08장. 집합의 처리

Youn-Hee Han
LINK@KOREATECH
http://link.koreatech.ac.kr

얼마 전 나는 학술회의 준비차 『다윈』을 읽었다.
읽으면서 그동안 읽지 않기를 잘했다는 생각이 들었다.
다른 때 『다윈』을 읽었더라면
여전히 이해할 수 없었을 것이기 때문이다.
… 결국, 읽을 준비가 되었을 때 읽어야 한다는 것이다.

–로저 생크

로저 생크

文A 10개 언어

위키백과, 우리 모두의 백과사전.

로저 생크(Roger Schank, 1946년 ~)는 소크라틱 아츠(Socratic Arts)의 회장이자 최고경영자이며, 인공 지능 분야의 선구자이다.

경력 [편집]

생크는 예일 대학의 컴퓨터 과학과 및 심리학과의 교수이자, 예일 인공 지능 프로젝트의 회장이었다. 1989년 그는 학습 과학 연구소를 설립하기 위해 노스웨스턴 대학에 고용되었다. 나중에 학습 과학 연구소는 세부 단과대학으로서의 교육학부로 흡수되었다. 그는 카네기 멜론 대학에 학습 과학을 연구하는 기관을 설립하는 데에도 도움을 주었다. 그는 ILS의 상업적 지부로 학습 과학 주식회사(Learning Sciences Corporation)를 설립하고, 2003년 매각할 때까지 운영하였다. 2005년에는 도널드 트럼프의 트럼프 대학의 수석 학습 사무관으로 임명되었다.[1]

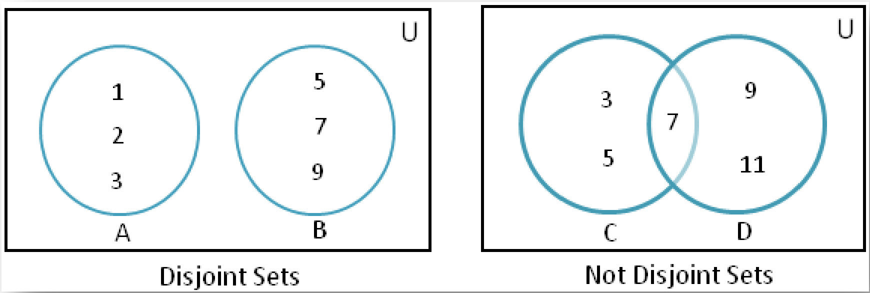
학습 목표

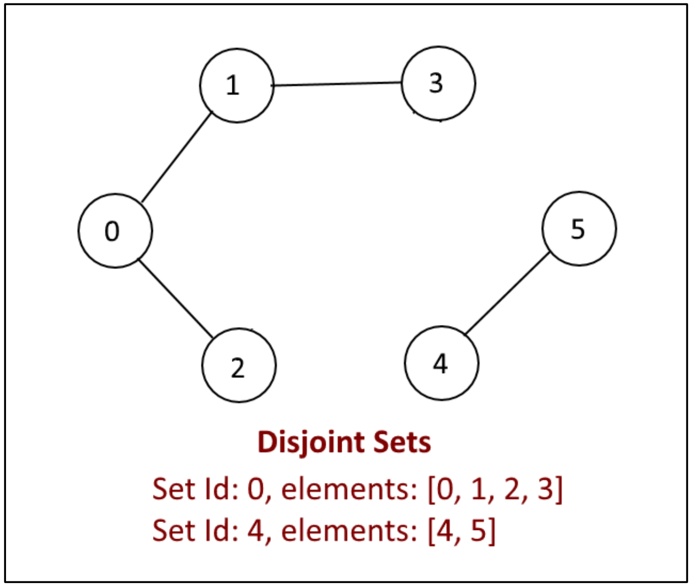
- ◈ 연결 리스트를 이용한 상호 배타적 집합의 처리 방법을 이해한다.
- ◈ 연결 리스트를 이용해 집합을 처리하는 연산들의 수행 시간을 분석할 수 있도록 한다.
- ◈ 트리를 이용한 상호 배타적 집합의 처리 방법을 이해한다.
- ◈ 트리를 이용해 집합을 처리하는 연산들의 수행 시간을 기본적인 수준에서 분석할 수 있도록 한다.

상호 배타적 집합

◈ 상호 배타적 집합 (Disjoint Set)

– Two sets are said to be **disjoint sets** if they have no element in common





– 구성 방법
- 연결 리스트를 이용하는 방법
- 트리를 이용하는 방법

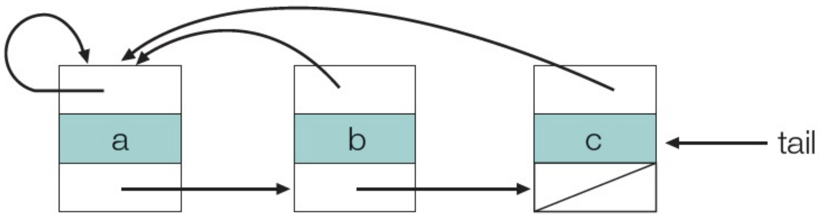
– 지원할 연산
- Make-Set(x): 원소 x로만 이루어진 집합을 만든다
- Find-Set(x): 원소 x를 가지고 있는 집합을 알아낸다
- Union(x, y): 원소 x를 가진 집합과 원소 y를 가진 집합을 합친다.

01. 연결 리스트를 이용한 집합의 처리

작업의 개요

◈ 연결 리스트를 활용한 상호 배타적 집합 구성

- 한 개 이상의 노드를 연결 리스트로 연결하는 것으로 집합 구성
  - 각 원소당 하나의 노드 구성



- <u>대표 노드</u>: 집합의 대표하는 노드
- <u>꼬리(Tail) 노드</u>: 집합의 마지막 노드
- 각 노드가 지니는 두 개의 포인터
  - 대표 노드(원소)를 가리키는 포인터
  - 다음 노드(원소)를 가리키는 포인터
- 꼬리 노드를 가리키는 포인터 (Tail 변수)
  - 집합이 지니고 있는 변수 (즉, 노드가 지니는 것이 아님)
  - 두 집합을 합칠 때 효율 향상 목적으로 유지

작업의 개요

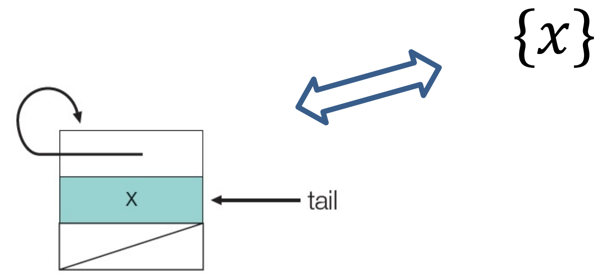


그림 8-1 연결 리스트를 이용하는 표현에서 하나의 원소로 구성된 집합

◈ 지원하는 연산

- Make-Set(x)
  - 원소 하나로 구성된 집합을 구성
  - 대표 노드로는 자신을 가리키도록 함
  - 다음 원소는 없으므로 다음 노트를 가리키는 포인터는 NIL로 정함
  - 꼬리 노드를 가리키는 포인터는 새롭게 구성된 노드를 가리키도록 함

- Find-Set(x)
  - 원소 x가 포함된 집합을 알아냄
    - ➢ 즉, 원소 x가 가리키는 대표 노드를 리턴한다.

