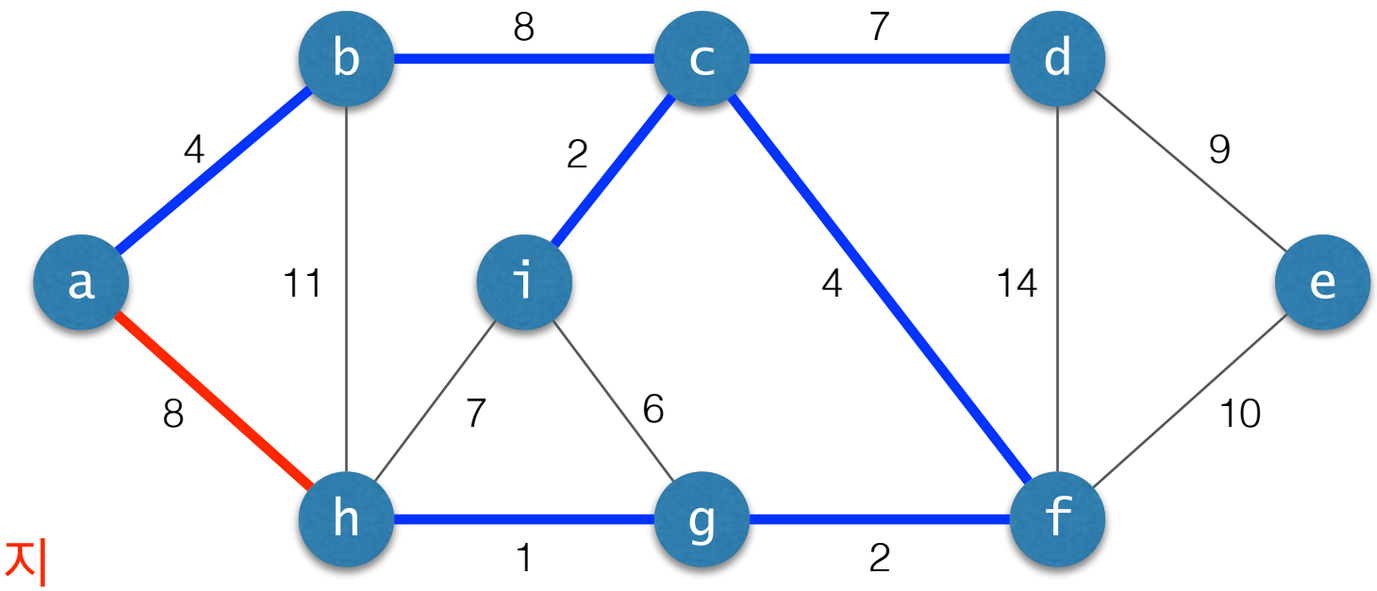
{a,b} {c,f,g,h,i,d} {e}

2. b와 c가 서로 다른 집합에 속함

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

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

사이클 검사



1. 가중치가 최소인 에지 (a,h)를 고려한다.

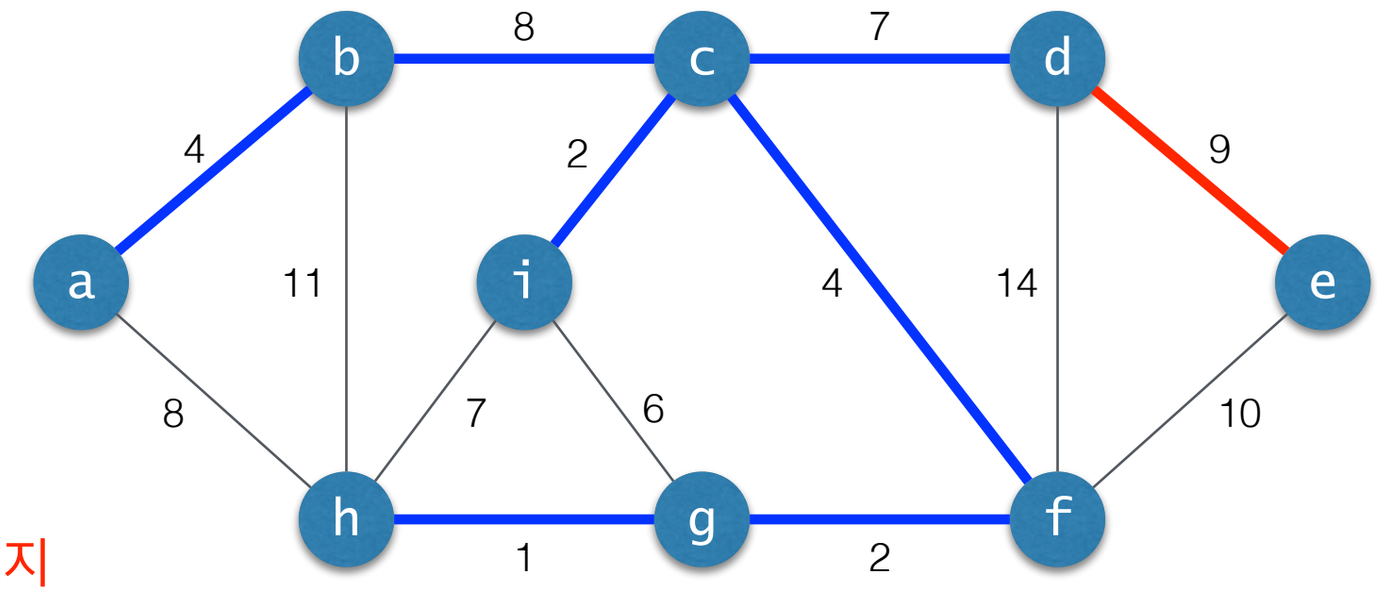
{a,b,c,f,g,h,i,d} {e}

2. a와 h는 이미 동일한 집합에 속함

3. 에지 (a,h)를 선택하지 않는다.

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

사이클 검사



1. 가중치가 최소인 에지 (d,e)를 고려한다.

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

2. d와 e는 서로 다른 집합에 속함

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

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

4. n-1개의 에지가 선택되었으므로 종료한다.

Kruskal의 알고리즘

```
MST-KRUSKAL(G, w)
1   A ← ∅
2   for each vertex v ∈ 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) ∈ E, taken in nondecreasing order by weight
6       do if FIND-SET(u) ≠ FIND-SET(v)
7          then A ← A ∪ {(u, v)}
8               UNION(u, v)
9   return A
```

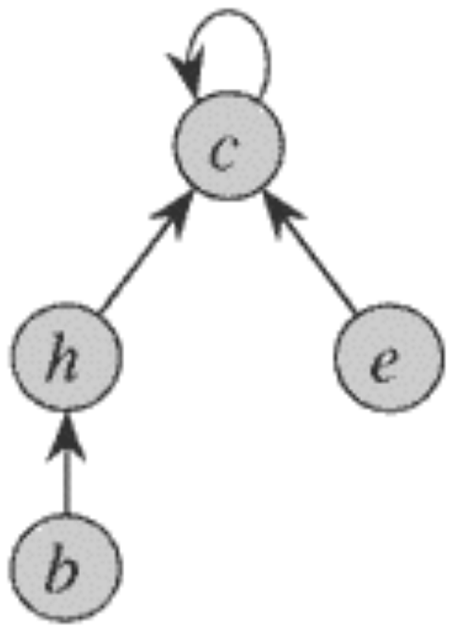
각각의 노드들을 유일한 원소로
가지는 집합들을 만들어라.

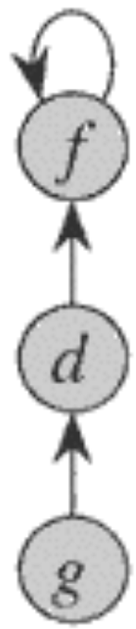
노드 v가 속한 집합을
찾아라

u와 v가 속한
두 집합을 하나로 합친다.

