알고리즘

8장 집합의 처리

학습내용

1. 연결 리스트를 이용한 집합의 처리
2. 트리를 이용한 집합의 처리

학습목표

- 연결 리스트를 이용한 상호배타적 집합의 처리 방법을 이해한다.
- 연결 리스트를 이용한 집합의 처리를 위한 연산들의 수행시간을 분석할 수 있도록 한다.
- 트리를 이용한 상호배타적 집합의 처리 방법을 이해한다.
- 트리를 이용한 집합의 처리를 위한 연산들의 수행시간을 기본적인 수준에서 분석할 수 있도록 한다.

집합(set)의 처리

- 이 장에서는 상호배타적 집합(disjoint set) 만을 대상으로 함. 따라서 교집합 연산은 다루지 않음
- 상호배타적 집합 처리에 필요한 작업(지원하는 연산)
  - Make-Set(x): 원소 x로만 이루어진 집합을 생성
  - Find-Set(x): 원소 x가 속한 집합을 알아냄
  - Union(x, y): 원소 x가 속한 집합과 원소 y가 속한 집합의 합집합을 구함

집합의 처리

- 두 가지 구현 방법
  1. Linked list를 이용하는 방법
  2. Tree를 이용하는 방법 ← 수행 시간면에서 더 효율적인 방법임

학습내용

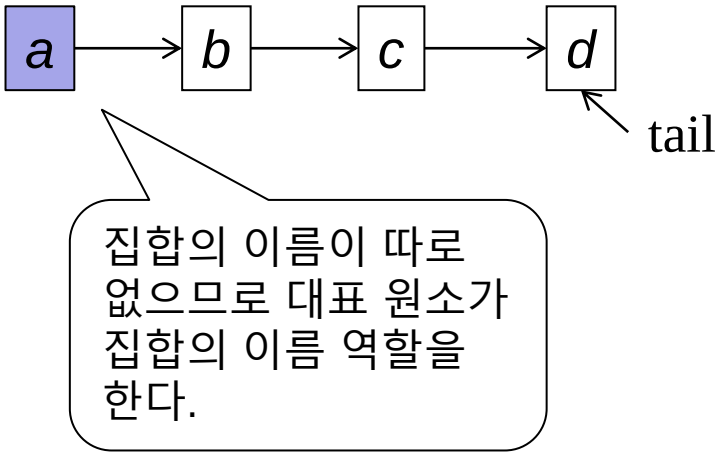
1. 연결 리스트를 이용한 집합의 처리

2. 트리를 이용한 집합의 처리

1. Linked List를 이용한 집합 처리

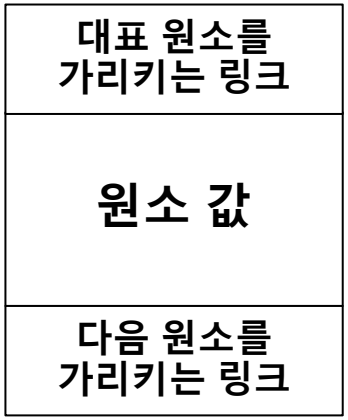
- 각 원소 당 하나의 노드를 만들고, 같은 집합에 속한 원소들을 하나의 연결 리스트로 관리한다.
  - 집합의 대표 원소 = linked list의 맨 앞의 원소
  - 마지막 원소를 가리키는 변수 tail을 둠(Union에 이용)
  - 예) {*a, b, c, d* }를 표현하는 연결리스트

대표 원소는 *a*



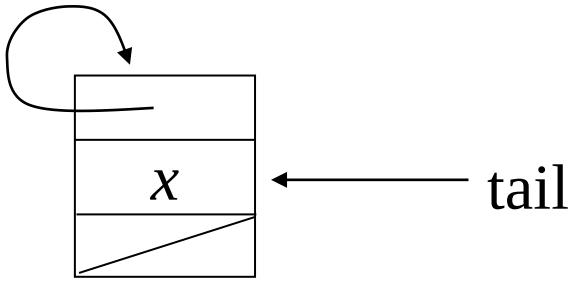
Linked List : 노드 구조

- 노드 구조



Linked List : 원소가 하나인 집합

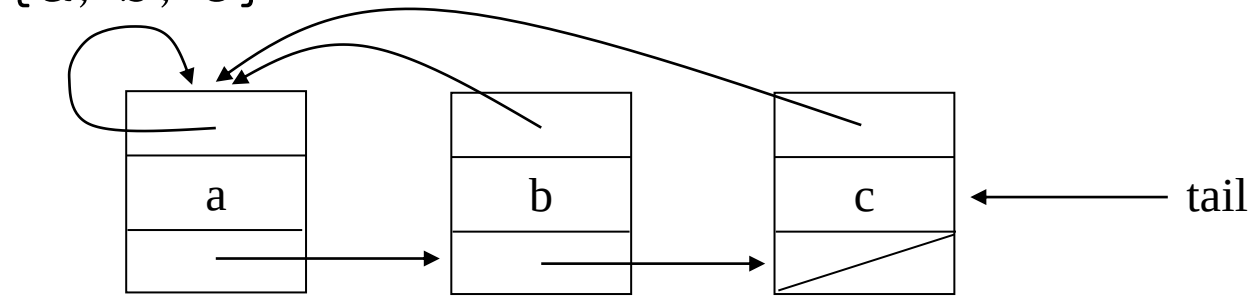
- Make-Set(x) : 원소 x로만 이루어진 집합 $\{x\}$ 를 만드는 연산
  1. 노드를 하나 만들어 원소 x 를 저장
  2. 대표 원소 링크는 자신을 가리키도록 함
  3. 다음 원소는 없으므로 다음 원소 링크는 NIL