  - 만약 원소 x와 원소 y가 동일한 집합에 속한다면

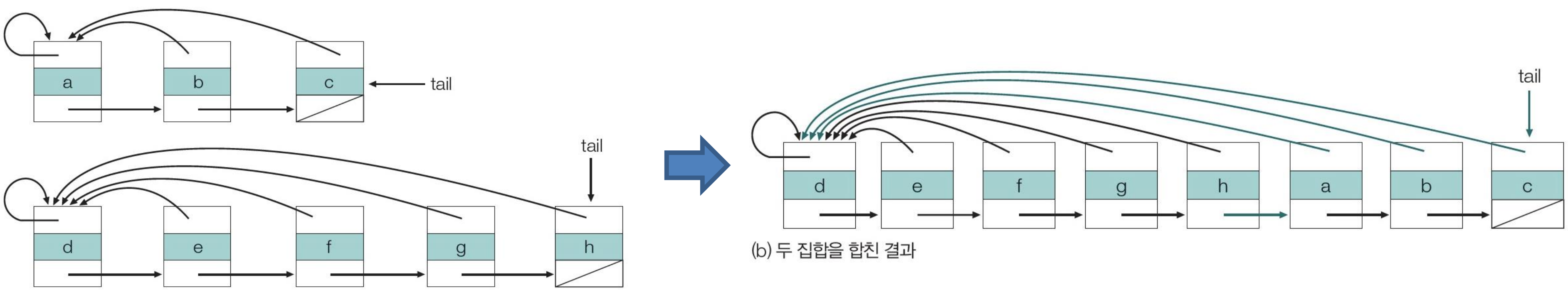
Find-Set(x) == Find-Set(y)

작업의 개요

◈ 지원하는 연산

– Union(x, y)

1) Find-Set(x)와 Find-Set(y)를 호출하여 x와 y가 속한 대표 노드를 각각 알아낸다.



(a) 합치고자 하는 두 집합

(b) 두 집합을 합친 결과

2) 두 집합 중 하나를 다른 집합 뒤에 붙인다.

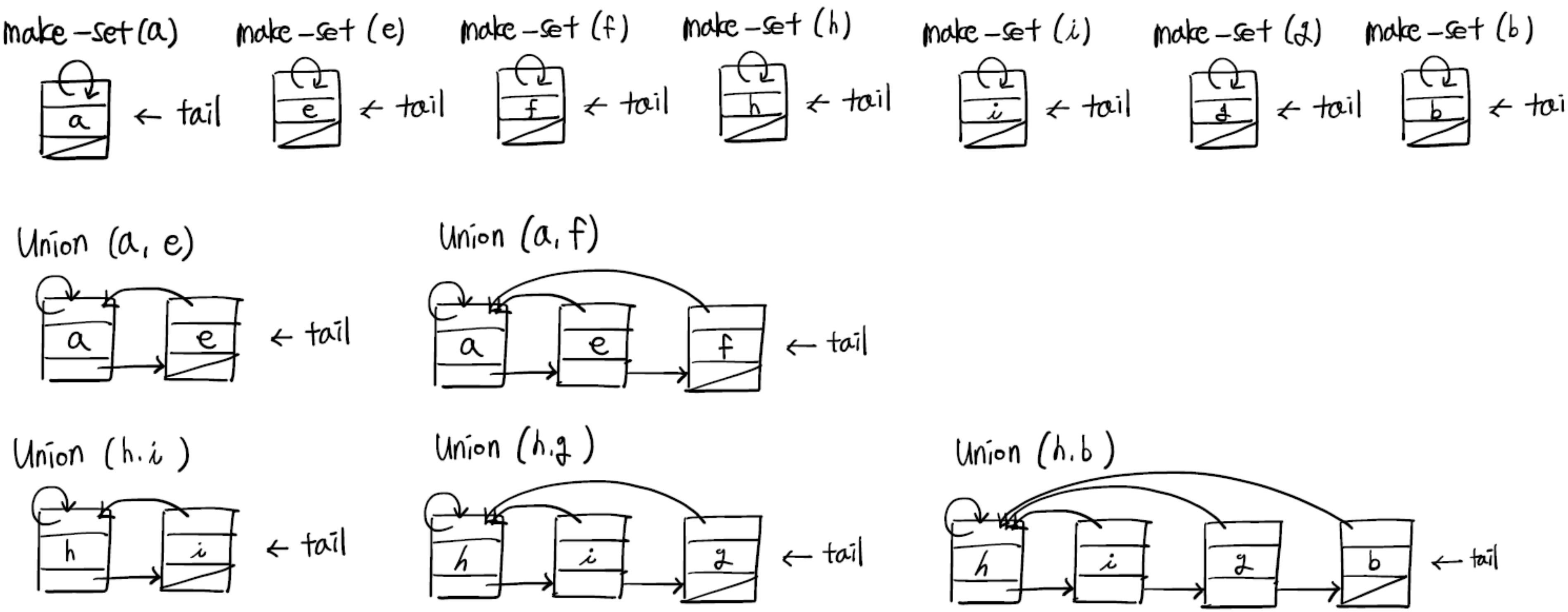
- ➢ 주집합: 앞쪽에 있는 집합, 부집합: 뒤에 붙는 집합
- ➢ 과정
  - » 1) 주집합의 tail 노드의 다음 원소 포인터를 부집합의 대표 노드를 가리키도록 함
  - » 2) 부집합의 모든 노드의 대표 노드 포인터는 주집합의 대표 노드를 가리키도록 함

작업의 개요

◈ 상호 배타적 집합 사용 예시

– [출처] https://yjg-lab.tistory.com/295

- 다음 두 개의 상호 배타적 집합이 연결리스트로 완성되기까지의 연산 과정을 도시하시오 (make-set, union, find-set 등 사용)
  - ➢ {a, e, f}, {h, i, g, b}



수행 시간

◈ 수행 시간 분석

- Make-Set(x)
  - 하나의 원소 x를 지니는 노드 하나 구성 → $\Theta(1)$
- Find-Set(x)
  - 해당 원소 x에 있는 대표 원소 포인터만 리턴 → $\Theta(1)$
- Union(x, y)
  - 가장 많은 시간이 드는 작업
    - ➢ 대표 원소를 가리키는 포인터를 변경하는 작업
  - 무게를 고려한 Union (Weighted Union)
    - ➢ 원소의 개수가 많은 집합은 주집합으로 정하고, 원소의 개수가 작은 집합을 부집합으로 정하는 것이 효율적
  - 부집합의 원소 개수를 k라고 가정 → $\Theta(k)$

수행 시간

◈ [정리 8-1]

- 연결리스트를 이용해 표현되는 배타적 집합들을 만들면서 Weight을 고려한 Union을 사용할 때, m번의 Make-Set, Union, Find-Set 수행 중 n번이 Make-Set이라면 이들의 총 수행시간은 $\boldsymbol{O(m + nlogn)}$이다. ($m \geq n$)
  - <u>처음에 전체 원소를 하나씩 나누어서 집합을 구성하고, 최종적으로 모든 원소를 하나의 집합으로 구성할 때까지의 총 수행시간</u>

- [증명 (1/3)]
  - Make-Set(x) → $\Theta(1)$ & Find-Set(x) → $\Theta(1)$
  - m번의 Make-Set, Union, Find-Set 수행 → 이들로 인한 복잡도 $\Theta(m)$
  - Make-Set이 n번 → 원소의 총 개수: n개

