Union-Find

그룹화된 정보를 저장하기 위한 자료구조

자료구조 의미를 다시 생각해보자.

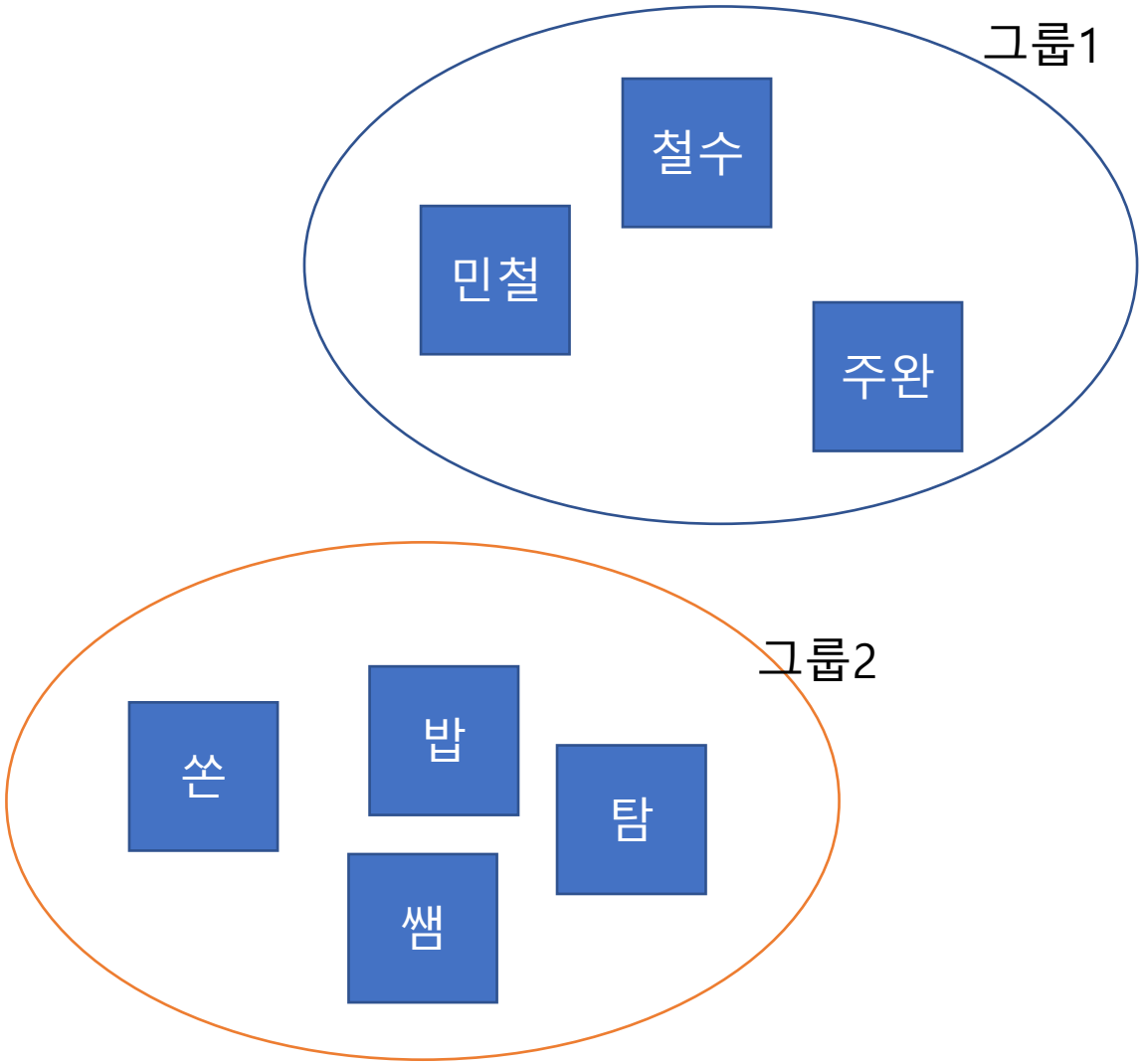
자료구조란? 데이터를 저장하고 관리하는 방법

- 배열 : 값을 저장, 순차적 저장
- 링크드리스트 : 값을 저장, 순차적 저장
- 그래프 : 값을 저장, 노드 관계 정보도 함께 저장
- Union-Find : 값을 저장, 노드 끼리 그룹 정보도 함께 저장

Union-Find 자료구조란?

여러 노드 값들이,
그룹으로 분류된 정보가 있을 때,

노드 값 뿐만 아니라,
그룹 정보도 함께 저장한다.



Union-Find 설명 방법

Union-Find 는 점점 구체화 되는 방식으로 설명한다.

1. **Union과 Find 기능 소개** : Union 기능과 Find 동작 설명
2. **대표 선정 규칙** : 대표 노드 선정 규칙
3. **내부 구현 원리** : Tree 동작 이해
4. **Tree 구현 이해** : Tree 구현 방법과 소스코드

Union과 Find 기능 이해

Union-Find가 어떤 정보를 저장하고, 어떤 정보를 알 수 있는지 확인

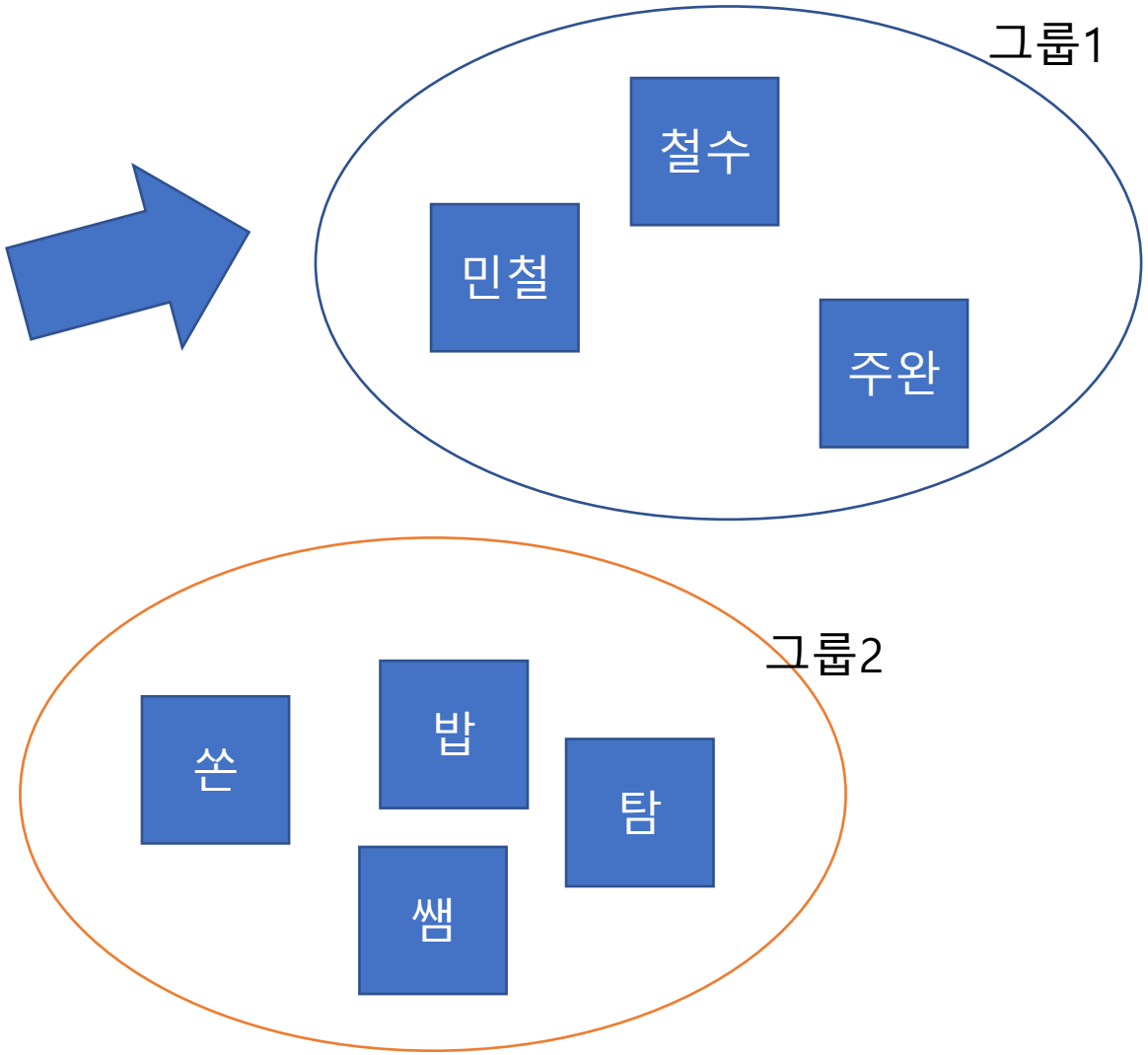
Union-Find 저장 방식 1

노드 2개씩 선택하여, 그룹화한다.

오른쪽 그림처럼 그룹화 하기 위해서
아래와 같이 두 명씩 그룹화 한다.

1. 민철, 철수는 같은 그룹이다.
2. 주완, 철수는 같은 그룹이다.

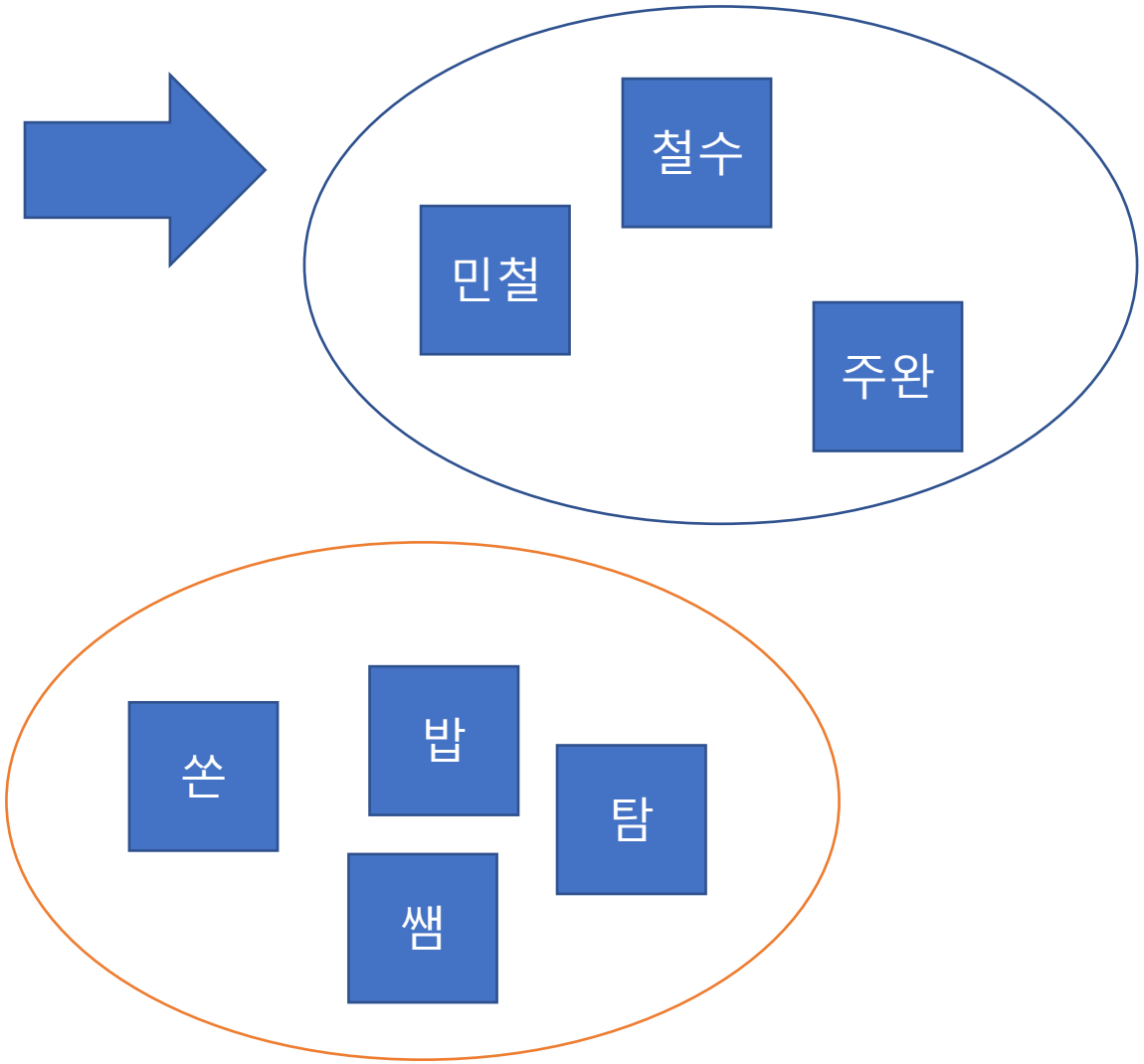
결과 : 1번과 2번을 수행하면,
{민철, 철수, 주완}은 모두 같은 그룹이다.



Union-Find 저장 방식 2

다음과 같이 순서를 바꾸어도 된다.

1. 주완, 민철은 같은 그룹이다.
2. 주완, 철수은 같은 그룹이다.



