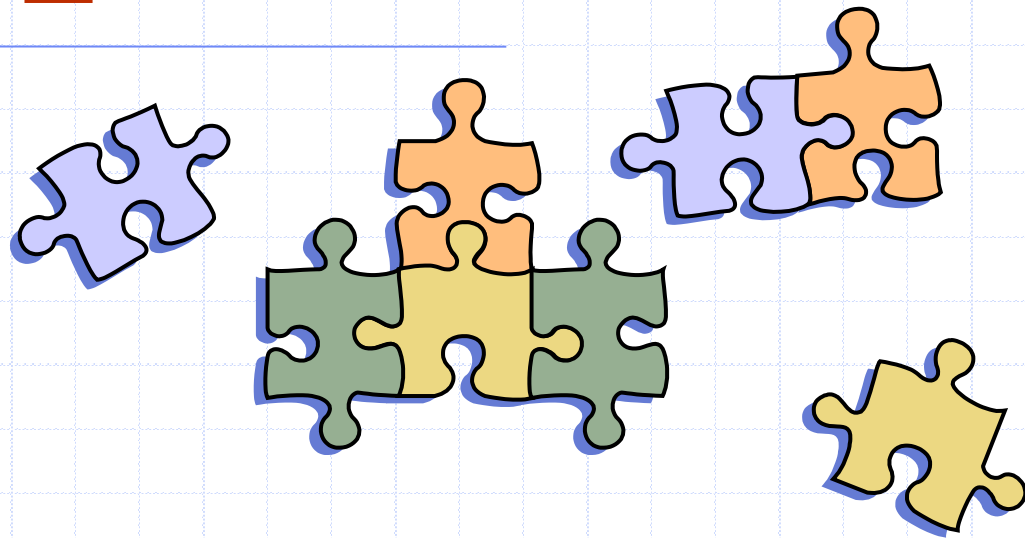


# 분리 집합



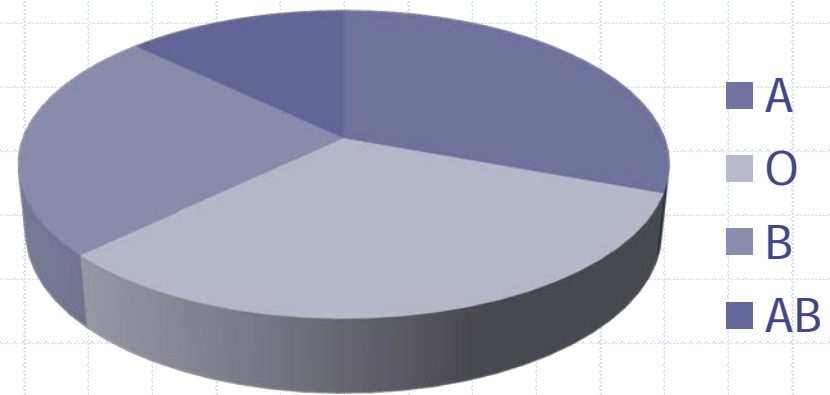
# Outline

- ◆ 9.1 분리집합 ADT
- ◆ 9.2 분리집합 ADT 메소드
- ◆ 9.3 분리집합 응용
- ◆ 9.4 분리집합 ADT 구현
- ◆ 9.5 응용문제

# 분리집합 ADT

- ◆ 분리집합 ADT는 분할, 즉, 상호배타적인 집합들을 모델링한다
- ◆ 집합 ADT의 특별한 버전
- ◆ 분리집합 간의 교집합이나 차집합 연산은 무의미

Blood Types



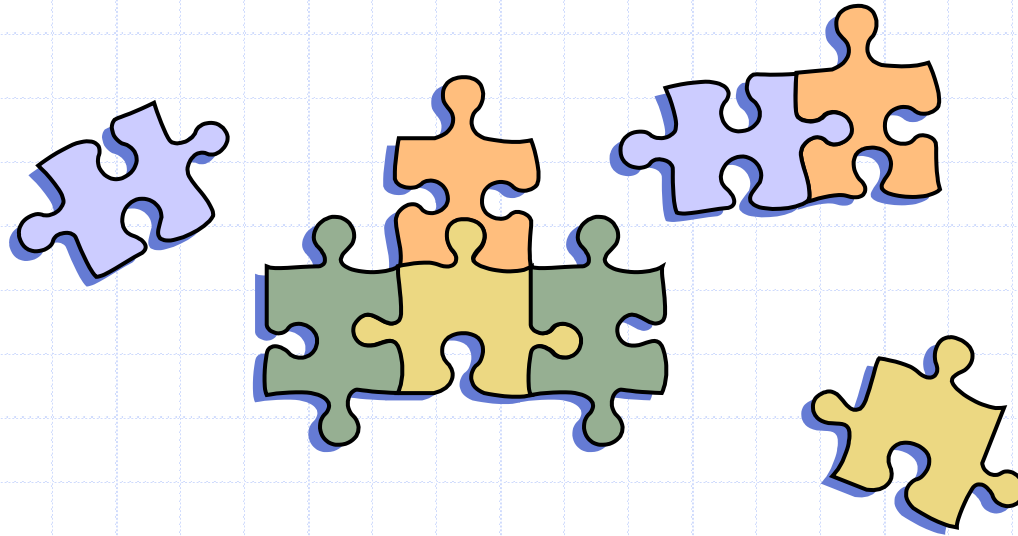
# 분리집합 ADT 메소드

## ◆ 주요 메소드

- set **find**( $e$ ): 원소  $e$ 가 속한 집합을 반환
- **union**( $x, y$ ): 집합  $x, y$ 를 통합