서로소인 집합들의 표현

- 각 집합을 하나의 트리로 표현
- 예: 2개의 집합



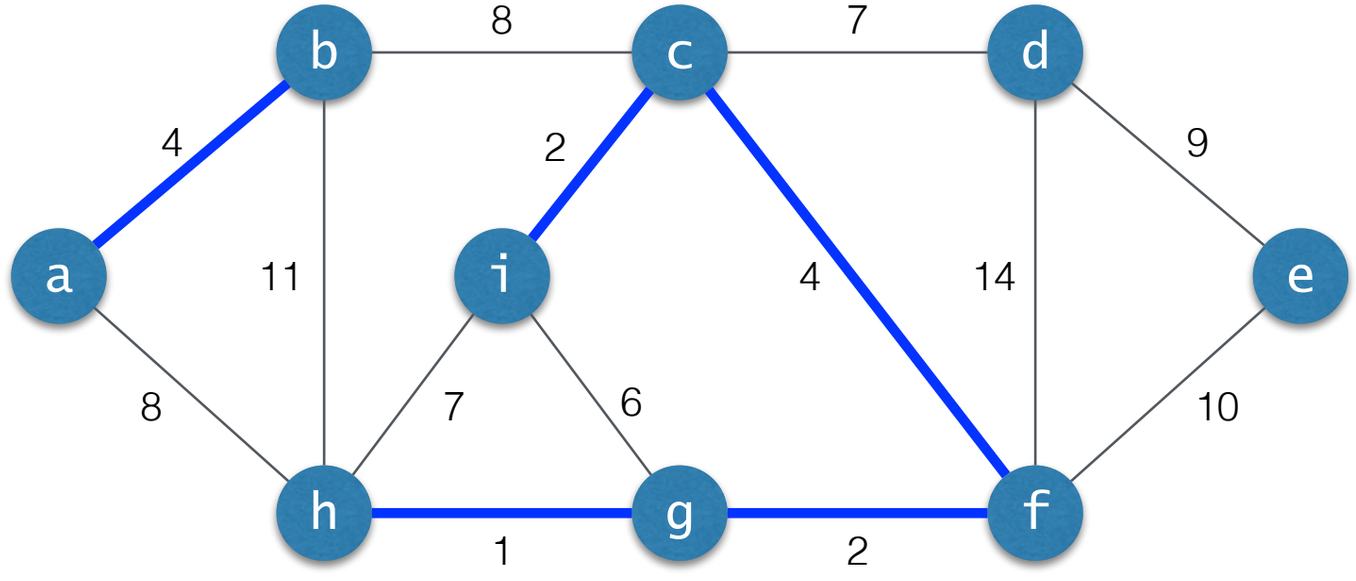


{b,c,e,h}, {d,f,g}

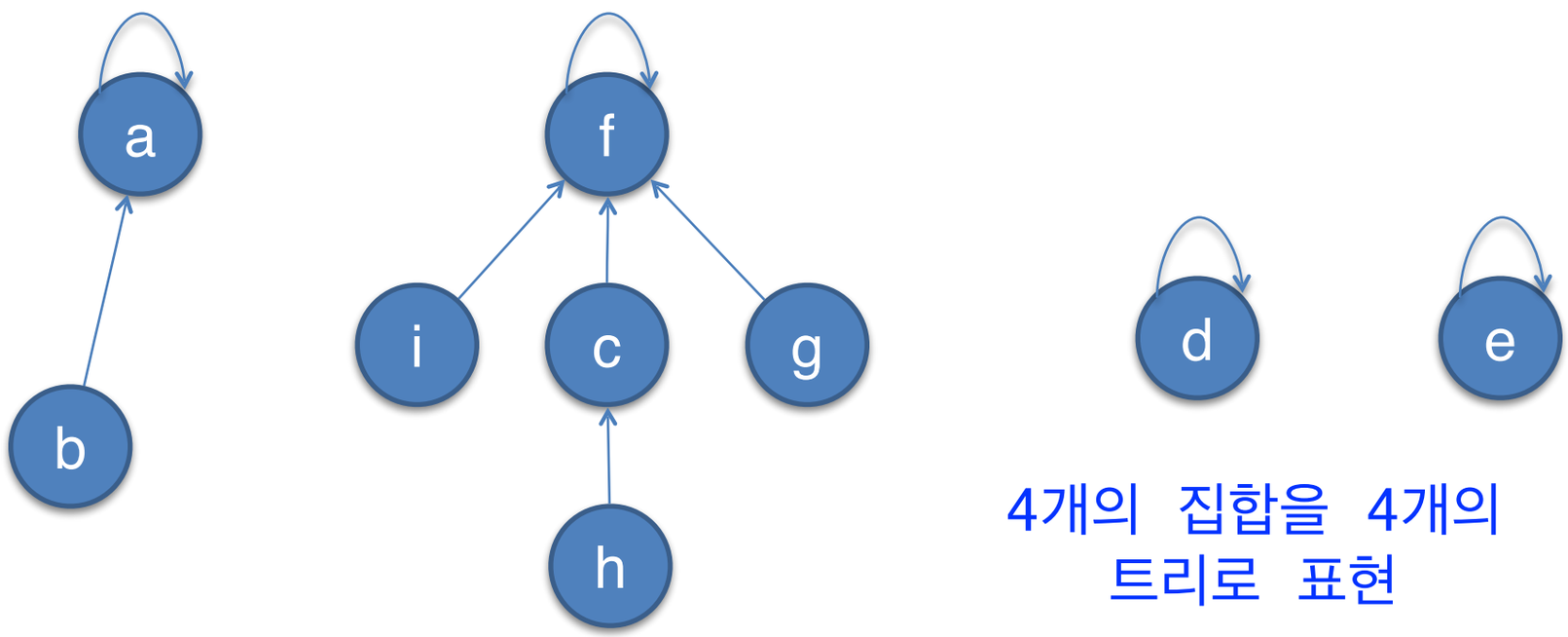
집합의 각 원소들이 트리의 노드가 됨. 누가 루트이고 누가 누구의 부모이든 상관없음.

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

서로소인 집합들의 표현



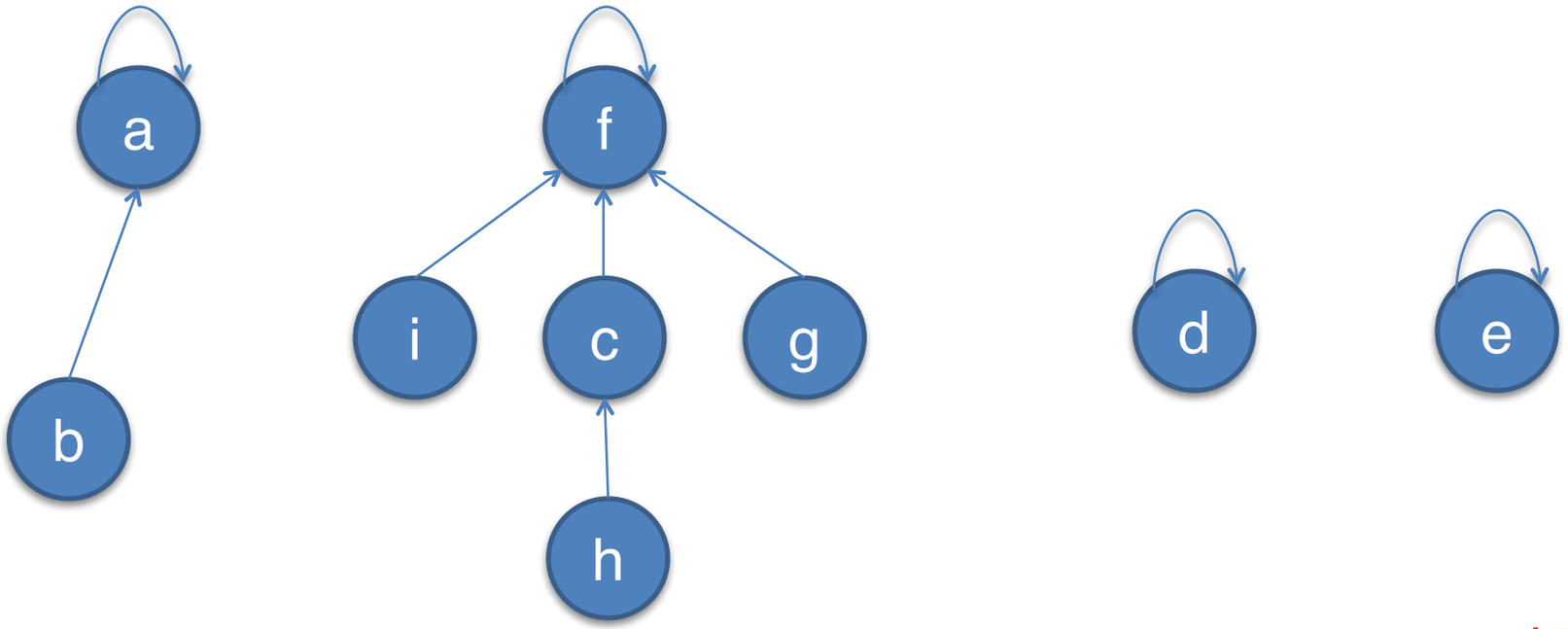
{a,b} {c,i,f,g,h} {d} {e}



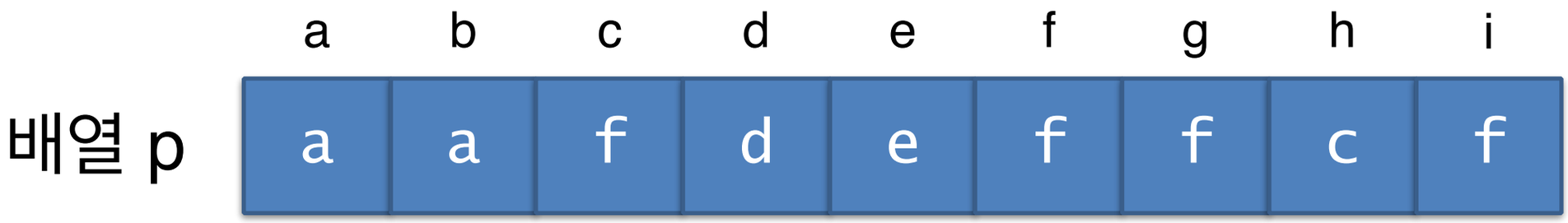
4개의 집합을 4개의 트리로 표현

서로소인 집합들의 표현

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

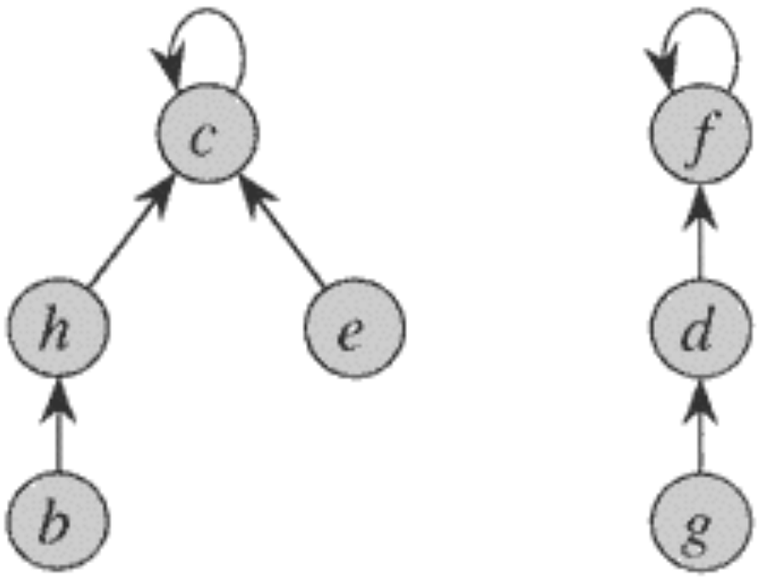


모든 트리를 하나의
배열로 표현



Find-Set(v)