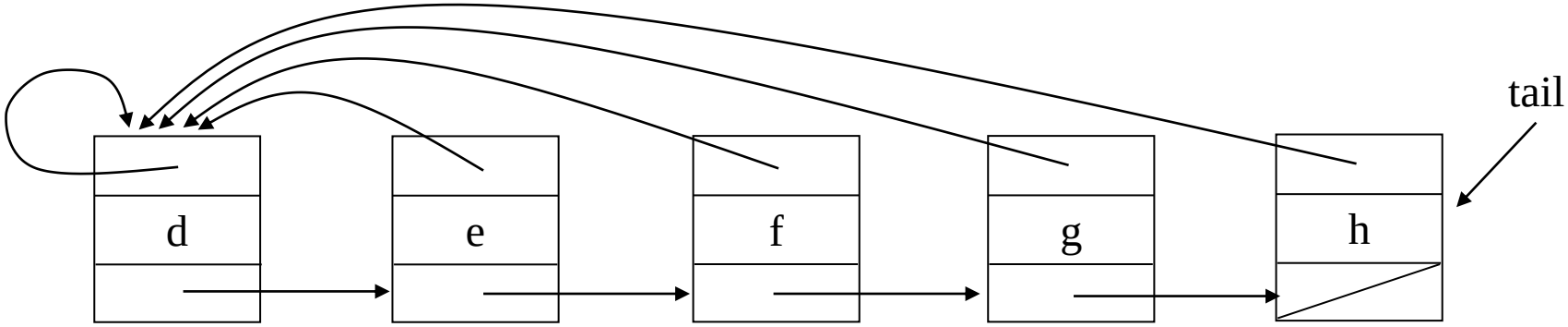
사선(/)은 NIL을 뜻함

Linked List : 원소가 여러 개인 집합

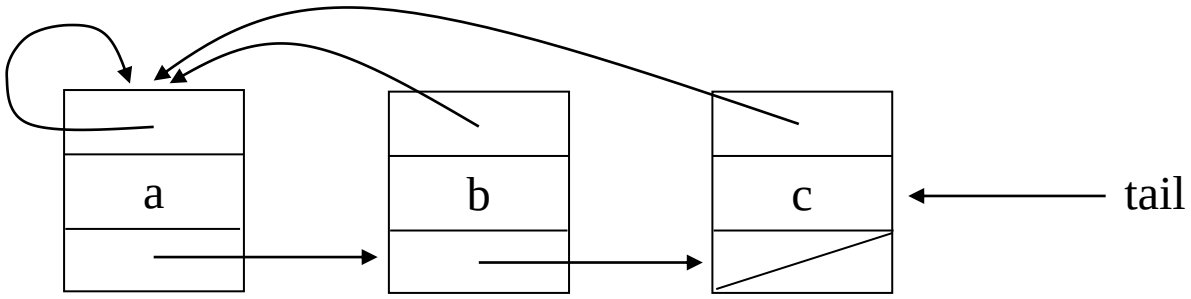
{a, b, c}



{d, e, f, g, h}

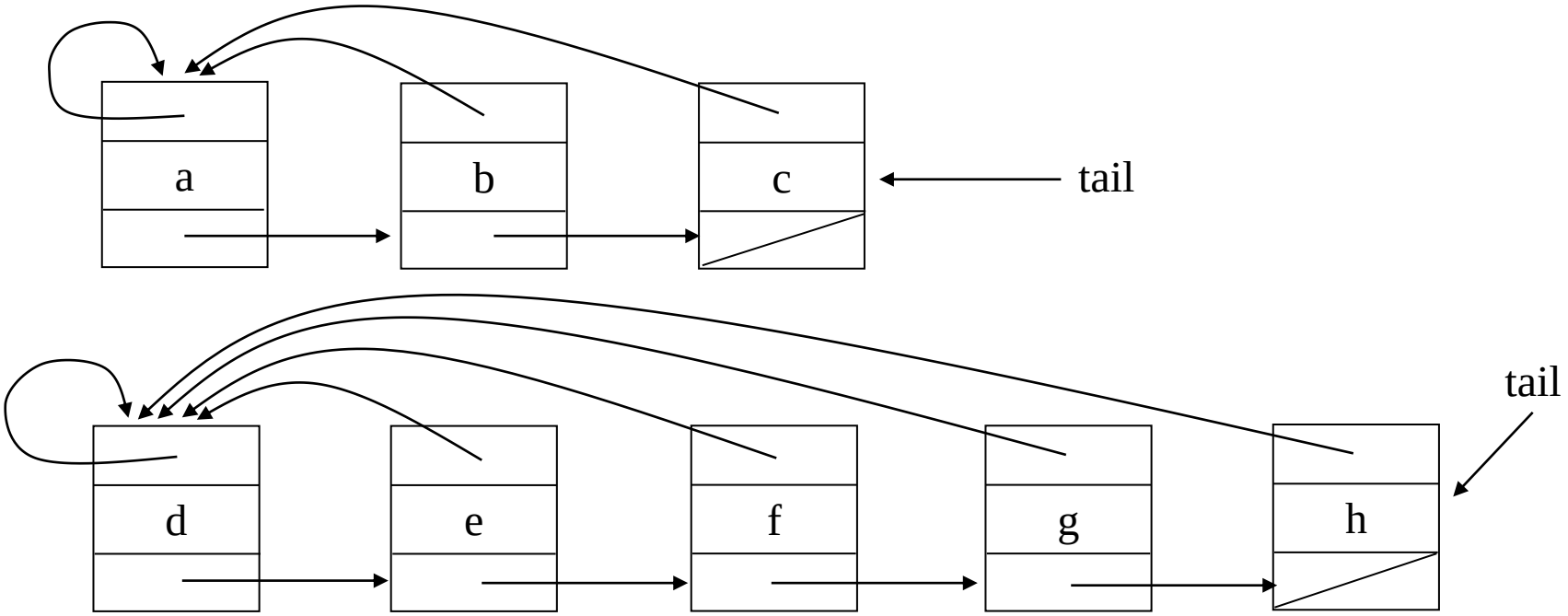


Find-Set 예



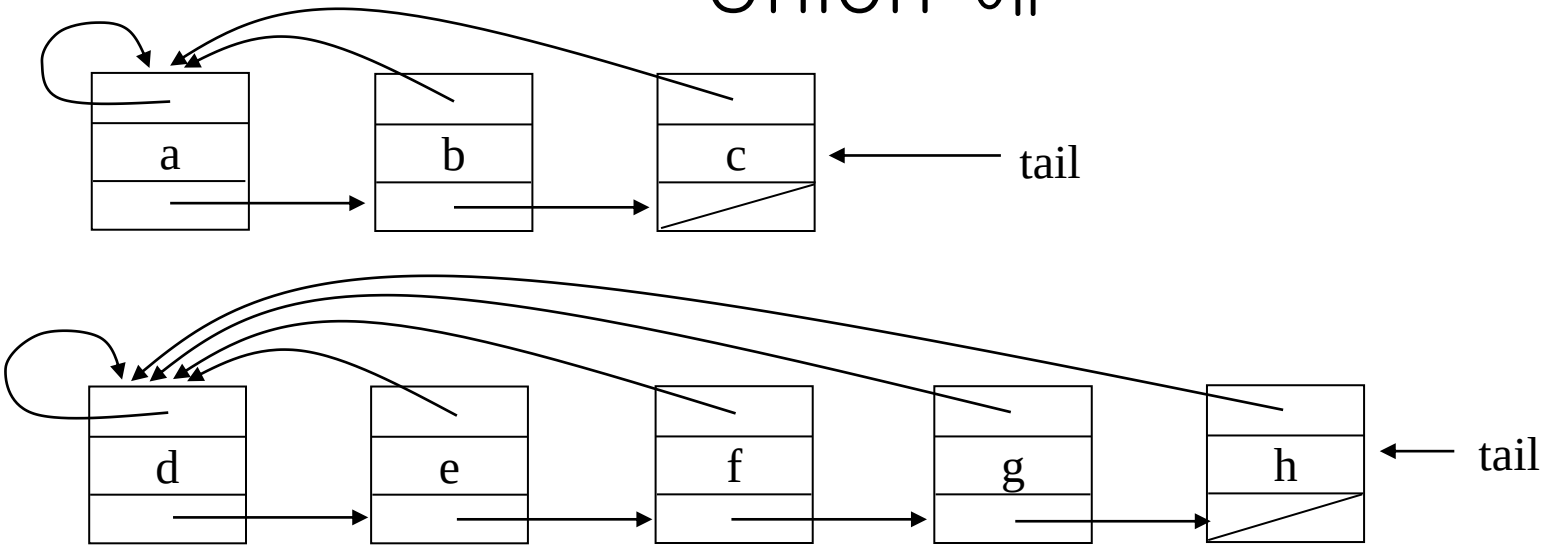
- Find-Set(a): 원소 a가 속한 집합을 알아냄 → 대표 원소 *a*
- Find-Set(*b*): 원소 *b*가 속한 집합을 알아냄 → 대표 원소 *a*
- Find-Set(*c*): 원소 *c*가 속한 집합을 알아냄 → 대표 원소 *a*