## ◆ 보조 메소드

- integer **size**( $S$ ): 집합  $S$ 의 원소 수를 반환



# 분리집합 응용

## ◆ 직접 응용

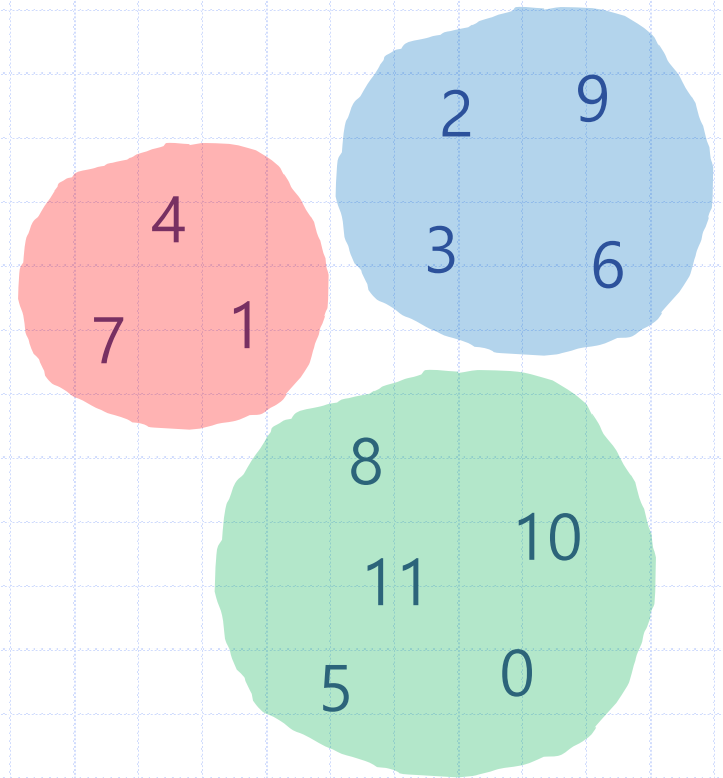
- 동치관계(equivalence relation) (예: 그래프)
- 최소신장트리(minimum spanning tree)

## ◆ 간접 응용

- 알고리즘을 위한 보조 데이터구조
- 다른 데이터구조를 구성하는 요소

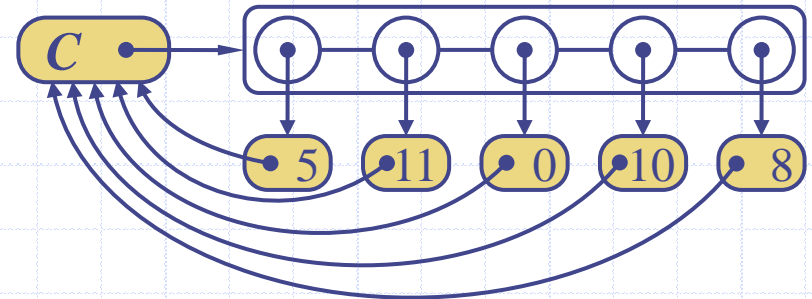
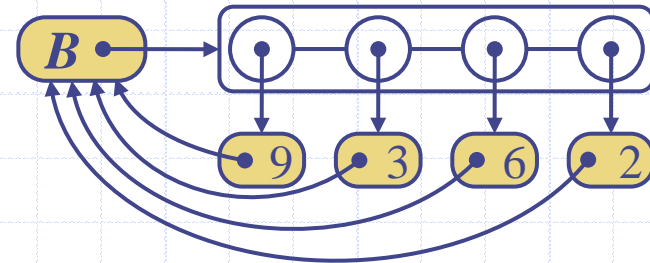
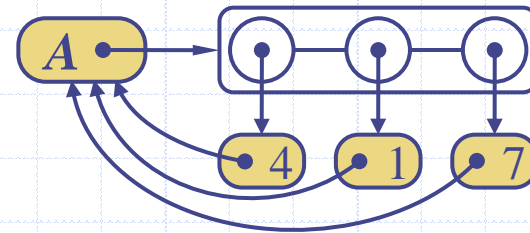
# 분리집합 구현

- ◆ 리스트에 기초한 구현
  - **find**는 빠르고 **union**은 느리다
  - 배열 또는 연결리스트 사용
- ◆ 트리에 기초한 구현
  - **find**는 느리고 **union**은 빠르다
  - **find** 성능 개선 가능



