유니온 파인드

서로 소 집합의 자료구조 (disjoint-set)란?

컴퓨터 사이언스에서 **disjoint-set(union-find data structure, merge-find set)** 이란 disjoint-set을 저장하는 자료구조를 의미한다.

그래프로 표현이 가능한 경우, 두 정점이 *연결되었는지* 확인하며 간선이 계속 추가되는 *동적상황* 에서 두 집합의 원소가 같은 집합에 속해있는 지를 알려주는 자료구조이다.

Disjoint-set 자료구조는 **Minimum-spanning tree** 를 찾는 **Kruskal Algorithm**을 구현하는데, 중요한 역활을 한다.

Disjoint-set의 특징

- 서로 소인 동적 집합들의 모임인 S = {s1, s2, ….sk}를 유지 관리한다.
- 각각의 집합은 어떤 원소인 **대표값**에 의해 식별된다.
  - 어떤 값을 **대표값**으로 설정한 것인지는 상관이 없다.
- 두 집합에는 어떠한 교집합의 원소가 없고, 모든 집합의 합집합은 전체집합이다.
- 무방향 그래프에서 연결 요소를 정하는 경우 많이 사용된다.

Union-Find의 연산

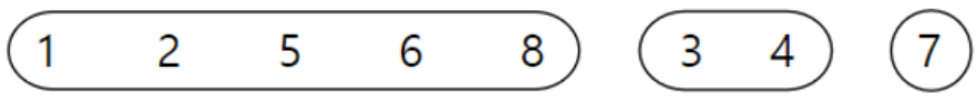
- **Find(x)** : x를 원소로 가지고 있는 유일한 집합의 대표값을 리턴한다.
- **Union(x,y)** : x와 y를 가지고 있는 집합 S(x)와 집합 S(y)를 합해서 새로운 집합을 생성한다.

Union-Find의 집합 표현 방식

위의 case를 보면 아직까지 아무 연산도 사용되지 않은 자료구조 입니다.

즉 집합으로 표현을 하자면

{1}, {2}, {3}, {4}, {5}, {6}, {7}, {8}이란 소리입니다.



위의 case는 몇 번의 연산을 한 후의 집합의 상태입니다.

집합으로 표현하자면

{1, 2, 5, 6, 8}, {3, 4}, {7}의 상태입니다.

즉 집합 {1, 2, 5, 6, 8}과 {3, 4}, {7}은 서로소 상태입니다.

서로소 집합은 어떻게 코드로 구현하나요??

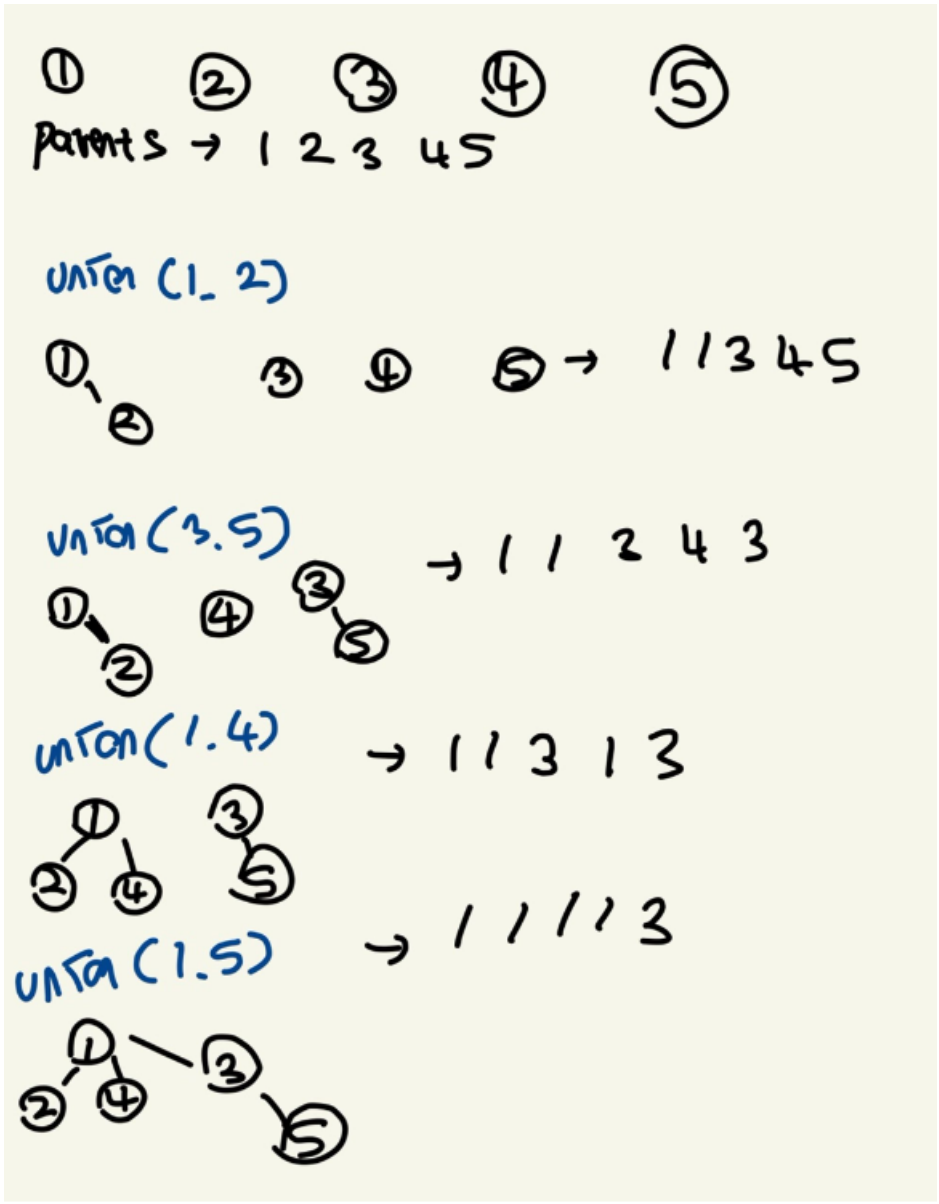
먼저 N개의 노드가 있다고 가정하겠습니다.

