Data Structures for Disjoint Sets

Heejin Park

Hanyang University

Contents

- Disjoint-sets
- Disjoint-set operations
- An application of disjoint-set data structures
- Disjoint-set data structures

Disjoint sets

- ***Disjoint sets***
  - Two sets A and B are disjoint if $A \cap B = \{\}$.
    Ex> $A = \{1, 2\}, B = \{3, 4\}$
  - Sets $S_1, S_2, \ldots, S_k$ are disjoint
    if every two distinct sets S_i and S_j are disjoint.
    Ex> $S_1 = \{1, 2, 3\},\ S_2 = \{4, 8\},\ S_3=\{5,7\}$

S_1, S_3
S_2, S_3
S_1, S_2

Disjoint sets

- A ***collection*** of disjoint sets
  - A set of disjoint sets is called a collection of disjoint sets.

    Ex> {{1, 2, 3}, {4, 8}, {5,7}}

  - Each set in a collection has a ***representative member*** and the set is identified by the member.

    Ex> {{1, 2, 3}, {4, 8}, {5,7}}

나머지 숫자

representative number 가능

Disjoint sets

- A collection of ***dynamic disjoint sets***
  - **Dynamic**: Sets are changing.
    - New sets are created.
      - $\{\{1, 2, 3\}, \{4, 8\}, \{5,7\}\} \rightarrow \{\{1, 2, 3\}, \{4, 8\}, \{5,7\}, \{9\}\}$
    - Two sets are united.
      - $\{\{1, 2, 3\}, \{4, 8\}, \{5,7\}\} \rightarrow \{\{1, 2, 3\}, \{4, 8, 5, 7\}\}$

Disjoint-set operations

- **Disjoint-set operations**
  - **MAKE-SET(*x*)** : X를 포함하는 set 생성
  - **UNION(*x*, *y*)**
  - **FIND-SET(*x*)** : Find representative member

Disjoint-set operations

- **MAKE-SET(*x*)**
  - Given a member *x*, generate a set for *x*.
  - MAKE-SET(9)

    {{1, 2, 3}, {4, 8}, {5,7}} ➔ {{1, 2, 3}, {4, 8}, {5,7}, {9} }

'9 포함 set 생성

9 → representative 됨

Disjoint-set operations

representative member

- **UNION(x, y)**
  - Given two members x and y, unite the set containing x and another set containing y.
  - UNION(1,4)
  - {{1, 2, 3}, {4, 8}, {5,7}} → {{1, 2, 3, 4, 8}, {5,7}}
- **FIND-SET(x)**
  - Find the representative of the set containing x.
  - FIND-SET(5): 7

Disjoint-set data structures

- **Problem**
  - ***Developing data structures*** to maintain a collection of dynamic disjoint sets supporting disjoint-set operations, which are MAKE-SET(x), UNION(x,y), FIND-SET(x).

Disjoint-set data structures

- **Parameters for running time analysis**
  - #Total operations: m
  - #MAKE-SET ops: n
  - #UNION ops: u
  - #FIND-SET ops: f
  - $m = n + u + f$

Disjoint-set data structures

union ≤ make −1

- $u \leq n - 1$

→ make로 n개의 set → 최대 n-1 union

  - n is the number of sets generated by MAKE-SET ops.
  - Each UNION op reduces the number of sets by 1. → set 개수 1씩 감소
  - So, after n-1 UNION ops, we have only 1 set and then we cannot do UNION op more.

u=n-1 → 1개의 set만 존재

Assumption

- The first n operations are MAKE-SET operations.

처음 n개의 operation → MAKE-SET만 존재한다고 가정

Contents

- Disjoint-sets
- Disjoint-set operations
- An application of disjoint-set data structures
- Disjoint-set data structures

Application

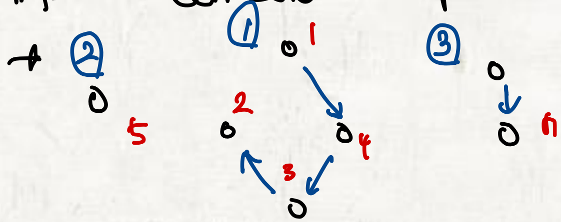
{{1,2,3,4},{5},{6,7}}

- **Computing connected components (CC)**

→ Connected components의 개수 세기

- Static graph → 변하지 않는 그래프
  - Depth-first search: $\Theta(V+E)$

DFS로 connected component



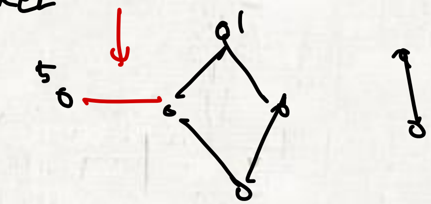
- Dynamic graph
  - Depth-first search is inefficient. → G 이상 | 진행 불가능할 때까지

    → 하나의 connected component
  - Maintaining a disjoint-set data structure is more efficient.