# 리스트에 기초한 구현

- ◆ 한 개의 분리집합에 대해 한 개의 **리스트**를 사용
- ◆ 각 원소는 소속집합으로 향하는 참조를 가진다
- ◆ 예: 분리집합  $A, B, C$ 
  - $A = \{1, 4, 7\}$
  - $B = \{2, 3, 6, 9\}$
  - $C = \{0, 5, 8, 10, 11\}$
- ◆ **find**( $e$ ):  $e$ 가 소속된 집합을 반환 – 실행시간은  $O(1)$
- ◆ **union**( $x, y$ ):  $x, y$  중 작은 집합의 원소들을 큰 집합으로 이동 (즉, 소속집합을 변경) – 실행시간은 연결리스트를 사용할 경우  $O(\min(|A|, |B|))$



# 배열을 사용할 경우

- ◆ 소속집합을 원소값으로 하는 크기  $n$ 의 배열  $S$ 로 분리집합을 표현
- ◆  $\text{find}(e)$ :  $S[e]$  접근 - 실행시간:  $O(1)$
- ◆  $\text{union}(x, y)$ : 집합  $y$  소속의 원소들을 모두 집합  $x$  소속으로 (혹은 반대로) 변경 - 실행시간은 배열 전체를 검사하므로  $\Theta(n)$
- ◆ 전제: 원소와 배열첨자, 즉  $[0, n-1]$  간의 대응관계 관리(예:  $\text{index}(e)$ )
- ◆ 예: 분리집합  $A, B, C$ 
  - $A = \{1, 4, 7\}$
  - $B = \{2, 3, 6, 9\}$
  - $C = \{0, 5, 8, 10, 11\}$

