Disjoint Sets

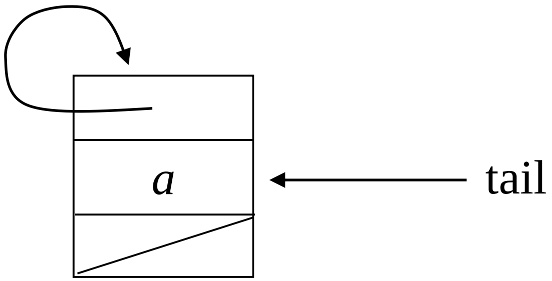
Set의 처리

- 이 장에서는 disjoint set 만을 대상으로 한다
- 그러므로 교집합은 없다
- 지원할 연산
  - Make-Set(x): 원소 x로만 이루어진 집합을 만든다
  - Find-Set(x): 원소 x를 가지고 있는 집합을 알아낸다
  - Union(x, y): 원소 x를 가진 집합과 원소 y를 가진 집합의 합집합을 만든다
- Linked list를 이용하는 방법과 tree를 이용하는 방법이 있다

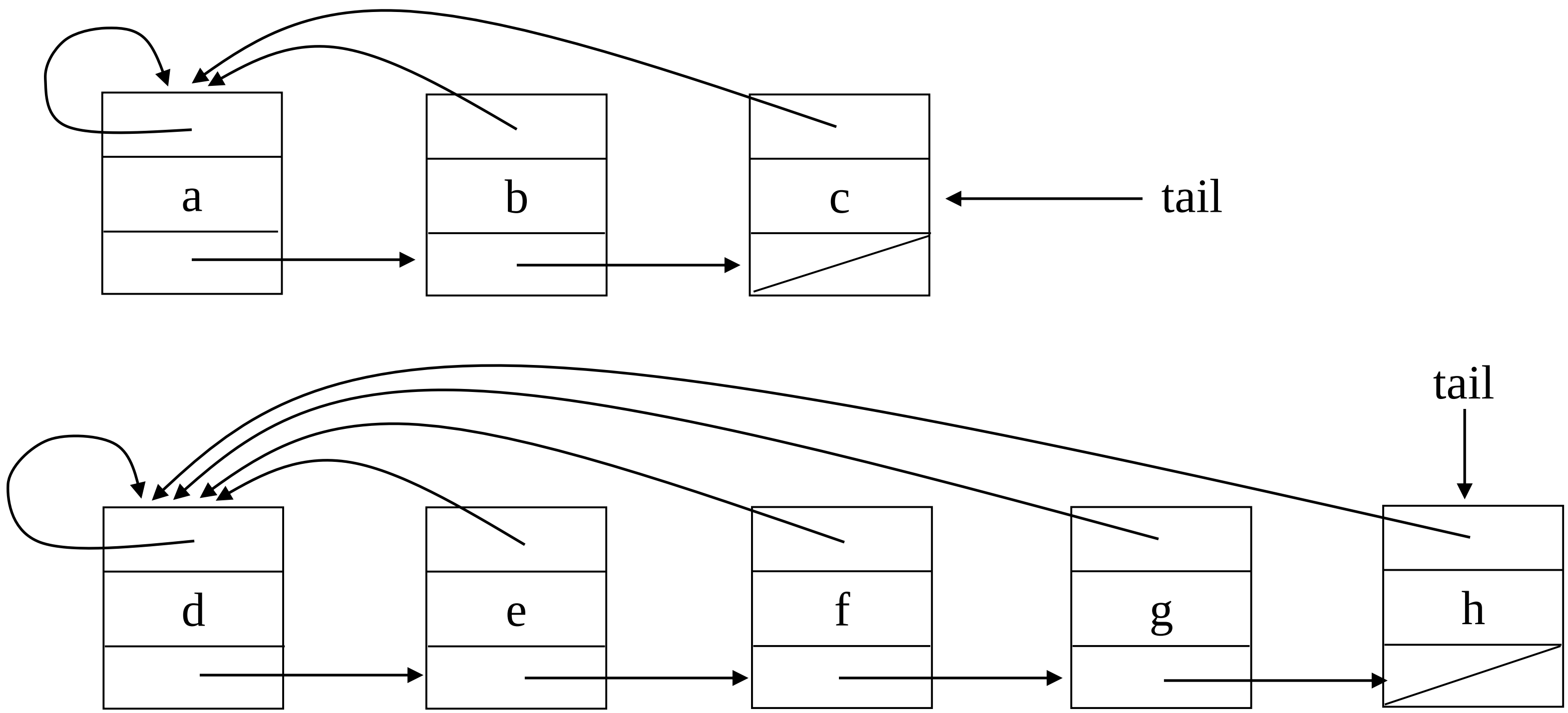
Linked List를 이용한 처리

- 같은 집합의 원소들은 하나의 linked list로 관리한다
- Linked list의 맨 앞의 원소를 집합의 대표 원소로 삼는다

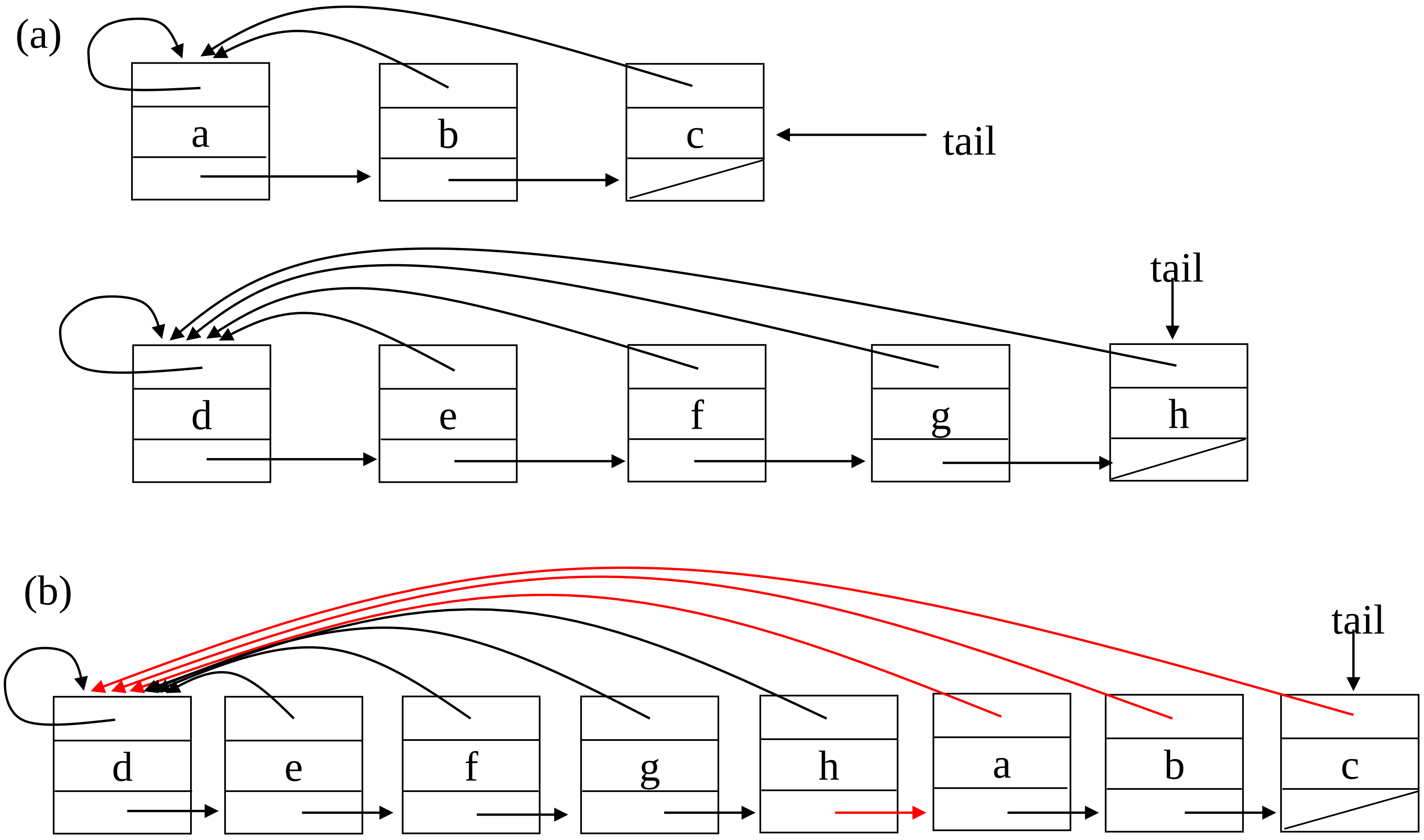
하나의 원소로 이루어진 집합