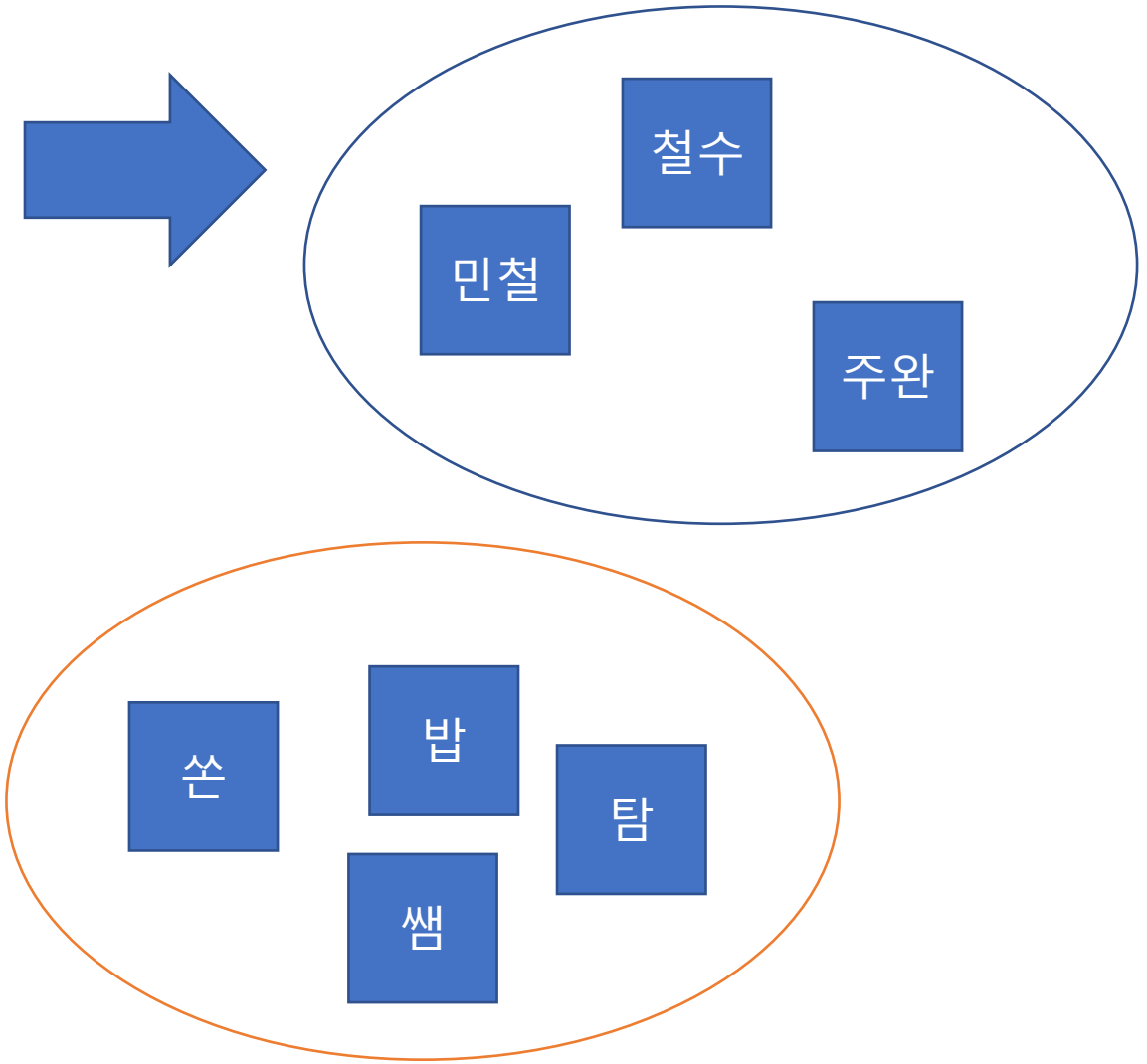
이전과 같이,

{민철, 철수, 주완}은 모두 같은 그룹이다.

Union-Find 저장 방식 3

두 명씩 그룹을 만드는 것을
다음과 같이 표현할 수 있다.

1. union(주완, 민철)
2. union(주완, 철수)



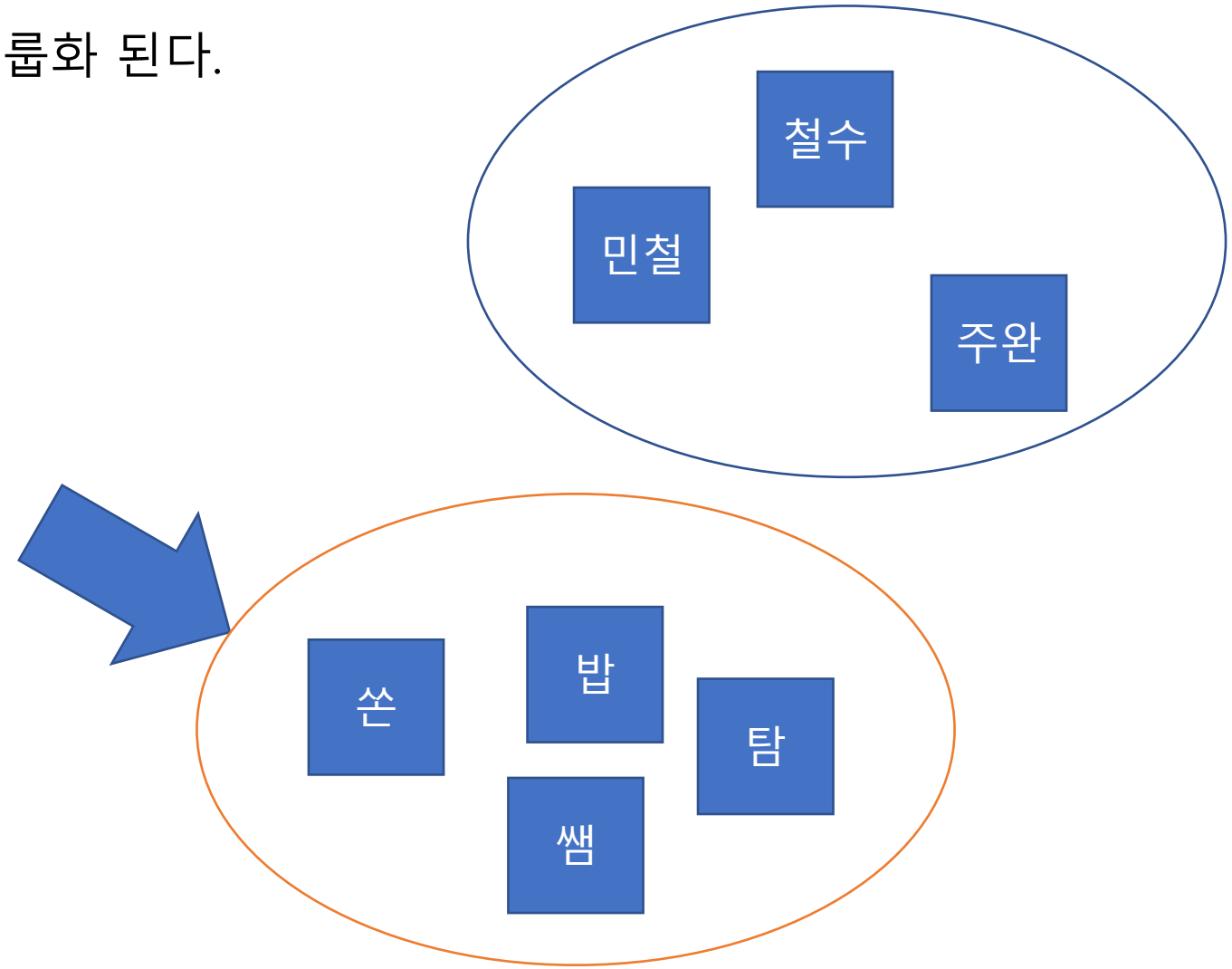
Union-Find 저장 방식 4

다음 내용을 수행하면 아래 그림처럼 그룹화 된다.

1. union(쏜, 밥)
2. union(탐, 쌤)
3. union(탐, 쏜)

1 ~ 4 번을 수행하면

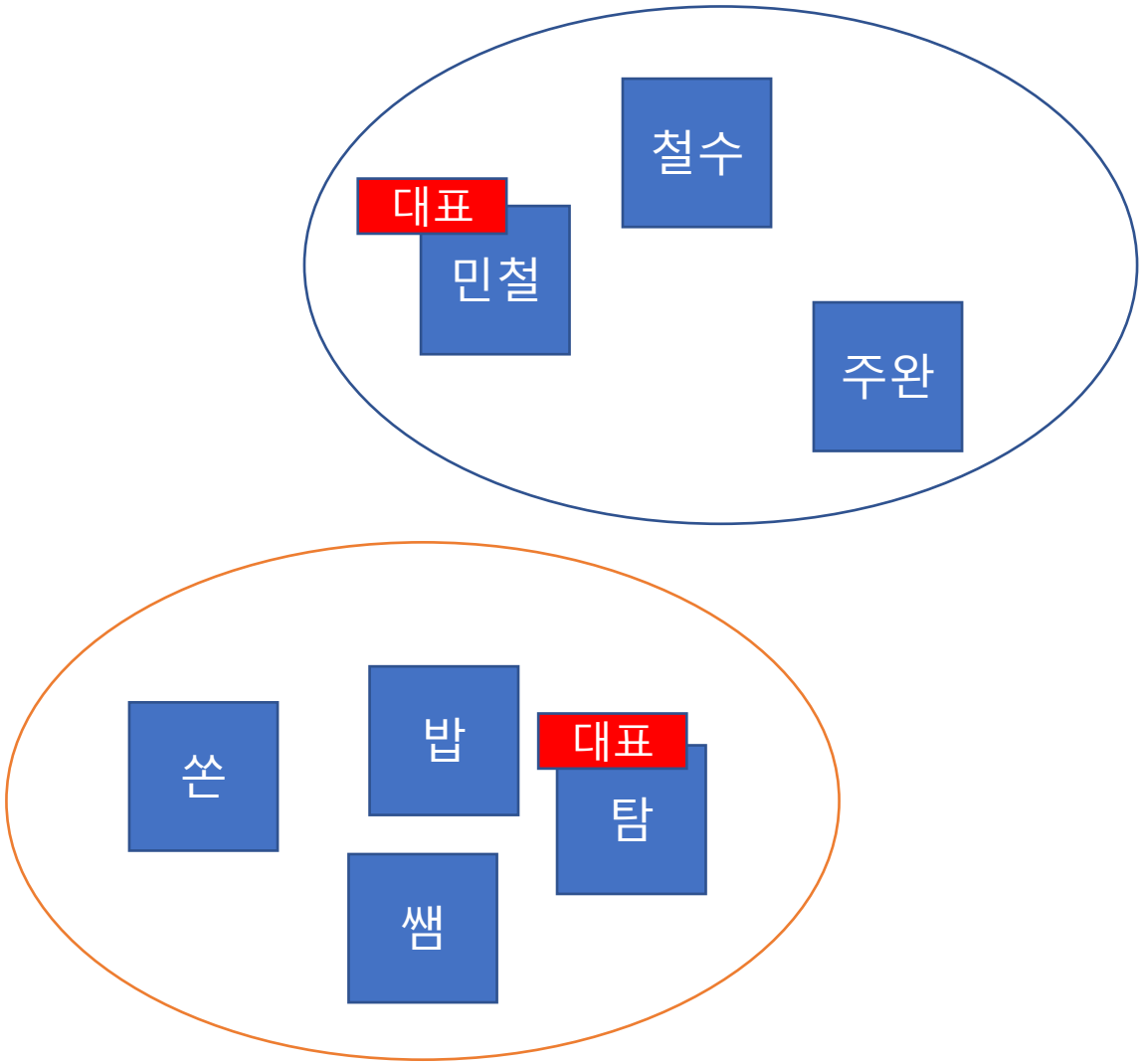
네 사람 모두 같은 그룹이다.



그룹 대표자

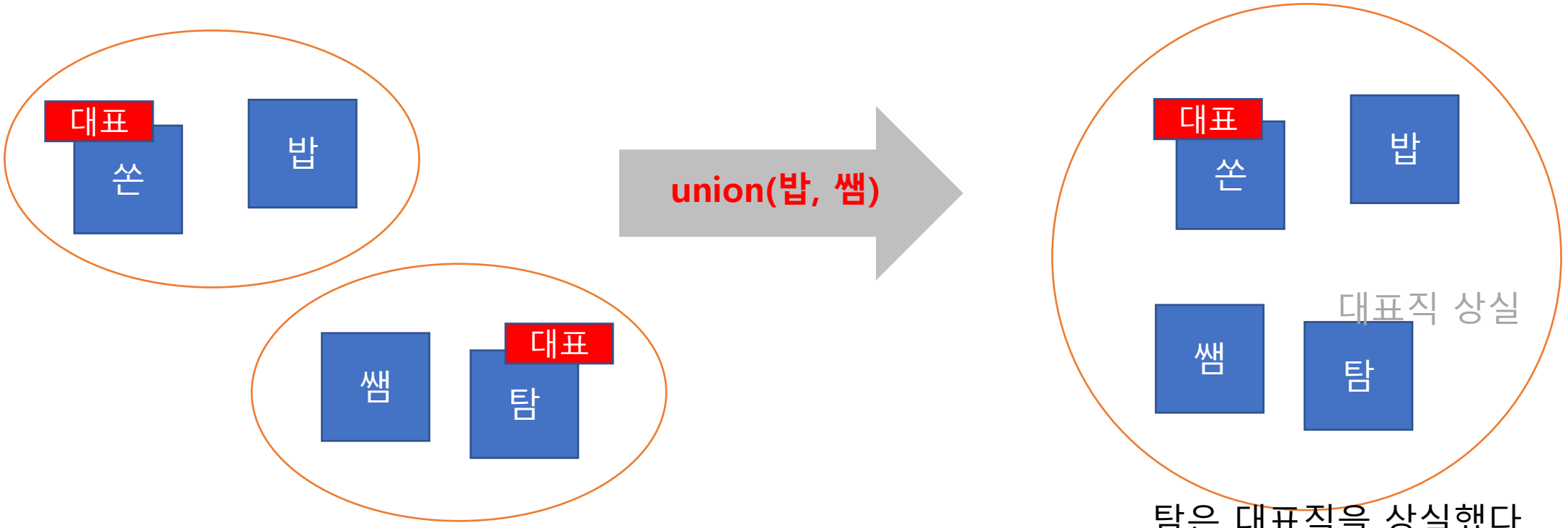
각 그룹에는 대표 노드가 존재한다.