Find-Set 예

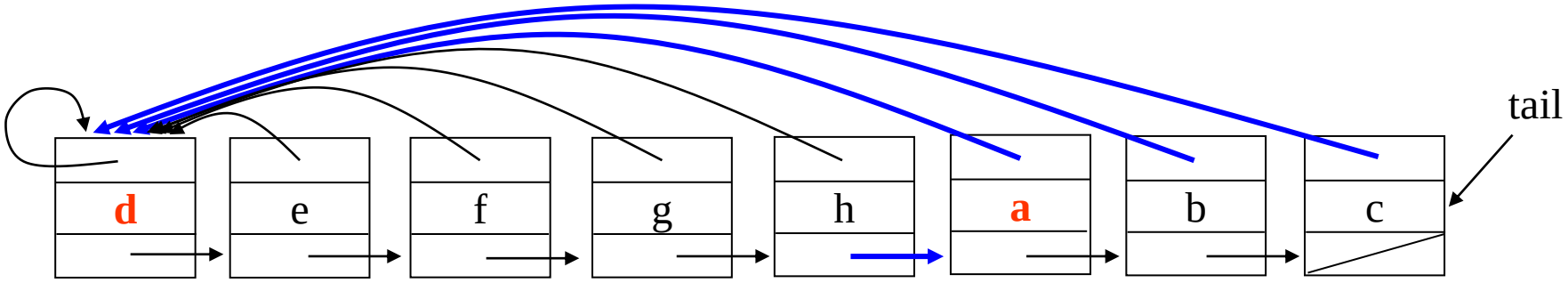


- Find-Set(b) : 원소 b가 속한 집합을 알아냄 → 대표 원소 a
- Find-Set(e): 원소 e가 속한 집합을 알아냄 → 대표 원소 d
- Find-Set(g): 원소 g가 속한 집합을 알아냄 → 대표 원소 d

Union 예



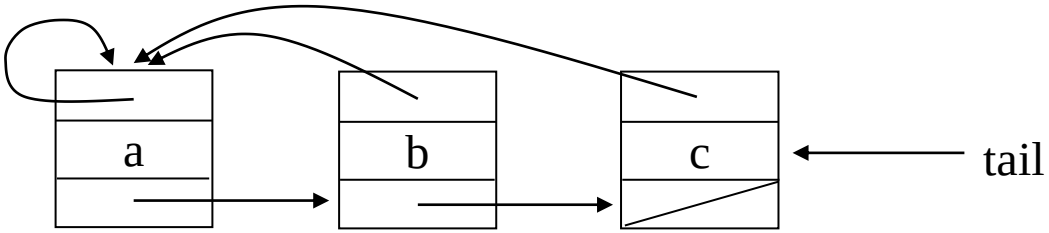
Union(c, g) 연산 : Find-Set(c)와 Find-Set(g)를 수행하여 각각의 대표 원소를 알아낸 후, 한 집합을 다른 집합의 뒤에 붙임

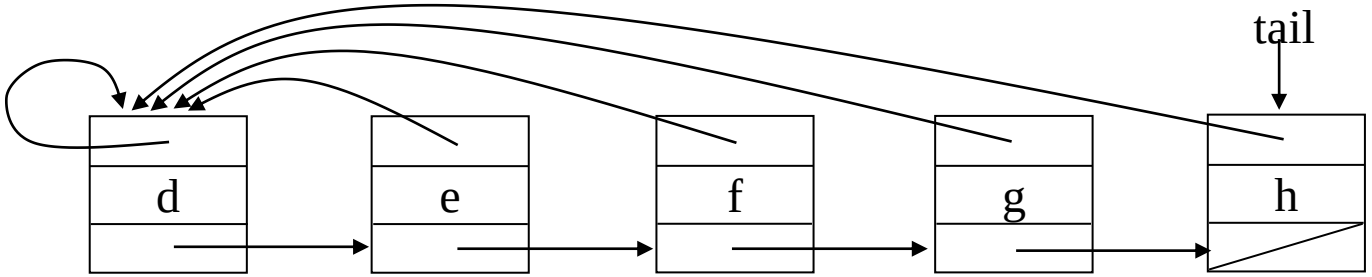


굵은 파란화살표는 변동이 생긴 간선들

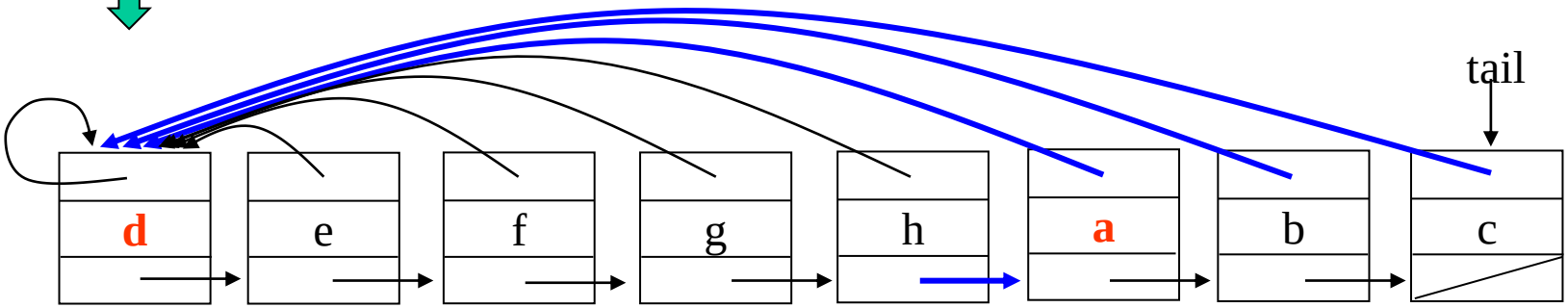
Weighted Union

- 무게를 고려한 Union
  - Linked list로 된 두 집합의 합집합을 구할 때 큰 집합 뒤에 작은 집합을 붙인다.
  - 대표 원소를 가리키는 링크 갱신 작업을 최소화 할 수 있다.
  - 예) 다음 슬라이드
    - Union(c, g) 결과는 (1), (2) 두 가지임
    - 이 중에서 Weighted Union 결과는 (1)