Linked List로 된 두 집합

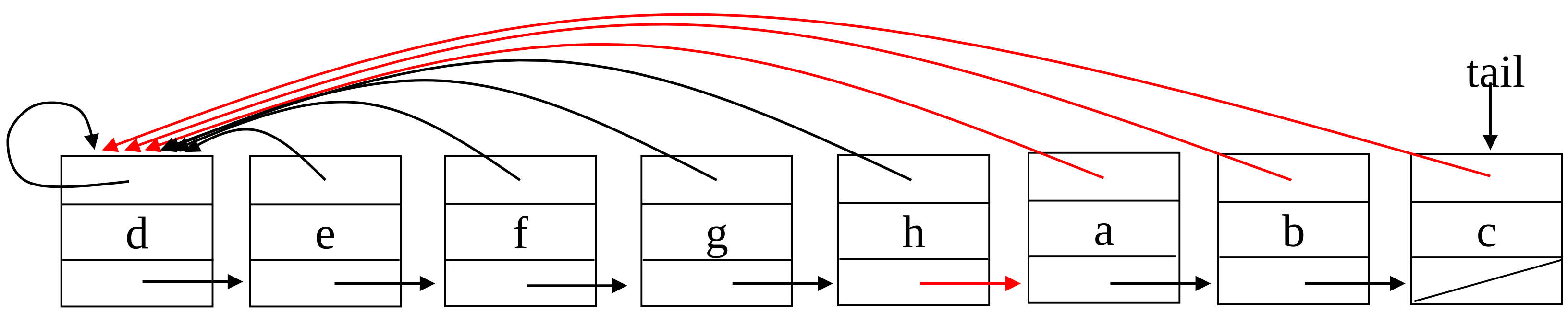


합집합을 만드는 예



Weight을 고려한 Union

- Linked list로 된 두 집합을 합할 때 작은 집합을 큰 집합의 뒤에 붙인다
  - 대표 원소를 가리키는 포인터 갱신 작업을 최소화하기 위한 것



수행시간

[Theorem 1]

Linked list를 이용하는 집합 처리에서 **Weight을 고려한 Union**을 사용할 때, m번의 Make-Set, Union, Find-Set 중 n번이 Make-Set이라면 이들의 총 수행시간은 $O(m + n \log n)$이다.

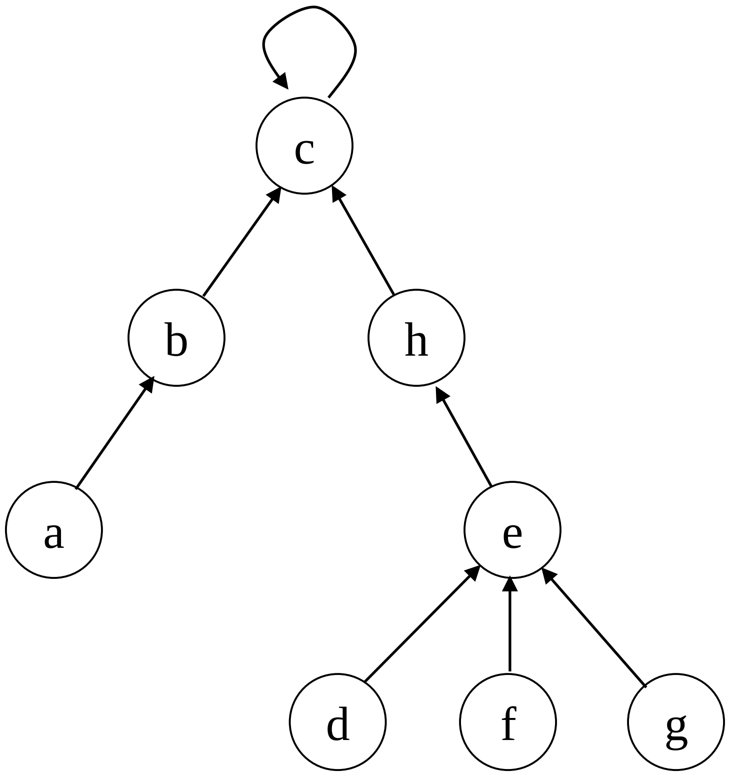
<Proof>

- The dominating cost for union is spent for updating the pointers to the representative.
- For any element x, each time x's pointer is updated (in Union), it belongs to the smaller set. (no update if it belongs to the larger set)
- Therefore, the set size containing x: $1 \rightarrow 2\uparrow \rightarrow 2^2\uparrow \rightarrow \ldots\ 2^k\uparrow$
  - $\therefore$ If the # of elements is n, there can be at most $\log_2 n$ updates for any x.
  - $\therefore$ The total time for update is $O(n \log n)$.
- Since each Make-Set and Find takes $O(1)$ time,
  the total cost for the entire seq. is $O(m + n \log n)$.