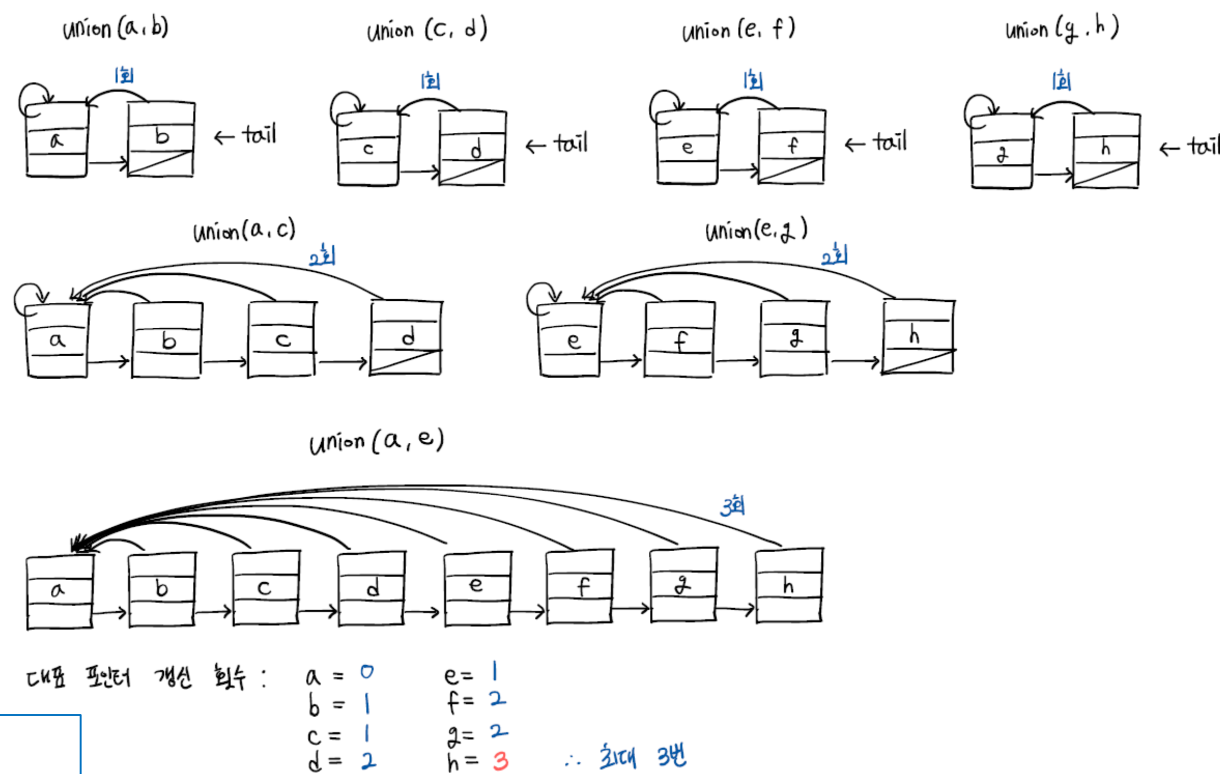
수행 시간

◈[정리 8-1]

– [증명 (2/3)]

- 임의의 원소 x에 대하여 반복적인 Union으로 발생되는 대표 노드 포인터 갱신 횟수 고찰
- 예시
  - ➢ 총 원소 개수: 8개
  - ➢ 8개의 원소 중 가장 많은 갱신이 요구되는 원소에 대한 갱신 횟수

    $log_2 8 = 3$



위 예에서 Make-Set(x) 호출 8회,
Union(x,y) 호출 7회, Find-Set(x) 호출 14회
→ $m = 8 + 7 + 14 = 29$

수행 시간

◈ [정리 8-1]

– [증명 (3/3)]

- 임의의 원소 x에 대해 반복적인 Union으로 생기는 대표 노드 포인터 갱신 횟수 고찰
- 예시
  - ➢ 총 원소 개수: 16개 → 임의의 원소에 대한 최대 갱신 횟수: $log_2 16 = 4$
  - ➢ 총 원소 개수: 32개 → 임의의 원소에 대한 최대 갱신 횟수: $log_2 32 = 5$
  - ➢ …

총 원소 개수: n개 → 임의의 원소에 대한 최대 갱신 횟수: $log_2 n$

- 따라서, 반복적인 Union 수행에 대한 점근적 복잡도: $O(nlog_2 n)$
- 종합적인 총 수행시간: $O(m + nlog_2 n)$

02. 트리를 이용한 집합의 처리

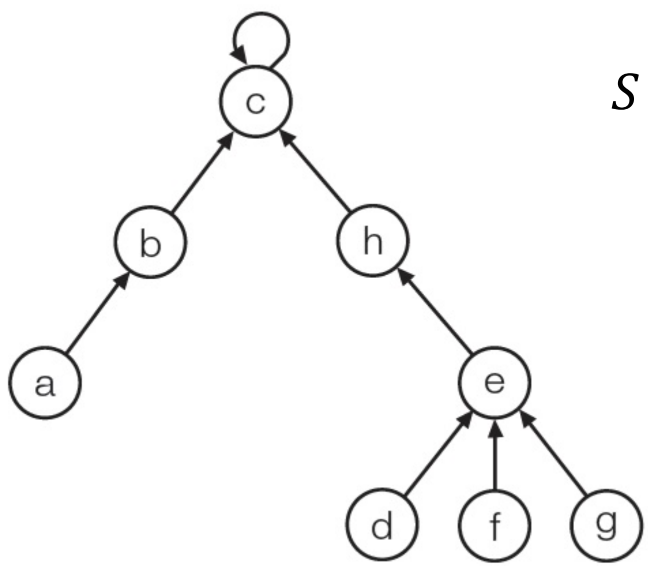
https://www.techiedelight.com/disjoint-set-data-structure-union-find-algorithm/

기본 원리

◈ 트리를 활용한 상호 배타적 집합 구성

– <u>자식 노드가 부모 노드를 포인트</u> 하도록 트리 구성

– 임의의 노드에서 부모를 가리키는 포인터를 계속 따라가면 자신이 속한 트리(즉, Disjoint Set)의 루트 노드 (즉, 대표 노드)를 만남



$S = \{a, b, \bar{c}, d, e, f, g, h\}$

그림 8–3 트리를 이용한 집합 표현의 예

기본 원리

◈ 트리를 활용한 상호 배타적 집합 구성

- Make-Set(x)
  - 하나의 원소로 구성되는 집합 생성
  - 노드를 하나 만들고 이 노드의 부모가 자신이 되도록 포인터 설정

Make-Set(x)
▷ 노드 x를 유일한 원소로 하는 집합을 만든다.
{
    $p[x] \leftarrow x$;
}

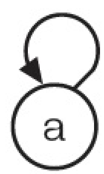


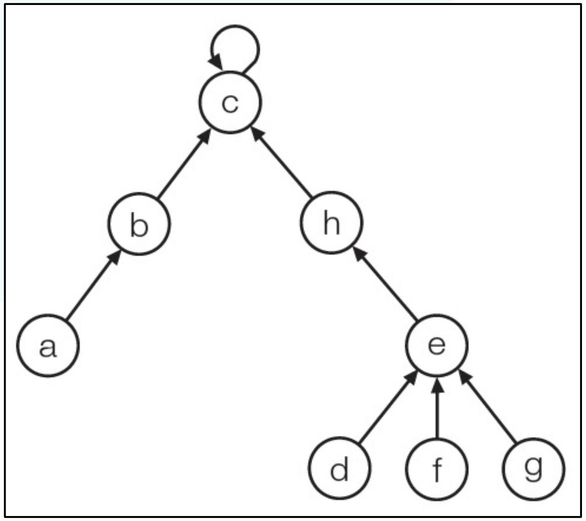
그림 8-5 트리를 이용하는 집합 표현에서 하나의 원소로 구성된 집합

기본 원리

◈ 트리를 활용한 상호 배타적 집합 구성

– Find-Set(x)
  - 노드 x가 속한 트리의 루트 노드를 리턴한다.
  - 즉, 노드 x의 부모 노드 포인터를 계속 따라가다가, 해당 포인터가 노드 자신을 가리키면 그 노드를 리턴

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



기본 원리

◈ 트리를 활용한 상호 배타적 집합 구성

- Union (x, y)
  - 두 개의 집합을 합치는 작업
  - 두 집합 중 하나의 집합의 루트 노드가 다른 집합의 루트 노드를 가리키도록 포인터 하나만 변경

Union (x, y)

▷ 노드 x가 속한 집합과 노드 y가 속한 집합을 합친다.