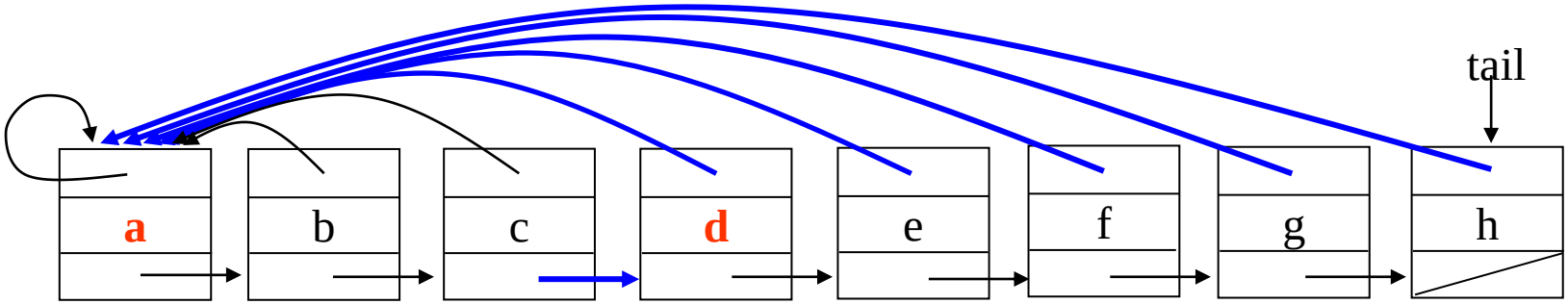




(1)



(2)



수행 시간

[정리 8-1]

- Linked list로 표현한 배타적 집합에서 Weighted Union을 사용할 때, m번의 Make-Set, Union, Find-Set 연산 중 n번이 Make-Set이라면 이들의 총 수행시간은 **O(m + n log n)**.

[증명]

- Make-Set이 n번이므로 원소 수는 n
- Make-Set, Find-Set은 각각 O(1)이므로 모두 합쳐도 O(m)
- Union에서 임의의 원소 x에 대해 대표 원소 포인터 갱신 수 $\leq \lceil \log_2 n \rceil$
  - x가 속한 집합은 갱신이 일어날 때마다 $1 \rightarrow 2 \rightarrow 2^2 \rightarrow 2^3 \rightarrow \ldots$ 이상의 비율로 커지므로

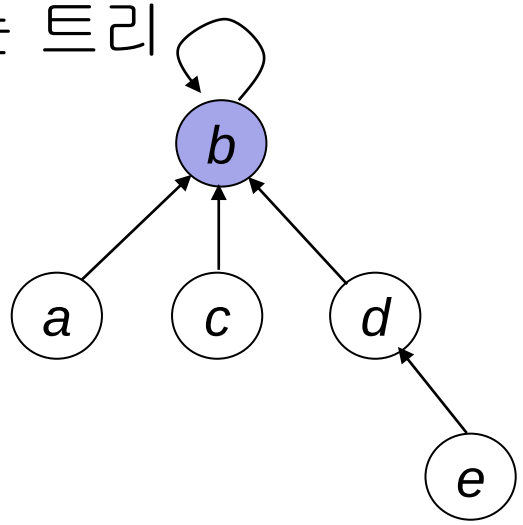
  ➔ 총 대표 원소 갱신 수 $\leq n \times \lceil \log_2 n \rceil$
- 따라서 전체 수행시간 = O(m + n log n)

학습내용

1. 연결 리스트를 이용한 집합의 처리
2. **트리를 이용한 집합의 처리**

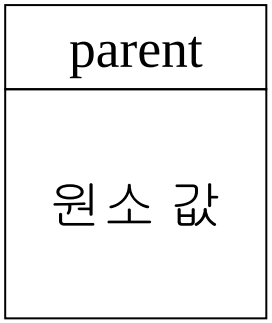
2. Tree를 이용한 집합 처리

- 각 원소 당 하나의 노드를 만들고, 같은 집합에 속한 원소들은 하나의 트리(tree)로 관리한다.
  - 일반적인 트리와 반대로 child가 parent를 가리키는 구조
  - 집합의 대표 원소 = 트리의 루트
  - 예: { *a, b, c, d, e* } 를 표현하는 트리



Tree : 노드 구조

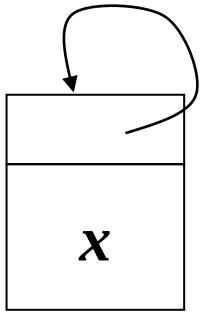
- 노드 구조



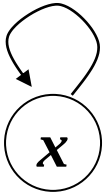
부모를 가리키는 링크

Tree : 원소가 하나인 집합

- Make-Set(x) : 원소 x로만 이루어진 집합 $\{x\}$ 를 만드는 연산
  1. 노드를 하나 만들어 해당 원소를 저장
  2. 부모 링크는 자신을 가리키도록 함

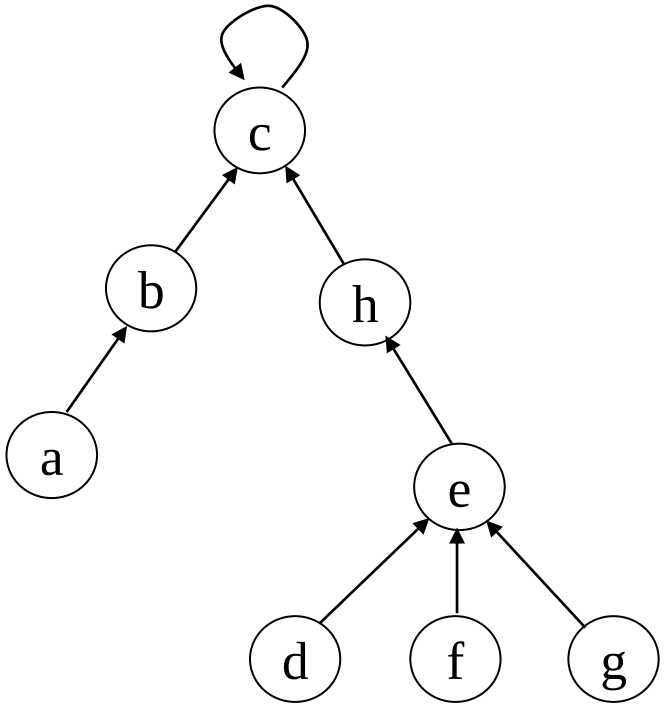


위의 노드 구조는 간단히 다음과 같이 표시하기로 하자.