N개의 노드는 자기 자신의 직계 부모만 pointing을 하고 있습니다. 즉, 초기 상태는 자기 자신을 값으로 가지고 있는 parents배열이 있겠죠

parents	**1**	**2**	**3**	**4**	**5**
value	1	2	3	4	5

```
    Union_Find(const int& n_input) : N(n_input), parents(N+1) {
        for(int i = 0; i <= N; ++i)
            parents[i] = i;
    }
```

그 후의 과정을 거쳐서 Union-Find를 몇 번 실행해보도록 하겠습니다.



이렇게 일련의 과정을 거쳐서 Union-Find함수를 구현해봤습니다.

(대표값 설정 방법 : 루트가 작은 것을 기준으로 붙임)

Code C++

```cpp
#include <iostream>
#include <vector>
using namespace std;

class Union_Find
{
public:
    int N;
    vector<int> parents;

    Union_Find() = default;
    Union_Find(const int& n_input) : N(n_input), parents(N+1) {
        for(int i = 1; i <= N; ++i)
            parents[i] = i;
    }


    void show_parents() const {
        for(auto& ele: parents)
            cout << ele << " ";
    }
    int Find(int a)
    {
        if (parents[a] == a) return a;
        return Find(parents[a]);
    }
    void Naive_Union(int a, int b) // 2개의 집합을 연결하는 함수
    {

        a = Find(a); b = Find(b);
        // 기준 : 더 작은 루트에다가 연결한다.
        if (a < b)
            parents[b] = a;
        else
            parents[a] = b;
    }
    void SameParents(int a, int b) // 2개의 원소가 같은 집합 안에 있는지를 return하는 값
    {
        if (Find(a) == Find(b))
            cout << "The values are same group";
```

```
        else
            cout << "The values are not same group";
    }
};

int main(void)
{
    Union_Find U(5);
    cout << "The Initial parents are : "; U.show_parents(); cout << endl; // 초기값
    U.Naive_Union(1,2);
    U.Naive_Union(3,5);
    U.Naive_Union(1,4);
    U.Naive_Union(1,5);
    cout << "The Final parents are : "; U.show_parents(); cout << endl; // 최종 값
    U.SameParents(1,5); cout << endl;
    U.SameParents(1,3); cout << endl;
    return (0);
}

결과
The Initial parents are : 0 1 2 3 4 5
The Final parents are : 0 1 1 1 1 3
The values are same group
The values are same group
```