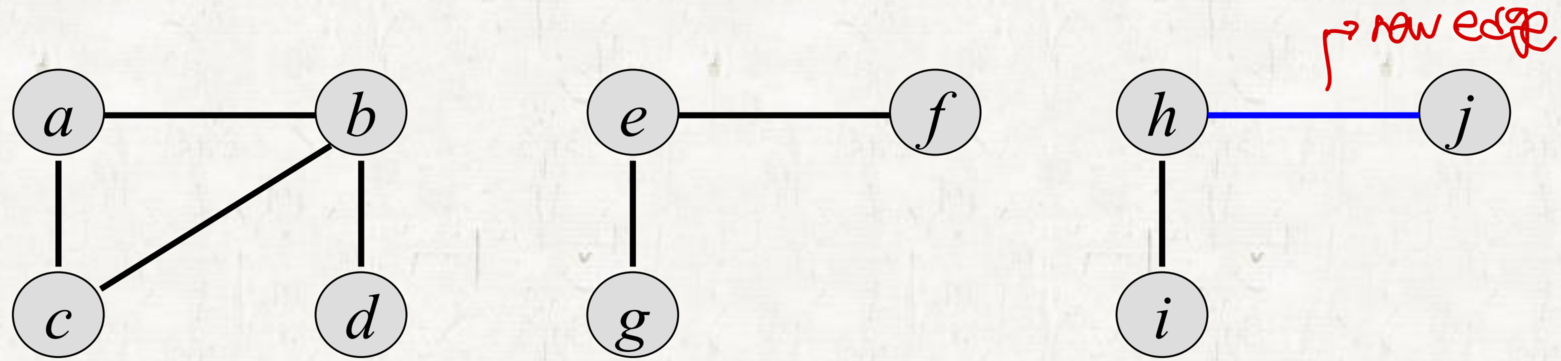
DFS시 1번 vertex와 5번 vertex에서 같은 곳을 2번 탐색

→ 비효율적

adding new edge



Connected component computation



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

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

위와같이 새로운 edge가 추가되면
DFS는 매번 실행되어야 하지만
Disjoint-set data structure은
UNION만 실행하면 된다.

Depth first search: $\Theta(V+E)$

Disjoint-set data structures: UNION(h, j)

Connected component computation

- **Computing CC using disjoint set operations**

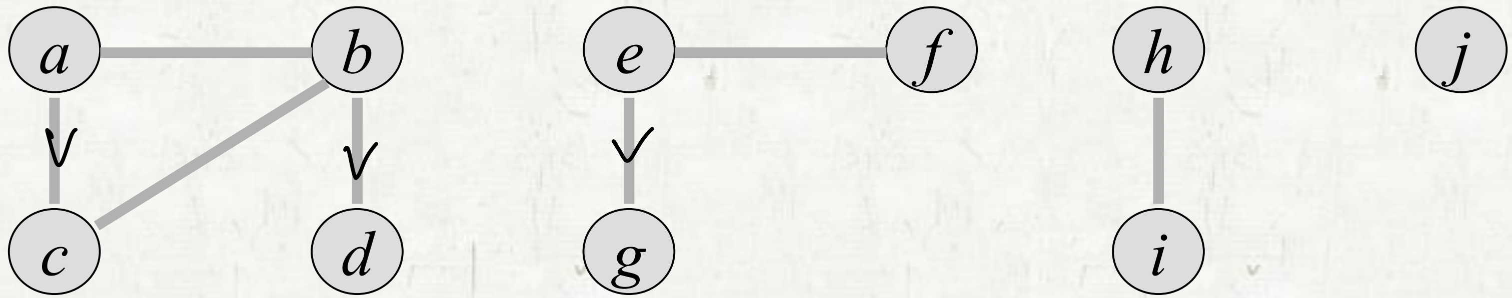
CONNECTED-COMPONENTS(*G*)

```
for each vertex v ∈ G.V
    MAKE-SET(v)
for each edge (u, v) ∈ G.E
    if FIND-SET(u) ≠ FIND-SET(v)
        UNION(u, v)
```