{

$p[\text{Find-Set}(y)] \leftarrow \text{Find-Set}(x);$

}

[예] Union(b, g)

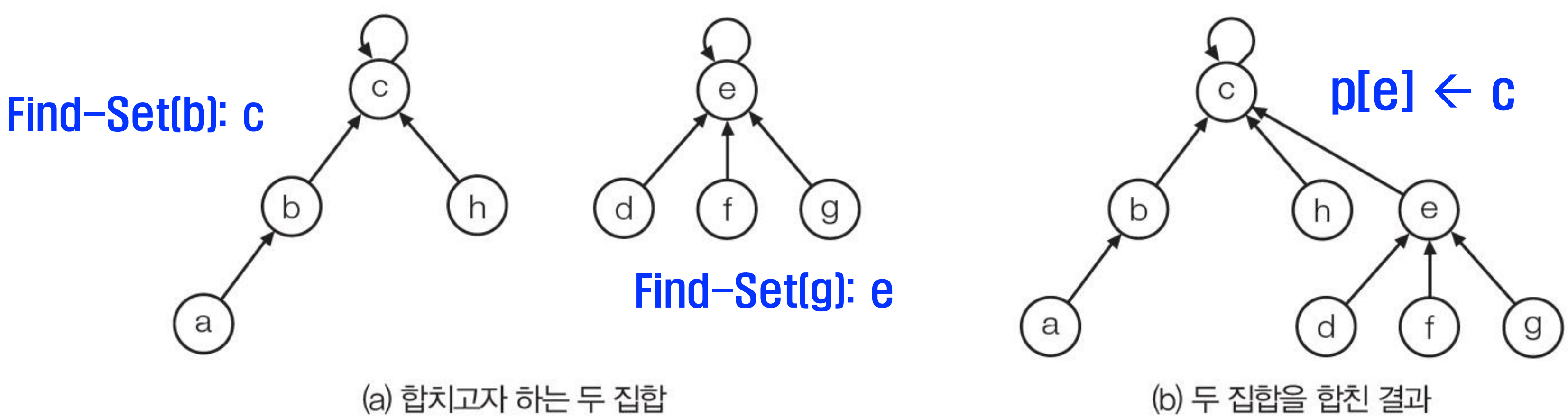


그림 8-4 트리를 이용한 표현에서 두 집합을 합치는 예

기본 원리

◈ 트리를 활용한 상호 배타적 집합 구성 코드

```
class DisjointSet:
    parent = {}   # dictionary (사전 자료구조)

    def make_set(self, universe):
        for i in universe:
            self.parent[i] = i

    def find(self, k):
        if self.parent[k] == k:
            return k
        return self.find(self.parent[k])

    def union(self, a, b):
        self.parent[self.find(b)] = self.find(a)

    @staticmethod
    def print_sets(universe, ds):
        print([ds.find(i) for i in universe])
```

```
if __name__ == '__main__':
    # universe of items
    universe = [1, 2, 3, 4, 5]

    # initialize disjoint set
    ds = DisjointSet()

    # create a singleton set for each element
    ds.make_set(universe)
    ds.print_sets(universe, ds)

    ds.union(3, 4) # 4 and 3 are in the same set
    ds.print_sets(universe, ds)

    ds.union(1, 2) # 1 and 2 are in the same set
    ds.print_sets(universe, ds)

    ds.union(3, 1) # 1, 2, 3, 4 are in the same set
    ds.print_sets(universe, ds)
```

기본 원리

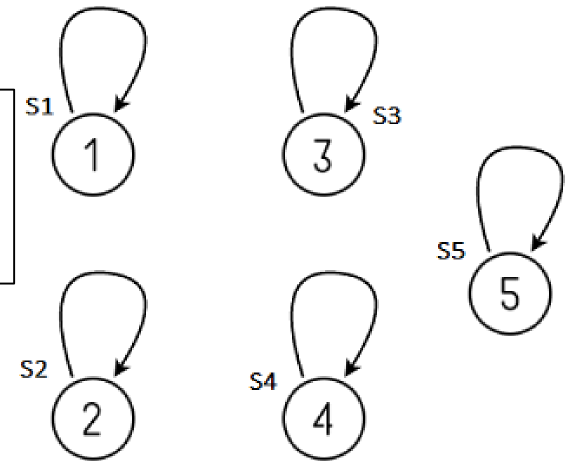
◈ 트리를 활용한 상호 배타적 집합 구성 코드 수행 예

– create a singleton set for each element

```
ds.make_set(universe)
ds.print_sets(universe, ds)

> [1, 2, 3, 4, 5]
```

```
def make_set(self, universe):
    for i in universe:
        self.parent[i] = i
```

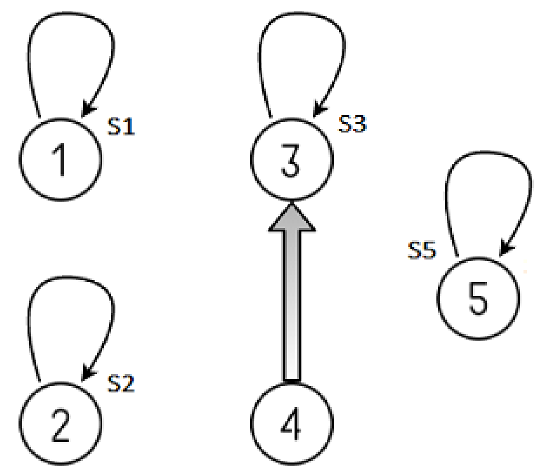


– Union(3, 4)

```
ds.union(3, 4)
ds.print_sets(universe, ds)

> [1, 2, 3, 3, 5]
```

```
def union(self, a, b):
    self.parent[self.find(b)] = self.find(a)
```



기본 원리

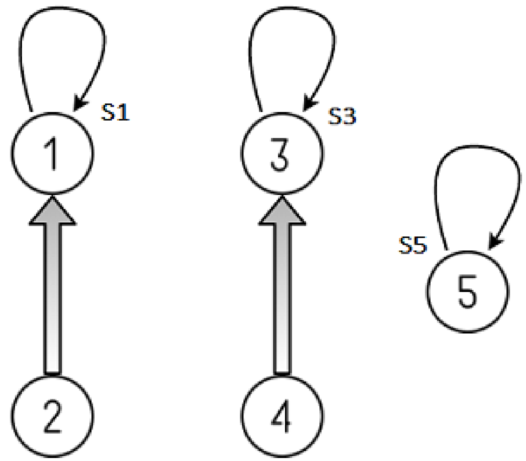
◈ 트리를 활용한 상호 배타적 집합 구성 코드 수행 예

– Union(1, 2)

```
ds.union(1, 2)
ds.print_sets(universe, ds)

> [1, 1, 3, 3, 5]
```

```
def union(self, a, b):
    self.parent[self.find(b)] = self.find(a)
```

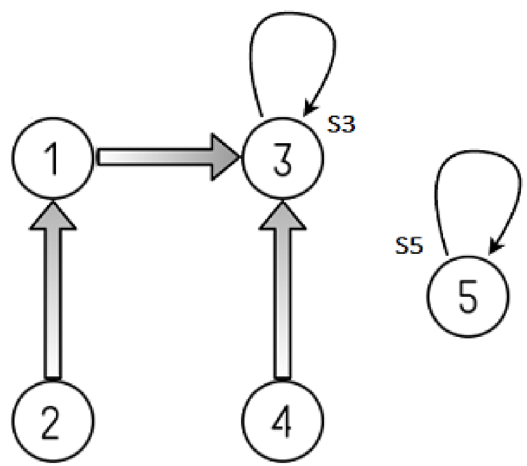


– Union(3, 1)

```
ds.union(3, 1)
ds.print_sets(universe, ds)

> [3, 3, 3, 3, 5]
```

```
def union(self, a, b):
    self.parent[self.find(b)] = self.find(a)
```

