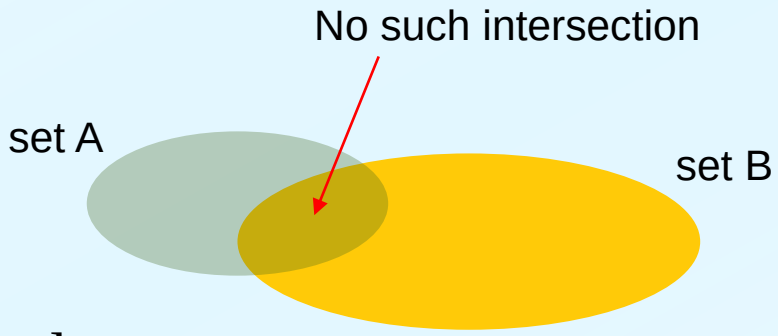
Disjoint Sets

Set Handling

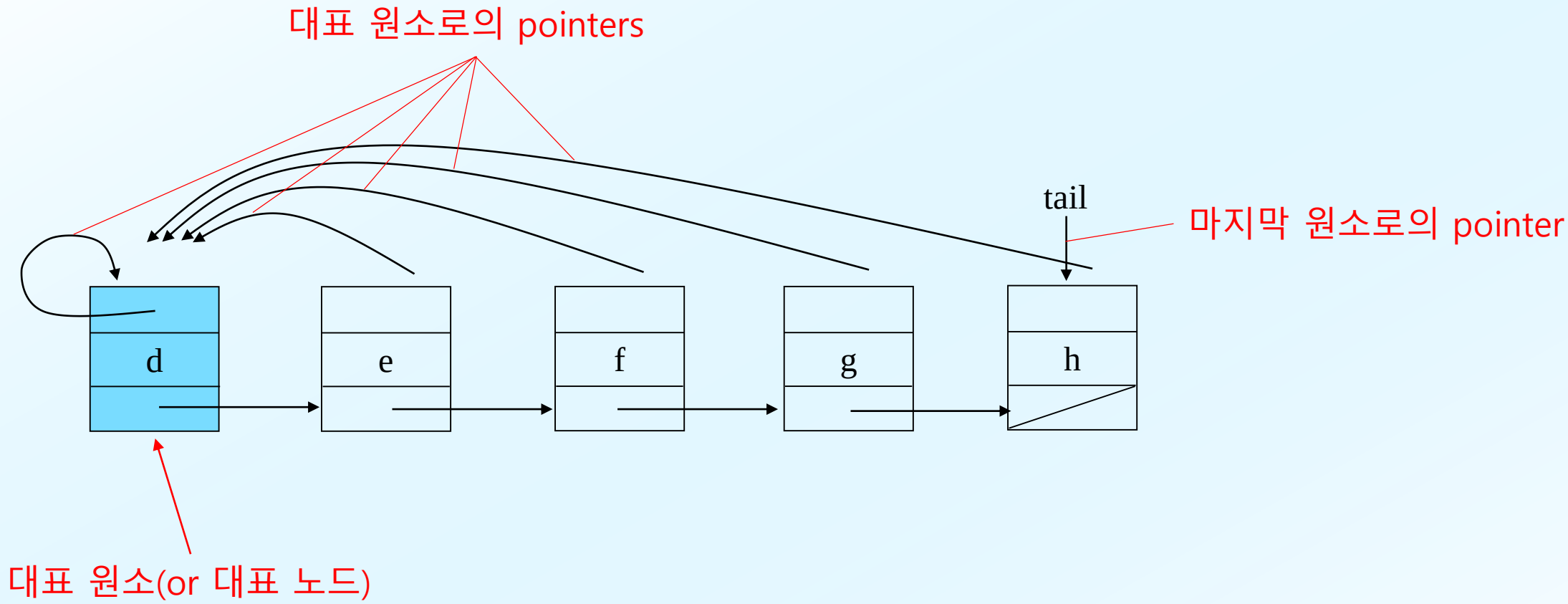


- 제한: disjoint sets, no intersection
- Operations
  - Make-Set(x): Create a singleton set with the only element x
  - Find-Set(x): Find the set to which element x belongs
  - Union(x, y): Merge the two sets of element x's and element y's
- Two Methods
  - Linked list based
  - Tree based

Linked List Based Method

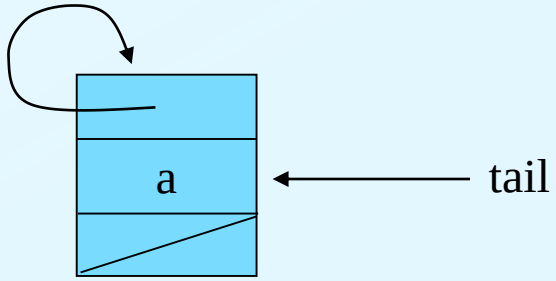
- a set = a linked list
- Linked list에서 맨 앞의 원소가 해당 set의 대표 원소