$\{a\}, \{b\}, \{c\}, \ldots \{i\}$

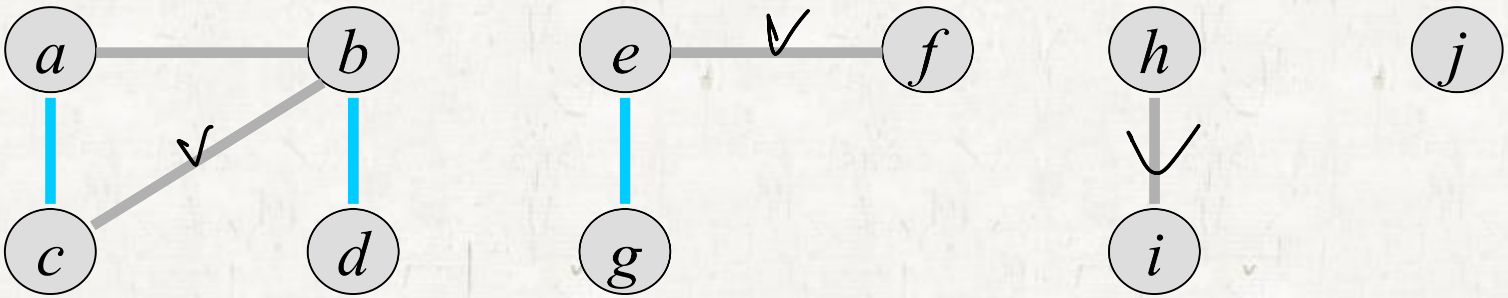
둘이 이미 같은 Set에 존재하는지 확인

Connected component computation



Initial sets	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b,d)	$\{a\}$	$\{b,d\}$	$\{c\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$	
(e,g)	$\{a\}$	$\{b,d\}$	$\{c\}$	$\{e,g\}$	$\{f\}$	$\{h\}$	$\{i\}$	$\{j\}$		
(a,c)	$\{a,c\}$	$\{b,d\}$	$\{e,g\}$	$\{f\}$	$\{h\}$	$\{i\}$	$\{j\}$			

Connected component computation



(*a*,*c*)	{*a*,*c*}	{*b*,*d*}	{*e*,*g*}	{*f*}	{*h*}	{*i*}	{*j*}
(*h*,*i*)	{*a*,*c*}	{*b*,*d*}	{*e*,*g*}	{*f*}	{*h*,*i*}		{*j*}
(*a*,*b*)	{*a*,*b*,*c*,*d*}		{*e*,*g*}	{*f*}	{*h*,*i*}		{*j*}
(*e*,*f*)	{*a*,*b*,*c*,*d*}		{*e*,*f*,*g*}		{*h*,*i*}		{*j*}
(*b*,*c*)	{*a*,*b*,*c*,*d*}		{*e*,*f*,*g*}		{*h*,*i*}		{*j*}

Connected component computation

```
SAME-COMPONENT(u, v)
if FIND-SET(u) == FIND-SET(v)
        return TRUE
else return FALSE
```

Connected-Component를 계산하면 SAME-COMPONENT로 확인할 수 있다.

Contents

- Disjoint-sets
- Disjoint-set operations
- An application of disjoint-set data structures
- **Disjoint-set data structures**

Disjoint-set data structures

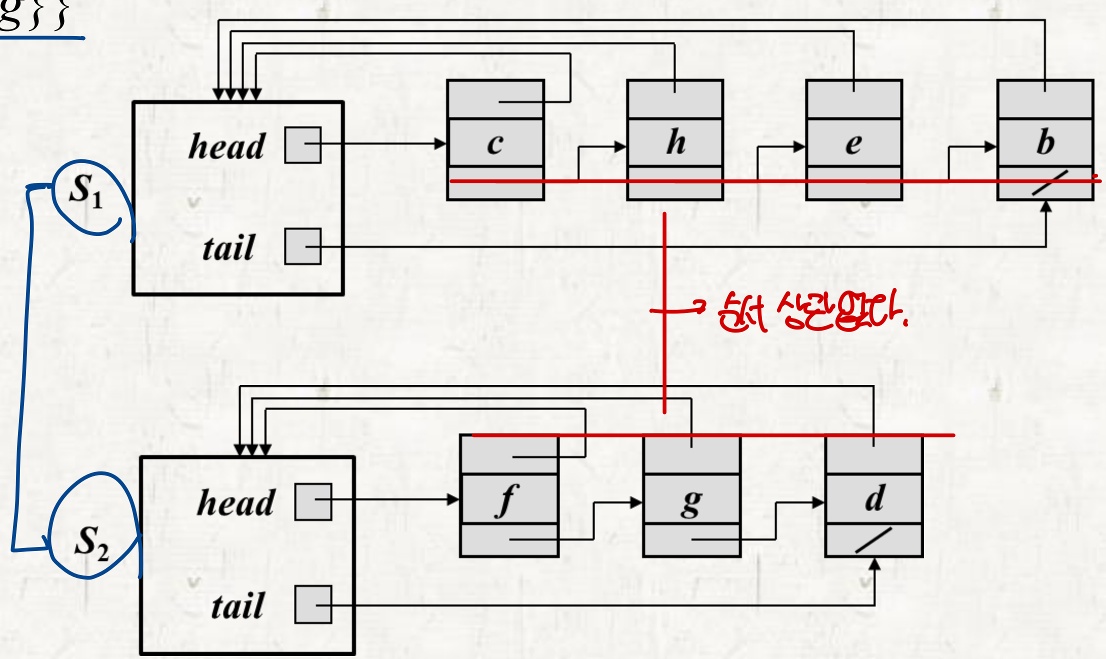
- **Disjoint-set data structures**
  - Linked-list representation
  - Forest representation