- 자신이 속한 트리의 루트를 찾음



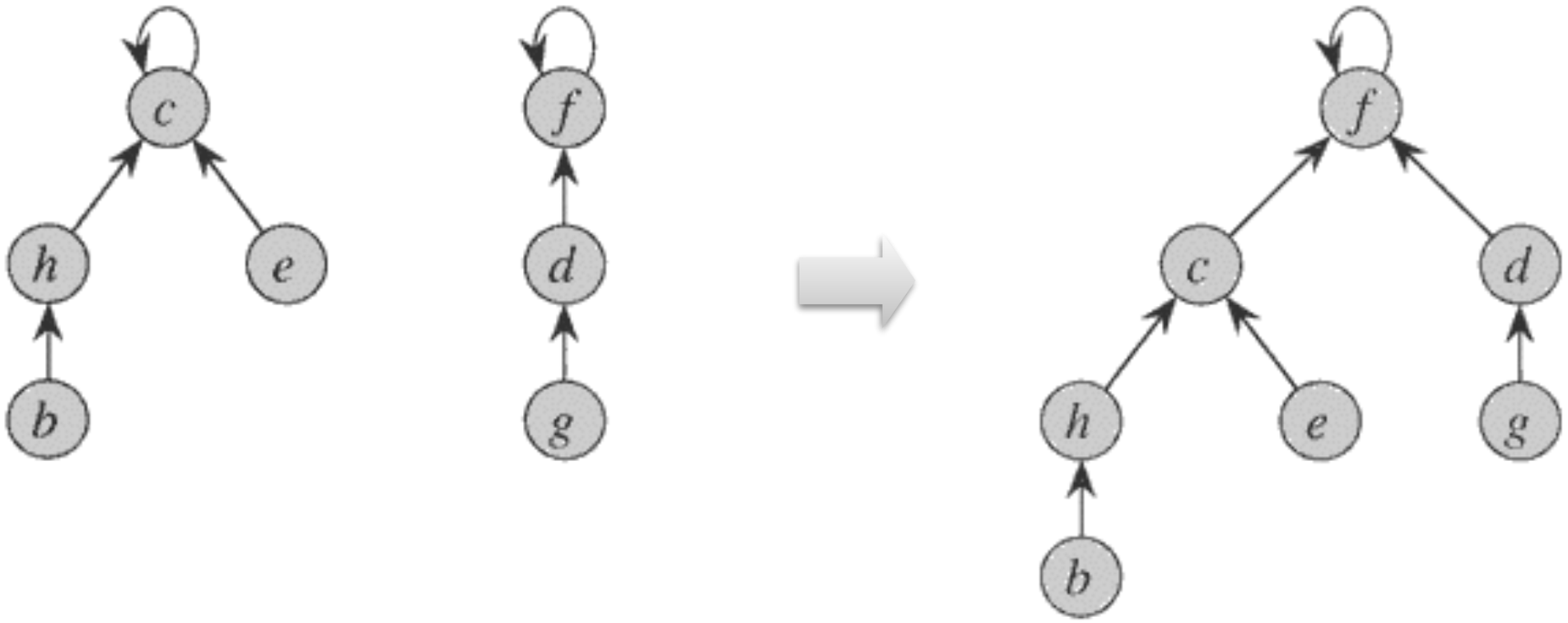

```
FIND-SET(x)
1  if x ≠ p[x]
2      then p[x] ← FIND-SET(p[x])
3  return p[x]
```

O(h), h는 트리의 높이

h는 최악의 경우 O(n)

Union(u,v)

- 한 트리의 루트를 다른 트리의 루트의 자식 노드로 만듬



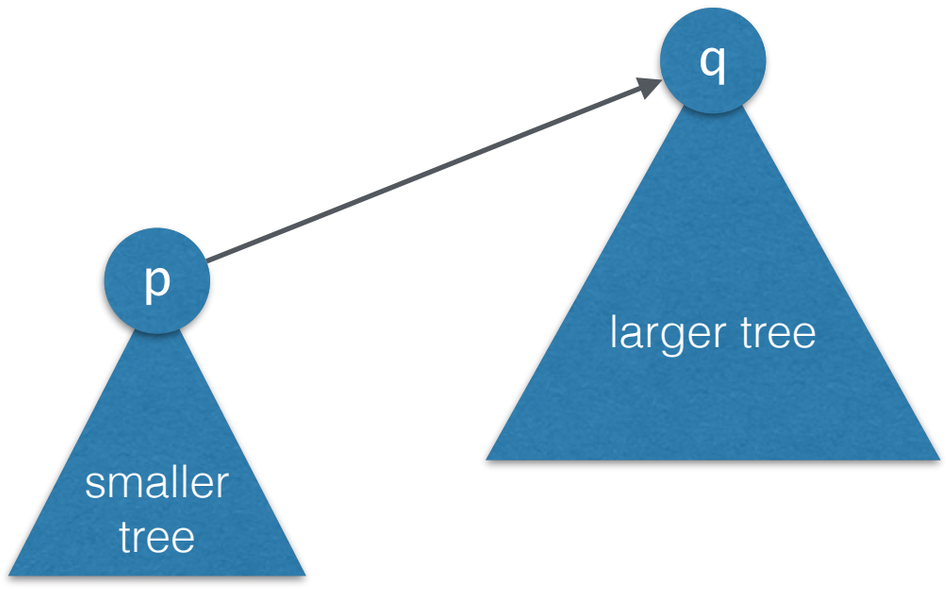

```
UNION(u,v)
1.   x ← FIND-SET(u);
2.   y ← FIND-SET(v);
3.   p[x] ← y;
```

루트 노드를 찾은 이후에는 O(1)

하지만 루트를 찾는데 O(h)

Weighted Union

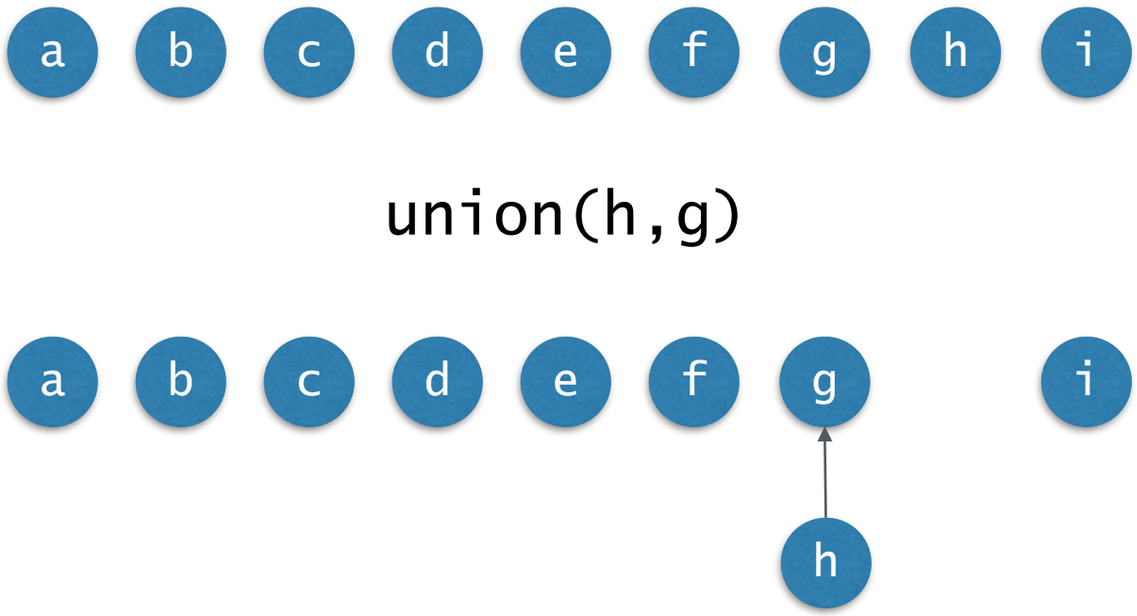
- 두 집합을 union할 때 작은 트리의 루트를 큰 트리의 루트의 자식으로 만듬 (여기서 크기란 노드의 개수)
- 각 트리의 크기(노드의 개수)를 카운트하고 있어야




```
WEIGHTED-UNION(u,v)
1.  x ← FIND-SET(u);
2.  y ← FIND-SET(v);
3.  if size[x]>size[y] then
4.     p[x] ← y;
5.     size[x] ←size[x] + size[y];
6.  else
7.     p[y] ← x;
8.     size[y] ←size[y] + size[x];
```