A Set in a Linked List

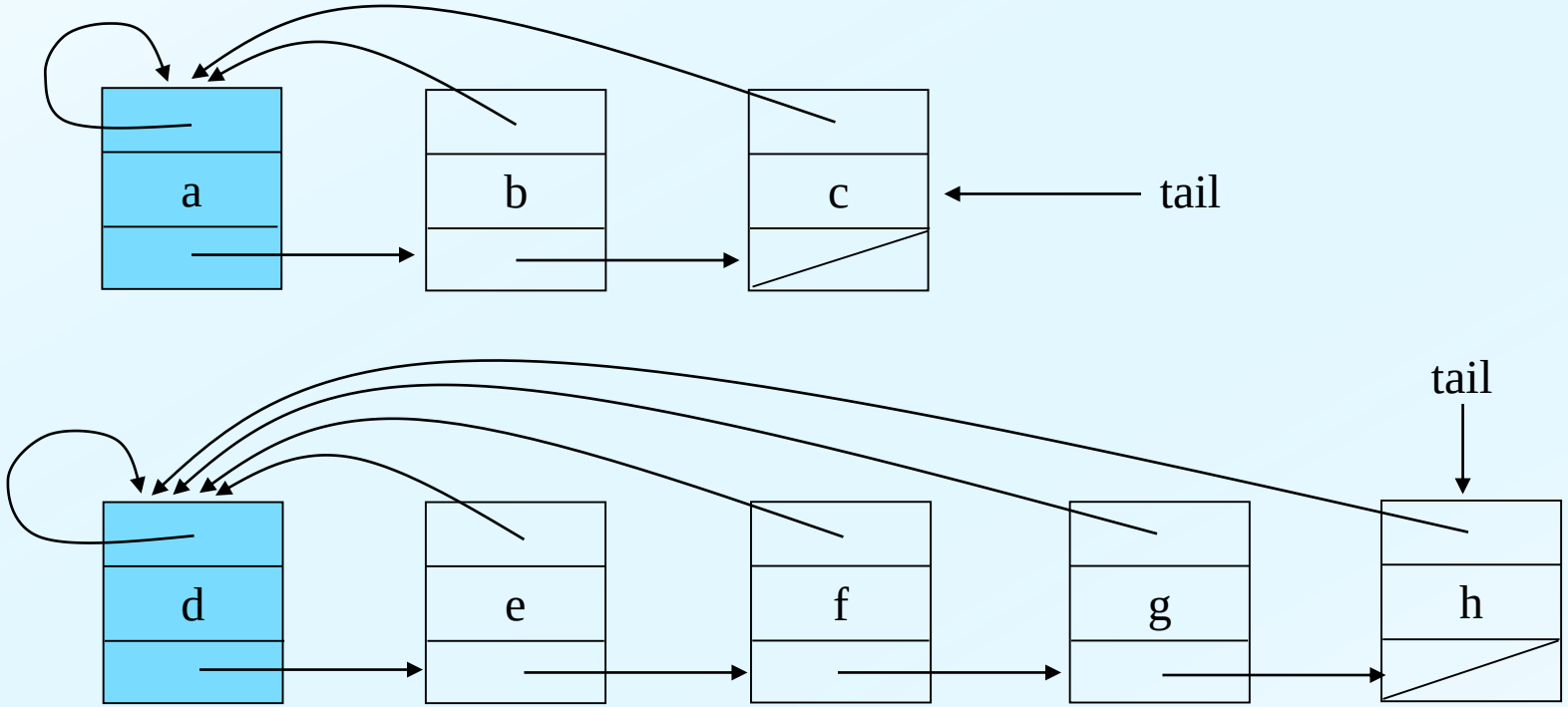


A Singleton Set

Operation: Make-Set

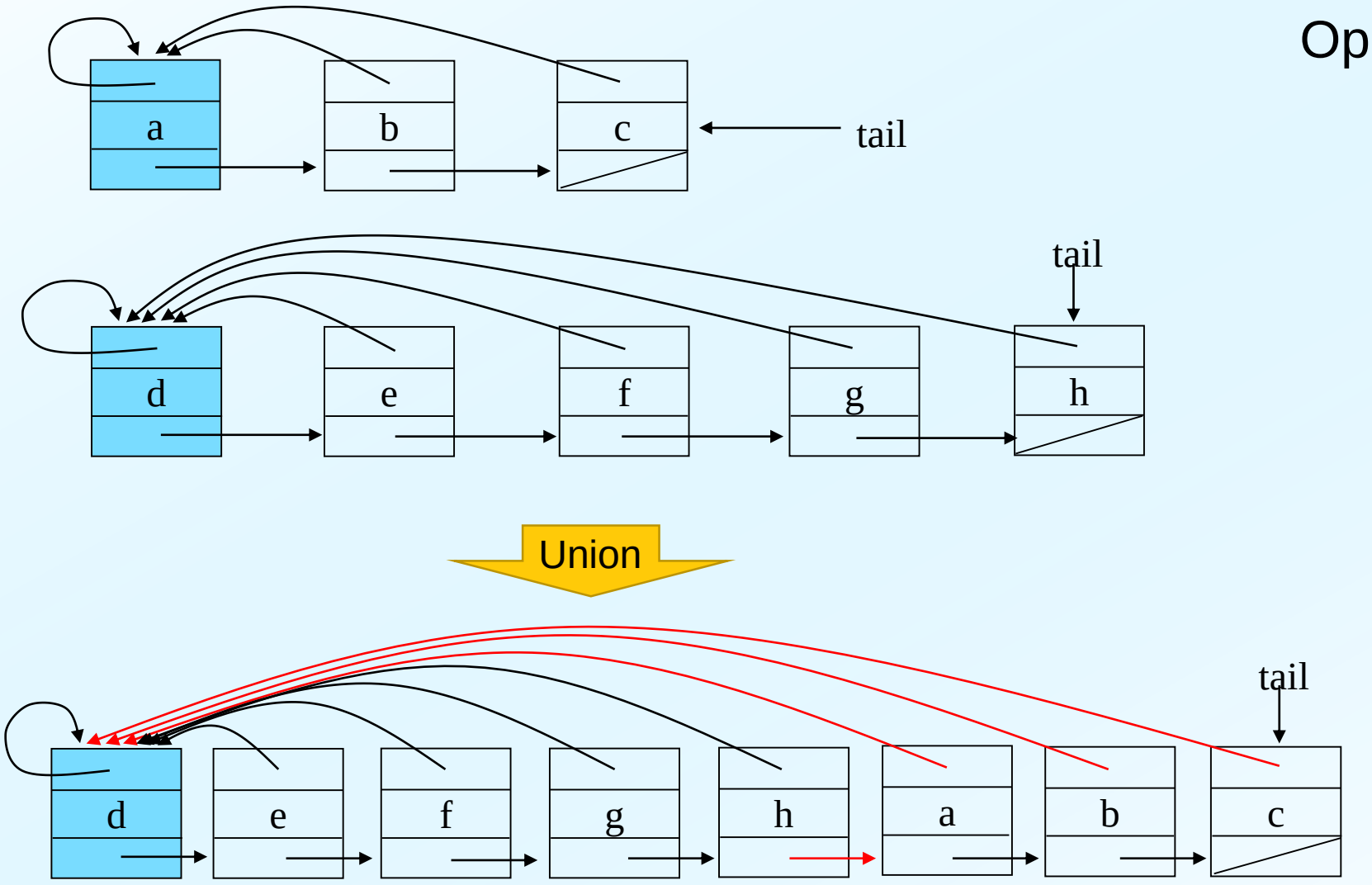


Two Sets in Linked Lists



Union of Two Sets

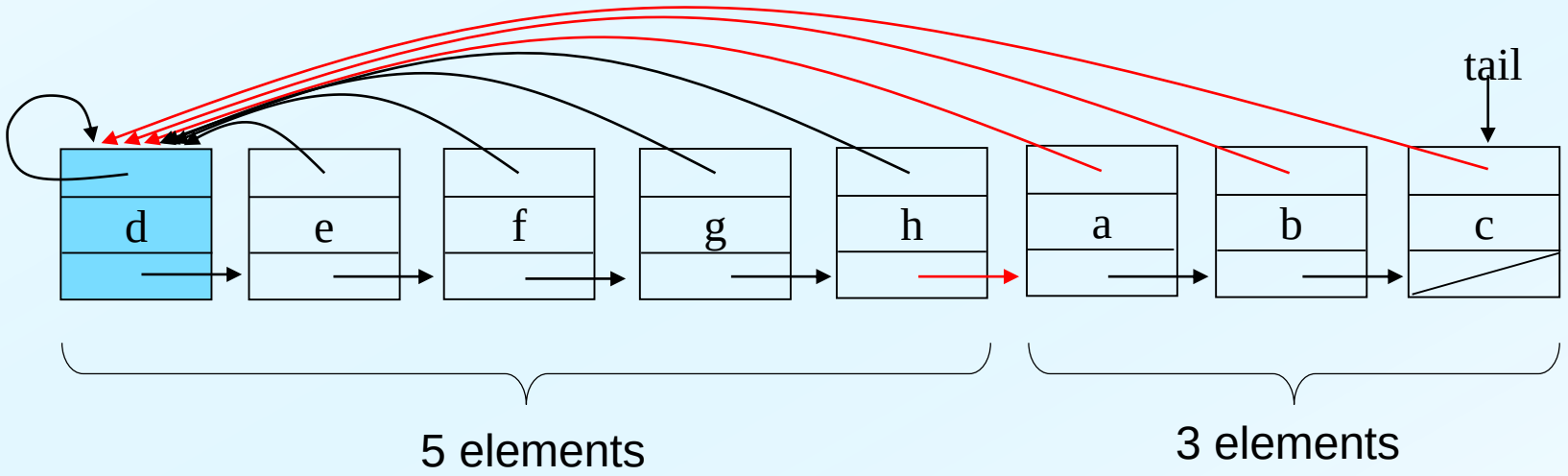
Operation: Union



Weighted Union

In the union of two sets in linked list, append the smaller set to the larger one.

- 대표 원소로의 pointer를 update하는 비용을 최소화하기 위함



Running Time

[Theorem 1]

In the set manipulation by linked lists using **Weighted-Union**, totally m Make-Sets, Unions, and Find-Sets including n Make-Sets, their total running time is $O(m + n\log n)$