Linked-list representation

- **Linked-list representation**
  - Each set is represented by a linked list. If a collection has two disjoint sets, two linked lists are needed.
  - Each set member is contained by an object in its linked list.
  - The objects may appear in any order in a linked list.
  - $\{\{b,c,e,h\}, \{d,f,g\}\}$

2개 set

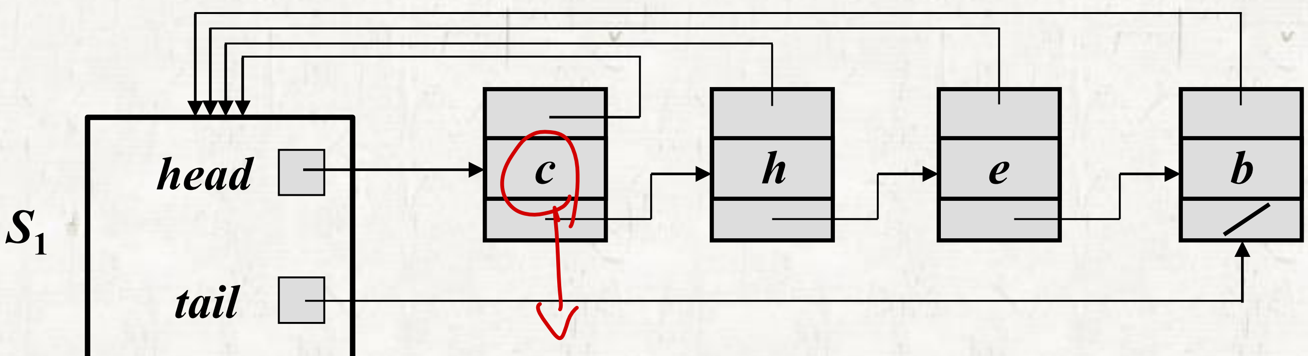
→ 2개 Linked List



Linked-list representation

Linked-list representation

- The object for each set has two attributes *head* and *tail*.
  - Attribute *head* points to the first object.
  - Attribute *tail* points to the last object.
- All objects have pointers to the set object.
- The first object in the linked list is the representative.



set object

representative

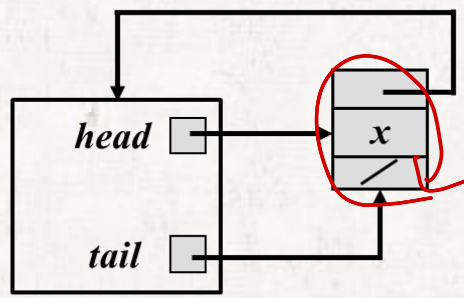
Linked-list representation

→ member

- **MAKE-SET(*x*)**

  → object including member x

  - Create a new linked list whose only object is *x*.
  - $\Theta(1)$



→ object

→ pointer to an set object

- **FIND-SET(*x*)**
  - Follow the pointer from *x* back to its set object and then return the member in the object that head points to.
  - $\Theta(1)$