대표노드는 하나의 그룹 상징한다.



그룹 합병

대표가 아니더라도,
그룹원 한 사람이, 다른 그룹원 사람과 union 되면, 그룹 자체가 union 된다.

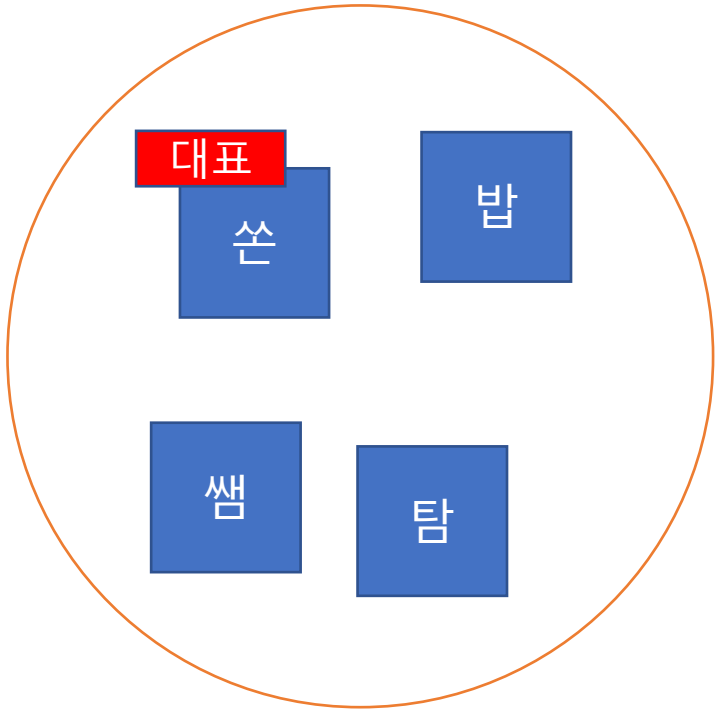


탐은 대표직을 상실했다.

find 기능

Union-Find 는 두 가지 기능을 합쳐서 부르는 용어이다.

1. Union(A, B) : A와 B는 같은 그룹이 된다.
2. Find(A) : A의 대표를 리턴한다.

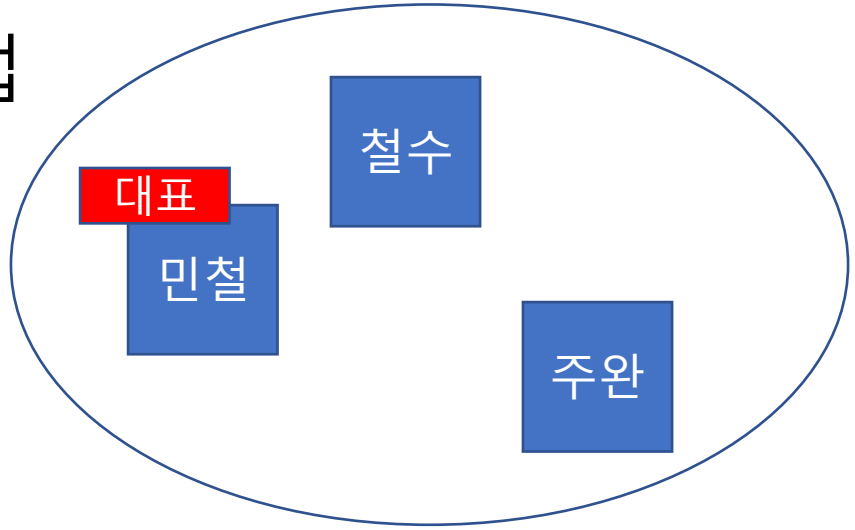


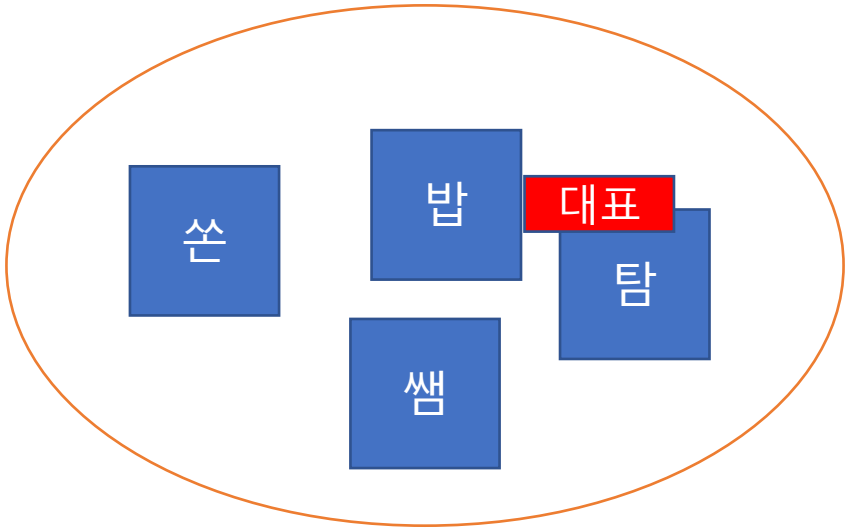
find('밥') 의 결과 → 쏜
find('쏜') 의 결과 → 쏜
find('탐') 의 결과 → 쏜
find('쌤') 의 결과 → 쏜

같은 그룹인지 확인하기

철수와 주완이 같은 그룹인지 확인하는 방법

```
if ( find('철수') == find('주완') ) {
    //같은 그룹!
}
else {
    //다른 그룹
}
```



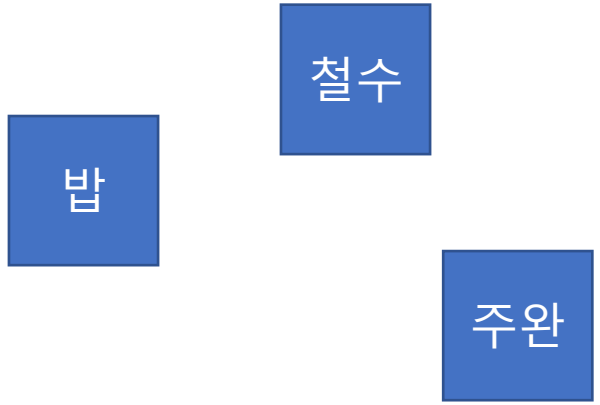


대표 선정 규칙

Union Find 에서 대표 노드 선정하는 규칙

조직생활

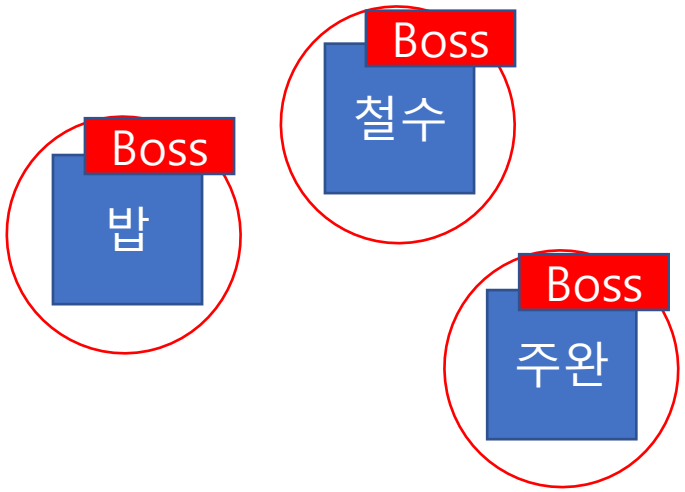
Union Find 를 쉽게 이해하기 위해 각 노드는,
조직생활에 몸을 담은 영화 속 사람들이라고 생각하자.



초기 상태

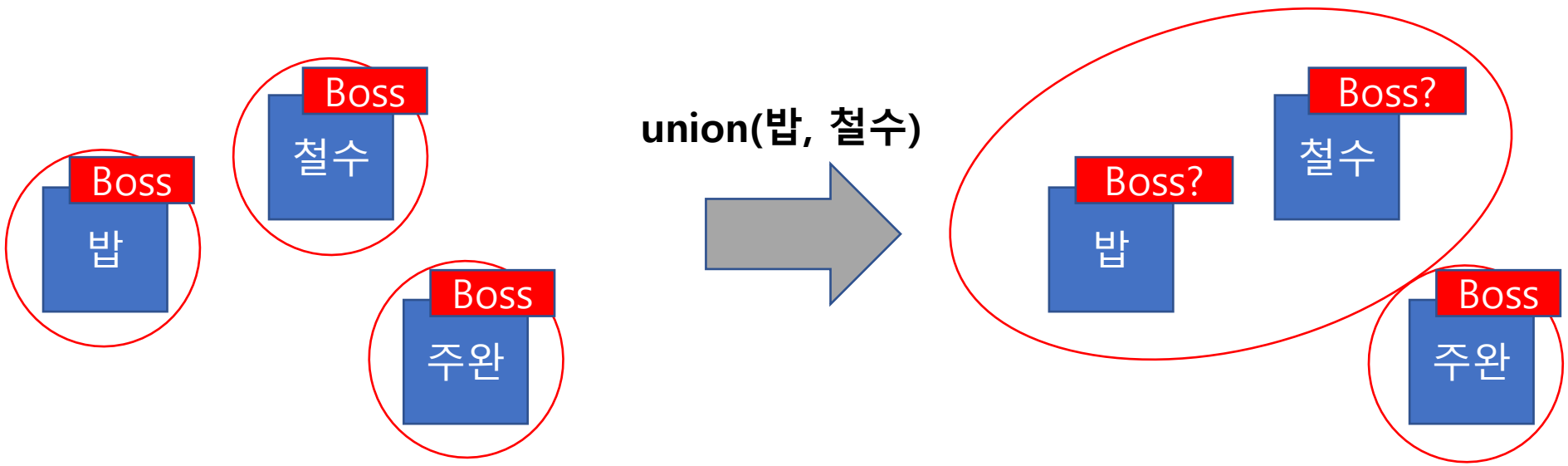
세 사람은 각자 그룹핑이 이미 되어 있는 상태이다.

- 밥은 '밥' 조직에 속한 맴버이자, Boss(대표 노드)이다.
- 철수는 '철수' 조직의 Boss 이다.
- 주완은 '주완' 조직의 Boss 이다.



Union 수행

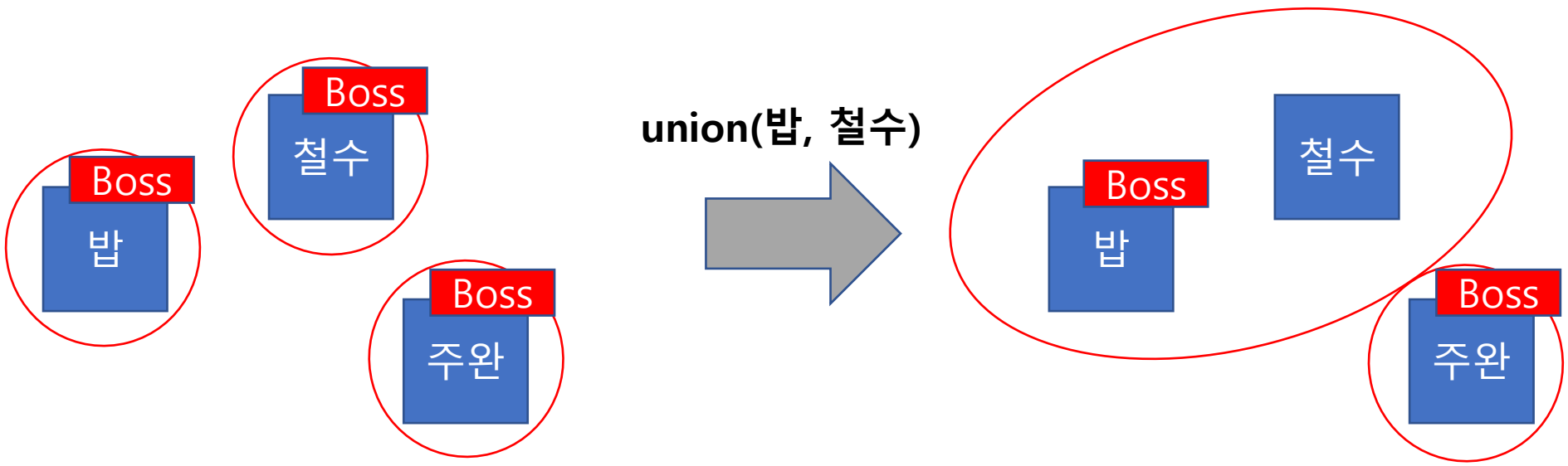
처음 Union 을 수행하면, 두 명중 한 명이 Boss가 되어야한다.