<Proof>

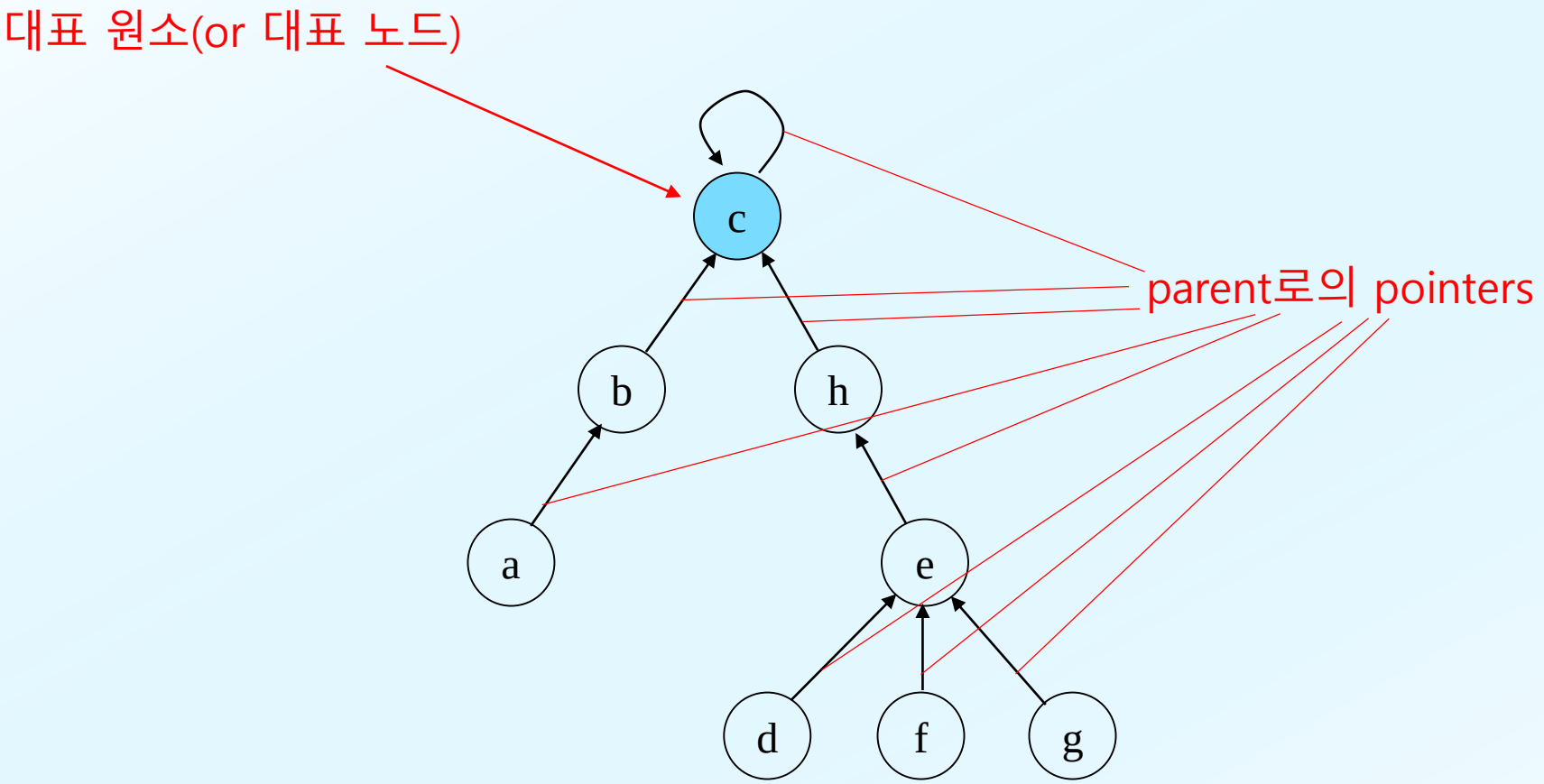
- Union의 cost는 대표 원소로의 update가 결정한다
- Union에서 어떤 원소 x의 pointer(to 대표 원소)가 update될 때마다 x는 두 집합 중 작은 집합에 속한다(if not, 큰 집합에 속해서 no update)
- 따라서, x를 포함하는 집합의 크기는 적어도 $1 \rightarrow 2\uparrow \rightarrow 2^2\uparrow \rightarrow \dots\ 2^k\uparrow$ 의 비율로 커진다
  - $\therefore$ 집합의 원소 수가 n이므로 임의의 원소 x에 $\log_2 n$ 번을 초과하는 update가 일어날 수 없다
  - $\therefore$ For every element, the cost for update is $O(\log n)$
  - $\therefore$ The total time for update is $O(n\log n)$
- Make-Set과 Find는 O(1) 시간 작업이므로,