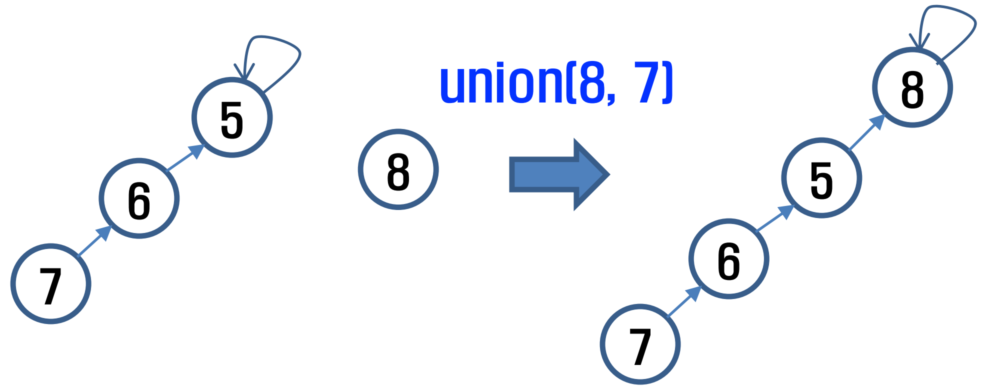


연산의 효율을 높이는 방법

◈ 기존 방법의 단점

– 트리의 균형이 심각하게 깨질 수 있음

➔ Find-Set(x) 효율 낮아짐



◈ 랭크를 이용한 Union

– 각 노드마다 **랭크** 값 유지

- 랭크: 자신을 루트로 하는 서브 트리의 높이
- 단 하나의 노드로 구성될 때 해당 노드의 랭크: 0

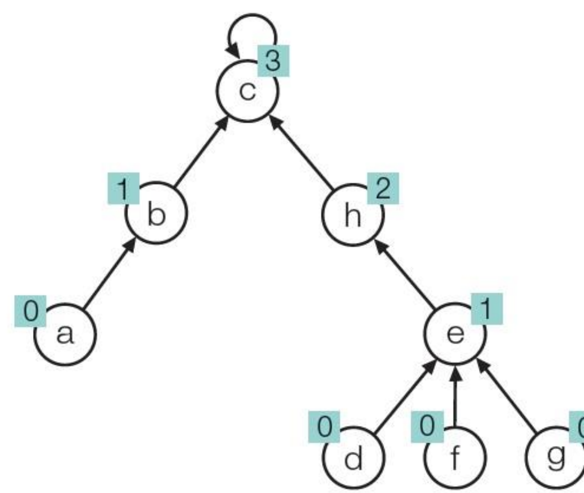
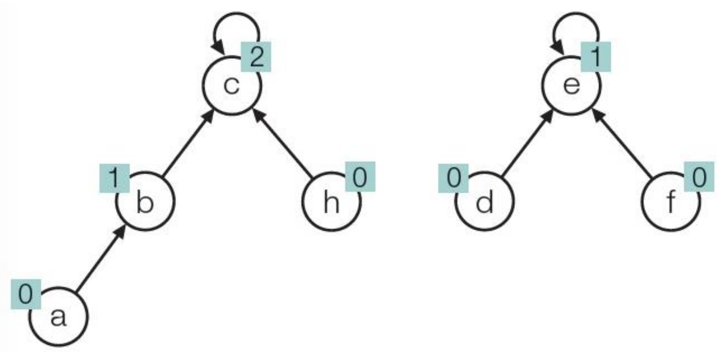


그림 8-6 집합을 표현한 트리에서 각 노드의 랭크가 표시된 예

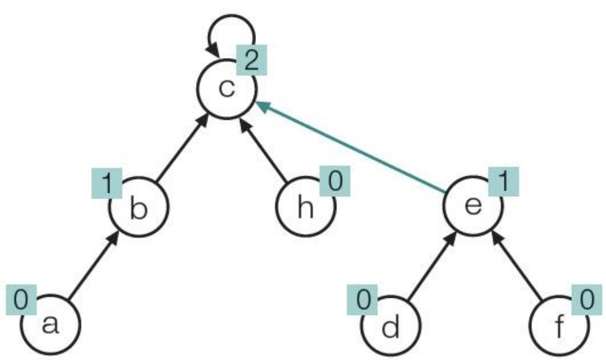
연산의 효율을 높이는 방법

◈ 랭크를 이용한 Union

- 랭크를 이용한 Union(x, y)
  - 랭크가 낮은 집합을 랭크가 높은 집합에 붙인다.



(a) 합치고자 하는 두 집합



(b) 두 집합을 합친 결과

알고리즘 8-2 랭크를 이용한 Union과 Make-Set

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

Union(x, y)
▷ 노드 x가 속한 집합과 노드 y가 속한 집합을 합친다.
{
    x′ ← Find-Set(x);
    y′ ← Find-Set(y);
    if (rank[x′] > rank[y′])
        then p[y′] ← x′;
        else {
            p[x′] ← y′;
            if (rank[x′] = rank[y′]) then rank[y′] ← rank[y′]+1;
        }
}
```

랭크가 낮은 노드의 부모를 수정

랭크가 동일할 때는 붙임을 받은 노드의 랭크를 1 증가

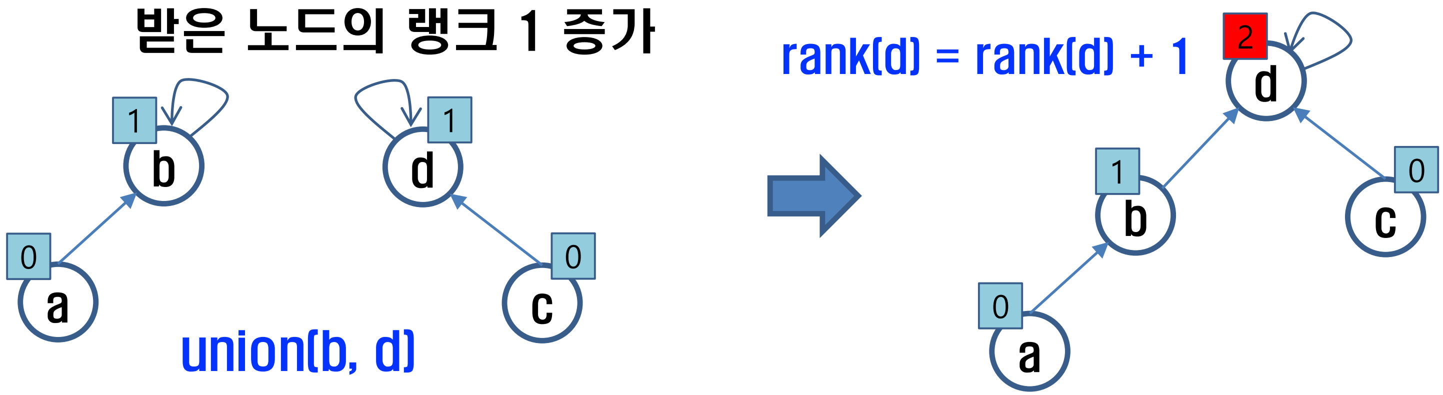
Union(c, e)
→
rank(c): 2
rank(e): 1
p[e] ← c

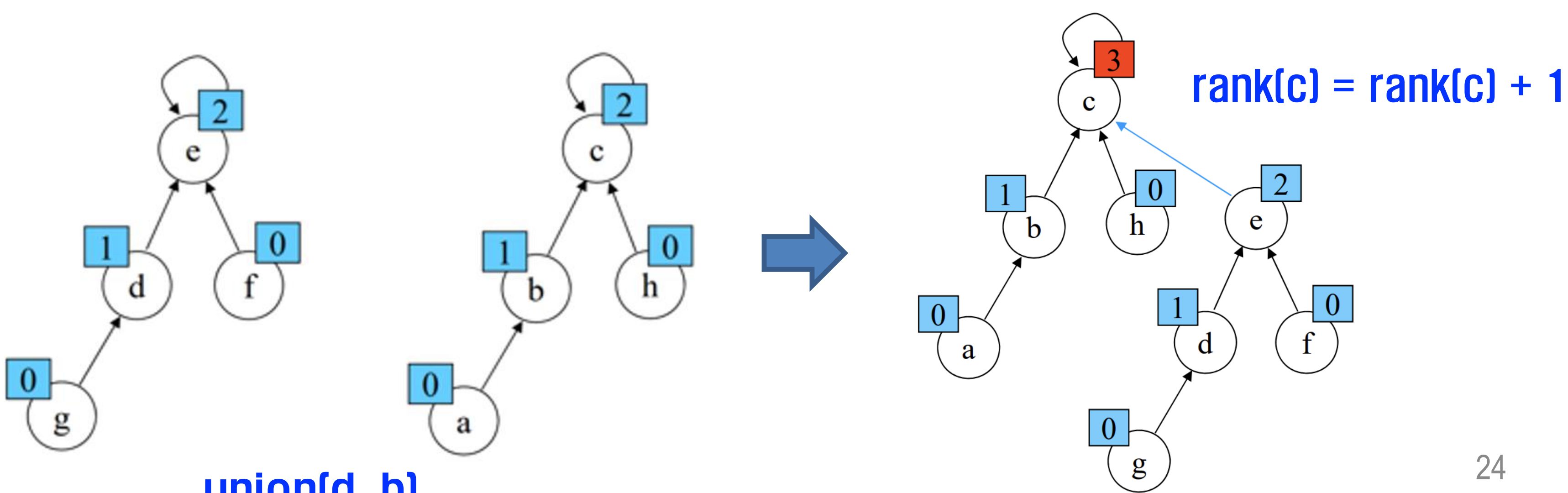
연산의 효율을 높이는 방법