[중요] Boss 선정 규칙

Union(A, B) 를 수행하면 다음 순서대로 진행된다.

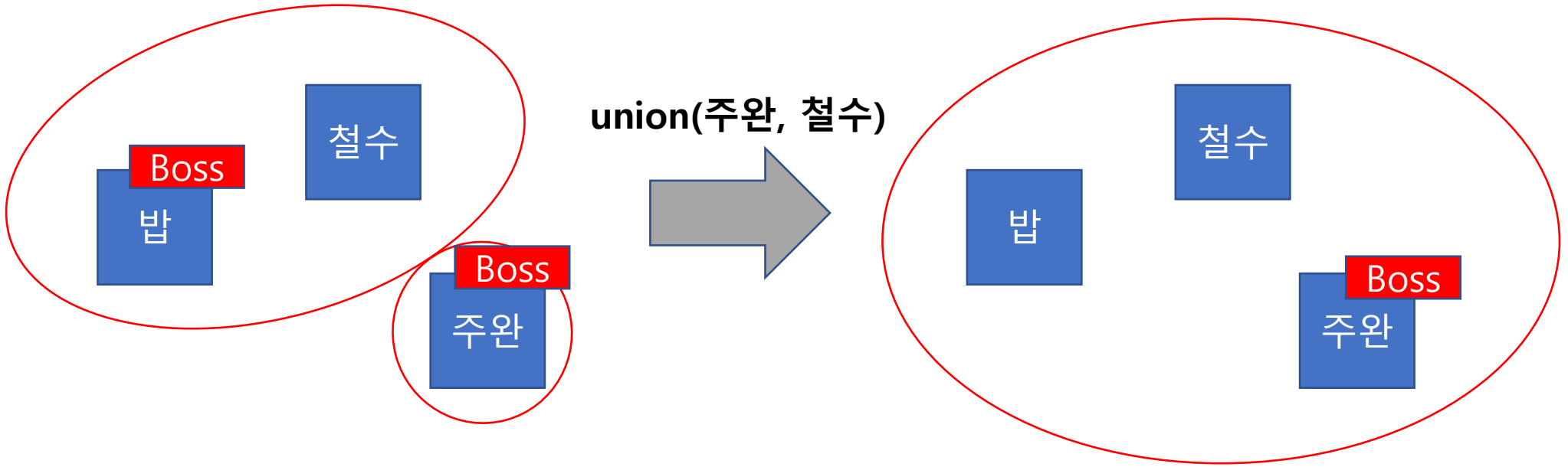
1. A의 보스와 B의 보스를 각각 찾아낸다.
2. B의 보스가 A의 보스 밑으로 들어가고, A는 통합된 조직의 Boss 이다.



Union 한번 더 수행

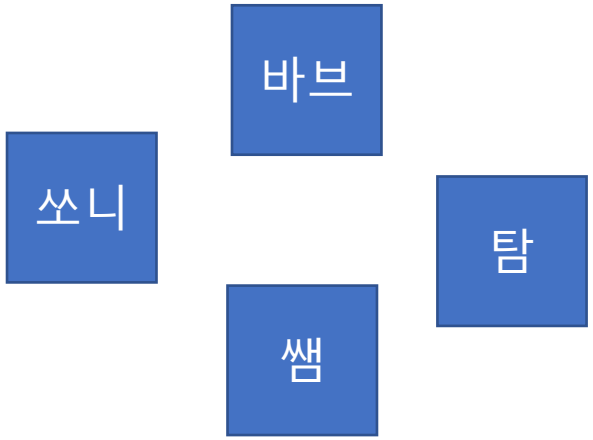
union(주완, 철수) 수행시,

- 주완의 Boss 는 '주완' 이다.
- 철수의 Boss 는 '밥' 이다.
- 따라서 '밥' 조직은 '주완' 조직으로 통합된다.



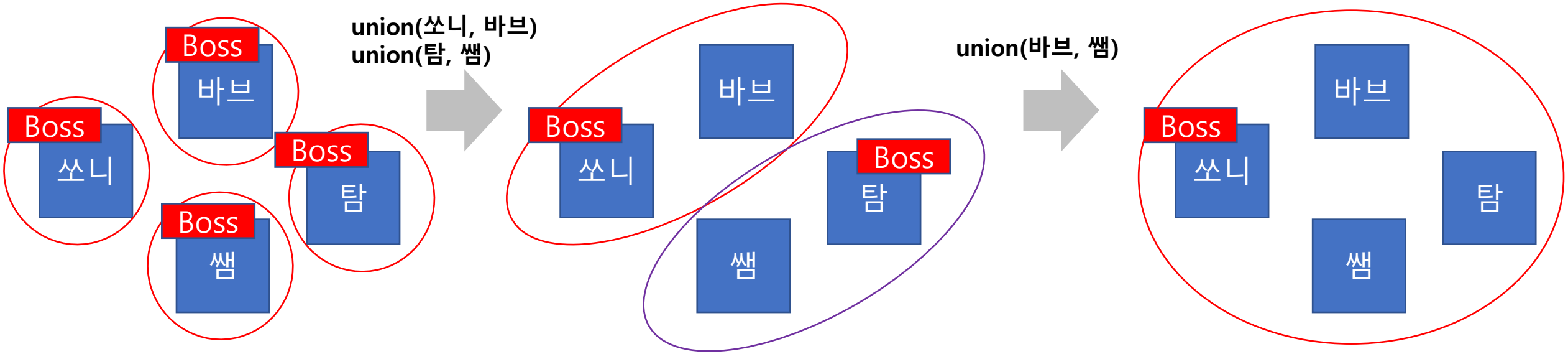
다음 union 수행시 최종 Boss는 누구인가?

union('쏘니', '바브')
union('탐', '쌤')
union('바브', '쌤')



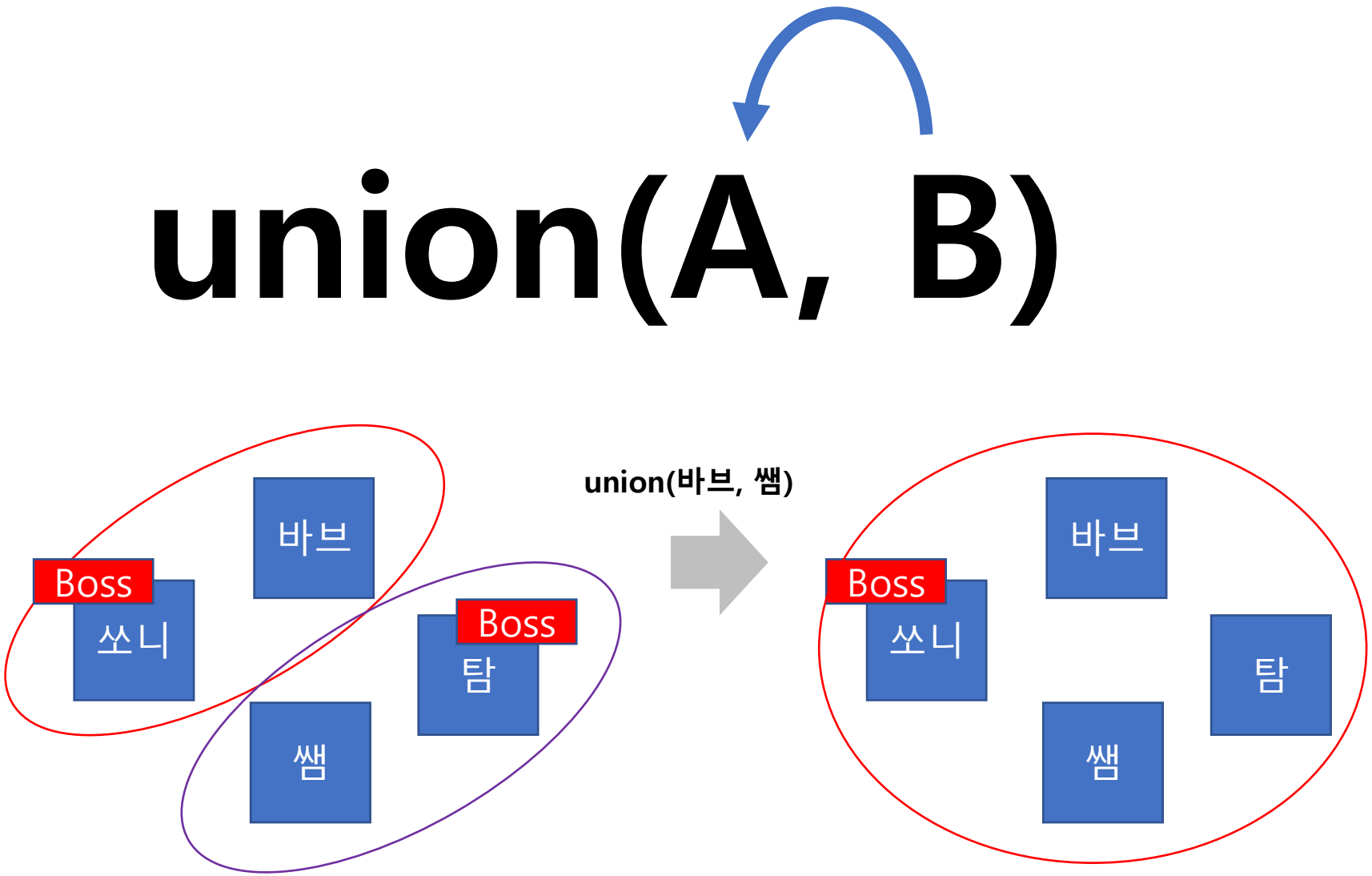
최종 Boss는 '쏘니' 이다.

union('바브', '쌤') 수행시,
'바브'의 boss는 '쏘니' 이며, '쌤'의 boss는 '탐' 이므로,
'탐'조직은 '쏘니' 조직 밑으로 들어간다. 따라서 '쏘니'가 boss이다.



정리, Union

B의 보스가, A의 보스 밑으로 들어감
(반대로 구현해도 되지만, 이 방향을 통일하도록 하자.)

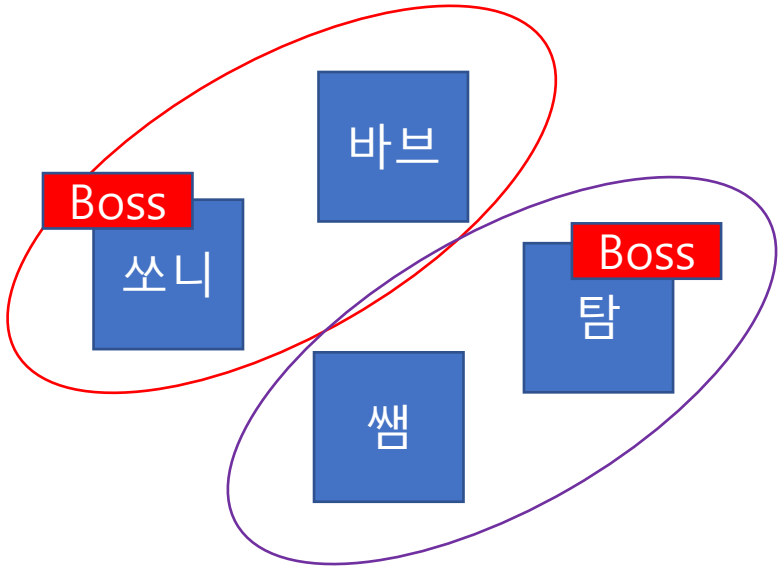


복습, 같은 조직인지 판별하는 방법

find('A') 은 'A'의 보스가 누구인지 알려준다.

쏘니와 쌤이 같은 조직인지 아닌지 확인하는 방법

- find(쏘니) == find(쌤) 이 참이라면, 경우 같은 조직이다.




```
if ( find('쏘니') == find('쌤') ) {
    //같은 그룹!
}
else {
    //다른 그룹
}
```

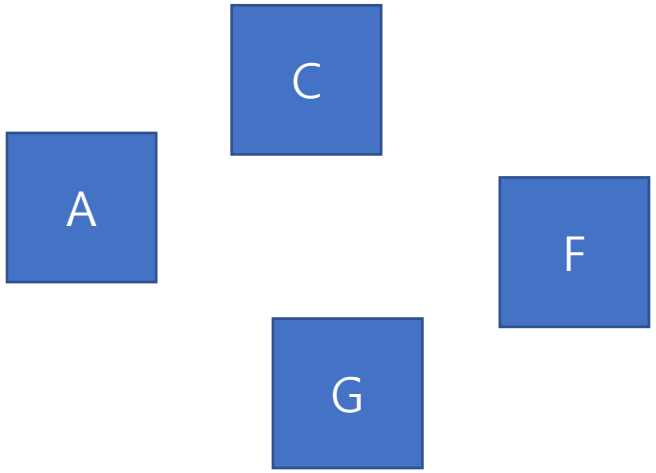
내부 구현 원리

Union과 Find 함수 구현 방법 소개

Union-Find는 Tree로 관리한다.