	0	1	2	3	4	5	6	7	8	9	10	11	$n$
$S$	C	A	B	B	A	C	B	A	C	B	C	C	



# find와 union

**Alg** *find*( $e$ )

**input** element  $e$

**output** set

1. **return**  $S[e]$

{Total  $O(1)$ }

**Alg** *union*( $x, y$ )

**input** set  $x, y$

**output** set  $x \cup y$

1. **for**  $i \leftarrow 0$  to  $n - 1$

**if** ( $S[i] = y$ )

$S[i] \leftarrow x$

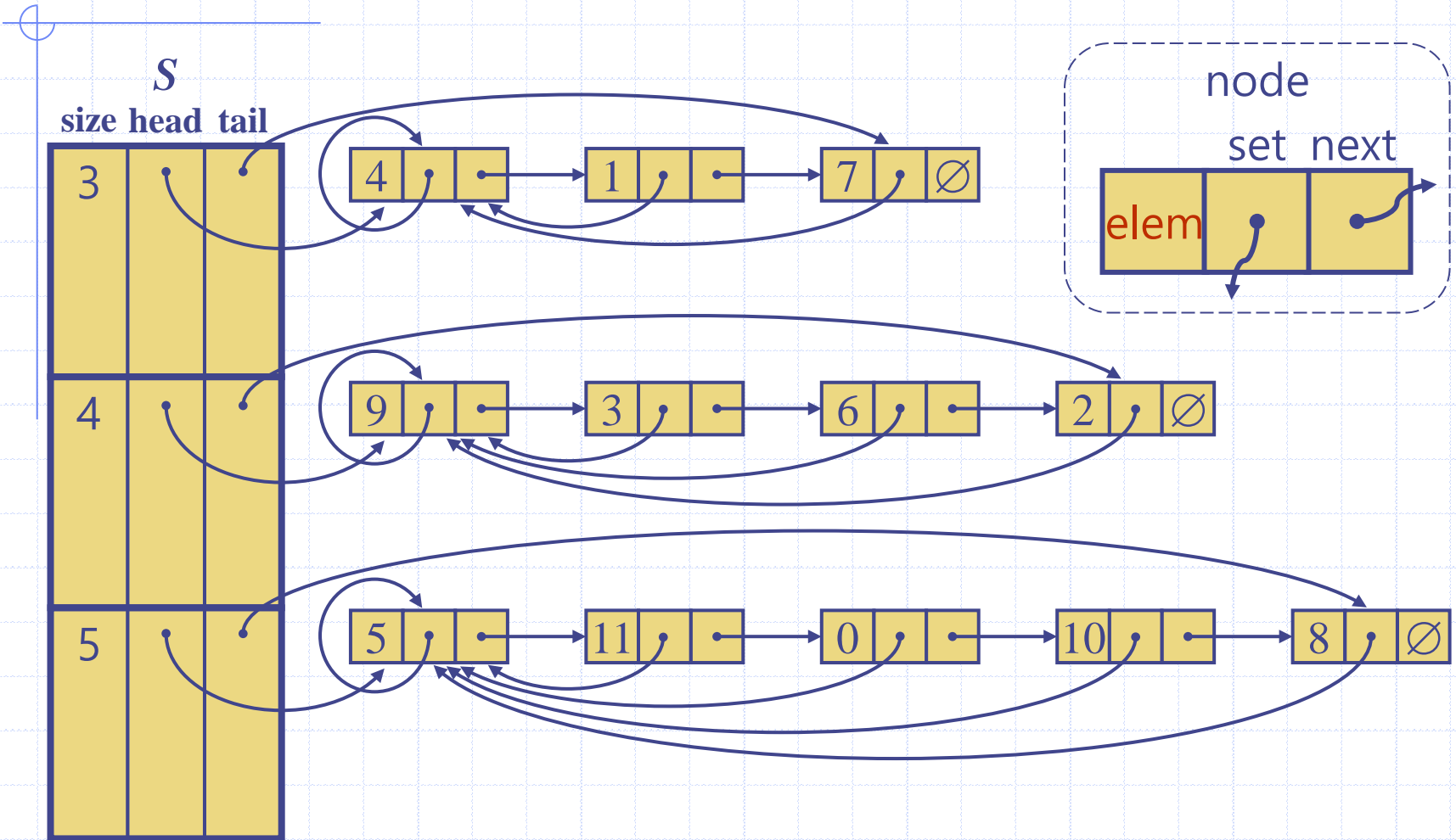
2. **return**

{Total  $O(n)$ }

# 연결리스트를 사용할 경우

- ◆ 분리집합 집단을 레코드의 배열로 표현
- ◆ 각 분리집합은 단일연결리스트로 구현
- ◆ 레코드 필드
  - **size**: 집합원소 수
  - **head**: 집합원소 노드들 중 첫 노드 주소
  - **tail**: 집합원소 노드들 중 마지막 노드 주소
- ◆ 집합원소 노드 저장내용
  - **elem**: 원소
  - **set**: 소속집합 노드 포인터
  - **next**: 다음 원소노드 포인터
- ◆ **find**( $e$ ):  $e$  노드의 set 필드를 접근한다 – 실행시간:  $O(1)$
- ◆ **union**( $A, B$ ):  $A, B$  중 작은 집합을 큰 집합에 병합시킨다 – 실행시간:  $O(\min(|A|, |B|))$ , 상각실행시간:  $O(\log n)$
- ◆ **전제**
  - 원소와 노드 주소 간의 대응관계 관리(예: **node**( $e$ ))
  - 각 집합의 헤드노드 원소와 집합식별자 간의 대응관계 관리(예: **setid**( $e$ ))

# 예



# find와 union

Alg **find**(*e*)

input element *e*

output set

1. return *setid*((*node*(*e*)).*set*).*elem*)

Alg **union**(*A*, *B*)

input set *A*, *B*

output set  $A \cup B$

1. if (*S*[*A*].*size* < *S*[*B*].*size*)  
    *smallerSet*, *largerSet*  $\leftarrow$  *A*, *B*  
  else  
    *smallerSet*, *largerSet*  $\leftarrow$  *B*, *A*

2. *headS*, *tailS*  $\leftarrow$  *S*[*smallerSet*].*head*,

*S*[*smallerSet*].*tail*

3. *headL*, *tailL*  $\leftarrow$  *S*[*largerSet*].*head*,

*S*[*largerSet*].*tail*

4. *p*  $\leftarrow$  *headS*

5. while (*p*  $\neq \emptyset$ )                   { **O**(*min*(|*A*|, |*B*|))

*p.set*  $\leftarrow$  *headL*

*p*  $\leftarrow$  *p.next*

6. *tailL.next*  $\leftarrow$  *headS*

7. *S*[*largerSet*].*tail*  $\leftarrow$  *tailS*

8. *S*[*largerSet*].*size*  $\leftarrow$  *S*[*A*].*size* + *S*[*B*].*size*

9. *S*[*smallerSet*].*head*, *S*[*smallerSet*].*tail*  $\leftarrow \emptyset$

10. *S*[*smallerSet*].*size*  $\leftarrow$  0

11. return

{ Total **O**(*min*(|*A*|, |*B*|))