Union-Find의 과정

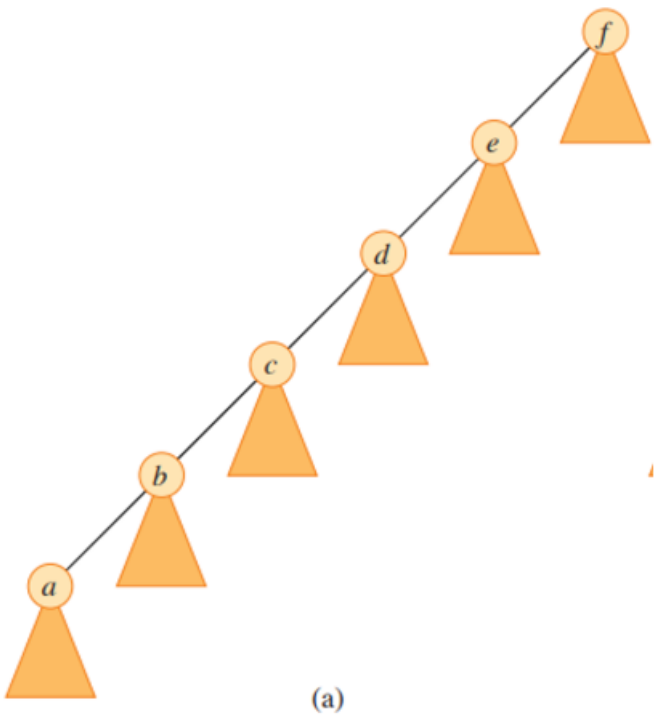
1. Node의 직계 부모를 나타내는 parents 배열을 구현한다 (자기 자신으로 초기화)
2. Node를 parameter로 받아서 해당 집합의 대표값을 리턴하는 Find함수를 구현한다.
3. 2개의 Node를 parameter로 받아서 2개의 집합을 합치는 Union함수를 구현한다.
    - 어떤 집합을 대표값으로 설정할 지를 기준을 정해서 Union연산을 실행한다.

Naive Union-Find의 문제점과 시간복잡도

문제점 1) Find 연산의 시간복잡도

그렇다면 Naive Union-Find의 시간복잡도는 어떻게 될까요?

아래의 그림을 참고해 보도록 하겠습니다



Union-Find의 자료구조에서는 각각의 집합은 **Forest(Tree의 집합)** 을 이루고 있습니다.

위의 상황에서 Find(a)를 진행한다고 가정을 해 본다면 5번의 재귀를 통해서 a의 대표값을 return합니다.

즉 **Find(a)** 연산을 한다고 가정하면, 최악의 시간복잡도는 **O(N)이 되겠죠!!**

PS) Find연산을 M번 실행하게 된다면 최악의 시간복잡도는 **O(MN)**이 된다.