  m회의 작업 전체 cost는 $O(m + n\log n)$

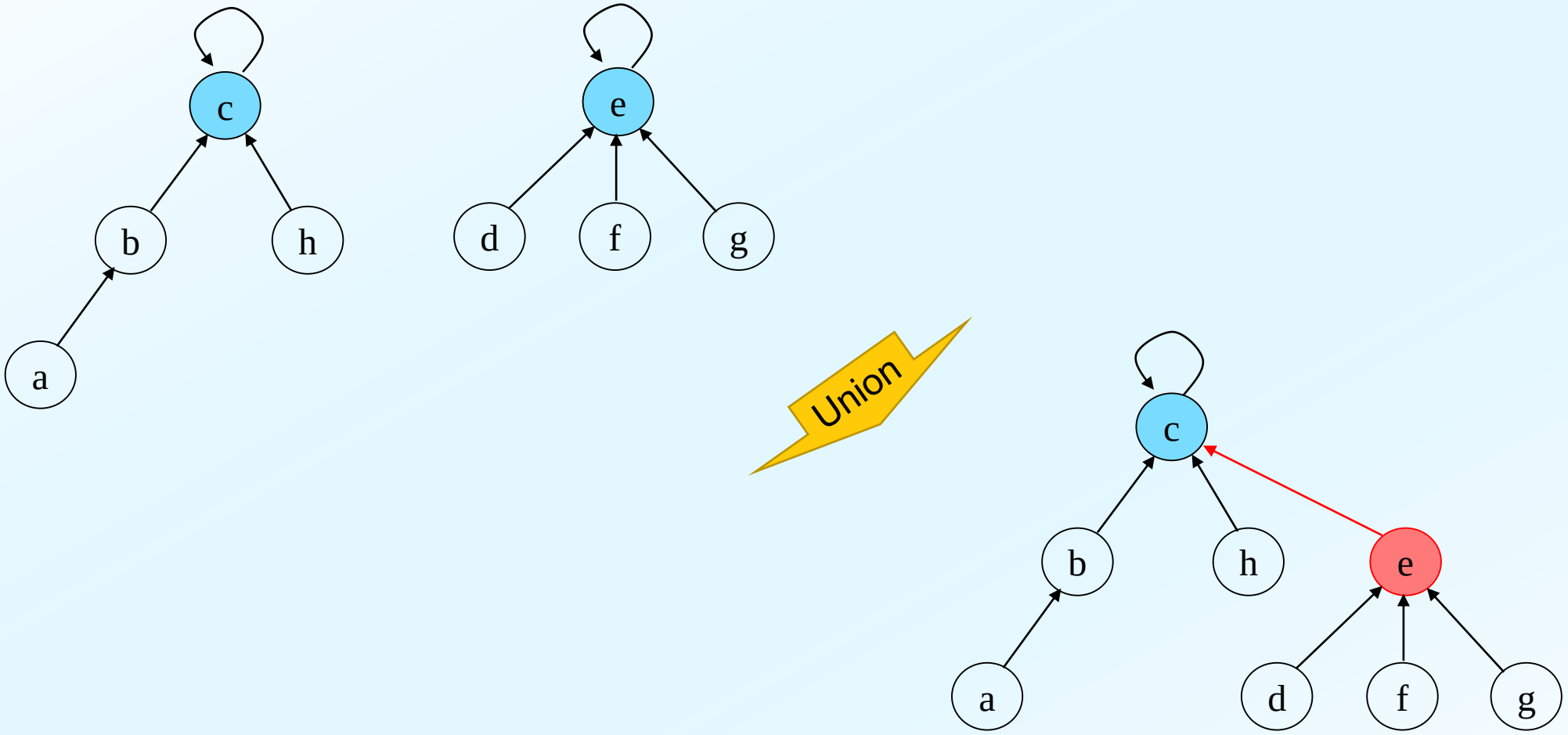
Tree Based Method

- a set = a tree
  - Each child points to its parent
- 트리의 root 원소(or 노드)가 대표 원소

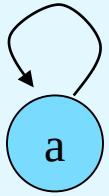
A Set in a Tree



Union of Two Sets



A Singleton Set



Codes of Operations

* 편의상 원소와 노드를 동일시(구현시에는 구분해야)

```
Make-Set(x):          ▷ Make a set with only one element x
    x.parent ← x
```

```
Find-Set(x):          ▷ Find the set to which x belongs. Return the root.
    if (x = x.parent)
        return x
    else
        return Find-Set(x.parent)
```

Inductively think why this works

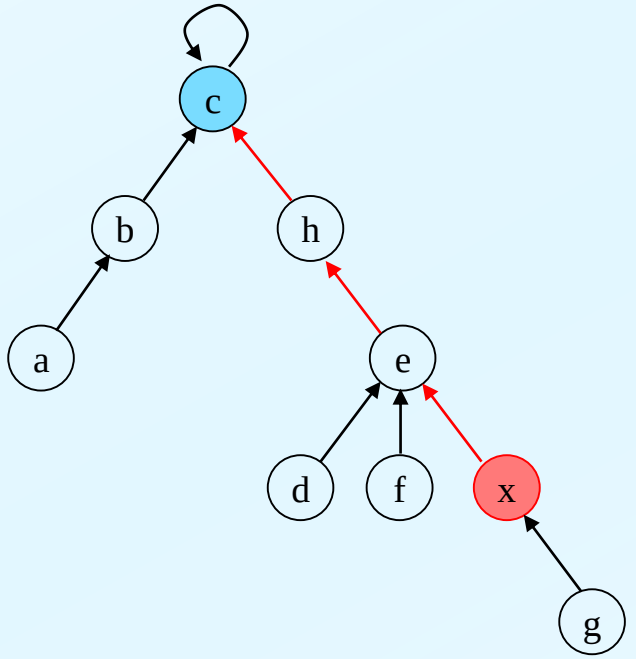
```
Union(x, y):          ▷ Merge the set of y to the set of x
    Find-Set(y).parent ← Find-Set(x)
```

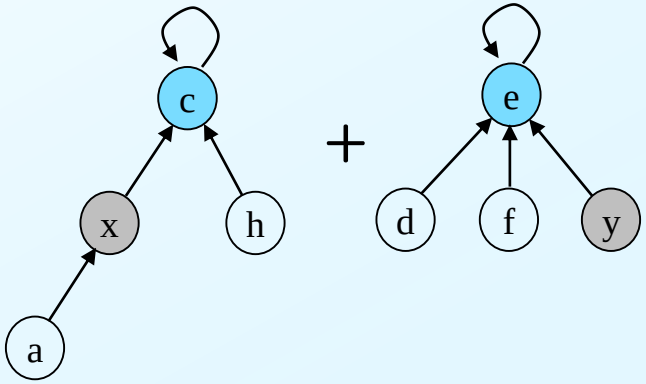
```
Find-Set(x):
    if (x ≠ x.parent)
        return Find-Set(x.parent)
    return x
```

동일

```
Find-Set(x):
    if (x ≠ x.parent)
        return Find-Set(x.parent)
    else
        return x
```

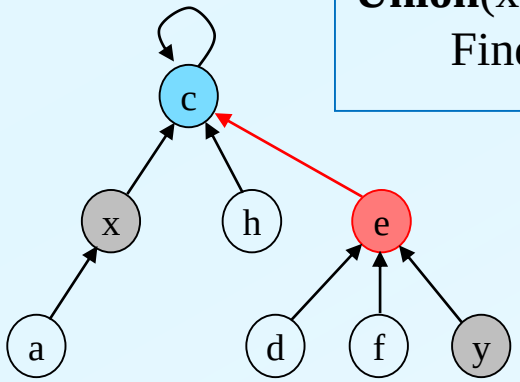
나중에 Pass Compression을 하는 Find-Set을 위해 모양을 미리 맞춤