# 트리에 기초한 구현

◆ 한 개의 분리집합에 대해 한 개의 **트리**를 사용하며 각 집합은 각 트리의 **루트**로 식별된다

## ◆ 연결트리 사용

- 각 노드는 원소 및 부모를 가리키는 포인터를 저장한다 – 단, 루트는 자신을 부모로 하는 포인터를 저장한다

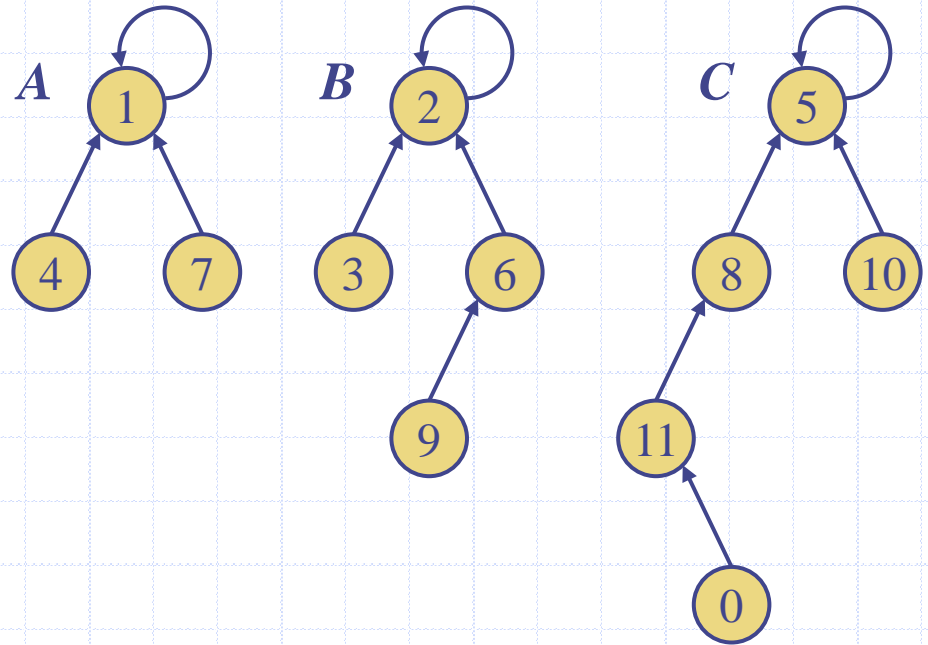
## ◆ 가상트리로 구현

- 자식을 가리키는 포인터가 불필요하므로, **트리** ADT 대신 **배열**에 의한 가상의 트리로 구현하면 충분 (**예: *Parent* 배열**)

# 트리에 기초한 구현 (conti.)

◆ 예: 분리집합  $A, B, C$

- $A = \{1, 4, 7\}$
- $B = \{2, 3, 6, 9\}$
- $C = \{0, 5, 8, 10, 11\}$

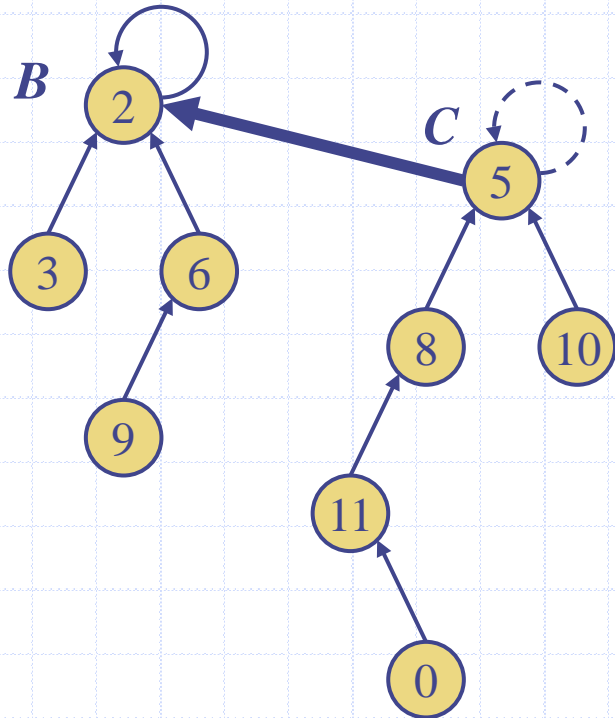


	0	1	2	3	4	5	6	7	8	9	10	11	$n$
Parent	11	1	2	2	1	5	2	1	5	6	5	8	

# 트리에 기초한 구현 (conti.)

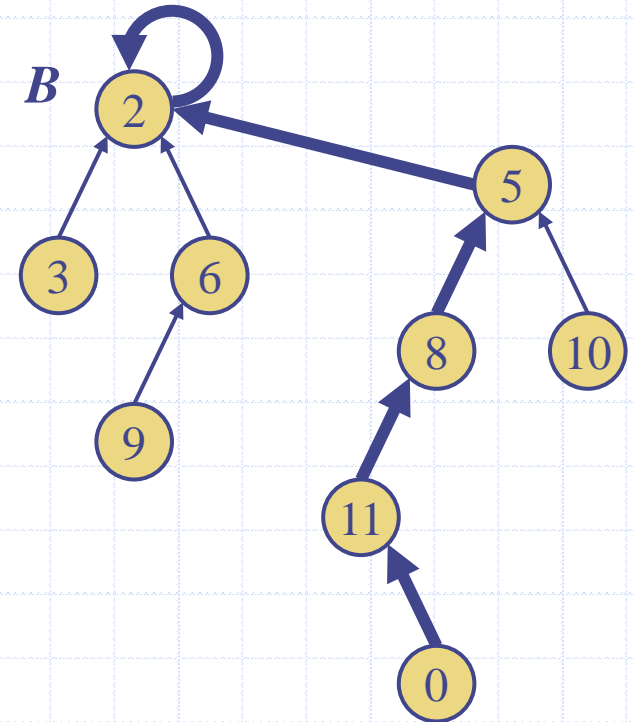
◆ **union**( $x, y$ ): 트리  $x, y$  중 하나를 다른 트리의 부트리로 만든다 - 실행시간:  $O(1)$

◆ 예: **union**( $B, C$ )