1. X가 가리키는 Object(set)으로 간다.
2. Object의 첫번째 element(representative) 본인 (head pointer)

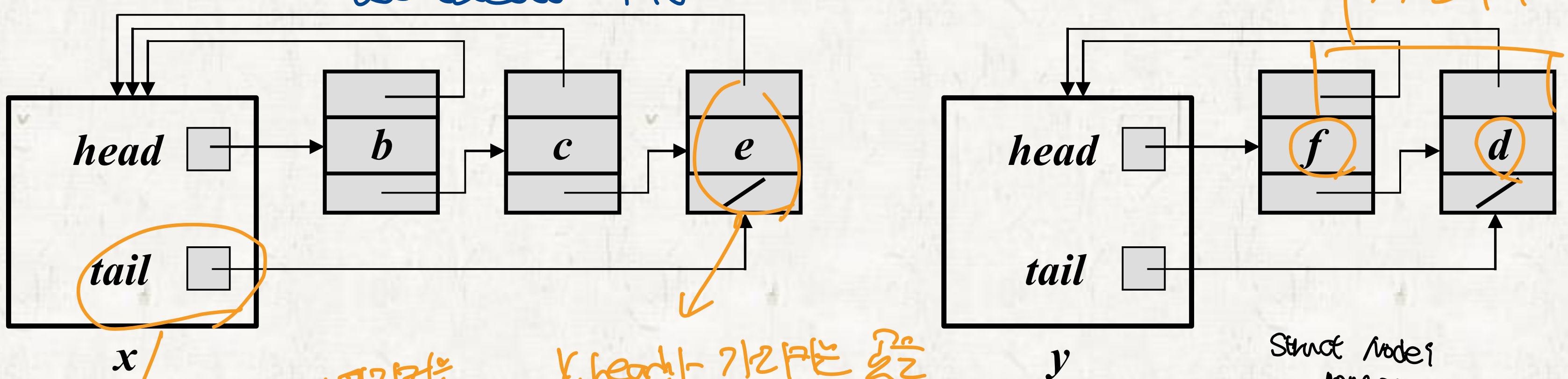
Linked-list representation

pointer to each linked list

- **UNION(*x*,*y*):** Attaching a linked list to the other

see object를 가리킴

y를 x에 이어붙임



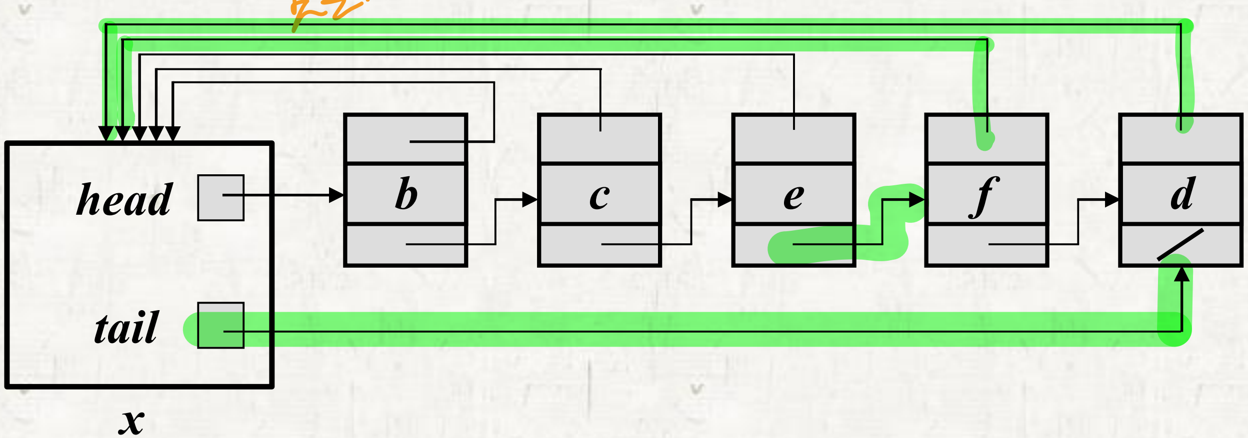
→ y.tail이 가리키는 곳을 가리키도록

y.head가 가리키는 곳을 가리키도록

```
struct node{
    member
    * next
    * parent
}
```

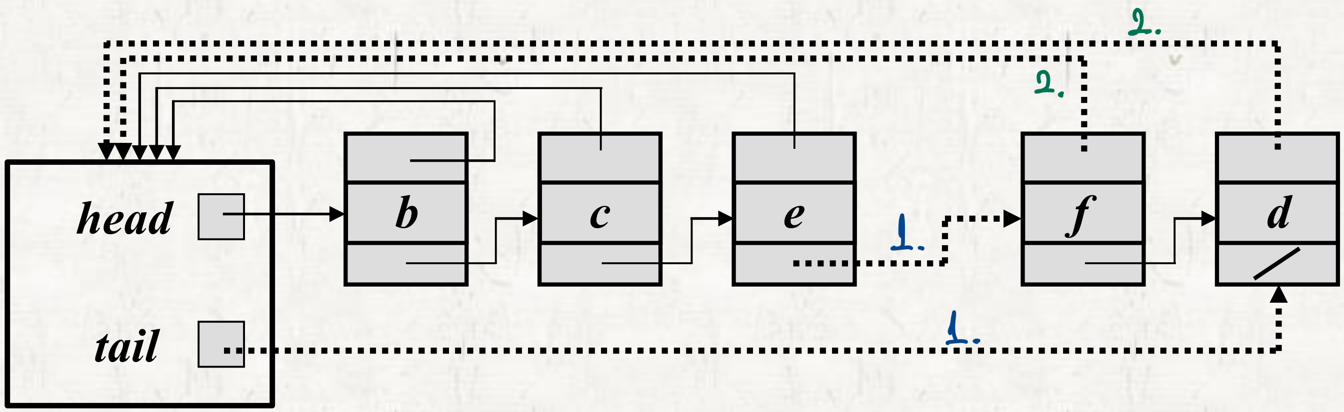
$\theta(1)$ { x.tail -> next = y.head
x.tail = y.tail

$\theta(m_y)$ current = y -> head
while (current != NULL)
current.parent = x
current = current -> next



Linked-list representation

- **UNION(x,y):** Time complexity



- $\Theta(m_y)$ time where m_y is the number of objects of y.
  1. Changing tail pointer & linking two linked lists: $\Theta(1)$
  2. Changing pointers to the set object: $\Theta(m_y)$

↳ y의 object 개수

$O(m_y + 2)$

Linked-list representation

worst case
→ $\Theta(n)$