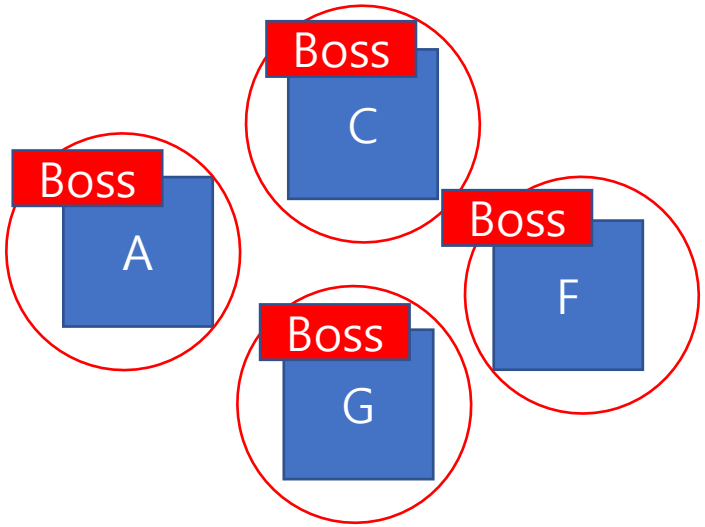
Union-Find 자료구조는 트리로 관리하면 가장 좋은 성능이 나온다.

- 그룹화된 정보를 트리로 관리한다.

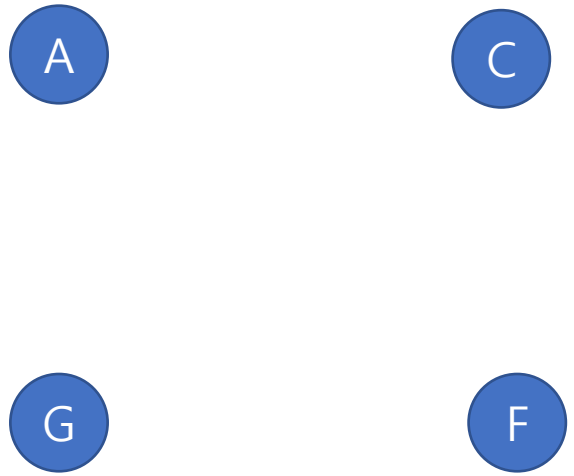


초기 상태

각 조직원들은
각자의 조직의 boss 이다.



그룹핑된
정보를 Tree로 저장



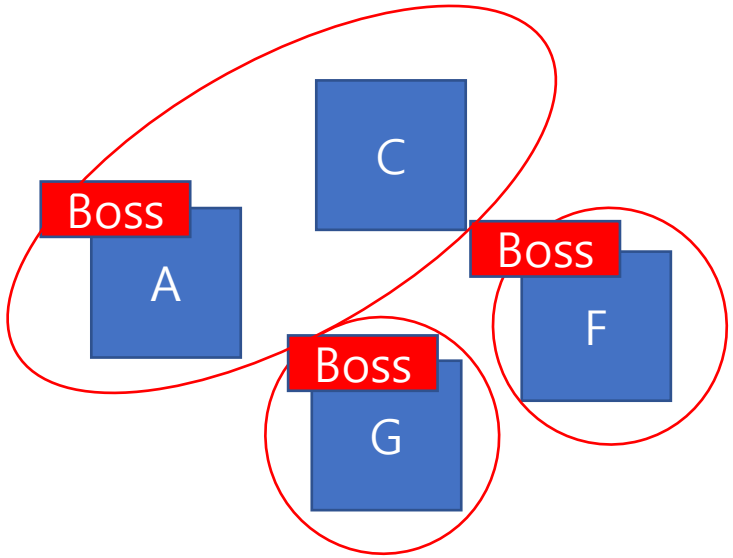
4개의 트리 존재

첫 번째 Union

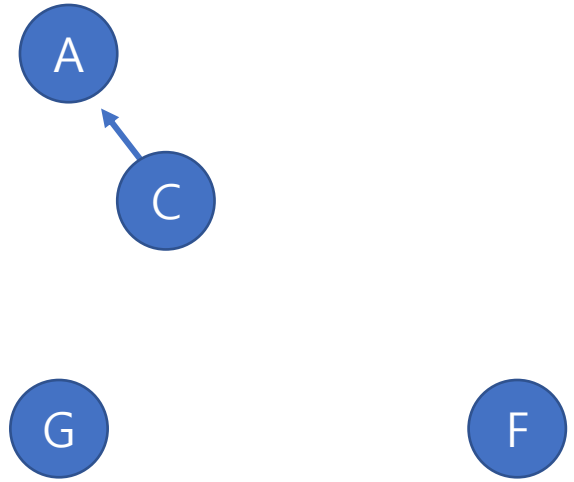
union(A, C)

- A의 최종 Boss : A
- C의 최종 Boss : C

→ C는 A 밑으로 들어간다.



그룹핑된
정보를 Tree로 저장



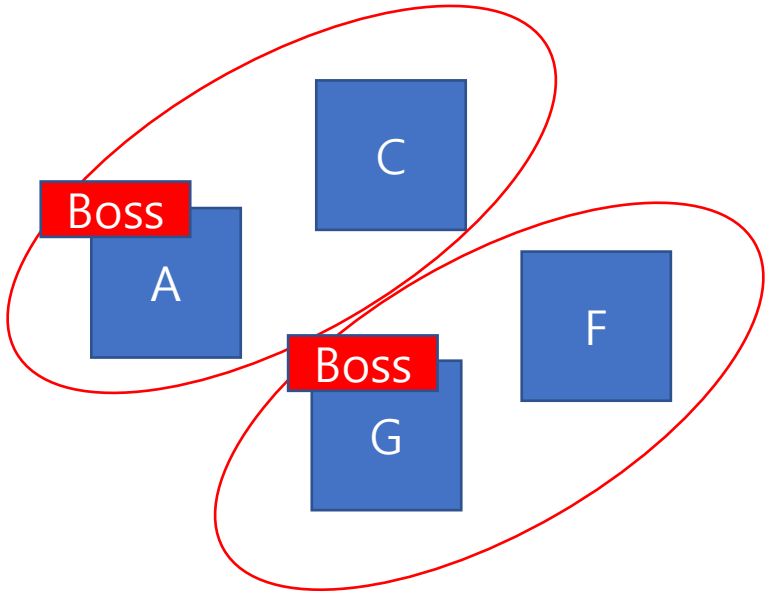
3개의 트리 존재

두 번째 Union

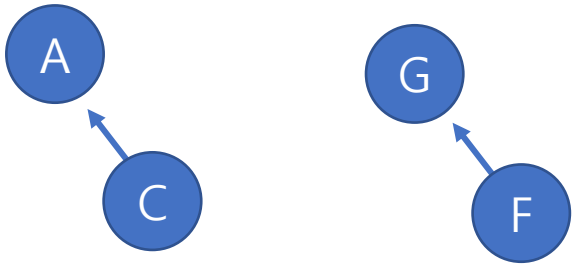
union(G, F)

- G의 최종 Boss : G
- F의 최종 Boss : F

→ F는 G 밑으로 들어간다.



그룹핑된
정보를 Tree로 저장



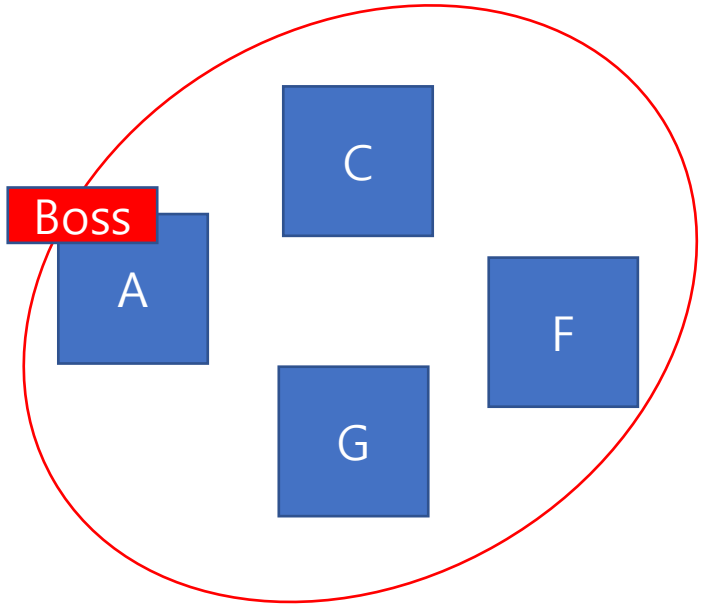
2개의 트리 존재

세 번째 Union

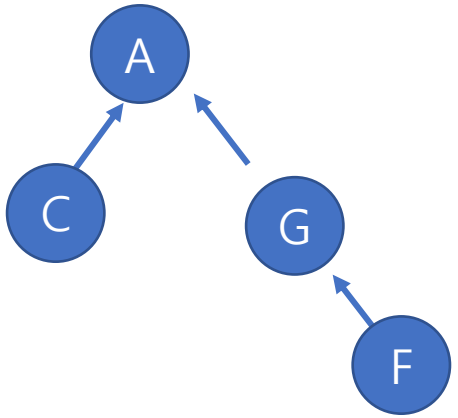
union(C, F)

- C의 최종 Boss : A
- F의 최종 Boss : G

→ G는 A 밑으로 들어간다.



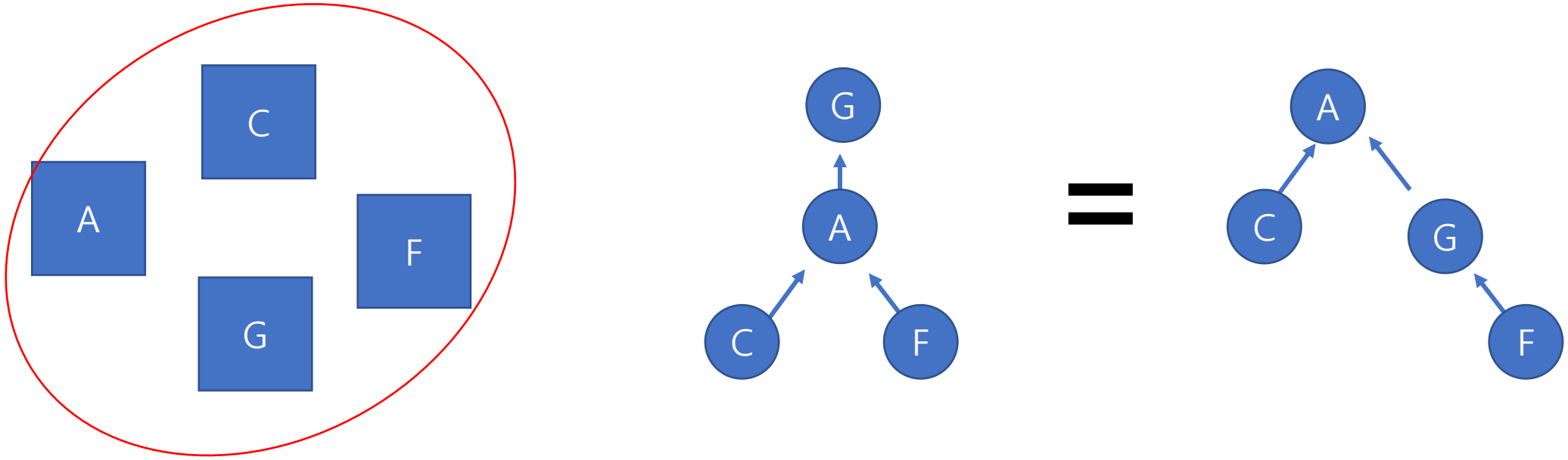
그룹핑된
정보를 Tree로 저장



1개의 트리 존재

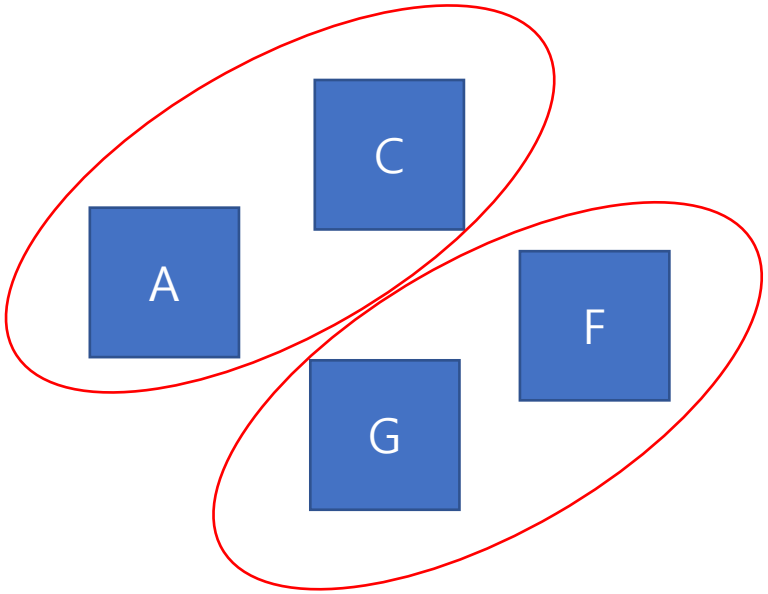
특징 1

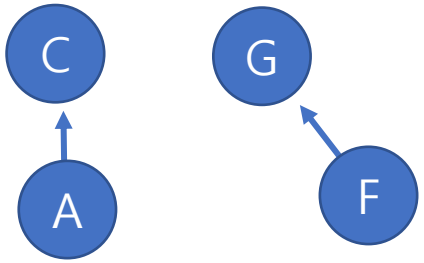
트리 모양이 달라도,
그룹핑된 정보는 같을 수 있다.



특징 2

트리의 개수가
곧, 그룹의 개수이다.