```
x′ ← Find-Set(x);
y′ ← Find-Set(y);
if (rank[x′] > rank[y′])
    then p[y′] ← x′;
    else {
        p[x′] ← y′;
        if (rank[x′] = rank[y′]) then rank[y′] ← rank[y′]+1;
    }
```

◈ 랭크를 이용한 Union

- 랭크를 이용한 Union(x, y)
  - 랭크가 동일할 때는 붙임을 받은 노드의 랭크 1 증가

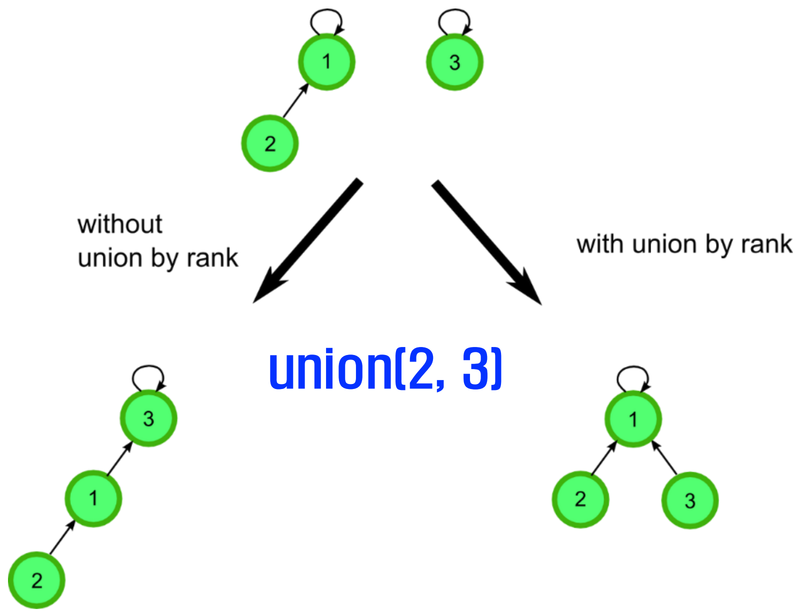


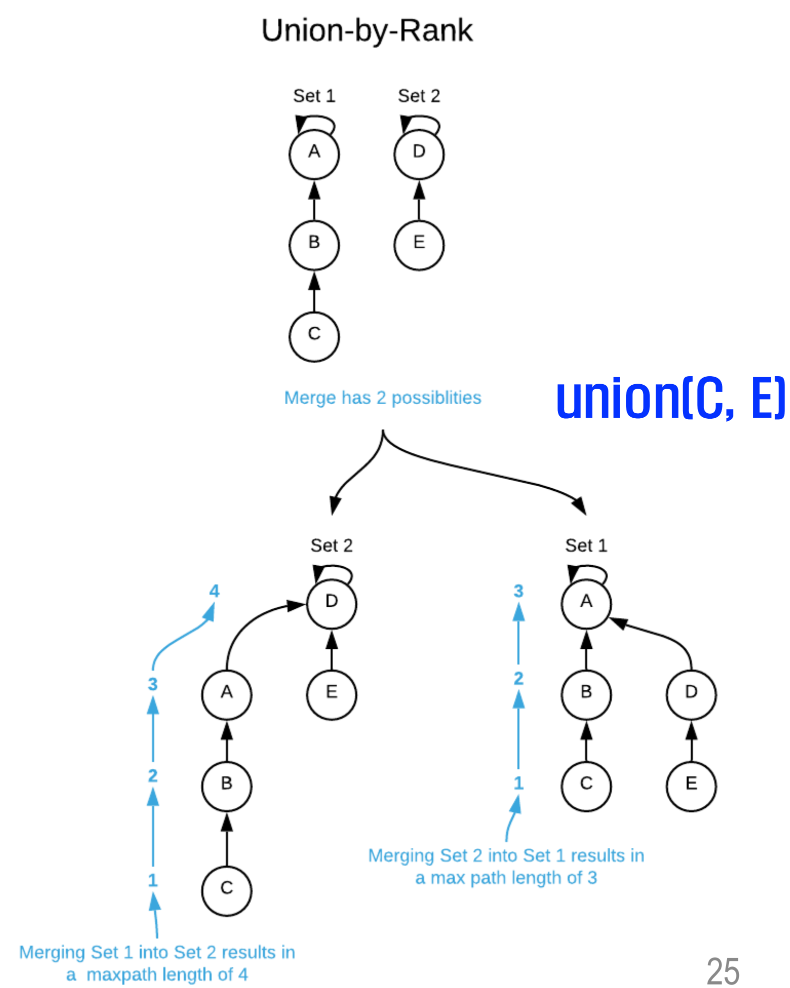


연산의 효율을 높이는 방법

◈ 랭크를 이용한 Union

– 단순 Union(x, y) vs. 랭크를 이용한 Union(x, y)





연산의 효율을 높이는 방법

◈ 경로 압축

– Find-Set(x)을 행하는 과정에 경로의 길이를 줄이는 작업 추가

- Find-Set(x)을 수행하는 과정에서 만나는 모든 노드가 직접 루트를 가리키도록 포인터 변경

Find−Set(x)
▷ 노드 x가 포함된 트리의 루트를 리턴한다.
{
    if ($p[x] \neq x$) then $p[x] \leftarrow$ Find−Set($p[x]$);
    return $p[x]$;
}

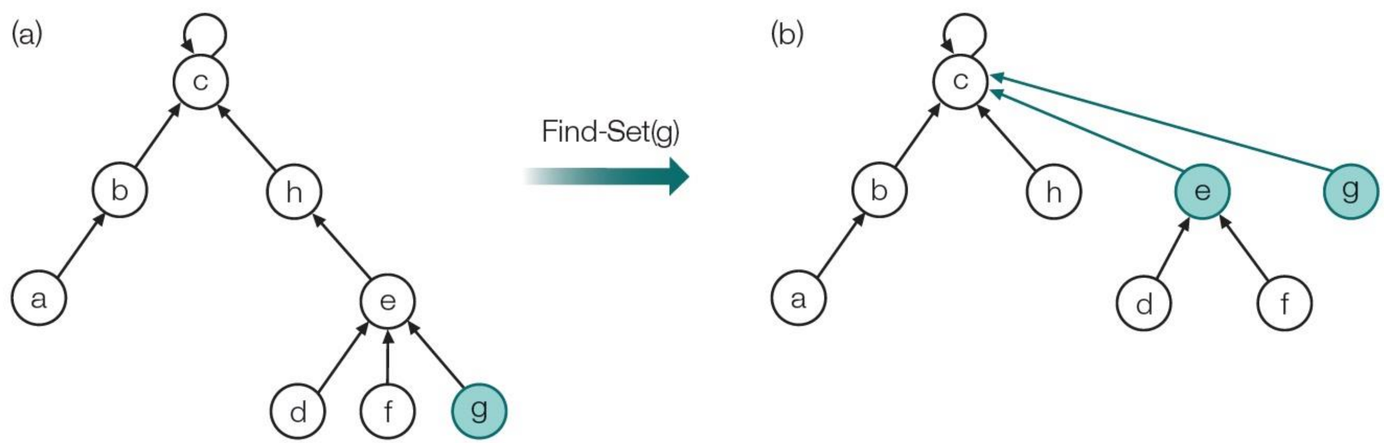


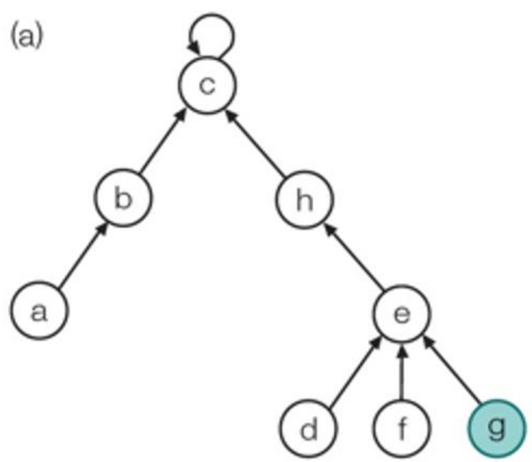
그림 8−8 트리를 이용한 표현에서 경로 압축을 사용하는 Find−Set(g) 직후의 모양

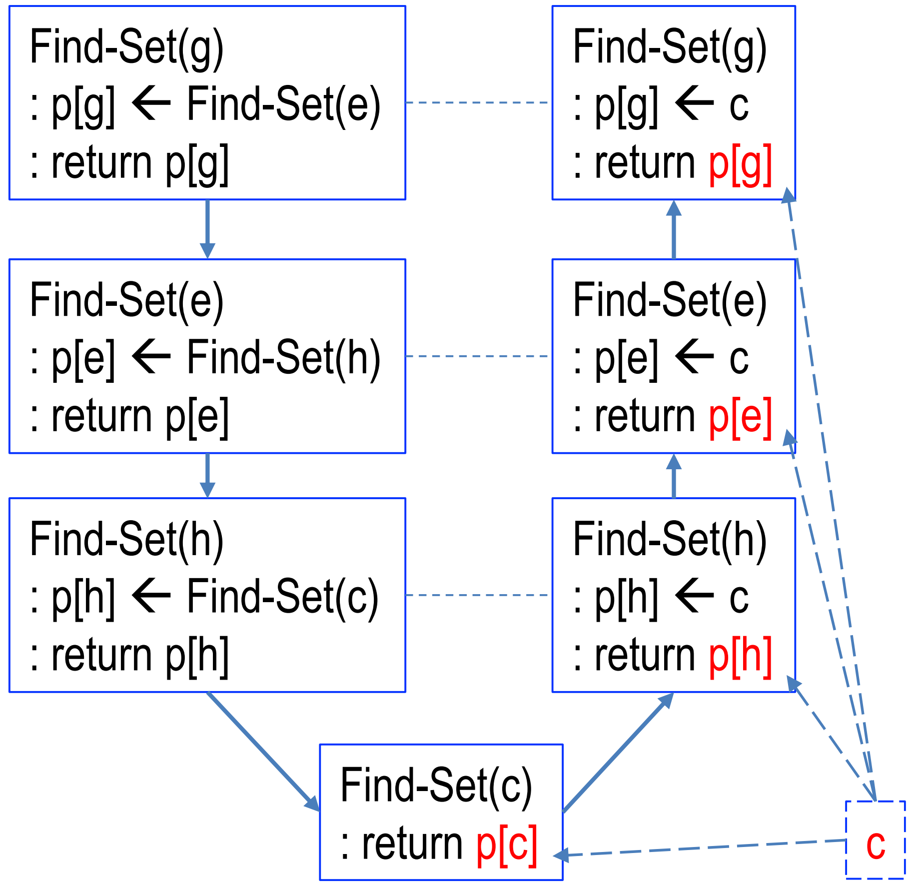
연산의 효율을 높이는 방법

◈ 경로 압축

– Find-Set(x)을 행하는 과정에 **경로의 길이를 줄이는 작업** 추가

Find−Set(x)
▷ 노드 x가 포함된 트리의 루트를 리턴한다.
{
    if ($p[x] \neq x$) then $p[x] \leftarrow$ Find−Set($p[x]$);
    return $p[x]$;
}

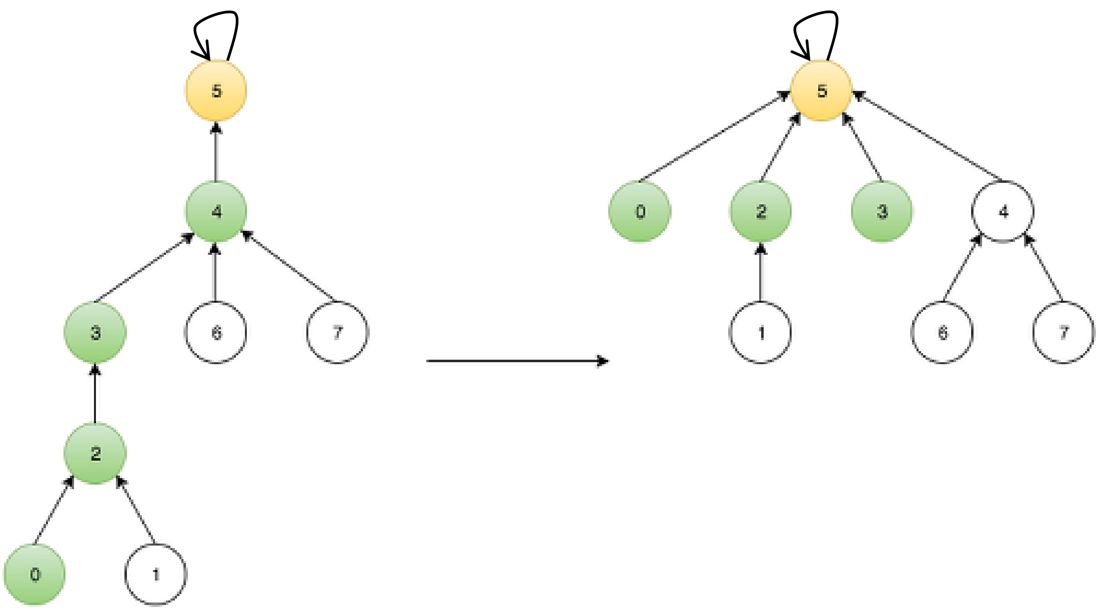




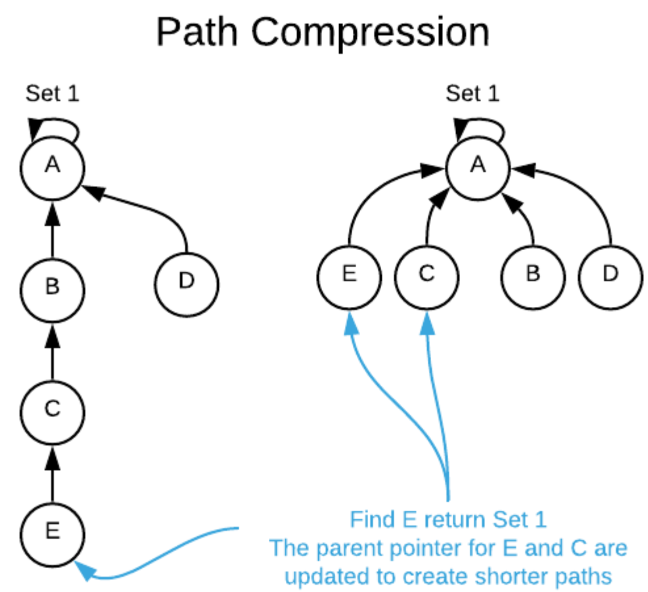
연산의 효율을 높이는 방법

◈ 경로 압축

– Find-Set(x)을 행하는 과정에 **경로의 길이를 줄이는 작업** 추가