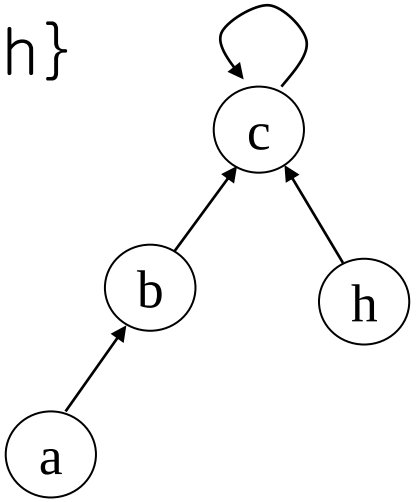
Tree : 원소가 여러 개인 집합

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

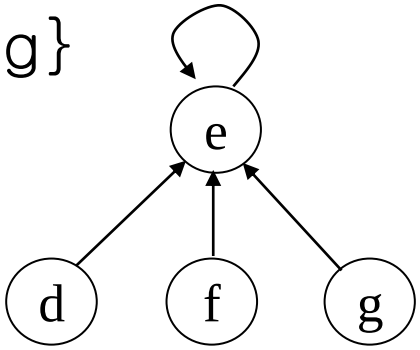


Find-Set 예

{a, b, c, h}



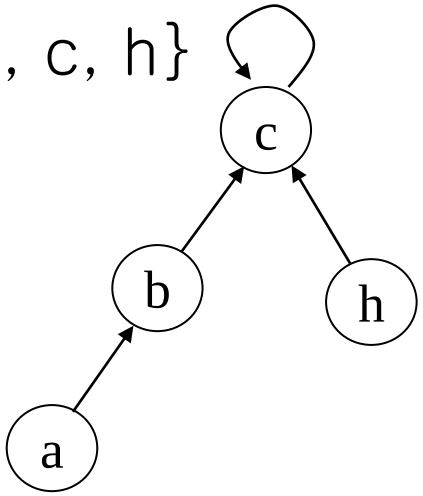
{d, e, f, g}



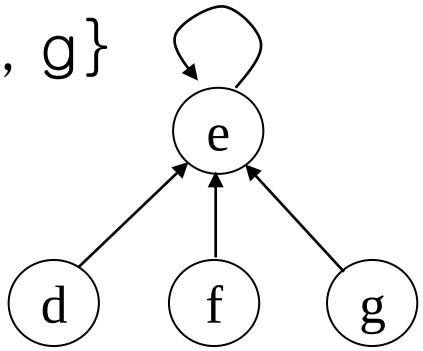
- Find-Set(a): 원소 a가 속한 집합을 알아냄 → 대표 원소 c
- Find-Set(e): 원소 e가 속한 집합을 알아냄 → 대표 원소 e
- Find-Set(g): 원소 g가 속한 집합을 알아냄 → 대표 원소 e

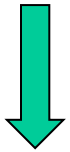
Union 예

{a, b, c, h}



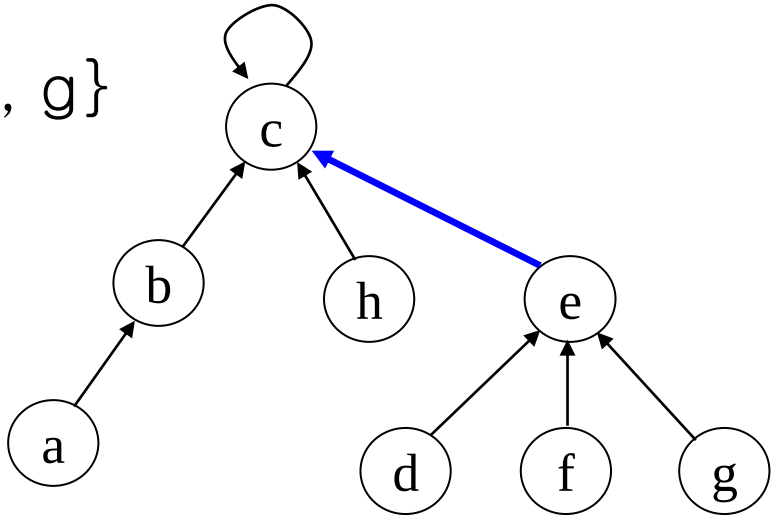
{d, e, f, g}





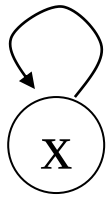
Union(a, g)

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



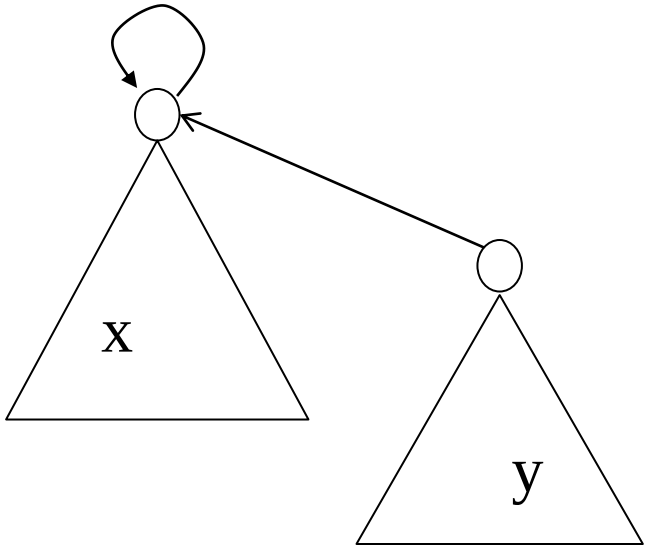
Tree를 이용한 집합 처리 알고리즘

```
Make-Set(x)        ▷ 노드 x를 유일한 원소로 하는 집합을 만든다.
{
      x.parent ← x ;
}
```