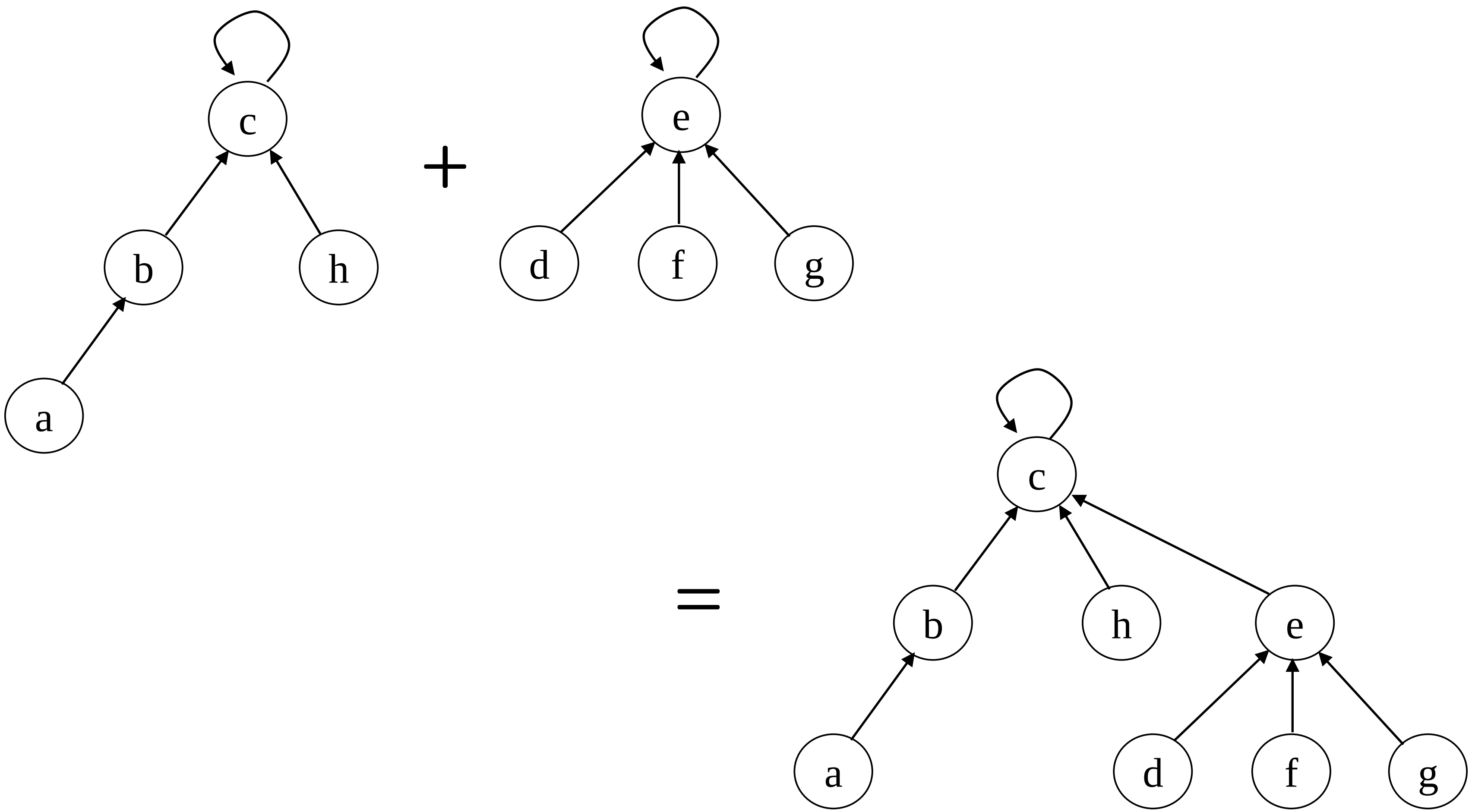
Tree를 이용한 처리

- 같은 집합의 원소들은 하나의 tree로 관리한다
  - child가 parent를 가리킨다
- Tree의 root를 집합의 대표 원소로 삼는다