(d) find (0) 수행 시 경로 압축 최적화

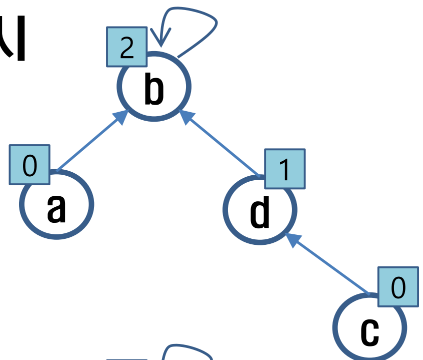


효율성 분석

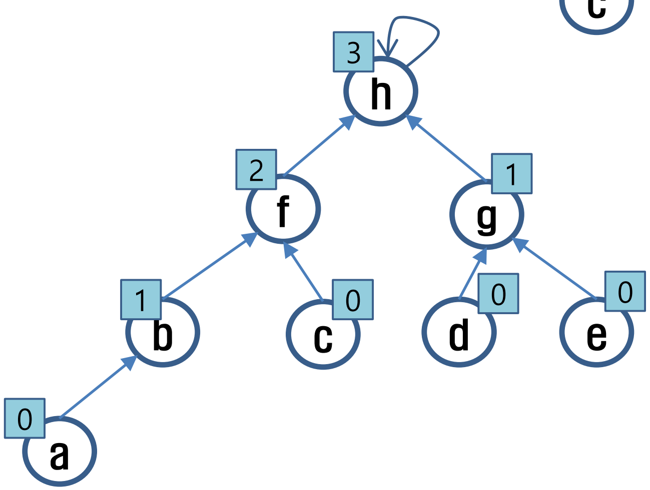
◈ [정리 8-2]

– 랭크를 이용한 Union을 사용하면 **랭크가 k인 노드를 대표로 하는 집합의 원소 수는 최소한 2^k**이다.

- 예시



rank(b) = 2 　　 왼쪽 집합의 원소 수: $2^2 = 4$



rank(h) = 3 　　 왼쪽 집합의 원소 수: $2^3 = 8$

효율성 분석

◈ [정리 8-2]

- 랭크를 이용한 Union을 사용하면 **랭크가 k인 노드를 대표로 하는 집합의 원소 수는 최소한 2^k**이다.
  - [증명 (수학적 귀납법)]
    - [경계조건 증명] 랭크가 0이면 원소가 1개: $2^0 = 1$ → 즉 위 정리가 증명된다.
    - [귀납적 가정] 랭크가 $r(< k)$인 노드를 대표로 하는 집합의 원소 수는 최소한 2^r이다.
    - [전개]
      - » 1) 랭크가 r인 노드를 대표로 하는 집합과 랭크가 r보다 작은 노드를 대표로 하는 집합을 Union하면 랭크는 여전히 r이며, 위 귀납적 가정에 의해 최소 원소의 개수는 2^r이다.
      - » 2) 랭크가 r인 노드를 대표로 하는 두 개의 집합을 Union하면 두 대표 노드 중 하나의 랭크가 $r + 1$로 증가함. 이 때 두 집합은 최소 원소의 개수가 각각 2^r이므로, 최소한 $2 \cdot 2^r = 2^{r+1}$의 원소 개수를 지닌다.
      - » 따라서, 랭크가 $r + 1(= k)$인 노드를 대표로 하는 집합의 원소 수는 최소한 $2^{r+1(=k)}$이다.

효율성 분석

◈ [정리 8-3]

– 랭크를 이용한 Union을 사용하면 **원소의 수가 n인 집합**을 표현하는 트리에서 **임의의 노드의 랭크는 $O(logn)$**이다.

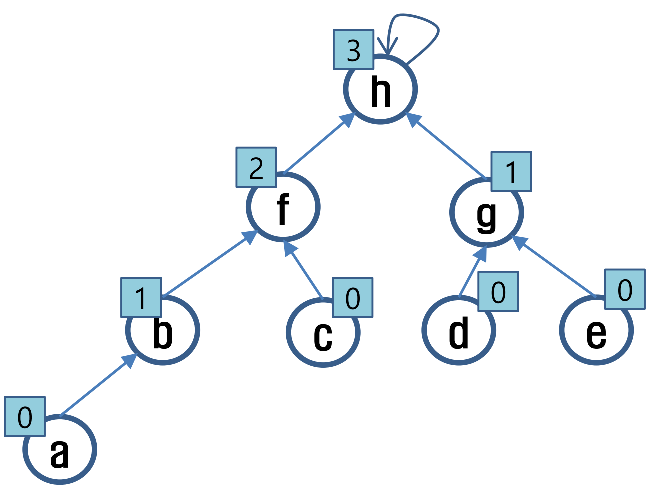
- [증명]
  - ➢ 원소의 수를 2^k이라고 가정 → $2^k = n$ → $k = logn$

정리 8-2를 다시 요약하면

만약 어떤 집합의 대표 노드의 랭크가 k → 그 집합의 원소 수는 최소한 2^k

즉, 만약 어떤 집합의 원소 수가 $2^k(= n)$ → 그 집합의 대표 노드의 랭크는 최대 $k(= logn)$

→ 임의의 노드의 랭크는 $O(logn)$