◆ **find**( $e$ ):  $e$ 의 부모포인터를 따라 루트까지 올라간다 - 실행시간: 최악의 경우  $O(n)$

◆ 예: **find**(0)



# find와 union

**Alg** *find*( $e$ )

**input** element  $e$

**output** set

1. **if** ( $Parent[e] = e$ )

**return**  $e$

**else**

**return**  $find(Parent[e])$

        {Total  $O(n)$ }

**Alg** *union*( $x, y$ )

**input** set  $x, y$

**output** set  $x \cup y$

1.  $Parent[y] \leftarrow x$

2. **return**

{Total  $O(1)$ }



# 성능 개선을 위한 전략

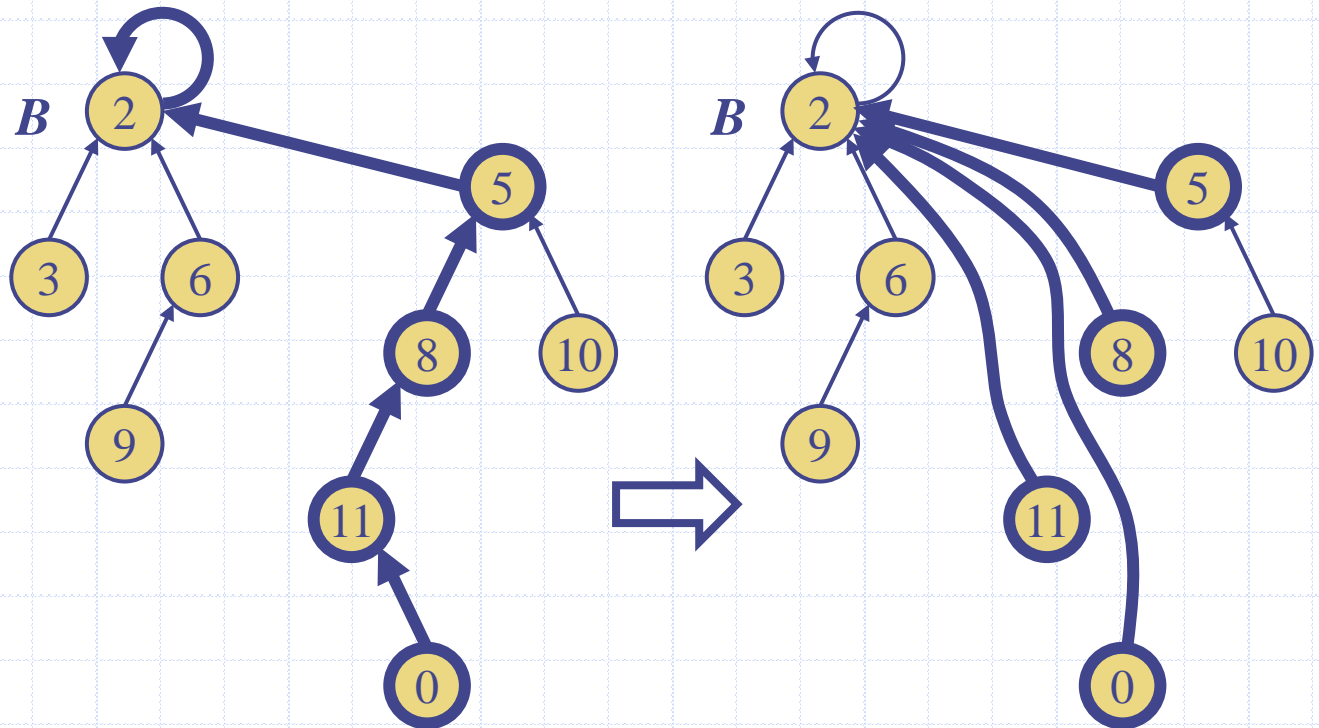
## ◆ 크기에 의한 union(union-by-size)

- 각 노드의 크기 필드에 그 노드를 루트로 하는 부트리의 노드 수를 저장하여(예: *Size* 배열), *Parent* 배열과 병행 사용
- union 작업 시에 두 개의 트리 중 작은 집합트리를 큰 집합트리의 부트리로 만들고 결과트리의 루트의 크기 필드 값을 갱신

	0	1	2	3	4	5	6	7	8	9	10	11	$n$
<i>Parent</i>	11	1	2	2	1	5	2	1	5	6	5	8	
<i>Size</i>	1	3	4	1	1	5	2	1	3	1	1	2	

# 성능 개선을 위한 전략 (conti.)

- ◆ **경로압축**(path-compression): **find** 수행시 작업 경로상의 모든 노드  $v$ 의 부모 포인터를 루트로 변경
- ◆ 예: **find**(0)



# find와 union

**Alg *find*(*e*)**

**input** element *e*

**output** set

```
1. if (Parent[e] = e)  
    return e  
else  
    Parent[e] ← find(Parent[e])  
    return Parent[e]
```