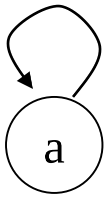
Tree를 이용한 집합 표현의 예



두 집합의 합집합



하나의 원소로 이루어진 집합



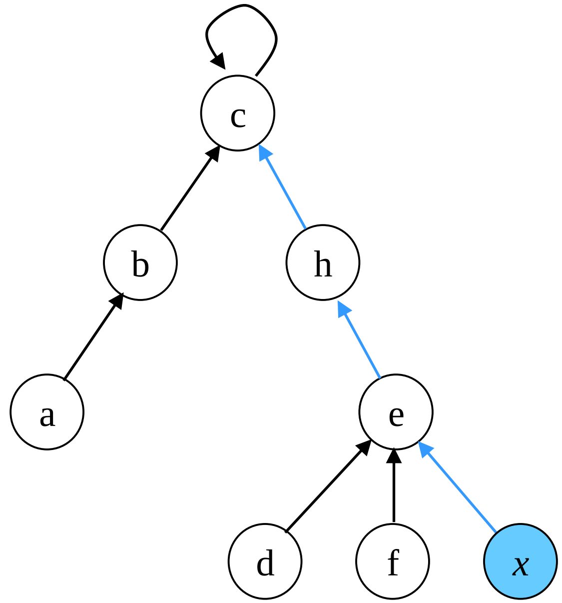
Tree를 이용한 집합 처리 알고리즘

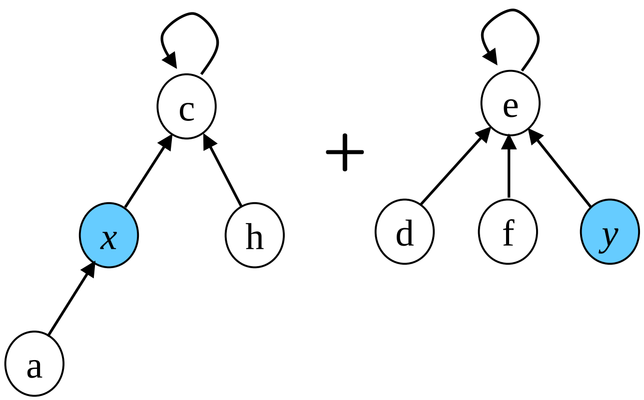
```
Make-Set(x)          ▷ 노드 x를 유일한 원소로 하는 집합을 만든다.
{
    x.parent ← x ;
}

Find-Set(x)          ▷ 노드 x가 속한 집합을 알아낸다.
                       노드 x가 속한 트리의 루트 노드를 리턴한다.
{
    if (x = x.parent)
        then return x ;
        else return Find-Set(x.parent) ;
}

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

이것이 제대로 작동한다는 것을 귀납적으로 생각해보자

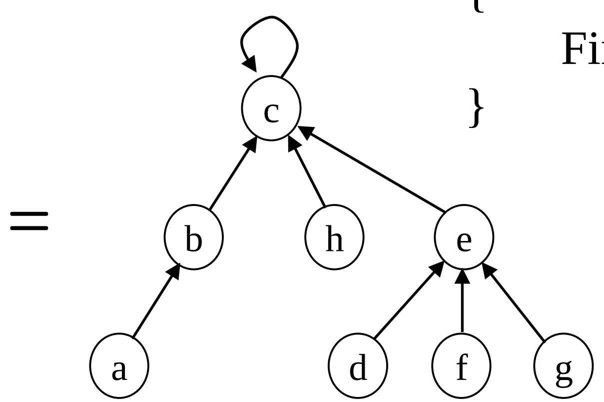
* 편의상 node와 원소를 동일시

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





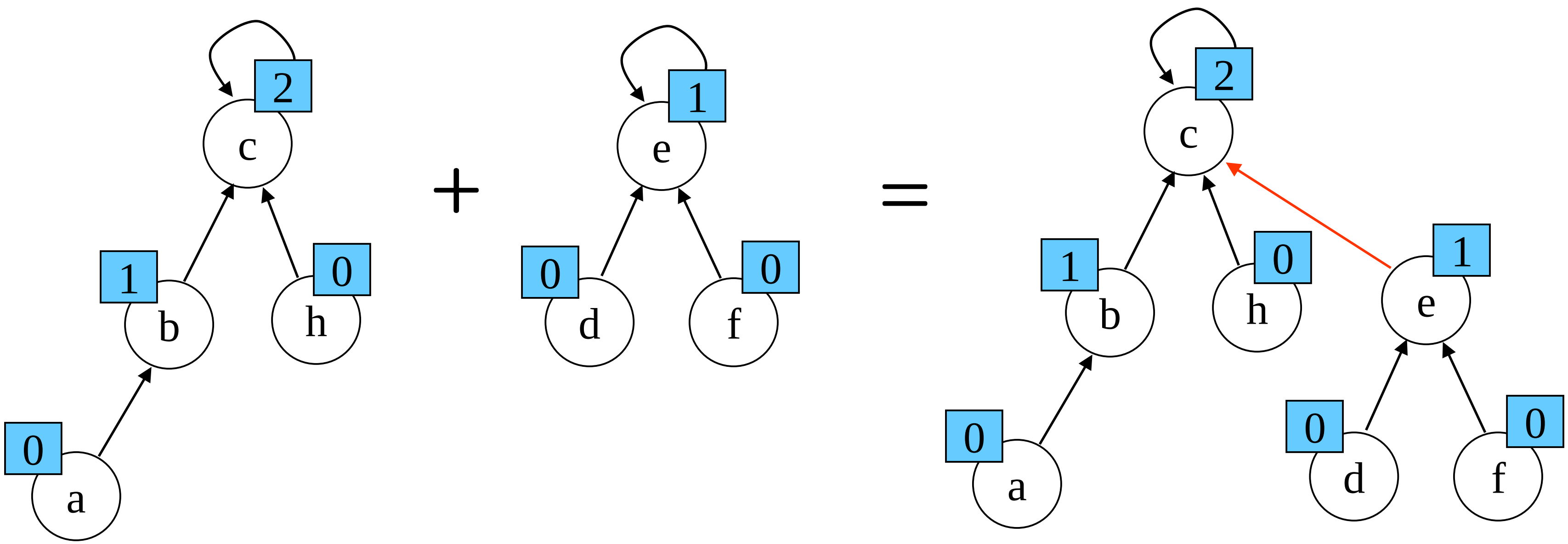