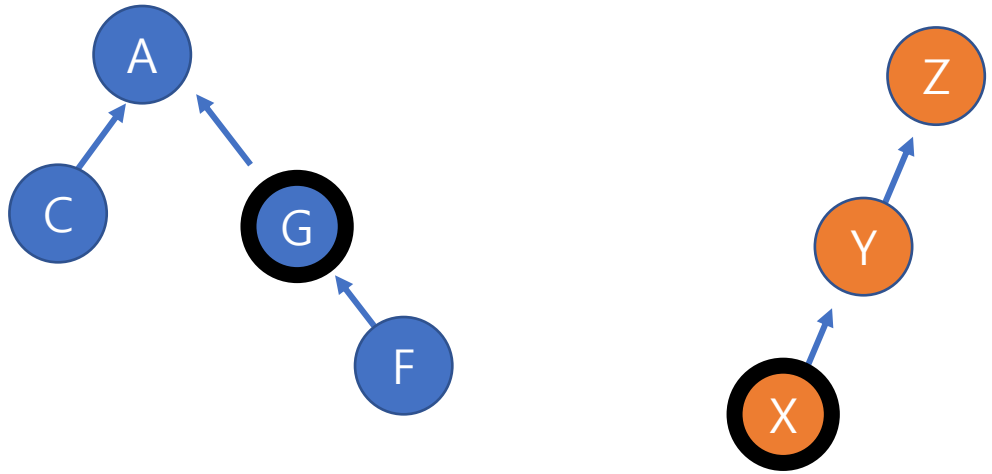


트리가 2개 이므로,
그룹의 개수도 2개이다.

[도전] 두 Tree Union

다음 두 그룹을 다음과 같이 Union 하면,
만들어지는 그림은?

- **union(G, X)**

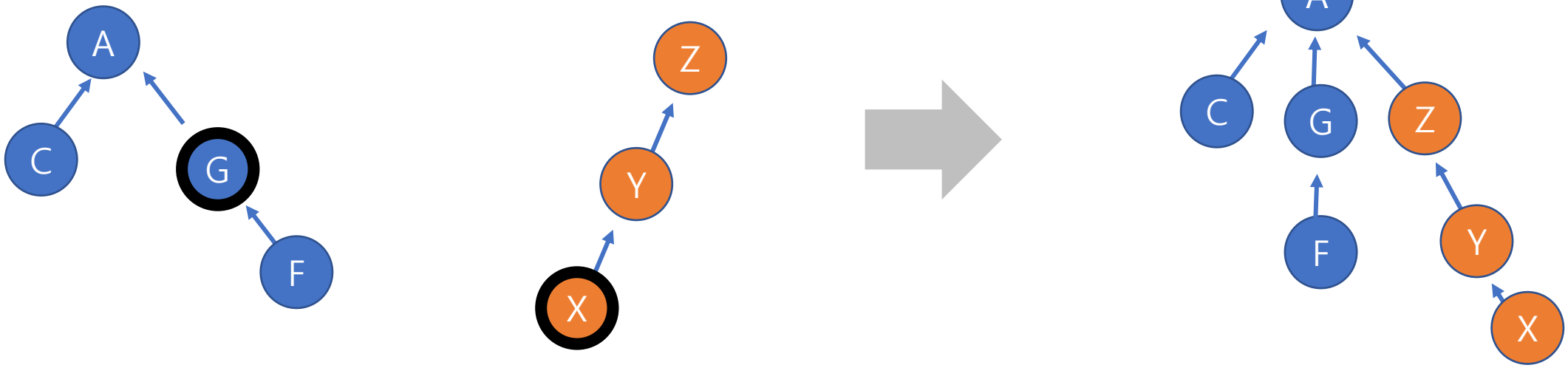


두 Tree Union 결과

union(G, X)

- G의 최종 Boss = find(G) = A
- X의 최종 Boss = find(X) = Z

→ Z가 A 밑으로 들어간다.

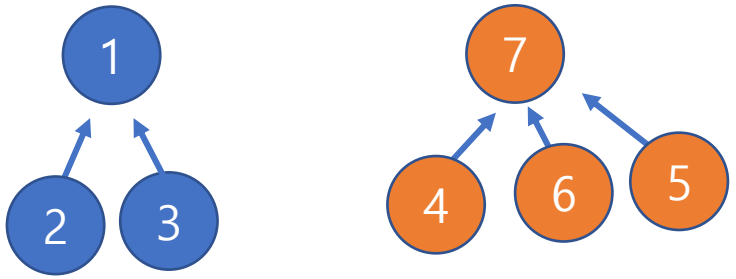


[도전] 만들어지는 Tree 그리기

union(1, 2)
union(2, 3)
union(7, 6)
union(6, 5)
union(5, 4)
union(10, 11)
union(13, 10)
union(13, 11)
union(4, 11)

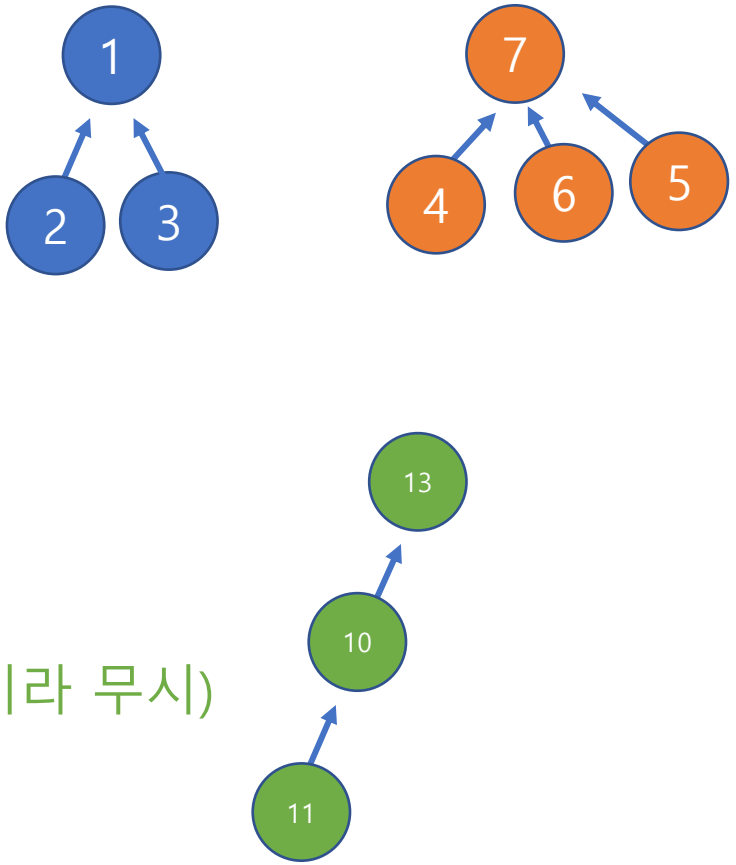
결과, 만들어지는 Tree - 1

union(1, 2)
union(1, 3)
union(7, 6)
union(6, 5)
union(5, 4)
union(10, 11)
union(13, 10)
union(13, 11)
union(4, 11)



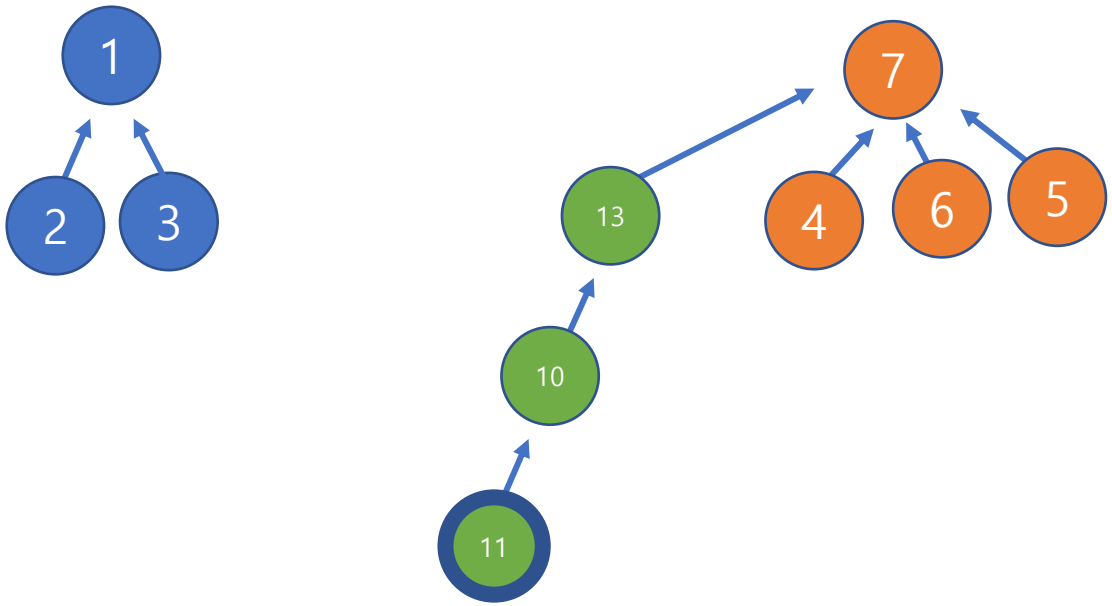
결과, 만들어지는 Tree - 2

union(1, 2)
union(1, 3)
union(7, 6)
union(6, 5)
union(5, 4)
union(10, 11)
union(13, 10)
union(13, 11) (이미 같은 그룹이라 무시)
union(4, 11)



결과, 만들어지는 Tree - 3

union(1, 2)
union(1, 3)
union(7, 6)
union(6, 5)
union(5, 4)
union(10, 11)
union(13, 10)
union(13, 11)
union(4, 11)



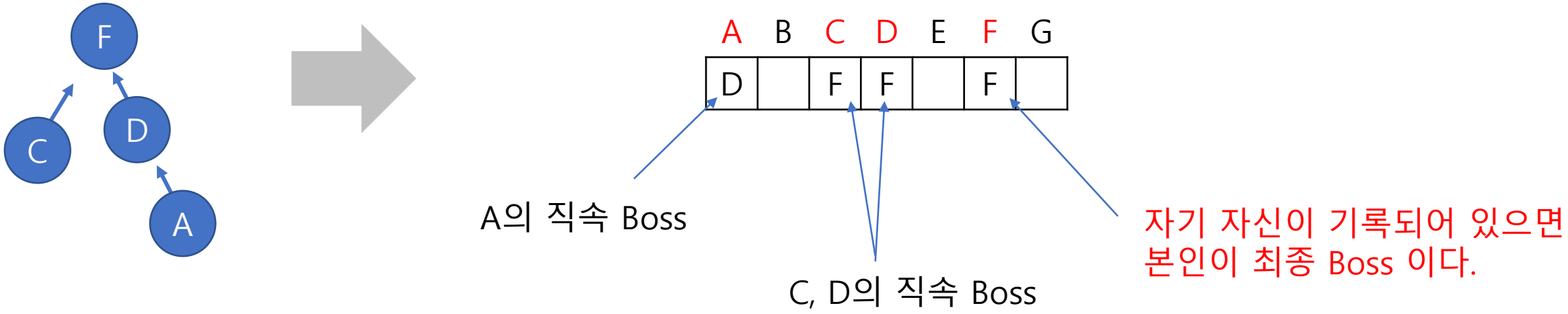
Tree 구현 이해

Union과 Find 함수 구현 방법 소개

1차원 배열로 Tree를 나타낸다.

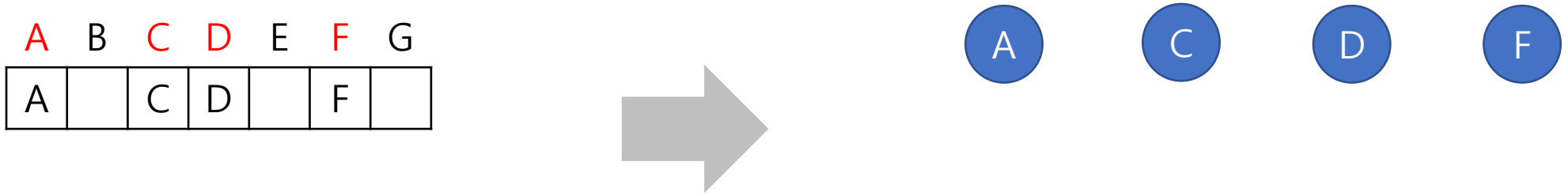
Tree 정보를 1차원 배열로 나타낸다.

- 각 노드는 본인들의 위한 칸이 한 칸씩 존재한다.
- 본인들의 직속 Boss가 누군지 기록한다.



초기 상태

초기 상태로, 각 노드는
자기 자신이 초기 Boss이다.

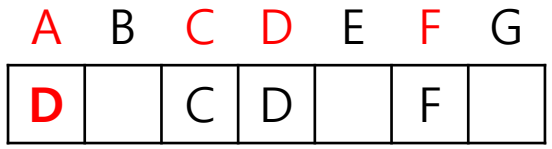


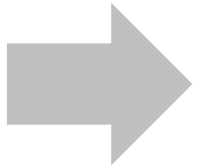
첫 번째 Union

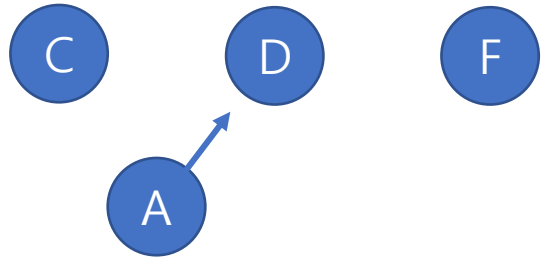
union(D, A)

- find(D) = D
- find(A) = A

→ A는 D 밑으로 들어간다.