{Total  $O(\log^*n)$ }

**Alg *union*(*x*, *y*)**

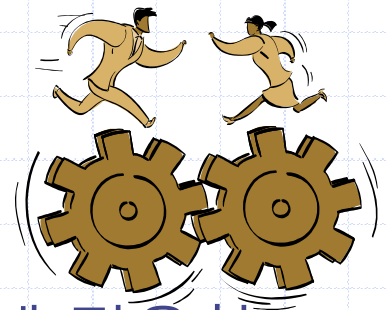
**input** set *x*, *y*

**output** set  $x \cup y$

```
1. if (Size[x] < Size[y])  
    Parent[x] ← y  
    Size[y] ← Size[x] + Size[y]  
else  
    Parent[y] ← x  
    Size[x] ← Size[x] + Size[y]  
3. return
```

{Total  $O(1)$ }

◆  $\log^*n$ 은 중첩  $\log n$ 을 의미한다  
즉,  $\log^*n = \log \log \log \dots n \approx 1$



# 두 전략의 호환성

- ◆ 경로압축을 크기에 의한 **union** 전략과 함께 적용할 경우,
- ◆ 경로압축을 하더라도 루트노드의 size 값은 불변이므로 차후 정확한 **union** 연산에 영향이 없다
- ◆ 다만, 루트를 제외한 경로상의 노드들의 size 값들은 부정확한 채로 남게 된다
- ◆ 경로압축 수행시 이들이 정확한 값을 갖도록 유지하는 것은 복잡한 계산만 수반할 뿐 차후 **union**이나 **find** 연산에서 사용되지도 않으므로 이 값들을 그대로 방치한다



# 전략 병행 시 성능

- ◆ 경로압축을 크기에 의한 **union** 전략과 함께 적용할 경우,
- ◆ **종합분석**: 전체  $n$ 개의 원소로 구성된 임의의 분리집합 집단에 대해  $n$ 회의 **union**과 **find** 작업을 수행하는데 걸리는 총 실행시간:  $O(n \log^* n)$
- ◆ 따라서, **find** 작업의 **상각실행시간**:  $O(\log^* n)$ 
  - 참고:  $\log^* n$ 은 중첩  $\log n$ 을 의미, 즉
$$\log^* n = \log \log \log \dots n \approx 1$$
- ◆ **union**의 실행시간이  $O(1)$ 이므로, **union**과 **find** 작업 모두 최선의 시간 성능으로 수행

# 응용문제: 높이에 의한 합집합

- ◆ 크기 대신, 높이(height)에 의한  $\text{union}(x, y)$ 은 트리로 구현된 분리집합  $x, y$  중 높이가 작은 트리를 다른 트리의 루트의 부트리로 만든다
- ◆ 선형시간에 수행하는, 높이에 의한  $\text{union}(x, y)$  알고리즘을 작성하라
- ◆ 전제: 각 노드는 크기 대신 높이 값을 유지하여야 하므로 *Size* 배열 대신 *Height* 배열을 사용한다
- ◆ 최초에 모든 원소가 단독노드로 존재한다고 가정하고, 이후 처리에서 높이에 의한  $\text{union}$  전략을 적용할 경우 총  $n$ 개의 원소로 이루어진 분리집합 집단에서 임의의 노드가 가질 수 있는 높이의 상한은 얼마인가?

# 해결: 높이에 의한 합집합 알고리즘

**Alg** *union*( $x, y$ ) {union by height}

**input** set  $x, y$

**output** set  $x \cup y$

1. **if** ( $Height[x] < Height[y]$ )

$Parent[x] \leftarrow y$

**else**

$Parent[y] \leftarrow x$

2. **if** ( $Height[x] = Height[y]$ )

$Height[x] \leftarrow Height[x] + 1$

3. **return**

{Total  $O(1)$ }

# 해결: 분석 (conti.)

- ◆ 높이가  $h$ 인 트리를 높이가  $h$ 보다 작은 트리와 합치는 경우에 높이는 여전히  $h$ 가 되면서 노드 수만 늘어난다
- ◆ 같은 높이  $h$ 인 두 개의 트리를 합치는 경우에만 높이가  $h+1$ 로 증가한다
- ◆ 높이  $k$ 인 트리의 노드 수는 최소  $2^k$ 이다(직관에 의함)
- ◆ 이를 역으로 적용하면 노드 수가  $n$ 인 트리의 높이는  $\log n$ 를 넘지 않는다 - 즉  $O(\log n)$ 이다

- ◆ 증명: 높이  $k$ 인 트리의 노드 수는 최소  $2^k$ 임  
(수학적귀납법에 의함)
  - $k=0$ 인 경우  $2^0 = 1$ 이므로 만족한다
  - 높이가  $h$ 인 트리의 최소 노드 수가  $2^h$ 이라 전제하면, 같은 높이  $h$ 인 두 개의 트리가 합쳐진 높이  $h+1$ 의 트리는 최소  $2 \cdot 2^h = 2^{h+1}$ 개의 노드를 가지므로 전제를 만족한다