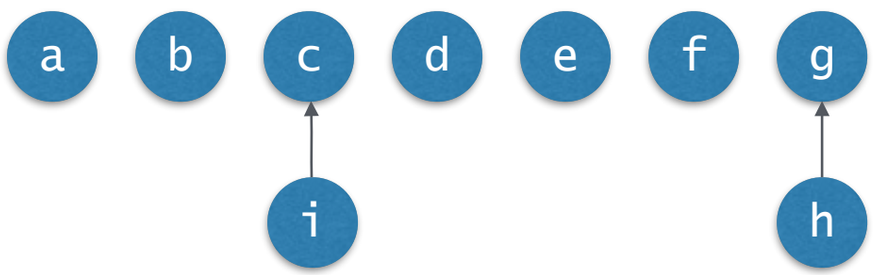
Worst vs. Weighted Union

worst

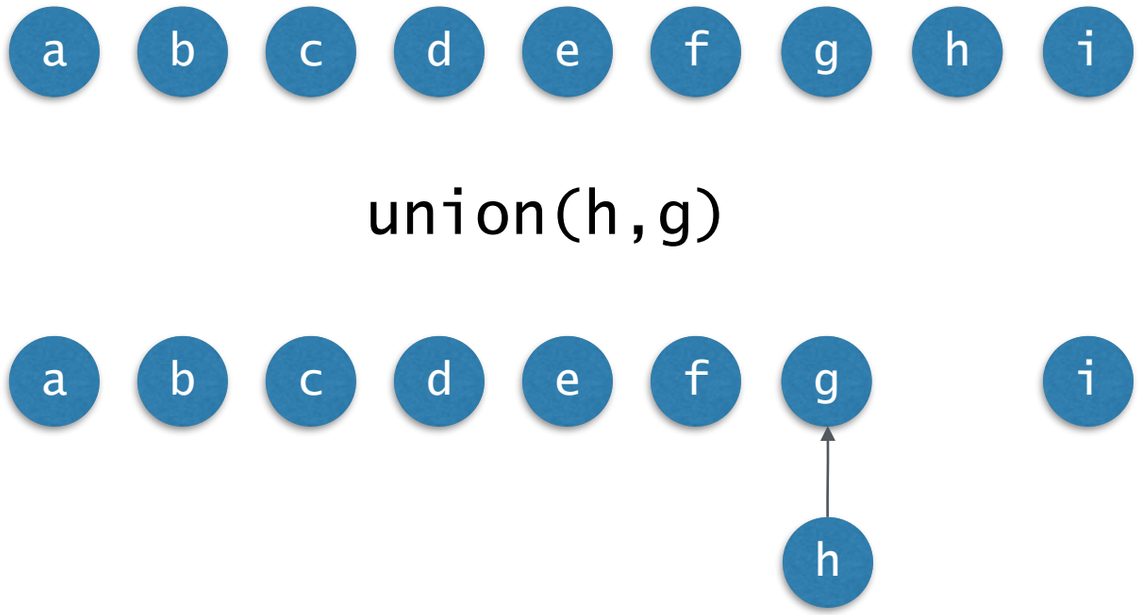


union(h,g)

union(c,i)

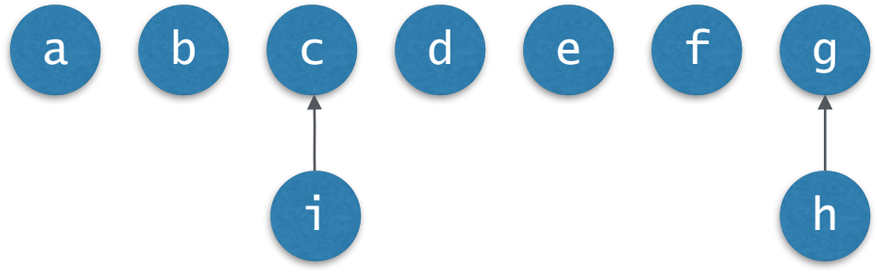


weighted



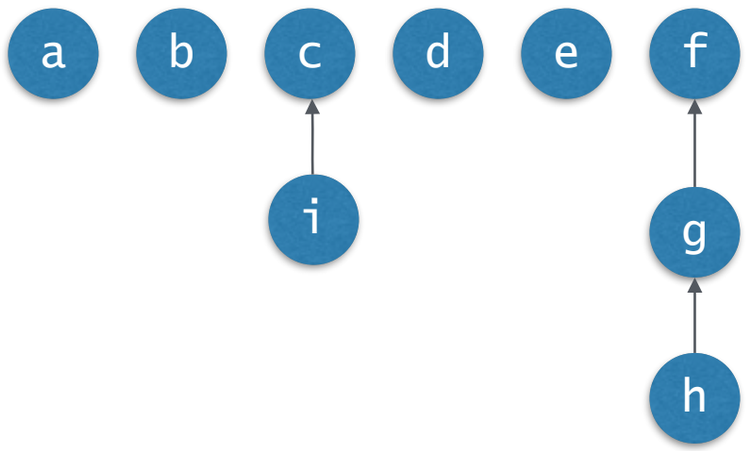
union(h,g)

union(c,i)

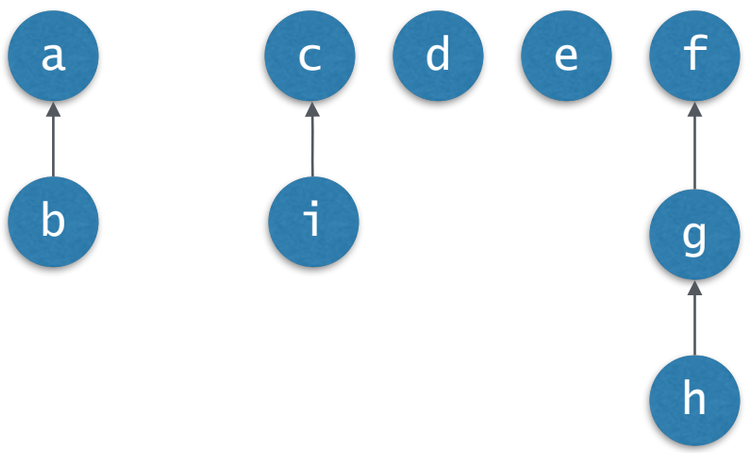


Worst vs. Weighted Union

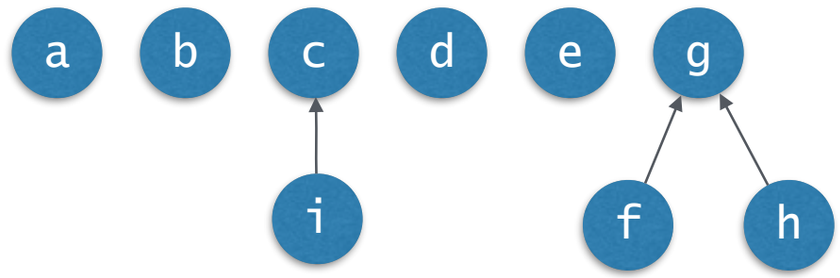
union(g,f)



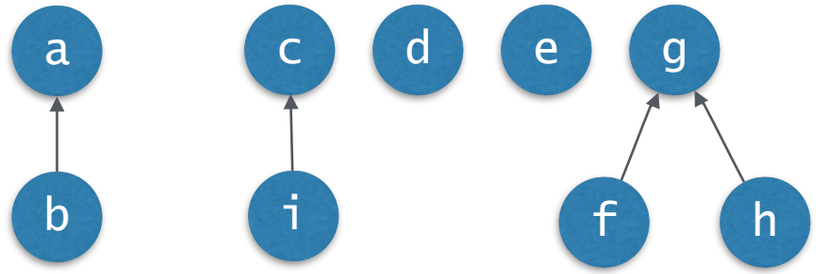
union(a,b)



union(g,f)

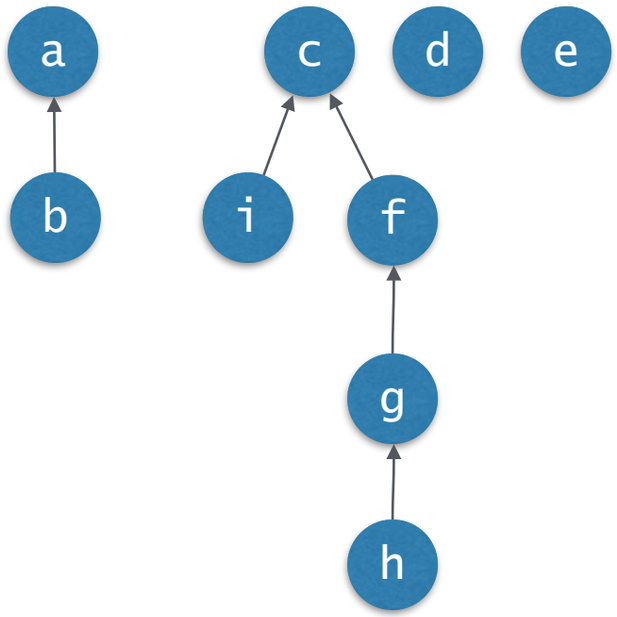


union(a,b)