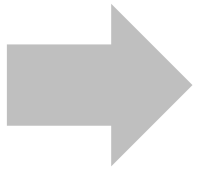
두 번째 Union

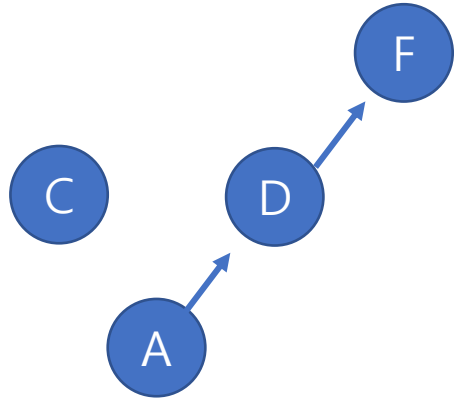
union(F, A)

- find(F) = F
- find(A) = D

→ D는 F 밑으로 들어간다.

A	B	C	D	E	F	G
D		C	**F**		F	

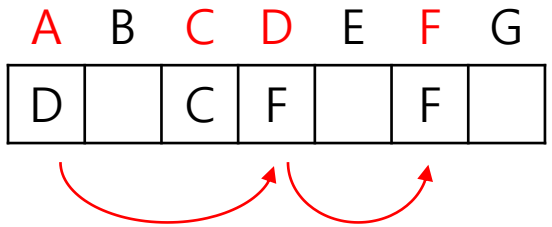




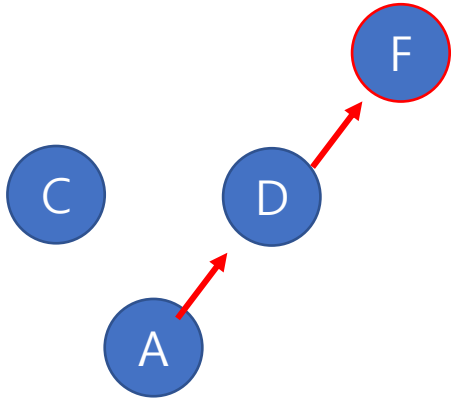
find(A) 값은?

find(A)

- A의 최종 Boss를 알아내는 방법
- 재귀호출로 구현한다.



1. A의 직속 Boss는 D 이고,
2. D의 직속 Boss는 F이고,
3. F의 직속 Boss는 F이므로, F가 최종 Boss이다.



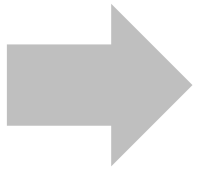
세 번째 Union

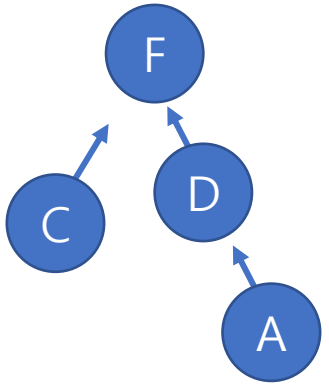
union(A, C)

- find(A) = F
- find(C) = C

→ A는 D 밑으로 들어간다.

A	B	C	D	E	F	G
D		**F**	F		F	

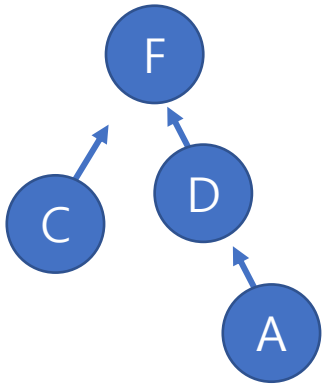




구현 – Main 함수

UnionFind 기본 구현 코드

```java
class UnionFind {
    char find(char a) {
        // 구현 필요
    }

    void union(char a, char b) {
        // 구현 필요
    }
}
```




```java
public class Main {
    void solution() {
        UnionFind uf = new UnionFind();

        uf.union( a: 'D', b: 'A');
        uf.union( a: 'F', b: 'A');
        uf.union( a: 'A', b: 'C');

        if (uf.find( a: 'C') == uf.find( a: 'A')) {
            System.out.println("같은 그룹");
        }
        else {
            System.out.println("다른 그룹");
        }
    }

    public static void main(String[] args) {
        new Main().solution();
    }
}
```

구현 – UnionFind class

UnionFind 기본 구현 코드

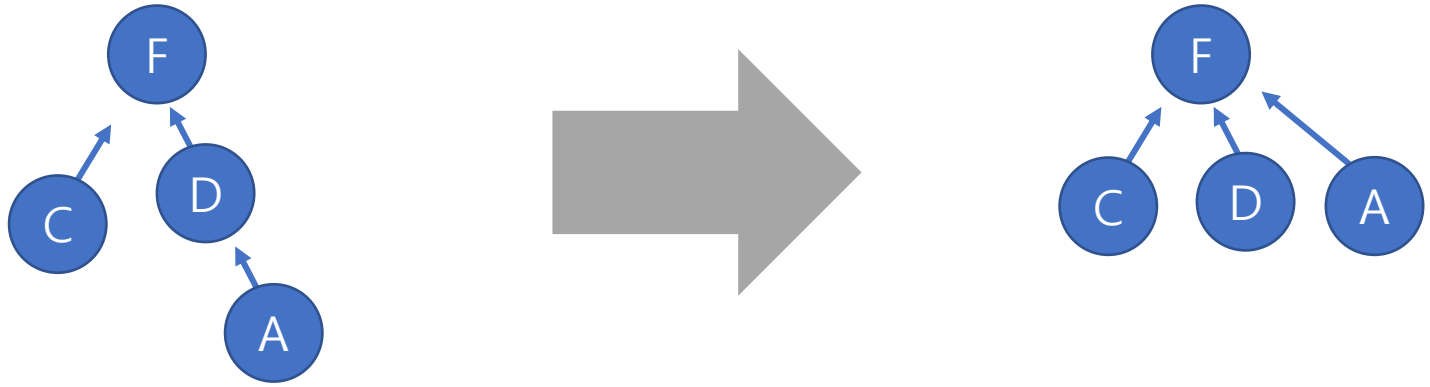
```java
public class Main {
    void solution() {
        UnionFind uf = new UnionFind();

        uf.union( a: 'D',  b: 'A');
        uf.union( a: 'F',  b: 'A');
        uf.union( a: 'A',  b: 'C');

        if (uf.find( a: 'C') == uf.find( a: 'A')) {
            System.out.println("같은 그룹");
        }
        else {
            System.out.println("다른 그룹");
        }
    }

    public static void main(String[] args) {
        new Main().solution();
    }
}
```

```java
class UnionFind {
    char[] arr = new char[200];

    public UnionFind() {
        //자기 자신을 가르키도록 초기 세팅
        for (int i = 0; i < 200; i++) {
            arr[i] = (char)i;
        }
    }

    char find(char a) {
        if (arr[a] == a) return a;

        char boss = find(arr[a]);
        return boss;
    }

    void union(char a, char b) {
        char t1 = find(a);
        char t2 = find(b);

        //같은 보스라면, 같은 그룹이므로 그룹을 지을 필요가 없다.
        if (t1 == t2) return;
        arr[t2] = t1; //t2는 t1 밑으로 들어간다.
    }
}
```

경로 압축

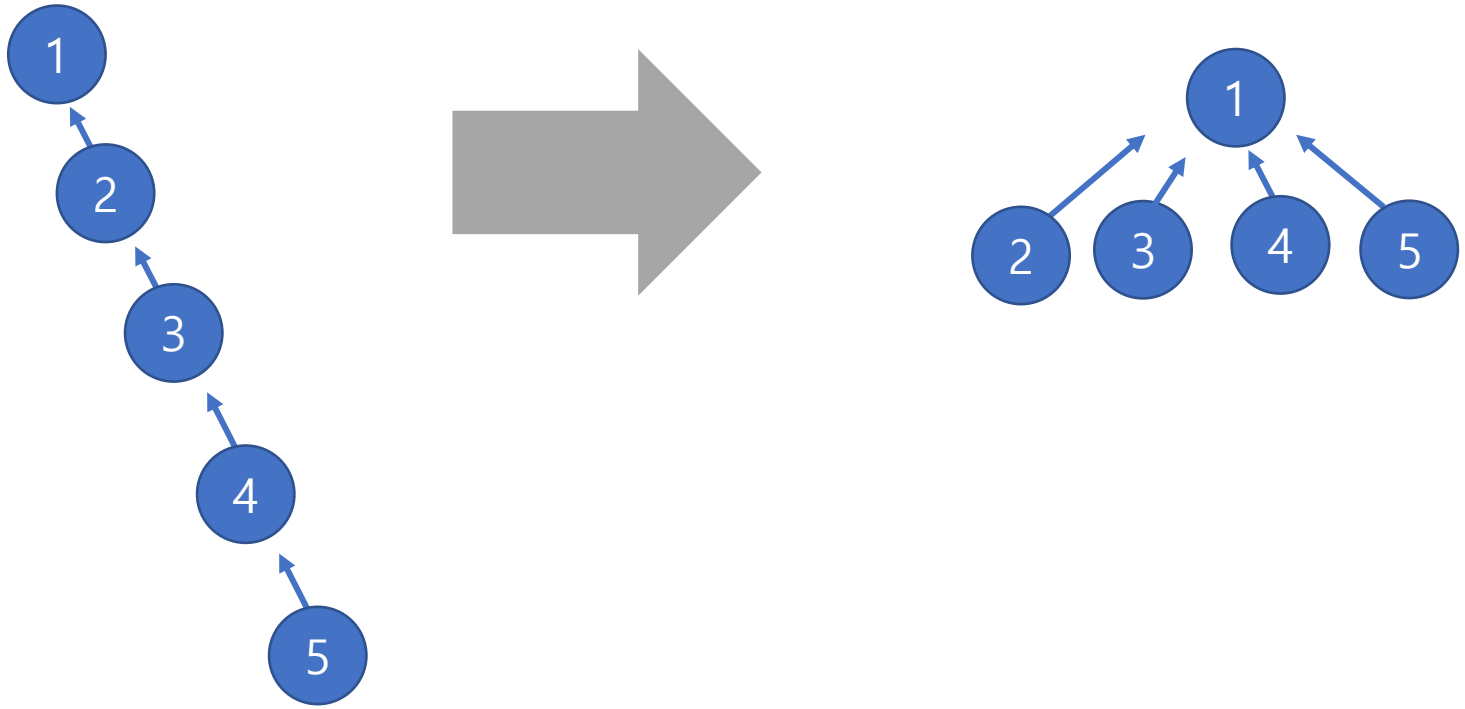
왼쪽과 오른쪽은 같은 그룹정보를 나타낸다.

- 성능은 다르다. find('A') 의 성능이 오른쪽이 더 빠르다.



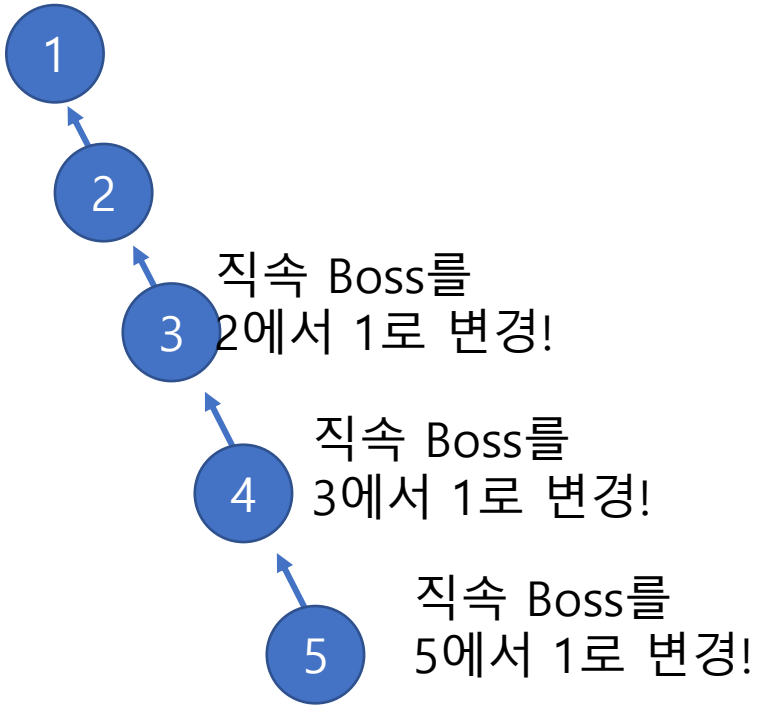
경로 압축

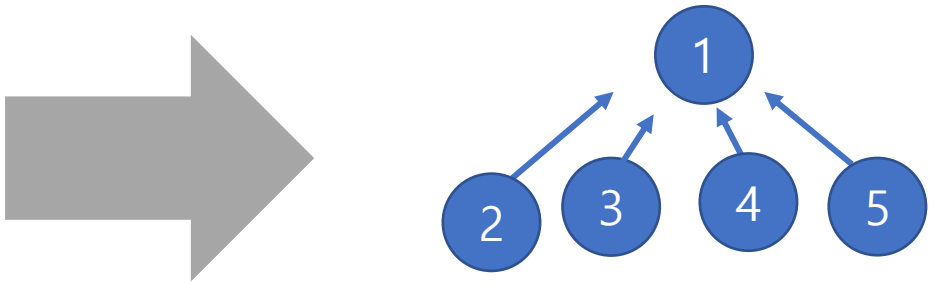
왼쪽 Tree를 오른쪽 Tree로 바꿔주면,
더 빠른 성능을 내는 Union-Find로 만들 수 있다.