worst case of union in linked list
— longer list to shorter list

- **Running time for *m* (= *n* + *f* + *u*) operations**

Union $\Theta(n)$
→ 하나의 원소 set에
n-1 개의 원소 set을
연결할 때

- Simple implementation of union
  - n×1 f×1
  - → n+f = m
  - $O(n+f+n^2)$ time ➔ $O(m+n^2)$ time
    - Because $u \leq n$-1

→ $(n-1) \times \Theta(n) = \Theta(n^2)$

- A weighted-union heuristic

→ shorter list to longer list

  - $O(n+f+u+n\lg n)$ time ➔ $O(m+n\lg n)$ time

set object pointer를
변경하는 횟수가 감소하기
때문이다.

Short list 의 length가 최대
→ 절반인 경우
최대 2배씩 늘어남 → short list의 2배가 된다.
UNION의 결과 Linked list는

Length: $1 \times 2 \times 2 \times \cdots \times 2 = 2^k$

Shortest listed $n \geq 2^k$ → $k \leq \log n$
→ set object pointer
update 수

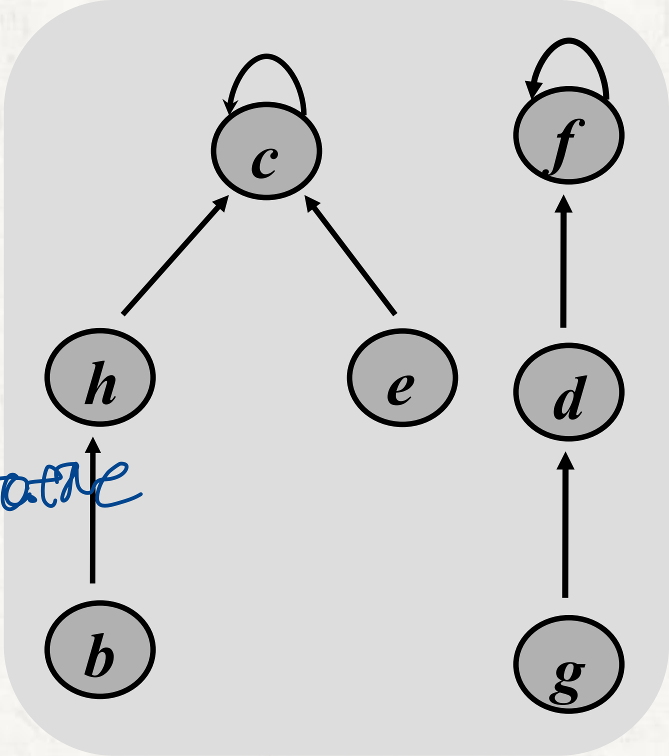
Forest representation

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

- **Forest representation**
  - Each set is represented by a tree.
  - Each member points to its parent.
  - The root of each tree is the rep.