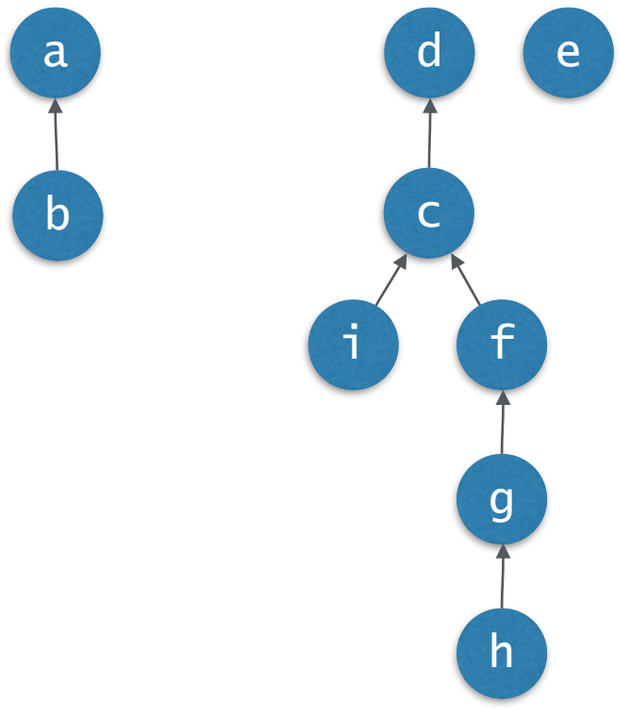


Worst vs. Weighted Union

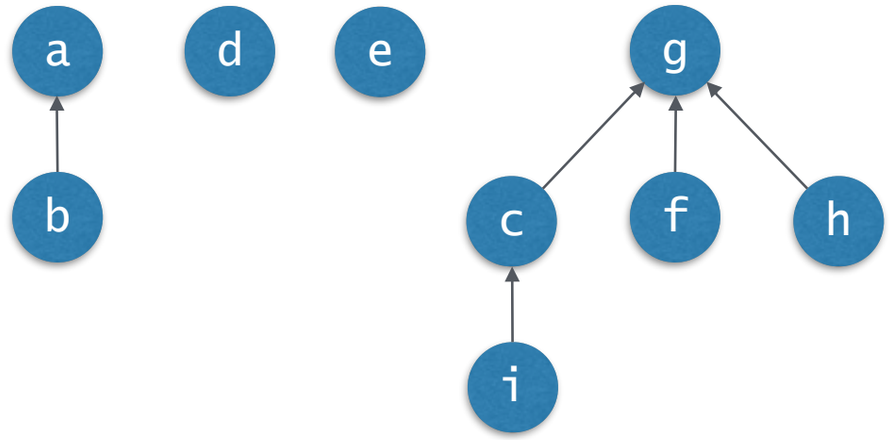
union(c,f)



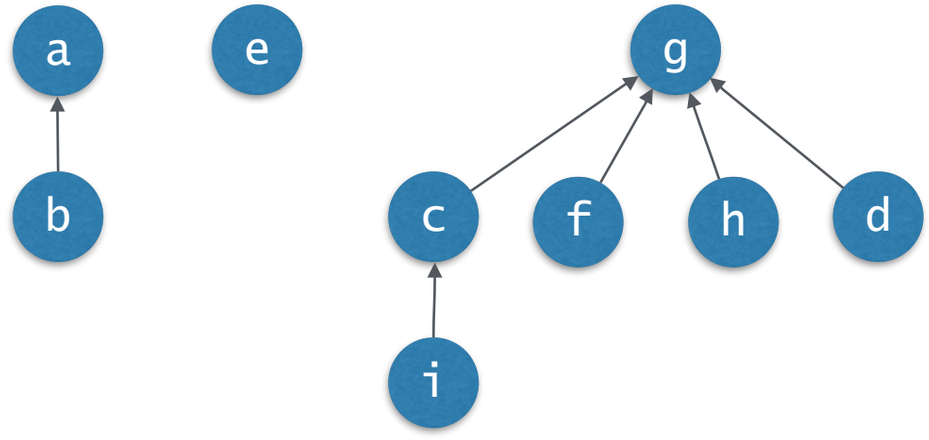
union(c,d)



union(c,f)

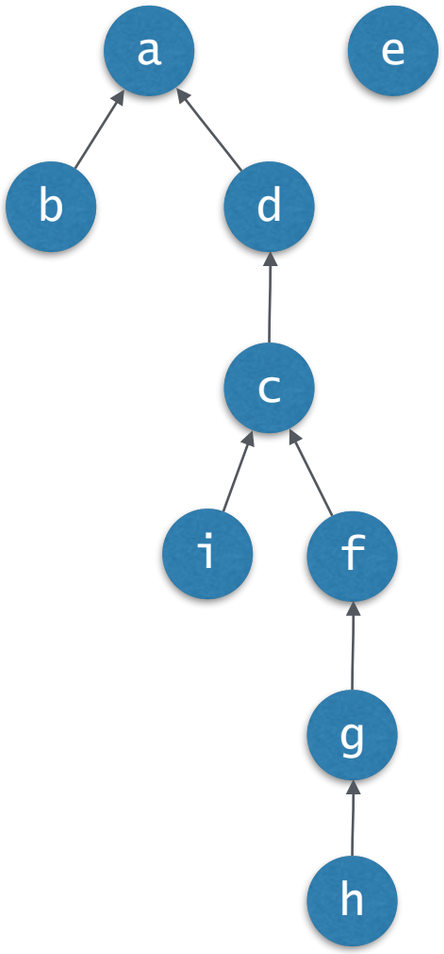


union(c,d)

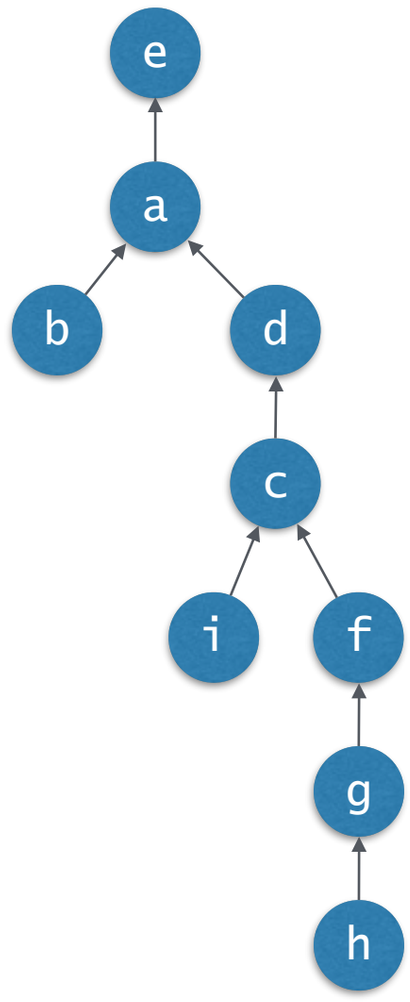


Worst vs. Weighted Union

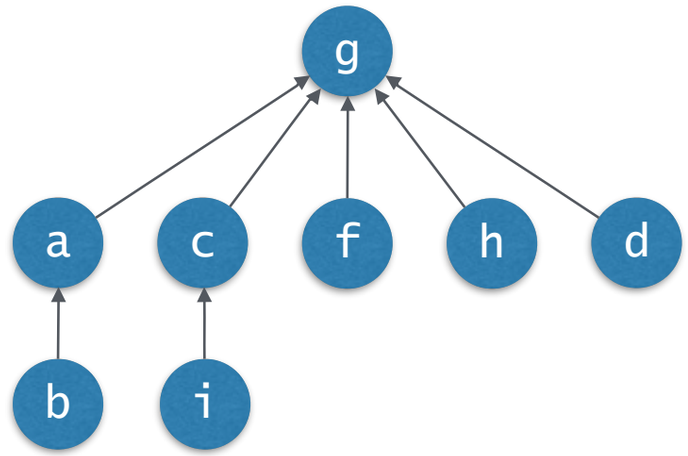
union(b,c)



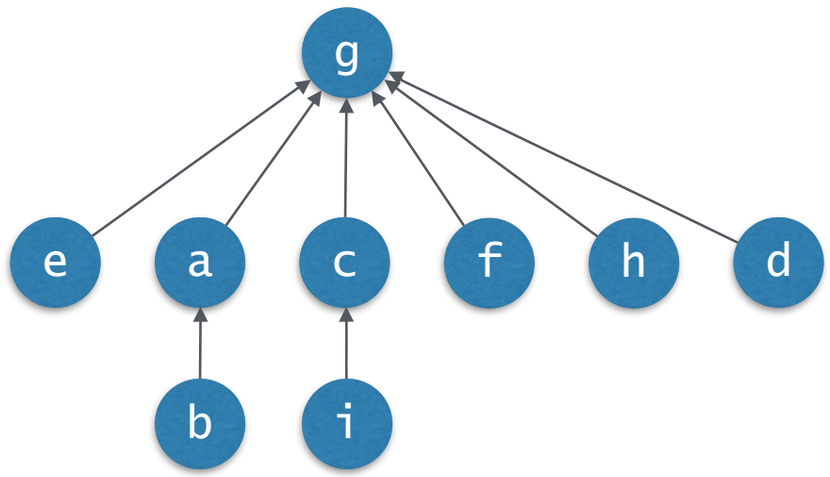
union(d,e)



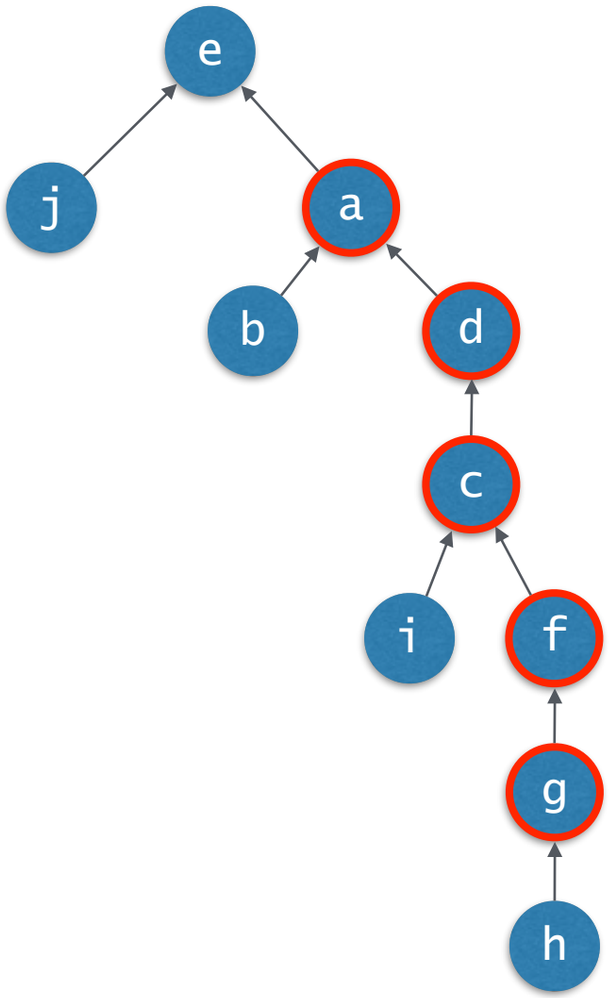
union(b,c)