Tree를 이용한 집합 처리 알고리즘

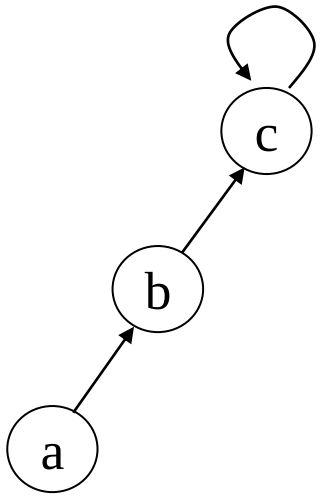
```
Union(x, y)          ▷ 노드 x가 속한 집합에 노드 y가 속한 집합을 합친다.
{
      Find-Set(y).parent ← Find-Set(x);
}
```



Tree를 이용한 집합 처리 알고리즘

Find-Set(x) ▷ 노드 x가 속한 집합을 알아낸다.
즉, 노드 x가 속한 트리의 루트 노드를 리턴한다.

```
{
    if (x = x.parent)
        then return x ;
        else return Find-Set(x.parent);
}
```




```
Find-Set(x) {
    if (x = x.parent) then return x ;
    else return Find-Set(x.parent) ;
}
```

```
Find-Set(x) {
    if (x = x.parent) then return x ;
    else return Find-Set(x.parent) ;
}
```

```
Find-Set(x) {
    if (x = x.parent) then return x ;
    else return Find-Set(x.parent) ;
}
```

```
y = Find-Set(a);
```

연산의 효율을 높이는 방법

- Tree를 이용한 집합 처리에서 연산의 효율을 높이는 방법 두가지

  (1) 랭크(rank)를 이용한 Union

  - Union 연산을 수행하여 트리가 확장될 때, 트리의 높이를 가능한 낮게 유지할 수 있도록 함

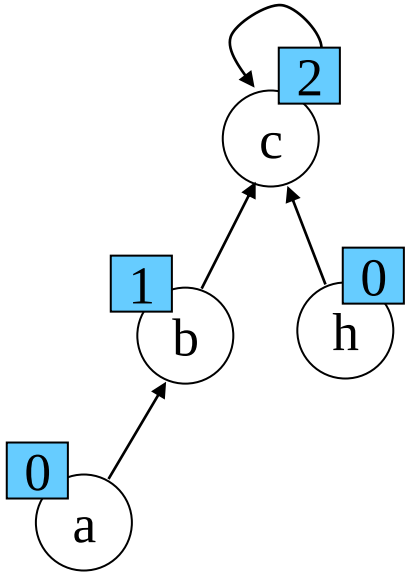
  (2) 경로 압축(path compression)

  - Find-Set 연산을 수행하는 과정에서 경로의 길이를 줄임

트리의 높이를 낮추는 두가지 방법

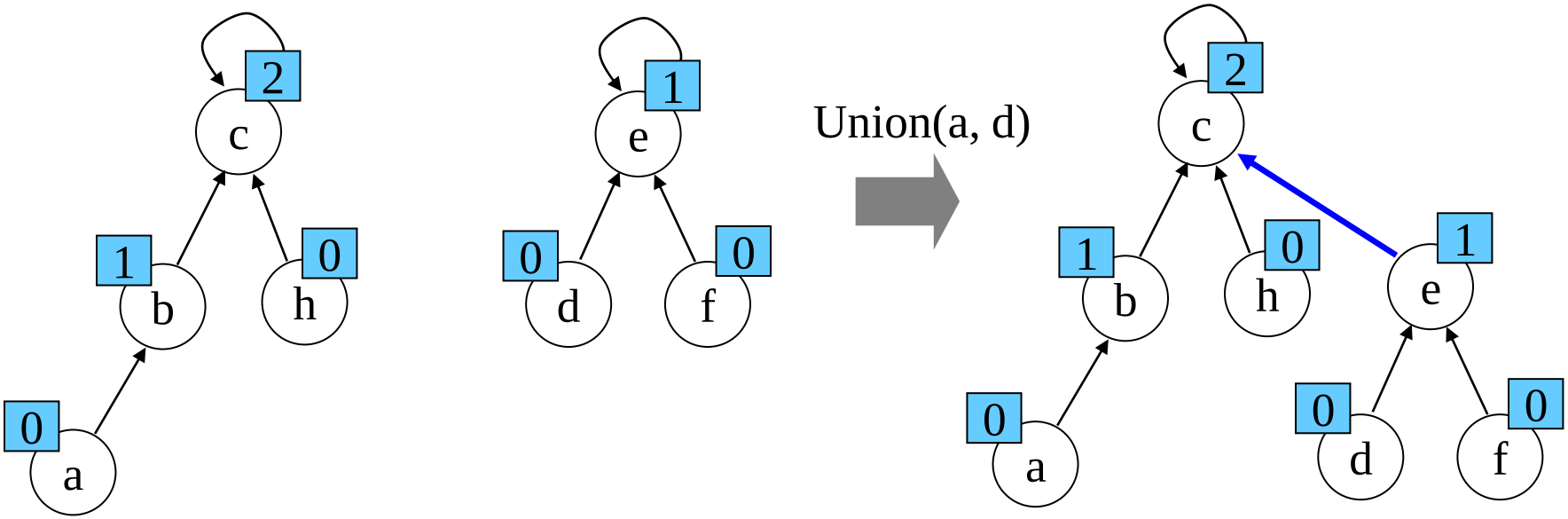
(1) 랭크를 이용한 Union

- Union 연산시 트리의 높이를 가능한 낮게 유지할 수 있도록 rank라는 개념을 이용
- 랭크(rank)
  - 각 노드에 랭크(rank)라는 이름의 필드를 두어 자신을 루트로 하는 서브트리의 높이를 저장한다.
  - 이 때 하나의 노드로 이루어진 트리(서브트리)의 높이는 0이라고 정의한다.
- 집합의 랭크(rank)
  - 루트 노드의 rank가 그 집합의 rank이다.
- 랭크를 이용한 Union
  - Union 연산시 rank가 낮은 집합을 rank가 높은 집합에 붙인다.