+

=




```
Union(x, y):
    Find-Set(y).parent ← Find-Set(x)
```

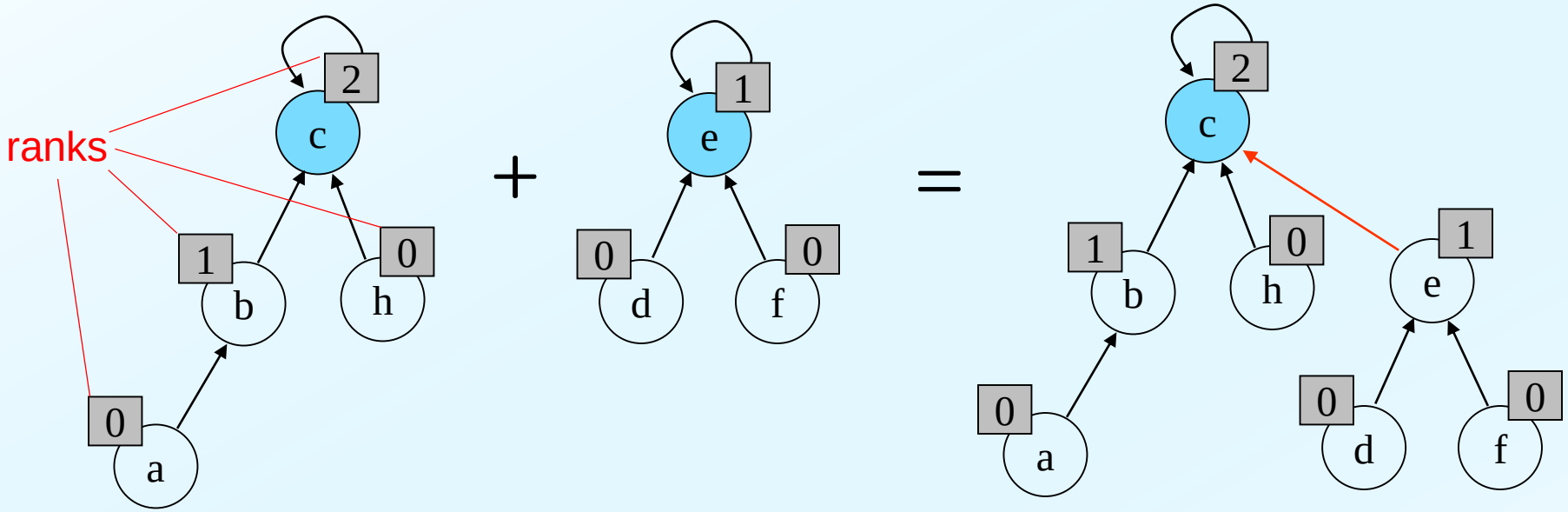
If we restrict the arguments to the roots(representative)

```
Union(c, e):
    e.parent ← c
```

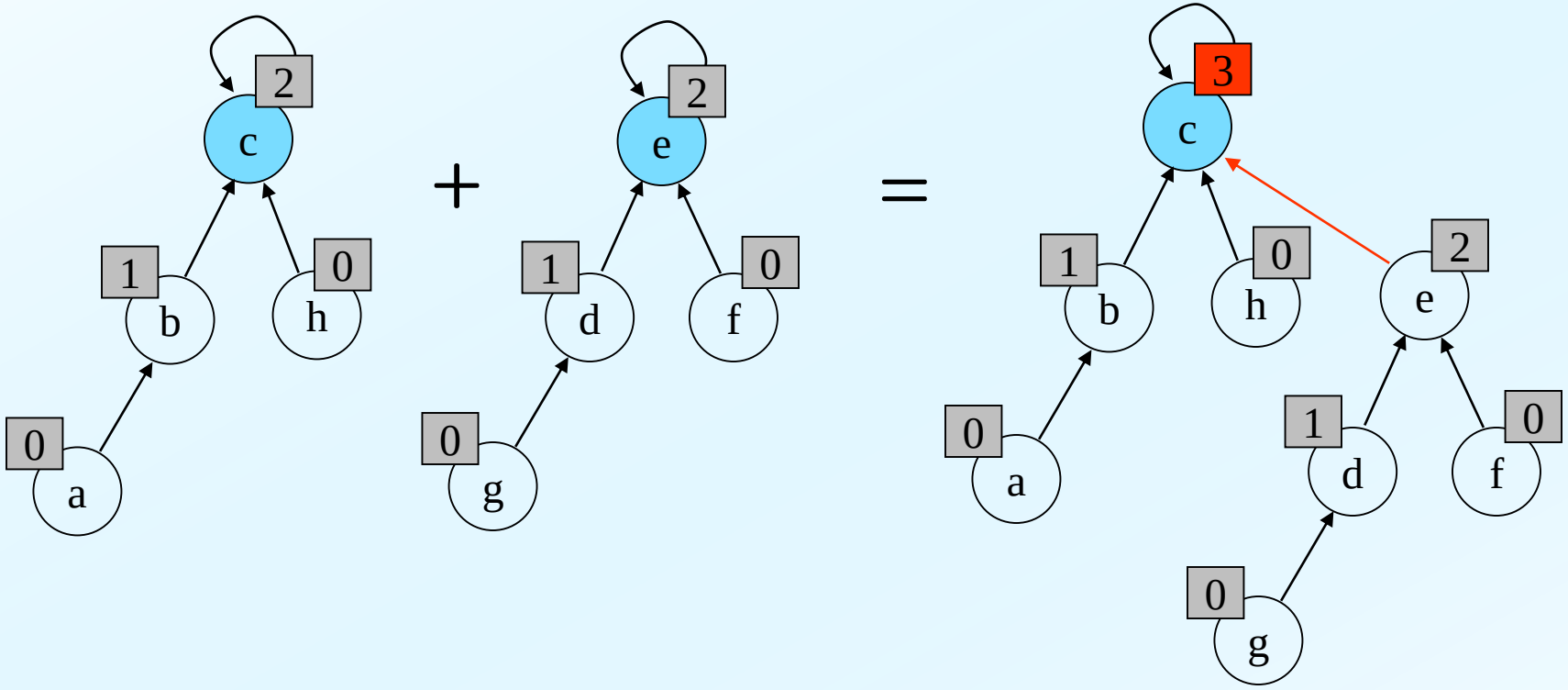
More Efficient Implementations

- Union by Rank
  - Each node has a rank, an **upper bound** of the **height** of the subtree rooted by itself
  - The union of two sets attaches the low-rank set(tree) to the high-rank one(tree)
- Path Compression in Find-Set(x)
  - Change the parents of all nodes in the path from x to the root
    to directly point the root(representative)