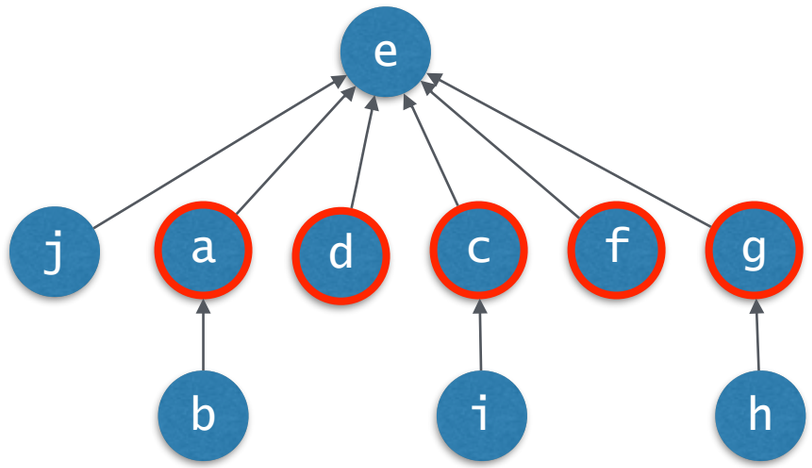
union(d,e)



Path Compression



Find(g)



FIND-SET-PC(x)

1. while x ≠ p[x] do
2. p[x] ← p[p[x]];
3. x ← p[x];
4. end.
5. return p[x];

Weighted Union with Path Compression (WUPC)

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

N	lg* N
1	0
2	1
4	2
16	3
65536	4
2^{65536}	5

lg* function

- 시간복잡도

Initialize A: $O(1)$

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

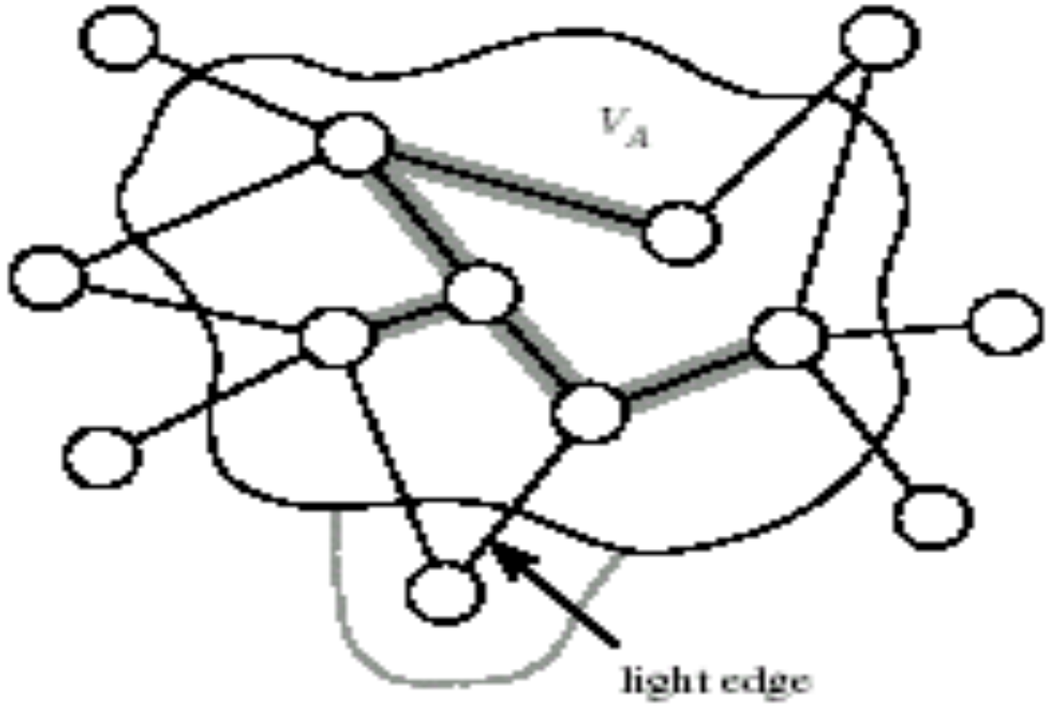
Sort E: $O(|E| \log_2 |E|)$

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

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

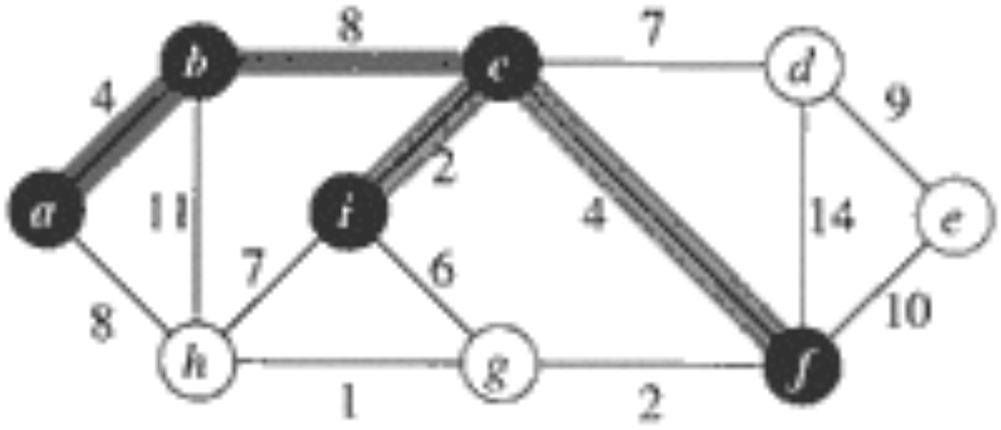
Prim의 알고리즘

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

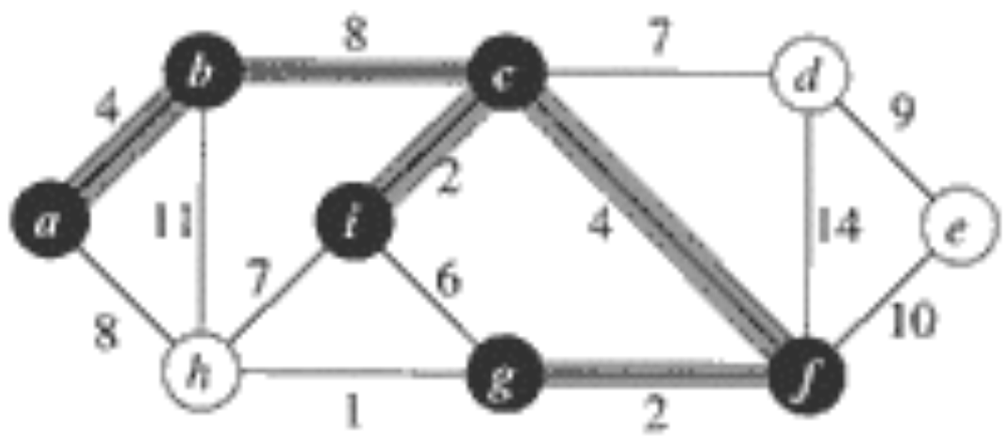


Prim의 알고리즘

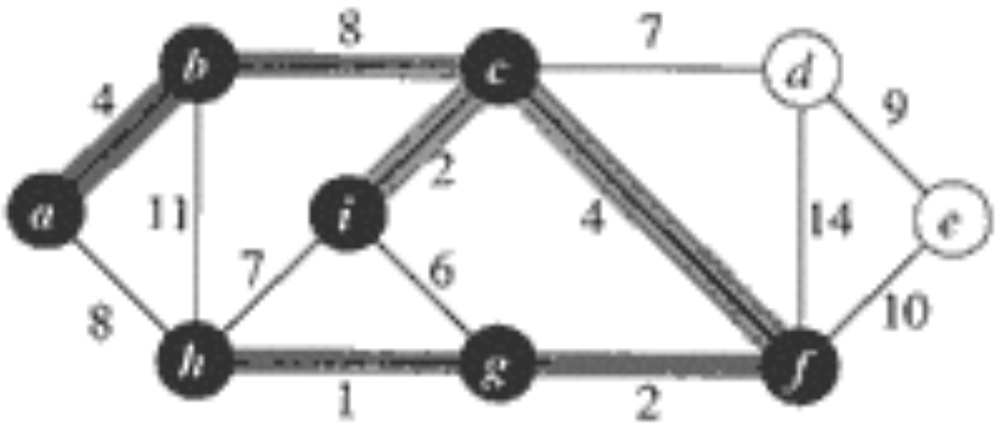
(e)