경로 압축

해결방법 :
최종 Boss를 알아내면,
직속 Boss를 최종 Boss로 바꿔주면 된다.





경로 압축 구현

경로 압축 구현

```
char find(char a) {
    if (arr[a] == a) return a;

    char boss = find(arr[a]);
    return boss;
}
```

기존 코드

경로 압축
코드 추가

```
char find(char a) {
    if (arr[a] == a) return a;

    char boss = find(arr[a]);
    arr[a] = boss;
    return boss;
}
```

코드 정리

```
char find(char a) {
    if (arr[a] == a) return a;
    return arr[a] = find(arr[a]);
}
```

최종 소스코드

Union Find + 경로압축

```java
class UnionFind {
    char[] arr = new char[200];

    public UnionFind() {
        //자기 자신을 가르키도록 초기 세팅
        for (int i = 0; i < 200; i++) {
            arr[i] = (char)i;
        }
    }

    char find(char a) {
        if (arr[a] == a) return a;
        return arr[a] = find(arr[a]);
    }

    void union(char a, char b) {
        char t1 = find(a);
        char t2 = find(b);

        //같은 보스라면, 같은 그룹이므로 그룹을 지을 필요가 없다.
        if (t1 == t2) return;
        arr[t2] = t1; //t2는 t1 밑으로 들어간다.
    }
}
```

```java
public class Main {
    void solution() {
        UnionFind uf = new UnionFind();

        uf.union( a: 'D',  b: 'A');
        uf.union( a: 'F',  b: 'A');
        uf.union( a: 'A',  b: 'C');

        if (uf.find( a: 'C') == uf.find( a: 'A')) {
            System.out.println("같은 그룹");
        }
        else {
            System.out.println("다른 그룹");
        }
    }

    public static void main(String[] args) {
        new Main().solution();
    }
}
```

Union-Find 응용

UnionFind 기본 응용 문제 소개

그룹 이름 관리

조직의 이름을 관리한다.

- F 는 조직이름이 없었으나,
  union 이후, "도끼파" 소속이다.

```
void solution() {
    UnionFind uf = new UnionFind();

    uf.setGroupName( a: 'A',  name: "도끼파");
    uf.setGroupName( a: 'B',  name: "대파");
    uf.setGroupName( a: 'C',  name: "쪽파");
    uf.setGroupName( a: 'D',  name: "실파");

    uf.union( a: 'A',  b: 'B');
    uf.union( a: 'C',  b: 'D');
    uf.union( a: 'E',  b: 'F');
    uf.union( a: 'B',  b: 'F');

    System.out.println(uf.names[uf.find( a: 'F')]);
}
```

도끼파

```
class UnionFind {
    char[] arr = new char[200];
    String[] names = new String[200];

    public UnionFind() {
        //자기 자신을 가르키도록 초기 세팅
        for (int i = 0; i < 200; i++) {
            arr[i] = (char)i;
        }
    }

    char find(char a) {
        if (arr[a] == a) return a;
        return arr[a] = find(arr[a]);
    }

    void union(char a, char b) {
        char t1 = find(a);
        char t2 = find(b);

        //같은 보스라면, 같은 그룹이므로 그룹을 지을 필요가 없다.
        if (t1 == t2) return;
        arr[t2] = t1; //t2는 t1 밑으로 들어간다.
    }

    void setGroupName(char a, String name) {
        int tar = find(a);
        names[tar] = name;
    }
}
```

그룹의 개수 관리

그룹의 개수 관리 방법

- dat 배열 (isNew)
- groupCnt 변수

```
1
2
3
2
```

```java
void solution() {
    UnionFind uf = new UnionFind();

    uf.union( a: 'A',  b: 'B');
    System.out.println(uf.groupCnt);
    uf.union( a: 'C',  b: 'D');
    System.out.println(uf.groupCnt);
    uf.union( a: 'E',  b: 'F');
    System.out.println(uf.groupCnt);
    uf.union( a: 'B',  b: 'F');
    System.out.println(uf.groupCnt);
}
```

```java
class UnionFind {
    char[] arr = new char[200];
    int[] isNew = new int[200];
    int groupCnt = 0;

    public UnionFind() {
        //자기 자신을 가르키도록 초기 세팅
        for (int i = 0; i < 200; i++) {
            arr[i] = (char)i;
        }
    }

    char find(char a) {
        if (arr[a] == a) return a;
        return arr[a] = find(arr[a]);
    }

    void union(char a, char b) {
        char t1 = find(a);
        char t2 = find(b);
        if (isNew[t1] == 0) groupCnt += ++isNew[t1];
        if (isNew[t2] == 0) groupCnt += ++isNew[t2];

        //같은 보스라면, 같은 그룹이므로 그룹을 지을 필요가 없다.
        if (t1 == t2) return;
        arr[t2] = t1; //t2는 t1 밑으로 들어간다.
        groupCnt -= 1; //그룹이 하나 줄었다.
    }
}
```

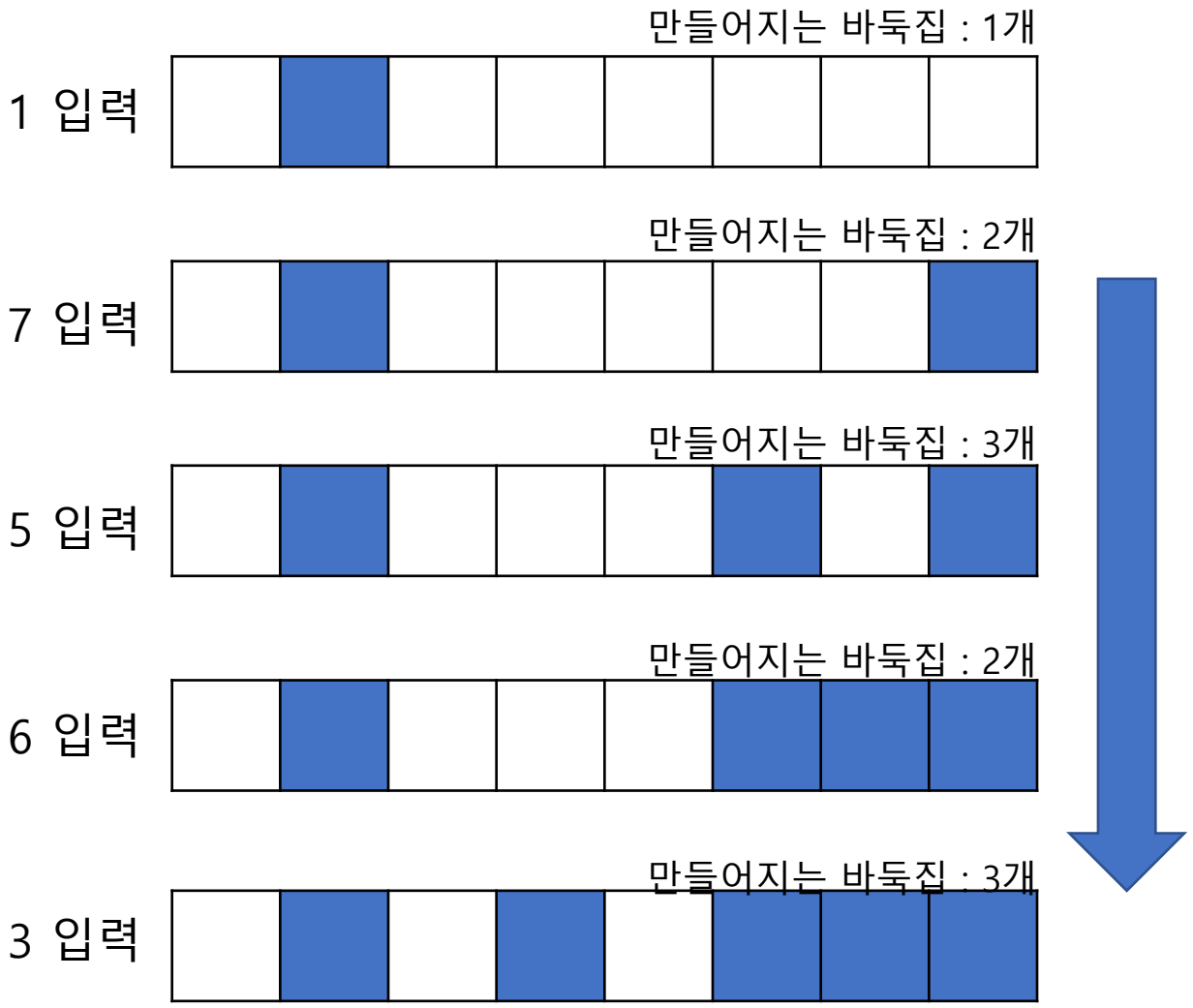
[도전] 바둑집

가로 바둑이 존재한다.
특정 index에 돌을 하나씩 둔다.

돌이 좌우로 붙어 있는 한 그룹을
“바둑 집”이라고 한다.

돌을 순차적으로 둘 때 마다
바둑집의 개수를 빠르게 구하여라

- 정답 : 1 2 3 2 3



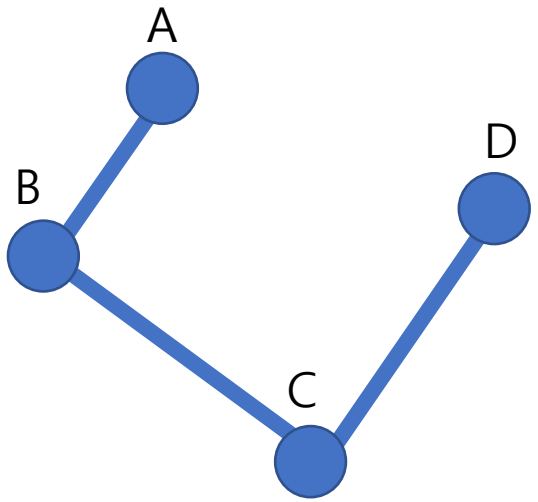
Cycle 판별법

간선이 연결된 정보가 주어진다.

Cycle 유무를 출력 해야한다.

- **예시 1**

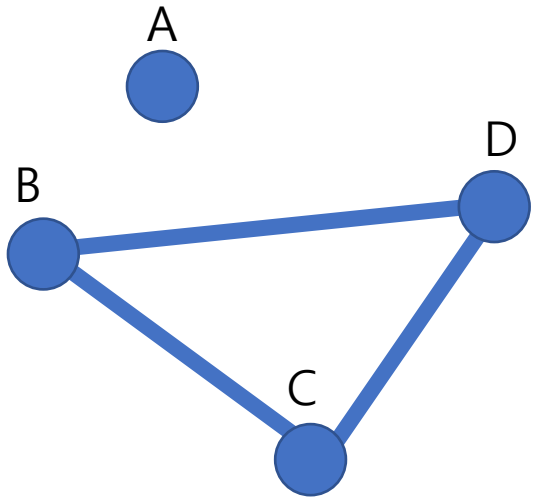
  A B
  B C
  C D
  → Cycle 없음



Cycle 없음

- **예시 2**

  B D
  C D
  B C
  → Cycle 존재



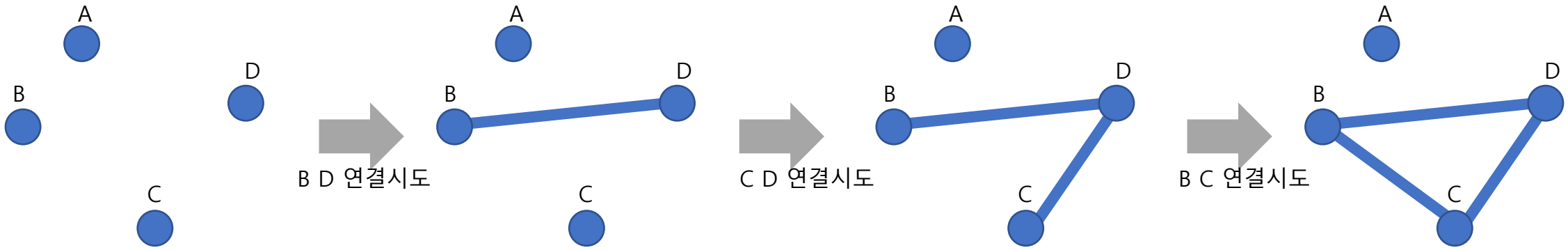
Cycle 존재

해결 방법

입력 받을 때 마다 Union 을 수행할 것이다.
Union 전, 이미 같은 그룹인지 검사한다.
이미 같은 그룹이었다면 → Cycle이 존재한 것이다.

[입력]

B D
C D
B C



초기 상태

B, D는 서로 다른 그룹임
union(B, D) 수행

C, D는 서로 다른 그룹임
union(C, D) 수행

B, C는
이미 같은 그룹이다.
→ Cycle 발생!

Cycle 판별 소스코드

Union 하기 전,
같은 그룹인지 검사하여

이미 같은 그룹이라면
Cycle 존재

```
UnionFind uf = new UnionFind();
char[][] input= {
        {'B', 'D'},
        {'C', 'D'},
        {'B', 'C'}
};

void solution() {
    boolean ret = isCycle();
    if (ret) System.out.println("Cycle 존재");
    else System.out.println("Cycle 없음");
}

private boolean isCycle() {
    for (int i = 0; i < input.length; i++) {
        if (uf.find( a: 'B') == uf.find( a: 'D')) {
            return true;
        }
        else {
            uf.union( a: 'B',  b: 'D');
        }
    }
    return false;
}
```

```
Cycle 존재
```