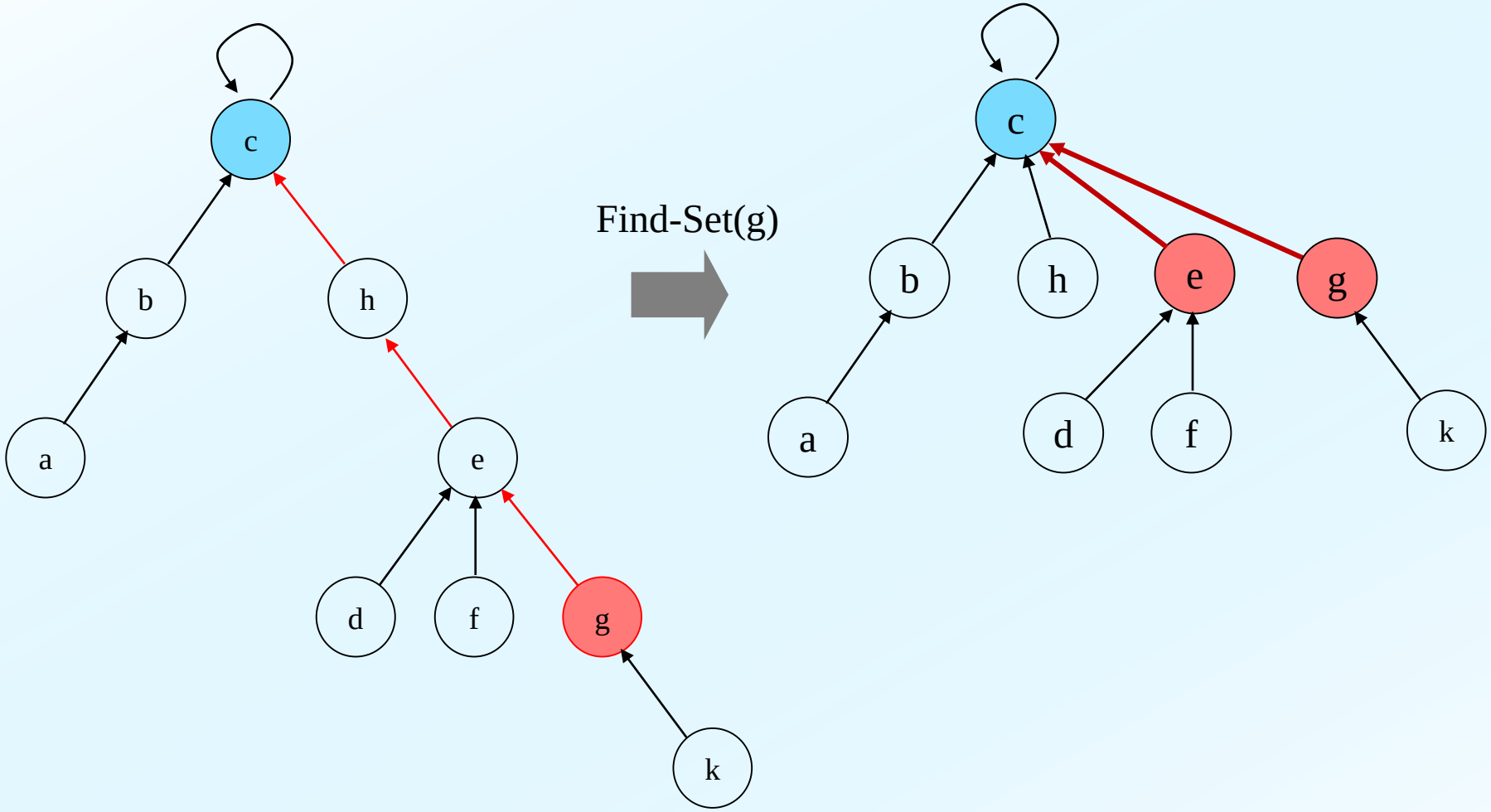
Example: Union by Rank



Example: Rank Increase in Union by Rank

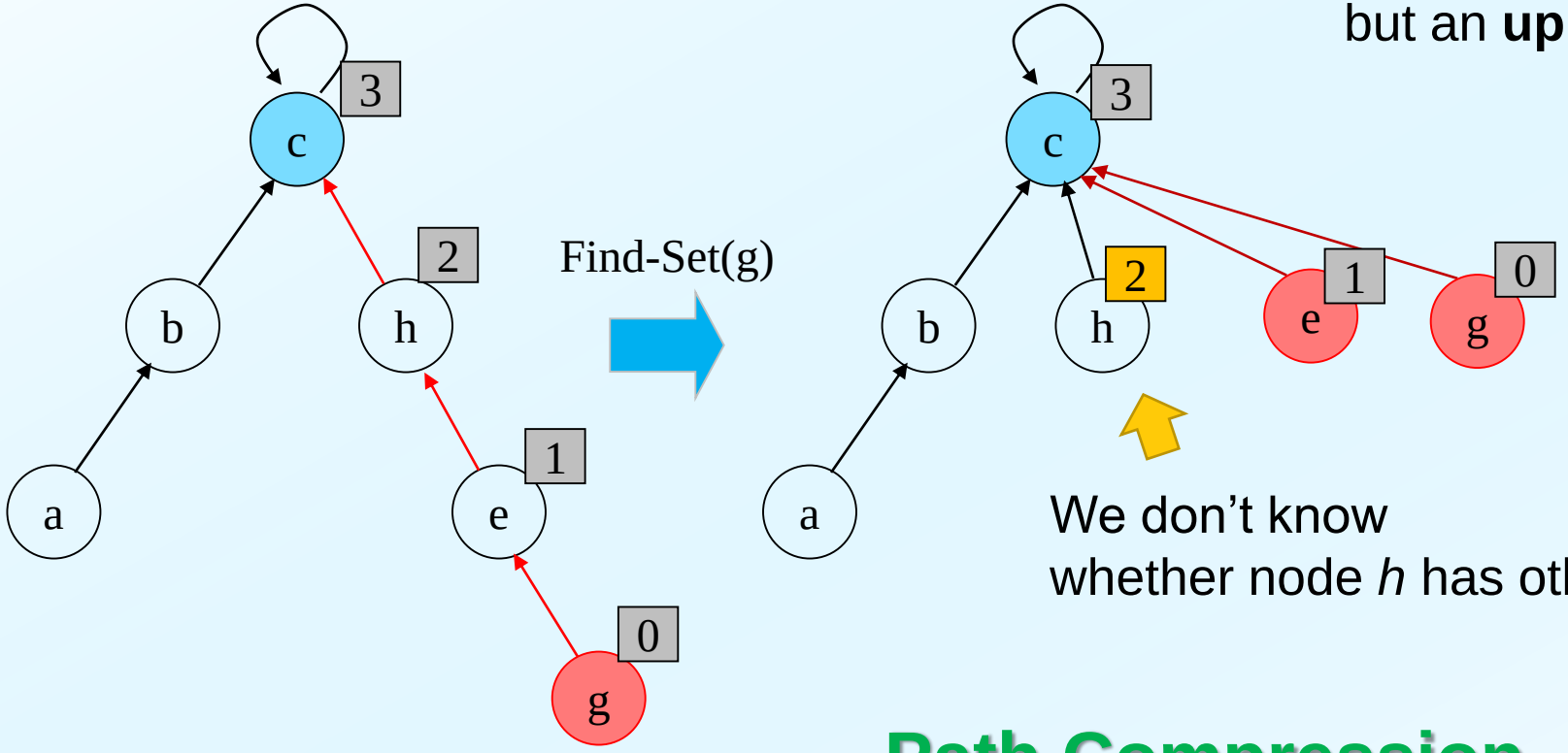


Example: Path Compression



Note: Rank is an *Upper Bound*

rank is not an exact height
but an **upper bound**



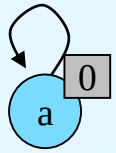
We don't know
whether node *h* has other children

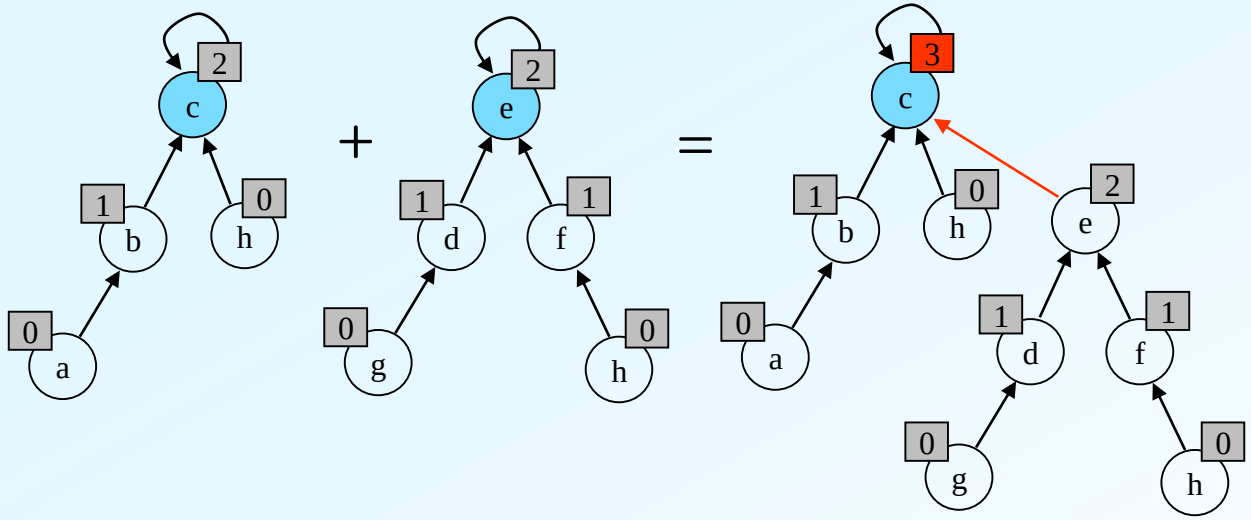
Path Compression

Codes: Union by Rank and 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)
        y′.parent ← x′
    else
        x′.parent ← y′
        if (x′.rank = y′.rank)
            y′.rank++
```

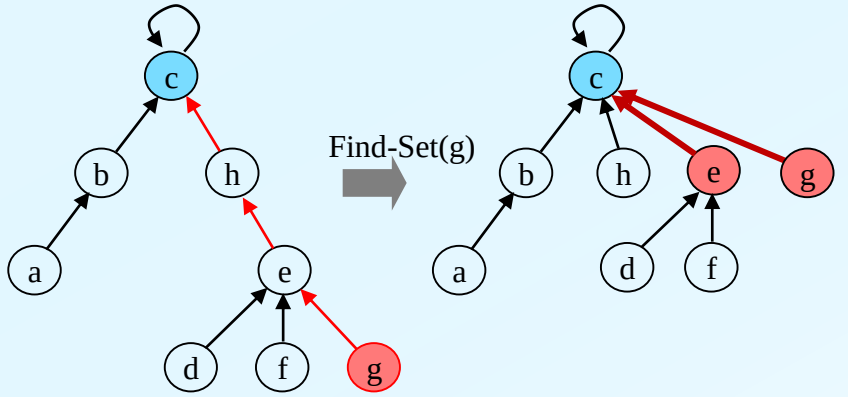




Codes: Find-Set with Path Compression

```
Find-Set(x):
    if (x ≠ x.parent)
        x.parent ← Find-Set(x.parent)
    return x.parent
```

Inductively think why this works



Originally,

```
Find-Set(x):
    if (x ≠ x.parent)
        return Find-Set(x.parent)
    else
        return x
```

Running Time

[Theorem]

In a tree-based set manipulation, if we use both **Union by Rank** and **Find-Set with PathCompression** together, totally m Make-Sets, Unions, and Find-Sets including n Make-Sets, their total running time is $O(m \log^* n)$.

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

← *de facto* constant

$$n = 2^{2^{2^{2^2}}} \quad : \log^* n = 5$$