a
b
c
d
e
f
g
h
i
4
8
11
8
7
7
2
4
14
9
6
10
1
2

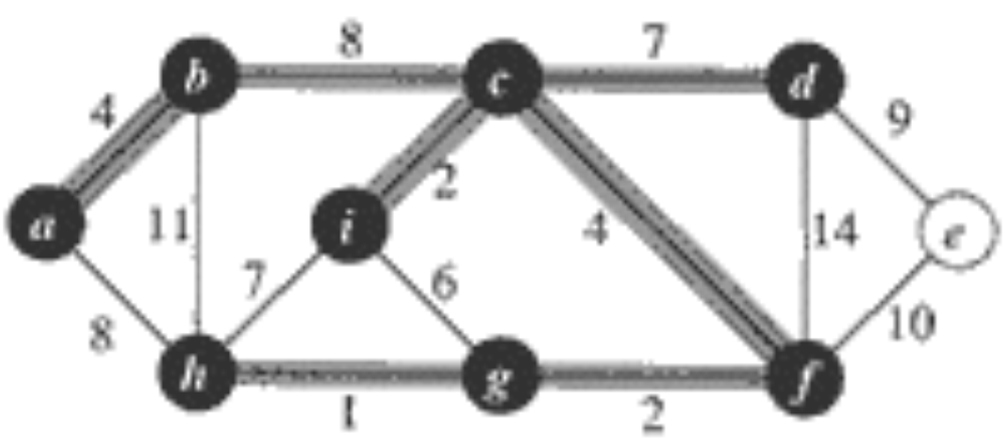
(f)


a
b
c
d
e
f
g
h
i
4
8
11
8
7
7
2
4
14
9
6
10
1
2

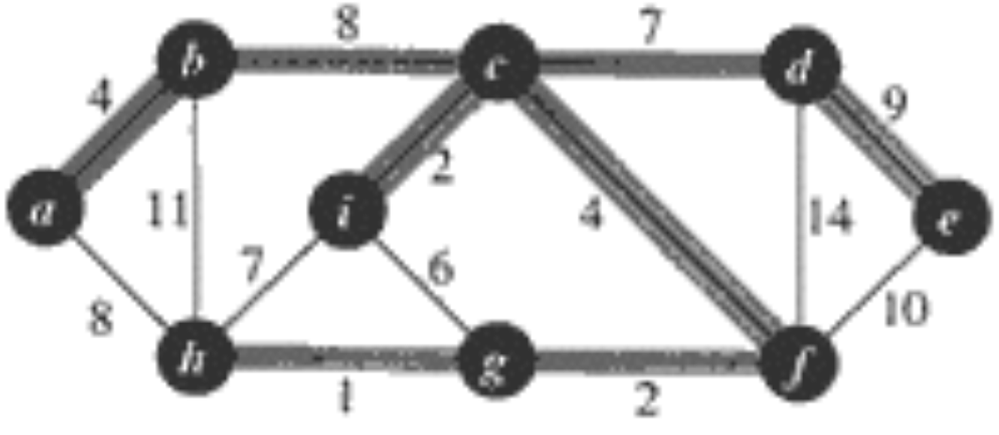
(g)


a
b
c
d
e
f
g
h
i
4
8
11
8
7
7
2
4
14
9
6
10
1
2

(h)

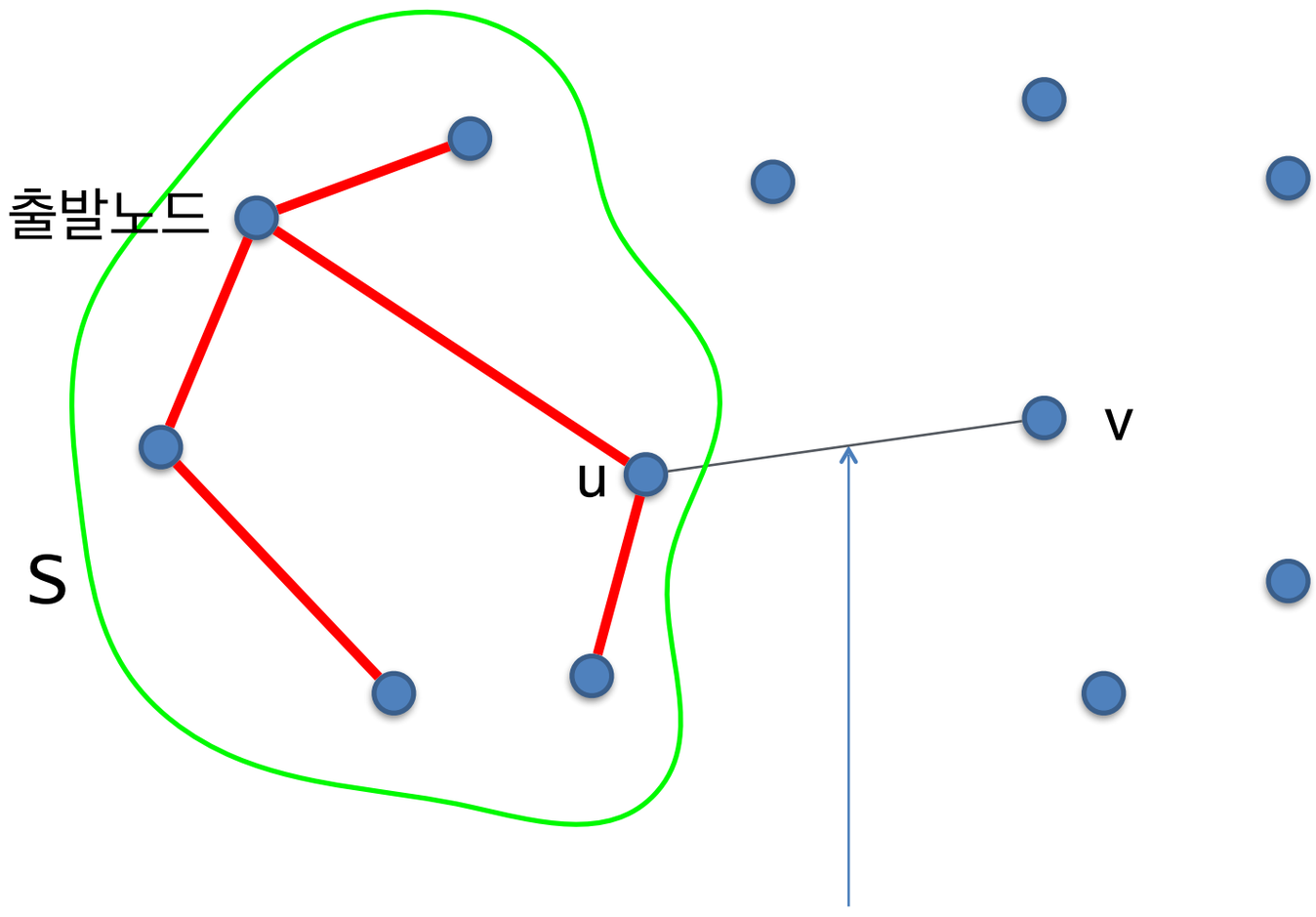

a
b
c
d
e
f
g
h
i
4
8
11
8
7
7
2
4
14
9
6
10
1
2

(i)


a
b
c
d
e
f
g
h
i
4
8
11
8
7
7
2
4
14
9
6
10
1
2

왜 MST가 찾아지는가?

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

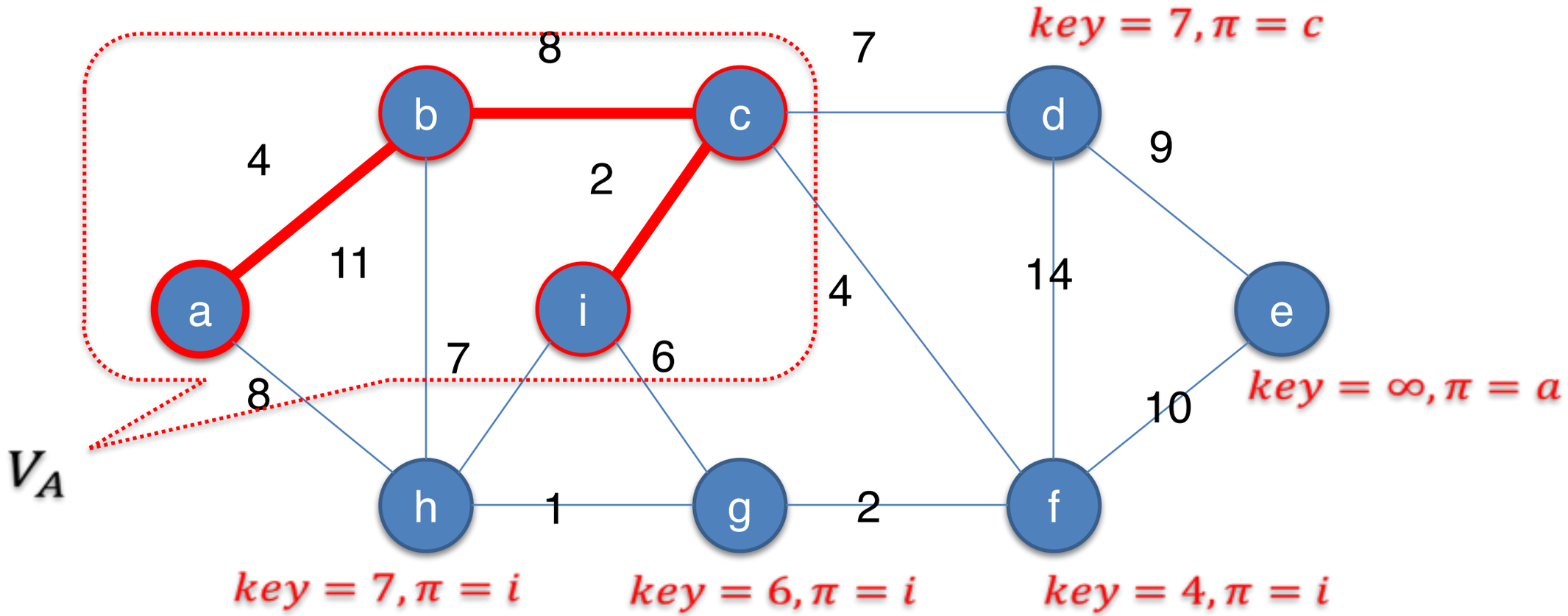


출발 노드에 이미 연결된 노드와 그렇지 않은 노드를 연결하는 에지들 중 lightest edge

가중치가 최소인 에지 찾기

- V_A: 이미 트리에 포함된 노드들
- V_A에 아직 속하지 않은 각 노드 v에 대해서 다음과 같은 값을 유지
  - key(v): 이미 V_A에 속한 노드와 자신을 연결하는 에지들 중 가중치가 최소인 에지 (u,v)의 가중치
  - π(v): 그 에지 (u,v)의 끝점 u