self loop

representative



Forest representation

object

MAKE-SET(x)

$x.p = x$ → self loop

FIND-SET(x)

if $x == x.p$ → If self loop

  return x

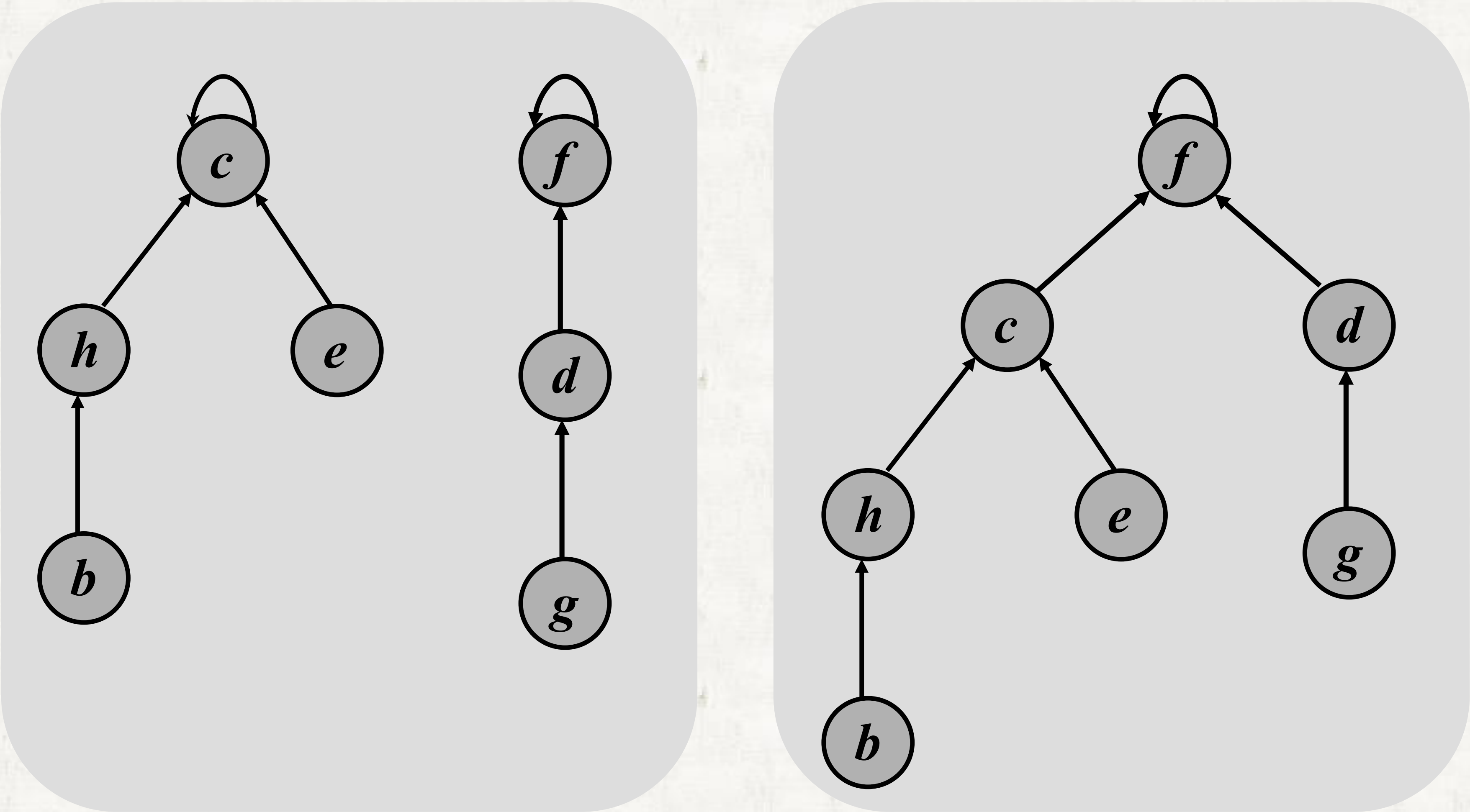
else return FIND-SET($x.p$)

→ parent

Forest representation

- **Union by rank** rank가 작은 것을 큰것에 이어붙임
  - ***Idea*: ** Attach the shorter tree to the higher tree.
  - Each node maintains a ***rank***, which is an upper bound on the height of the node.
  - Compare the ranks of the two roots and attach the tree whose root's rank is smaller to the other.