# 해결: 경로압축과 병용 (conti.)

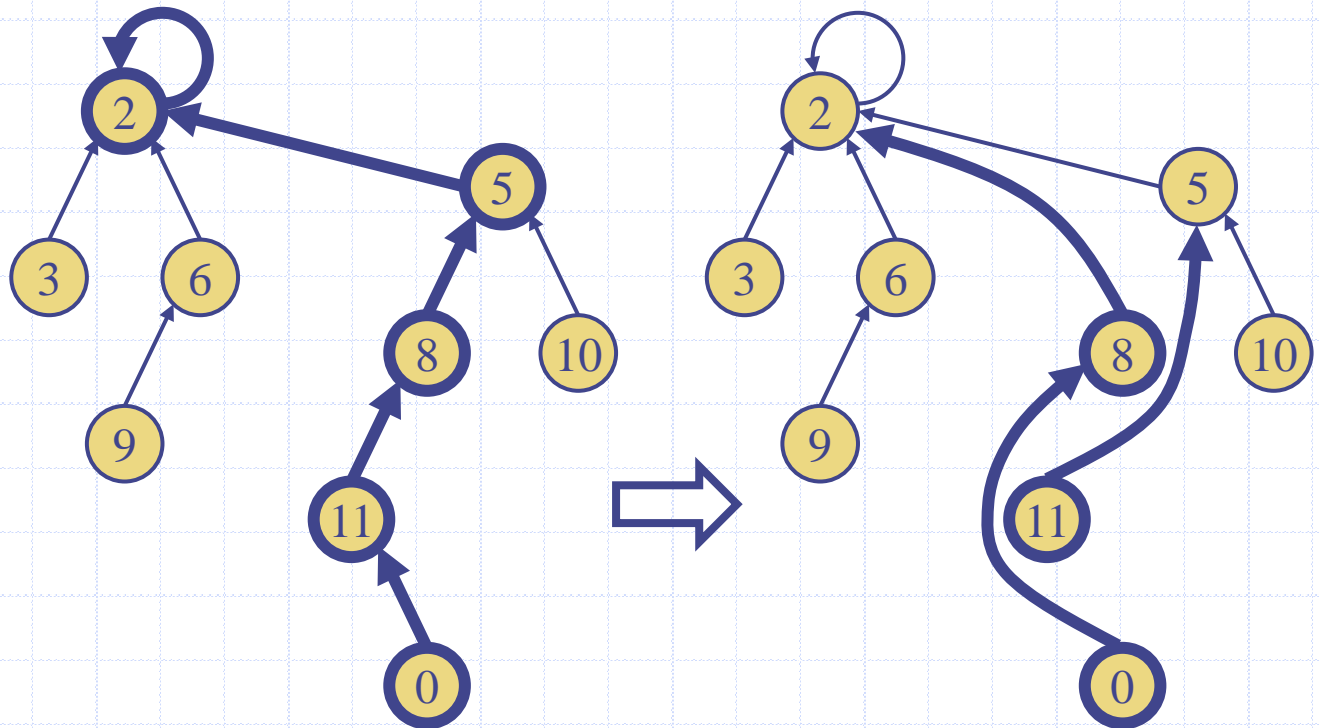
- ◆ **높이에 의한 union 전략을 (전체적 또는 부분적) 경로압축과 함께 적용하는데 관한 고려사항**
  - 경로압축을 수행하면 루트노드의 size 값은 변치 않는 것과 달리 height 값은 낮아지는 쪽으로 변한다
  - 하지만 루트노드가 정확한 height 값을 갖도록 유지하는 연산은 매우 복잡해서 실용성이 떨어지므로 수행하지 않는 것이 보통이다
  - 따라서 정확하지 않은 대략의 height이므로 이를 **rank**라고도 부르며, 그런 의미에서 이 전략을 "**랭크에 의한 union**"이라고도 부른다

# 응용문제: 부분적 경로압축

- ◆ 트리로 구현한 분리집합에 대한 **find** 작업 수행시 **부분적 경로압축**(partial path-compression) 전략을 적용하기로 한다
- ◆ **부분적 경로압축**이란 **find** 작업 수행시 **find** 경로상의 노드들의 부모포인터를 루트가 아닌, 그 노드의 **조부모노드**를 가리키도록 변경함을 말한다 – 다만 조부모가 없는 노드의 부모포인터는 변경 처리하지 않는다
- ◆ 이 방식의 **find** 전략이 **크기** 또는 **높이**에 의한 **union** 전략과 병행 적용되면, **종합분석**을 통해 **find** 작업의 **상각실행시간**은  $O(\log^* n) \approx O(1)$
- ◆ 위의 전략을 수행하는 **find(e)** 알고리즘을 작성하라

# 응용문제: 부분적 경로압축 (conti.)

예: find(0)



# 해결

**Alg** *find*(*e*) {nonrecursive}

**input** element *e*

**output** set

1.  $p \leftarrow e$
2. **while** ( $\text{Parent}[p] \neq p$ )  
     $par \leftarrow \text{Parent}[p]$   
    **if** ( $\text{Parent}[par] \neq par$ )  
         $\text{Parent}[p] \leftarrow \text{Parent}[par]$   
     $p \leftarrow par$
3. **return**  $p$