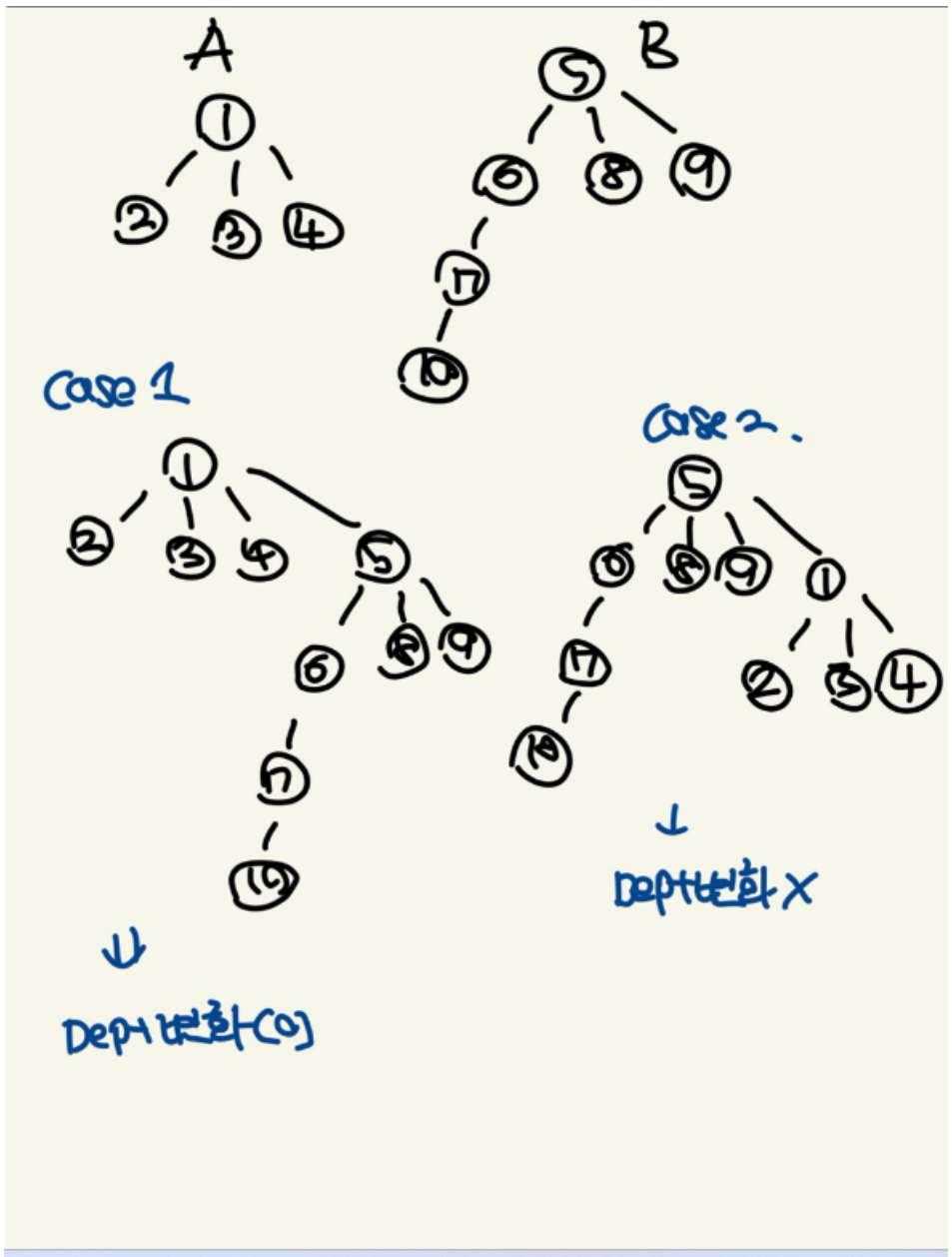
PS) 재귀 연산 자체도 컴퓨터 연산에 많은 부담을 줍니다!!

즉 우리는 Tree의 depth를 낮게 하면서 최적화를 진행해야 합니다.

2번 문제점) Union연산의 대표값 설정

위에서 살펴보았 듯, 우리는 union을 진행을 할 때, 2개의 집합을 연결할 때, 대표값이 더 작은 값에다가 tree를 합쳤습니다.

자 그렇다면 아래의 상황을 한 번 보도록 하겠습니다.



앞서 살펴보았 듯이,

Union-Find의 시간복잡도를 결정하는 dominant한 연산은 Find(a) 연산이고, 이 Find연산은 Tree의 깊이와 관련이 있다

를 알아보았습니다.

그렇다면 위의 A와 B Tree를 단순히 대표값(root)이 작은 tree에 연결을 한다면 합쳐진 tree의 depth가 증가하는 현상이 발생합니다. 반대로 B tree에다가 A tree를 연결하게 되면 tree의 depth의 변화가 없습니다.

위를 통해 어떤 Tree에다가 Tree를 연결하는 것을 설정하는 것이 Union-Find연산을 최적화에 도움을 줄 수 있다는 것을 알 수 있습니다.