랭크를 이용한 Union 예

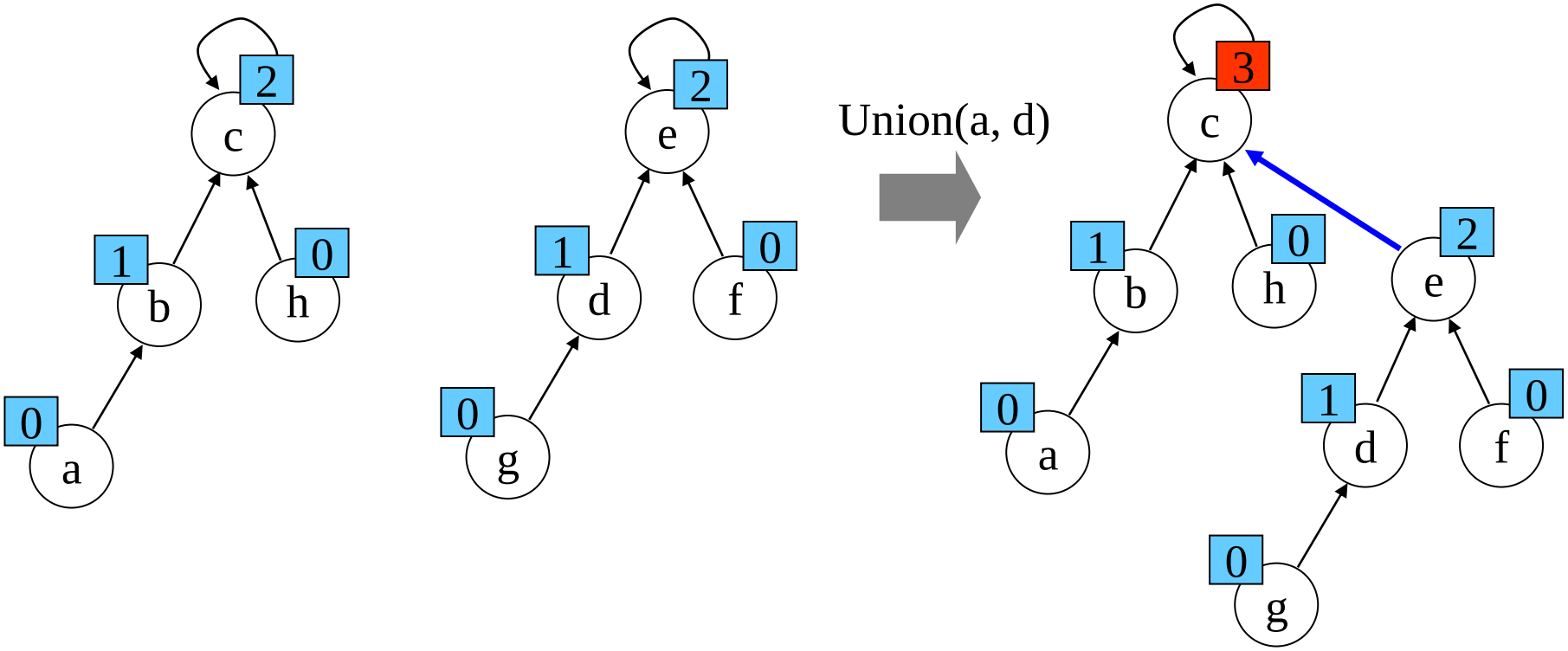
랭크가 변하지 않는 예



rank가 낮은 집합을 rank가 높은 집합에 붙인다.

랭크를 이용한 Union 예

랭크가 증가하는 예

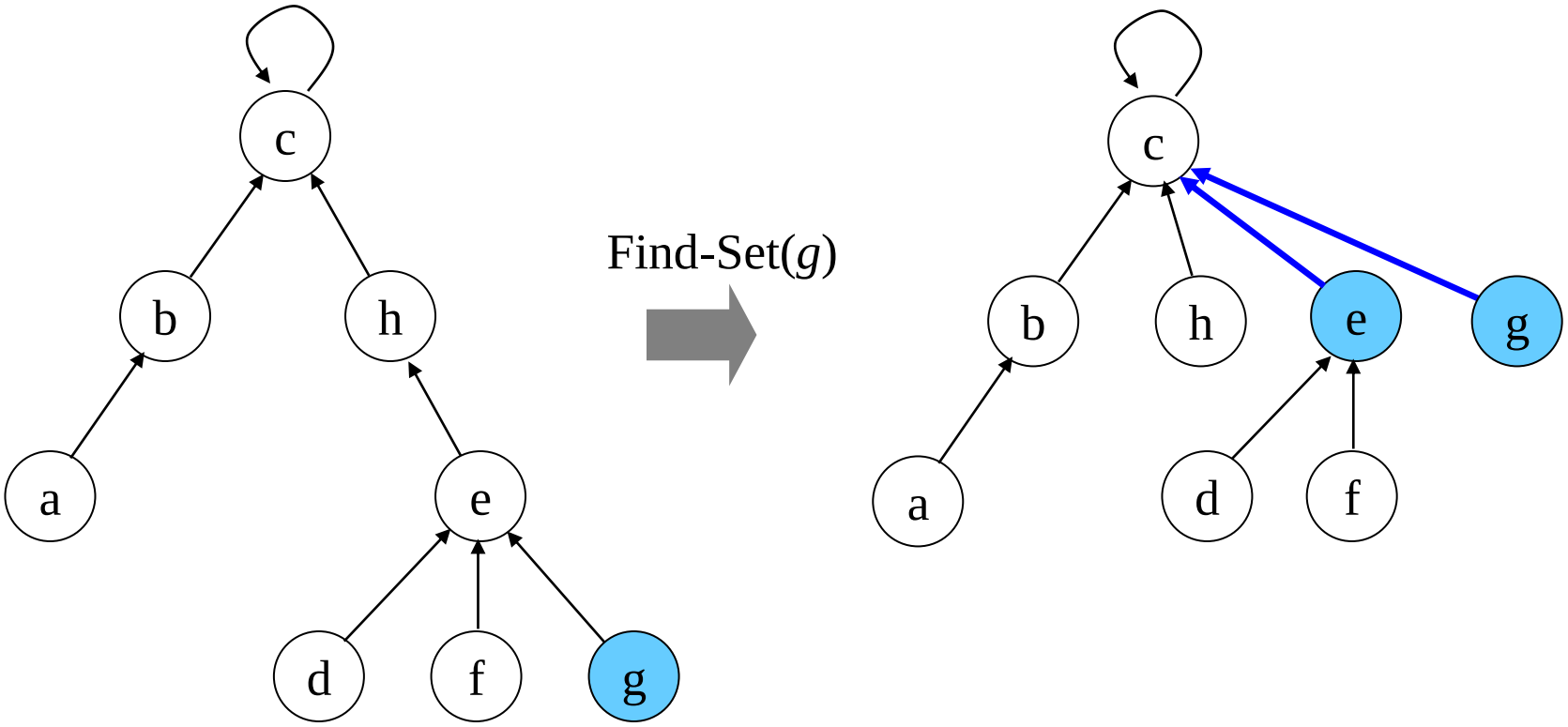


두 트리의 rank가 같으면 랭크가 증가한다.