왼쪽 집합의 원소 수: $2^3 = 8$

rank(h) = 3

효율성 분석

◈ [정리 8-4]

- 트리를 이용해 표현되는 배타적 집합들을 만들면서 랭크를 고려한 Union을 사용할 때, m번의 Make-Set, Union, Find-Set 수행 중 n번이 Make-Set이라면 이들의 총 수행시간은 $\boldsymbol{O(m\log n)}$이다. ($m \geq n$)
  - 처음에 전체 원소를 하나씩 나누어서 집합을 구성하고, 최종적으로 모든 원소를 하나의 집합으로 구성할 때까지의 총 수행시간

- [증명 (1/2)]
  - Make-Set(x) → $\Theta(1)$ & Union(x, y) → $\Theta(1)$
  - m번의 Make-Set, Union, Find-Set 수행 → 이들로 인한 복잡도 $\Theta(m)$
  - Make-Set이 n번 → 원소의 총 개수: n개

효율성 분석

◈ [정리 8-4]

– [증명 (2/2)]

- [정리 8-3]에 따르면 임의의 노드의 랭크는 $O(logn)$이므로 Find-Set(x)의 수행 시간도 $O(logn)$이다.

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

- 그러므로, 종합적인 총 수행시간: $O(mlogn)$

위 식에서 m과 $logn$ 사이가 곱하기인 이유: m번의 Make-Set, Union, Find-Set 수행 중 일정 횟수의 Find-Set 수행이 있을 것이고 그 수행마다 $O(logn)$ 만큼의 수행 시간이 소요되므로

효율성 분석

◈ [정의]

- $log^*n = min\{k | \underbrace{loglog\cdots logn}_{k} \le 1\}$
- 즉, n에 log를 계속 적용할 때 최초로 1 이하가 되는 k값
- 예: $2^{65536} \approx \infty$
  - $log^*2^{65536} = 5$
- 따라서, $O(log^*n) \approx O(1)$

◈ [정리 8-5]

- 트리를 이용해 표현되는 배타적 집합들을 만들면서 랭크를 고려한 Union과 경로 압축을 이용한 Find-Set을 동시에 사용하면, m번의 Make-Set, Union, Find-Set 수행 중 n번이 Make-Set일 때 이들의 총 수행시간은 $O(mlog^*n)$이다. ($m \ge n$)
  - 위 정의에 따르면 결국 $O(mlog^*n) \approx O(m)$

Questions & Answers