```
Union(x, y)
{
    Find-Set(y).parent ← Find-Set(x);
}
```



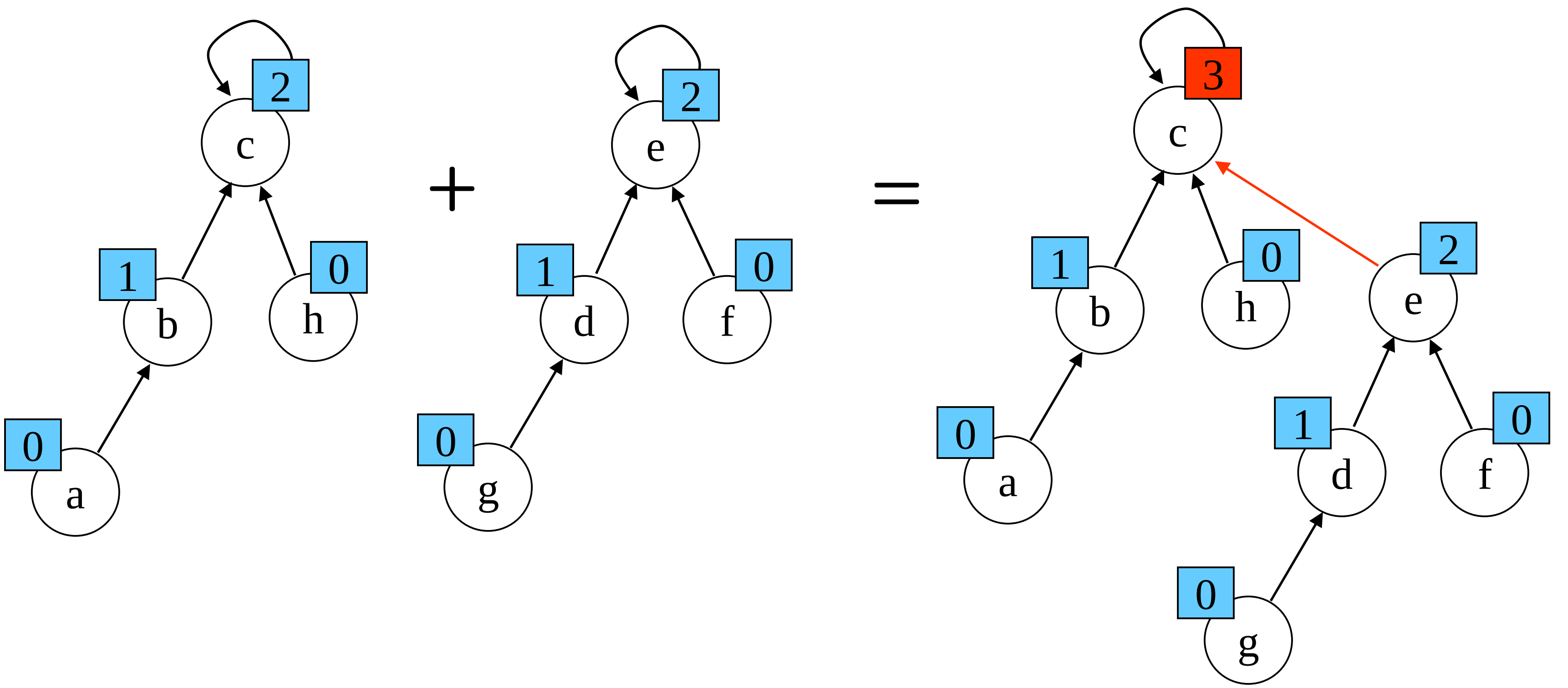
연산의 효율을 높이는 방법

- Rank를 이용한 Union
  - 각 노드는 자신을 루트로 하는 subtree 높이의 상한을 랭크Rank라는 이름으로 저장한다
  - 두 집합을 합칠 때 rank가 낮은 집합을 rank가 높은 집합에 붙인다
- Path compression
  - Find-Set을 행하는 과정에서 만나는 모든 노드들이 직접 root를 가리키도록 포인터를 바꾸어 준다

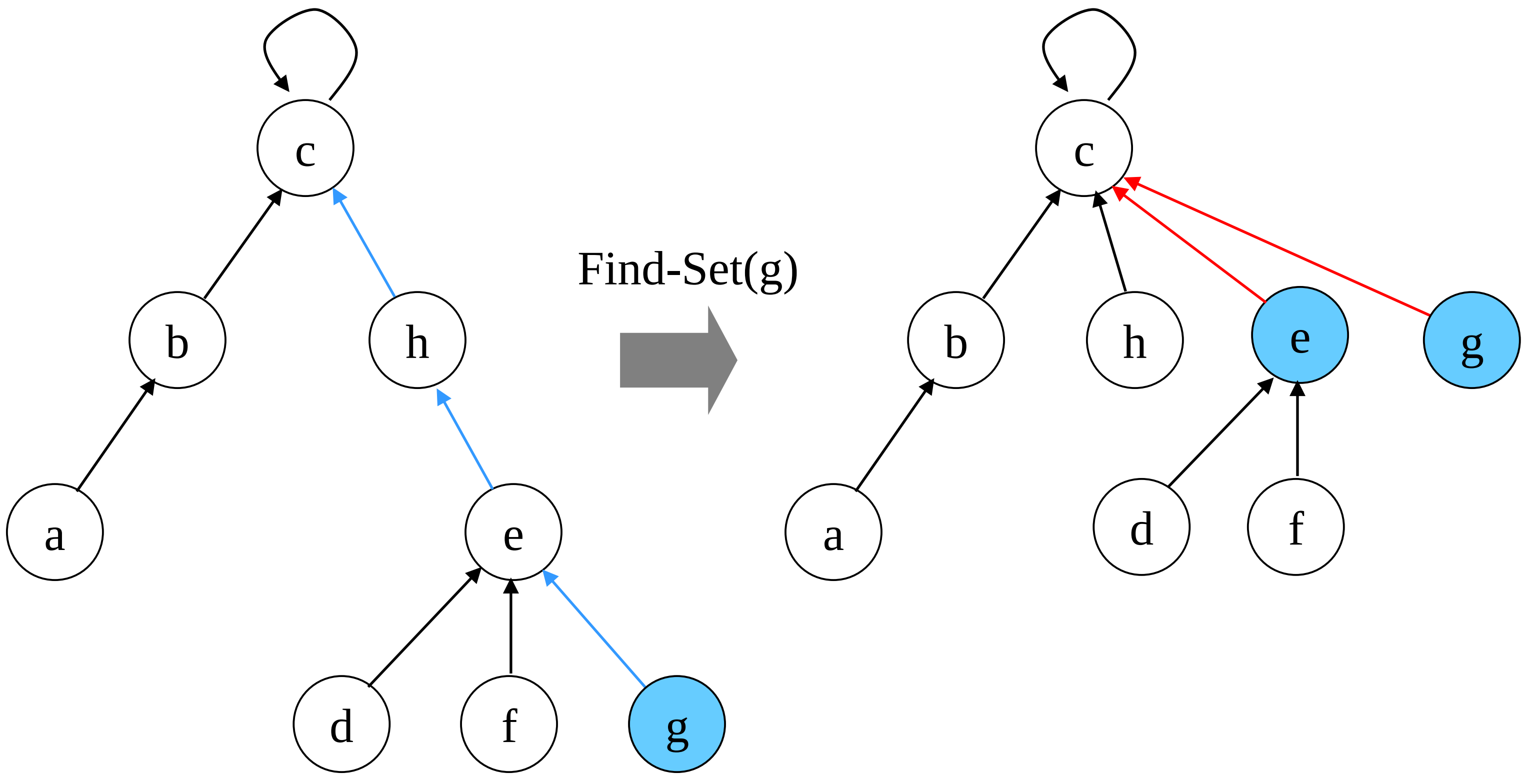
랭크를 이용한 Union의 예