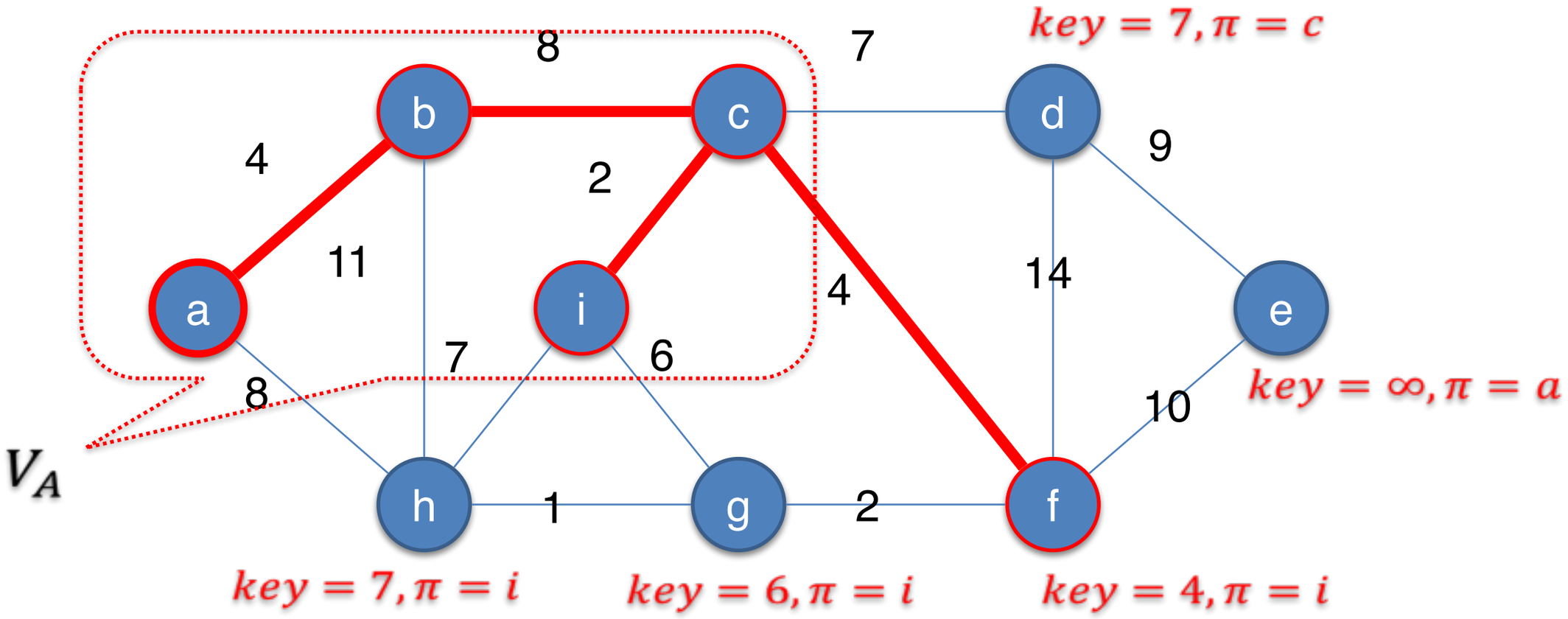
가중치가 최소인 에지 찾기



가중치가 최소인 에지를 찾는 대신

key값이 최소인 노드를 찾는다.

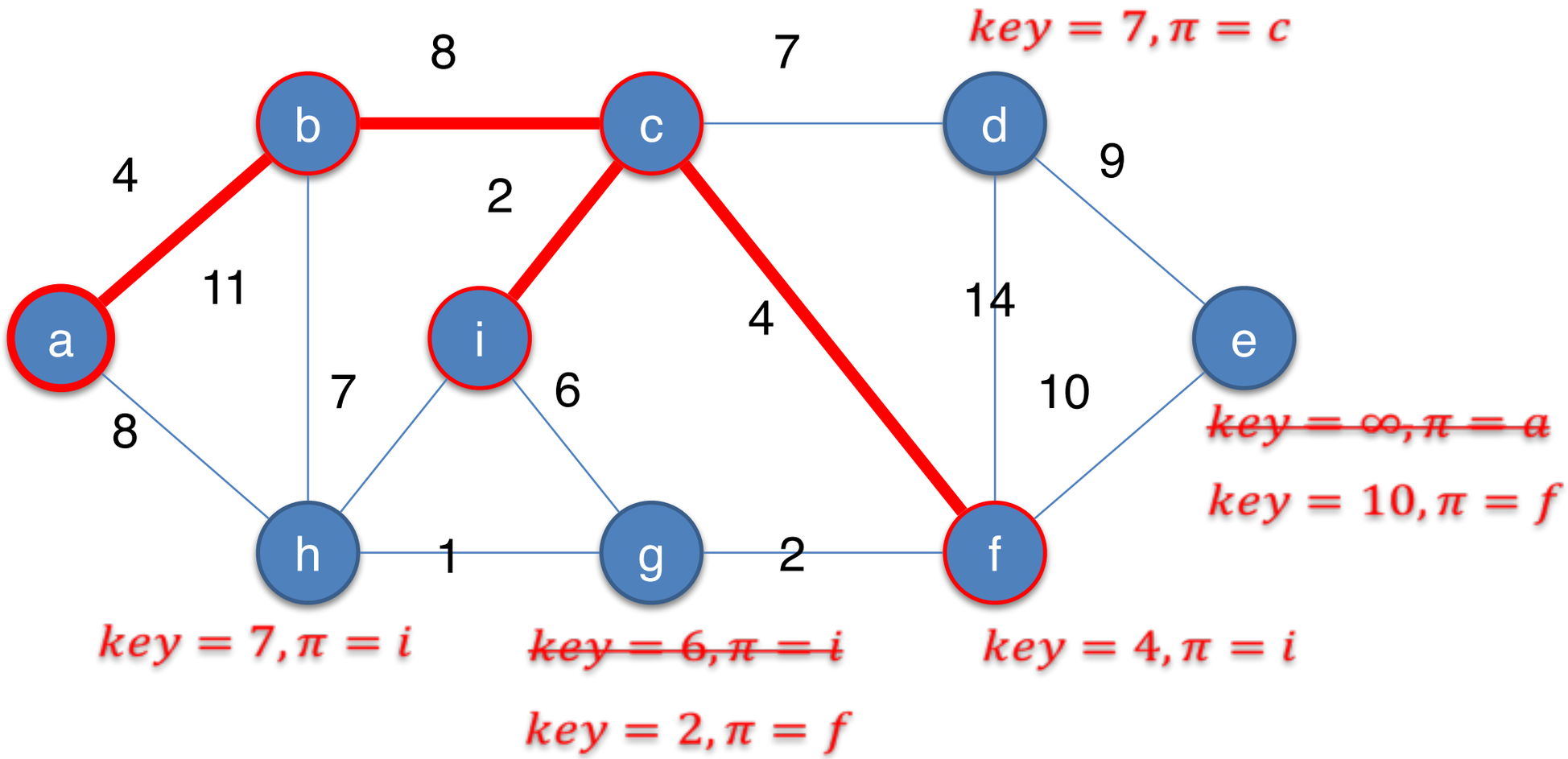
가중치가 최소인 에지 찾기



key 값이 최소인 노드 f를 찾고,

에지 $(f, \pi(f))$를 선택한다.

가중치가 최소인 에지 찾기



노드 d, g, e의 key 값과 π값을 갱신한다.

Prim의 알고리즘

```
MST-Prim(G, w, r)
1.  for each u∈V do
2.      key[u] ← ∞
3.      π[u] ← NIL
4.  end.
5.  V_A ← {r} empty set;
6.  key[r] ← 0
7.  while |V_A|<n do
8.      find u∉V_A with the minimum key value;
9.      V_A ← V_A ∪ {u}
10.     for each v∉V_A adjacent to u do
11.         if key[v] > w(u,v) then
12.             key[v] ← w(u,v)
13.             π[v] ← u
14.         end.
15.     end.
16. end.
```

while문은 n-1번 반복

최소값 찾기 O(n)

degree(u) = O(n)

시간복잡도 $O(n^2)$

Key값이 최소인 노드 찾기

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

Prim의 알고리즘

```
MST-PRIM(G, w, r)
 1  for each u ∈ V[G]
 2       do key[u] ← ∞
 3          π[u] ← NIL
 4  key[r] ← 0
 5  Q ← V[G]
 6  while Q ≠ ∅
 7       do u ← EXTRACT-MIN(Q)
 8          for each v ∈ Adj[u]
 9               do if v ∈ Q and w(u, v) < key[v]
10                    then π[v] ← u
11                         key[v] ← w(u, v)
```

최소값 찾기 O(logn)

우선순위큐에서 key값 decrease: O(logn)

Prim의 알고리즘: 시간복잡도

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