(2) 경로 압축(path compression)

- Find-Set 연산시 경로의 길이를 줄이는 시도를 함
  - Find-Set을 수행하는 과정에서 만나는 모든 노드들에 대해, 현재 부모를 가리키는 대신, 직접 root를 가리키도록 parent 포인터를 바꾸어 준다.
  - 이 과정에서 트리의 높이가 줄어들 가능성이 높다.

경로 압축의 예



(1) 랭크를 이용한 Union - 알고리즘

```
Make-Set(x) ▷ 노드 x를 유일한 원소로 하는 집합을 만든다.
{
    x.parent ← x;
    x.rank ← 0;
}
```

(1) 랭크를 이용한 Union – 알고리즘

Union(x, y) ▷ 노드 x가 속한 집합과 노드 y가 속한 집합을 합친다.

```
{
    x' ← Find-Set(x);
    y' ← Find-Set(y);
    if (x'.rank > y'.rank)
        then y'.parent ← x' ;
        else {
            x'.parent ← y' ;
            if (x'.rank = y'.rank) then y'.rank ← y'.rank + 1;
        }
}
```

Find-Set(x) ▷ 랭크를 이용하지 않은 알고리즘과 동일

(2) 경로 압축을 이용한 Find-Set – 알고리즘

Find-Set(x) ▷ 노드 x가 속한 집합을 알아낸다.
즉, 노드 x가 속한 트리의 루트 노드를 리턴한다.

```
{
    if (x.parent ≠ x) then                  ▷ x가 대표 원소(루트 노드)가 아니면
        x.parent ← Find-Set(x.parent);      ▷ x의 parent의 대표 원소를 찾아
                                               x의 parent로 삼음
    return x.parent;
}
```

비교: 경로 압축을 이용하지 않은 기존 Find-Set

```
Find-Set(x) {
    if (x = x.parent)
        then return x ;
        else return Find-Set(x.parent);
}
```

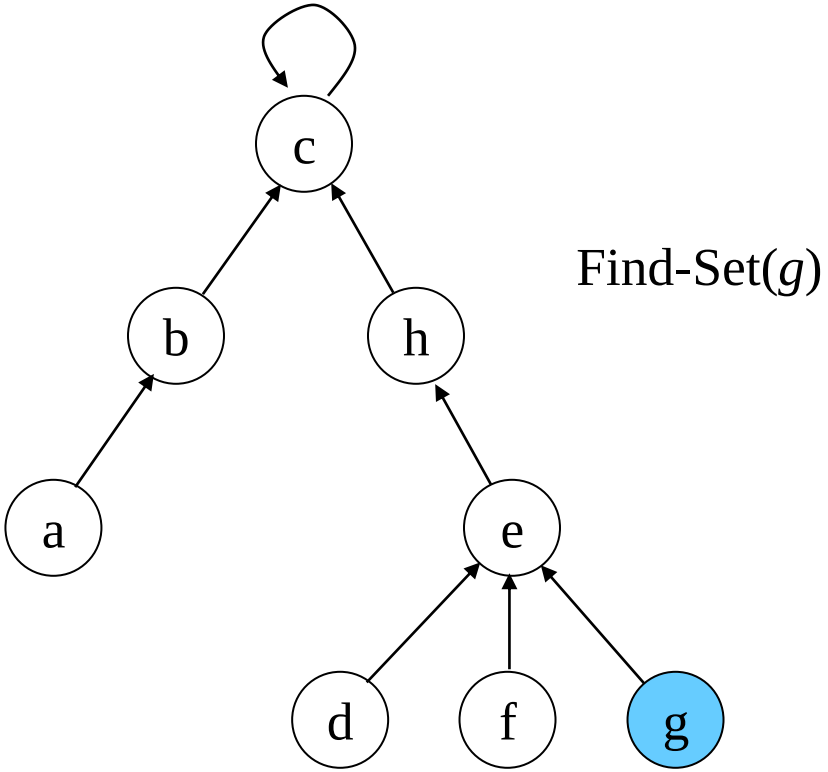
```
Find-Set(x)
{
    if (x.parent ≠ x) then
        x.parent ← Find-Set(x.parent);
    return x.parent;
}
```

```
Find-Set(x)
{
    if (x.parent ≠ x) then
        x.parent ← Find-Set(x.parent);
    return x.parent;
}
```

```
Find-Set(x)
{
    if (x.parent ≠ x) then
        x.parent ← Find-Set(x.parent);
    return x.parent;
}
```

```
Find-Set(x)
{
    if (x.parent ≠ x) then
        x.parent ← Find-Set(x.parent);
    return x.parent;
}
```

```
y = Find-Set(g)
```



Find-Set(*g*)

수행 시간

[정리 8-2]

- 랭크를 이용한 Union을 사용하면, 랭크가 k인 노드를 대표로 하는 집합의 원소 수는 최소한 2^k 개다.

[정리 8-3]

- 랭크를 이용한 Union 을 사용하면, 원소 수가 n인 집합을 표현하는 트리에서 임의의 노드의 랭크는 $O(\log n)$이다.

[정리 8-4]

- 랭크를 이용한 Union을 사용할 때, m번의 Make-Set, Union, Find-Set 중 n번이 Make-Set이라면, 이들의 총 수행 시간은 $O(m \log n)$이다.

수행 시간

[정리 8-5]

- Tree로 표현한 배타적 집합에서 랭크를 이용한 Union과 경로압축을 이용한 Find-Set을 동시에 사용하면, m번의 Make-Set, Union, Find-Set 중 n번이 Make-Set일 때 이들의 총 수행시간은 **$O(m \log^* n)$.**

$\log^* n$ 은 "로그스타 n" 이라고 읽고, 다음과 같이 정의된다.

$$\log^* n = \min \{k : \underbrace{\log\log \ldots \log}_{k} n \le 1\}$$

[이 정리의 의미]

- 현실적인 n 값에 대해 $\log^* n$ 는 상수라고 봐도 될 정도로 작은 값이다. 예) $\log^* 2^{65536} = 5$ 참고: $65536 = 2^{16}$
- 따라서 $O(m \log^* n)$은 사실상 $O(m)$, 즉 선형시간임
  - m 번의 연산을 수행하는 시간이 $O(m)$이므로 각 연산에 평균 상수 시간이 걸린다.

요약

- 상호 배타적 집합을 표현하는 대표적인 방법은 연결 리스트를 이용한 방법, 트리를 이용한 방법이다.
- 상호 배타적 집합 처리에 필요한 작업은 Make-Set, Find-Set, Union
- 연결 리스트를 이용한 방법은 잘 구현하면 Make-Set, Find-Set은 O(1), Union은 평균 O(log n)
- 트리를 이용한 방법은 잘 구현하면 m번의 Make-Set, Find-Set, Union 작업에 O(m)