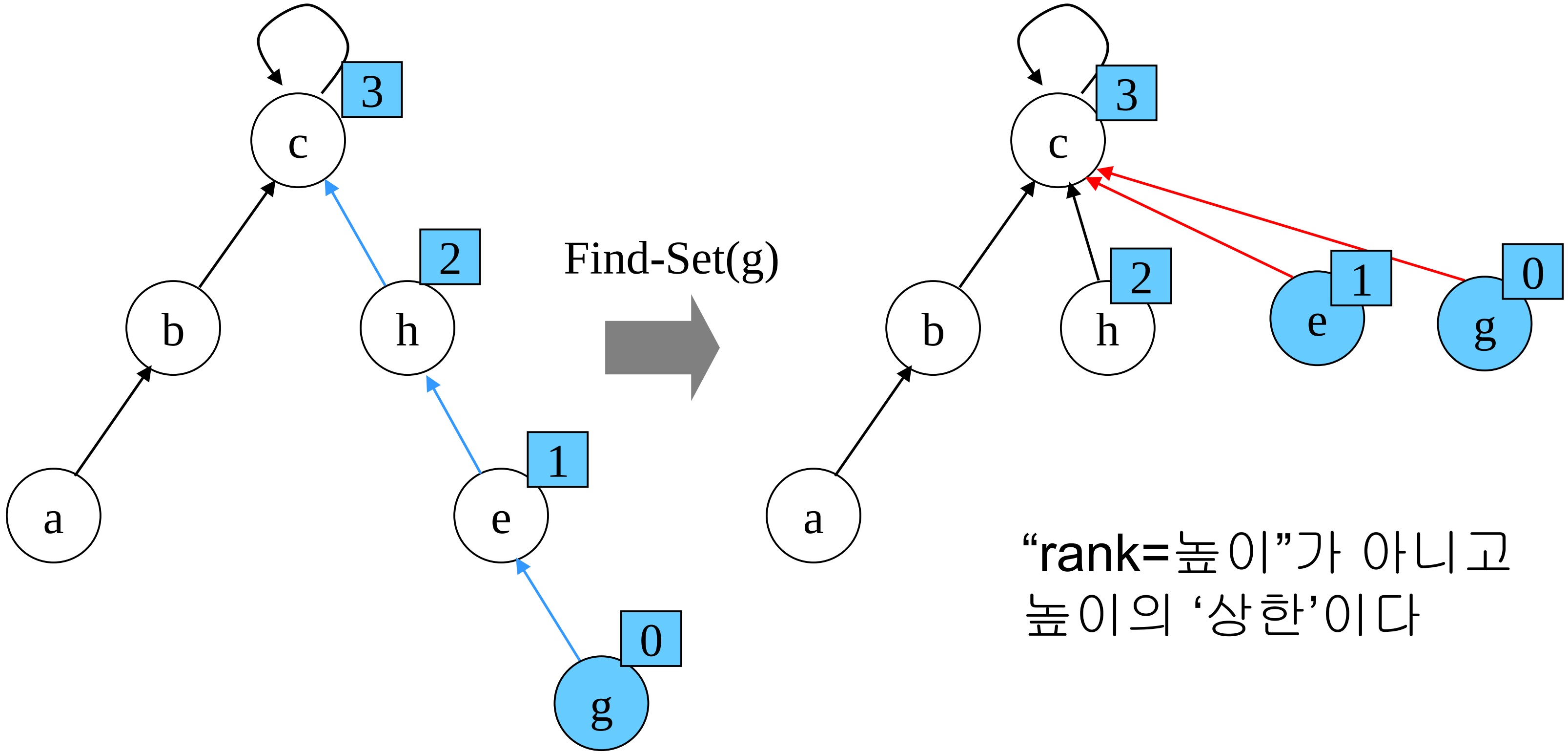
랭크를 이용한 Union에서 랭크가 증가하는 예



Path Compression의 예



다시 Path Compression



"rank=높이"가 아니고
높이의 '상한'이다

Rank를 이용한 Union과 Make-Set

Make-Set(x)
{
    x.parent ← x;
    x.rank ← 0;
}

Union(x, y)
{
    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++;
        }
}

Path Compression을 이용한 Find-Set

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

이것이 제대로 작동한다는 것을
귀납적으로 생각해보자

Originally,

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

수행시간

[Theorem]

Tree를 이용하는 집합 처리에서
랭크를 이용한 Union과 **경로압축을 이용한 Find-Set**을
동시에 사용하면,
m번의 Make-Set, Union, Find-Set 중 n번이 Make-Set일 때
이들의 수행시간은 $O(m \log^* n)$이다.

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

$n = 2^{2^{2^{2^{2}}}}$ 이면 $\log^* n$은 어느 정도?