Forest representation



Forest representation

MAKE-SET(x)

$x.p = x$

$x.rank = 0$

UNION(x, y)

LINK(FIND-SET(x), FIND-SET(y))

root of each tree

LINK(x, y)

if $x.rank > y.rank$

$\quad y.p = x$

else $x.p = y$

$\quad$ **if** $x.rank == y.rank$

$\quad\quad y.rank = y.rank + 1$

$x \leftarrow y$

find-set에는
[illegible]가 없다.

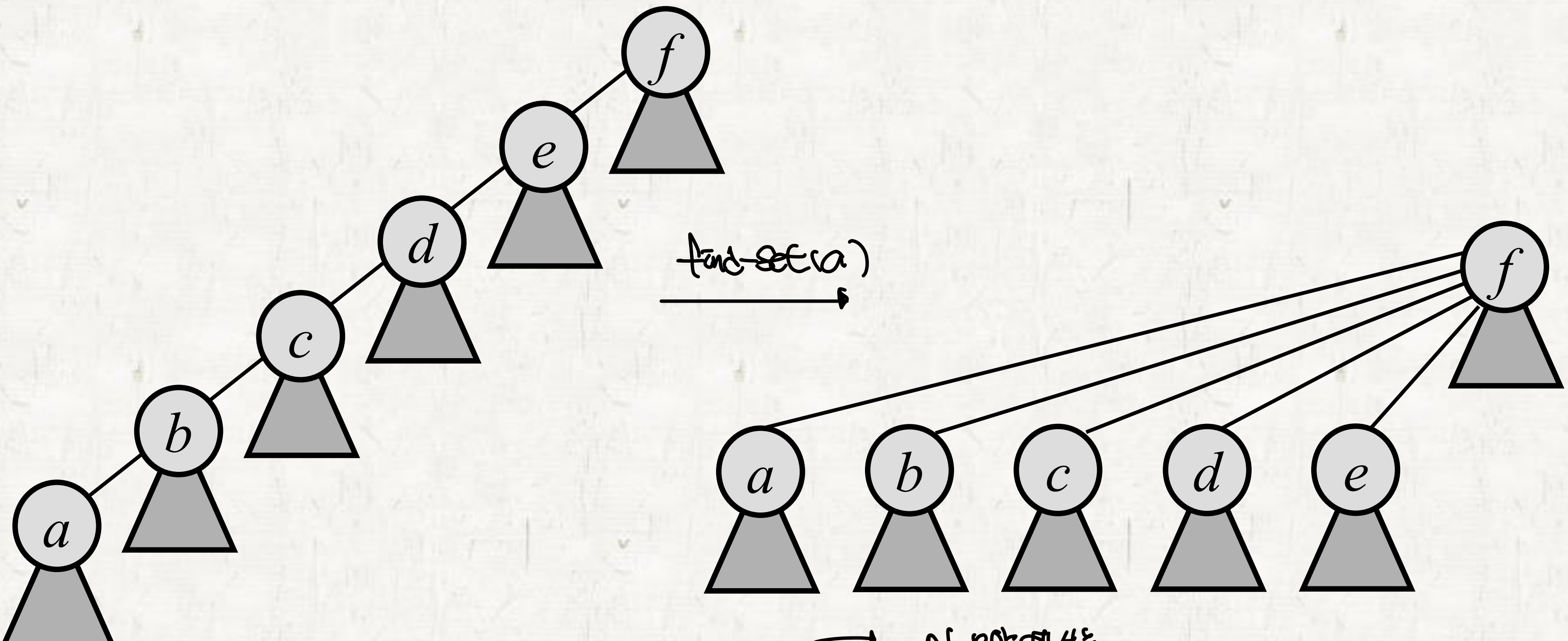
Forest representation

- **Path compression**
  - Change the parent to the root during FIND-SET(x).

parent를 root node로 변경

```
FIND-SET(x)
if x ≠ x.p          x != root
    x.p = FIND-SET(x.p)
return x.p
```

Forest representation



이 모양에서는
find-set(a) = $\Theta(1)$

Forest representation

m = n+f+u

→ for m operations

- Worst case running time : O(m $\boxed{\alpha(n)}$) → make-set 개수

→ 종음 X

↳ very slowly increasing function

- $\alpha(n) \leq 4$:for all practical situations.

↳ O(m) ≈ O(1) slowly increase

constant time

$\boxed{O(m\alpha(n)) \approx O(m) \approx O(1)}$