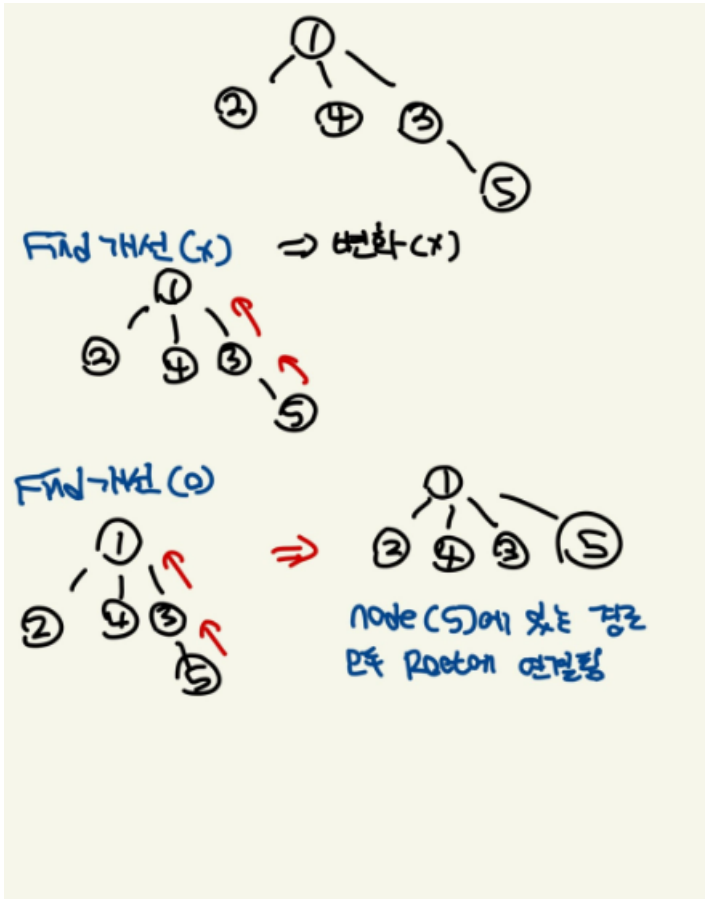
정리

1. union-find는 Find와 Union연산이 있다.
2. 시간복잡도를 지배하는 연산은 Find연산이다
3. Find 연산은 트리의 깊이와 연관이 있다 (최악의 시간복잡도 N)
4. 트리의 깊이를 최대한 낮게 하기 위해서는 Find와 Union연산의 최적화의 필요성이 대두된다.

Find연산의 최적화 (경로 압축)

경로 압축의 핵심 아이디어는 **특정 노드가 속합 집합의 루트를 찾을 때, 경로 상의 모든 노드를 직접 루트에 연결하여 트리의 높이를 줄이는 것입니다.**

Example Case



이전의 Find함수

```
int Find(int a)
{
    if (parents[a] == a) return a;
    return Find(parents[a]);
}
```

위의 그림에서 Find(5) 연산을 진행한다고 가정을 해보겠습니다.

이전의 Find(5)를 실행하게 된다면 루트 1까지 경로상에 있는 {3, 5} 노드에 대한 변화는 일어나지 않습니다. 그렇다면 다음에 또 Find(5)를 실행하게 된다면 경로 상의 {3,5}를 재귀를 통해 탐색을 진행하게 됩니다.

Find최적화의 핵심은 경로상의 경로상의 노드 {3,5}를 대표값 1 바로 밑에 붙이겠다 입니다. 이렇게 된다면 다음에 Find(5)를 실행하게 된다면 {5}만 거치게 되겠죠

수정된 코드는 이렇게 됩니다

수정된 Find함수

```
int Find(int a)
{
    if (parents[a] == a) return a;
    return parents[a] = Find(parents[a]);
}
```

Full Code

```
#include <iostream>
#include <vector>
using namespace std;

class Union_Find
{
public:
    int N;
    vector<int> parents;

    Union_Find() = default;
    Union_Find(const int& n_input) : N(n_input), parents(N+1) {
        for(int i = 1; i <= N; ++i)
            parents[i] = i;
    }

    void show_parents() const
    {
        for(auto& ele: parents)
            cout << ele << " ";
    }
    int Find(int a)
    {
        if (parents[a] == a) return a;
        return parents[a] = Find(parents[a]);
```

```
    }
    void Naive_Union(int a, int b) // 2개의 집합을 연결하는 함수
    {

        a = Find(a); b = Find(b);
        // 기준 : 더 작은 루트에다가 연결한다.
        if (a < b)
            parents[b] = a;
        else
            parents[a] = b;
    }
    void SameParents(int a, int b) // 2개의 원소가 같은 집합 안에 있는지를 return하는 값
    {
        if (Find(a) == Find(b))
            cout << "The values are same group";
        else
            cout << "The values are not same group";
    }
};

int main(void)
{
    Union_Find U(5);
    cout << "The Initial parents are : "; U.show_parents(); cout << endl; // 초기값
    U.Naive_Union(1,2);
    U.Naive_Union(1,4);
    U.Naive_Union(1,3);
    U.Naive_Union(3,5);
    cout << "The Final parents are : "; U.show_parents(); cout << endl; // 최종 값
    U.SameParents(1,5); cout << endl;
    U.SameParents(1,3); cout << endl;
    return (0);
}
```

정답

PS) Find최적화만 사용하게 된다면 시간복잡도는

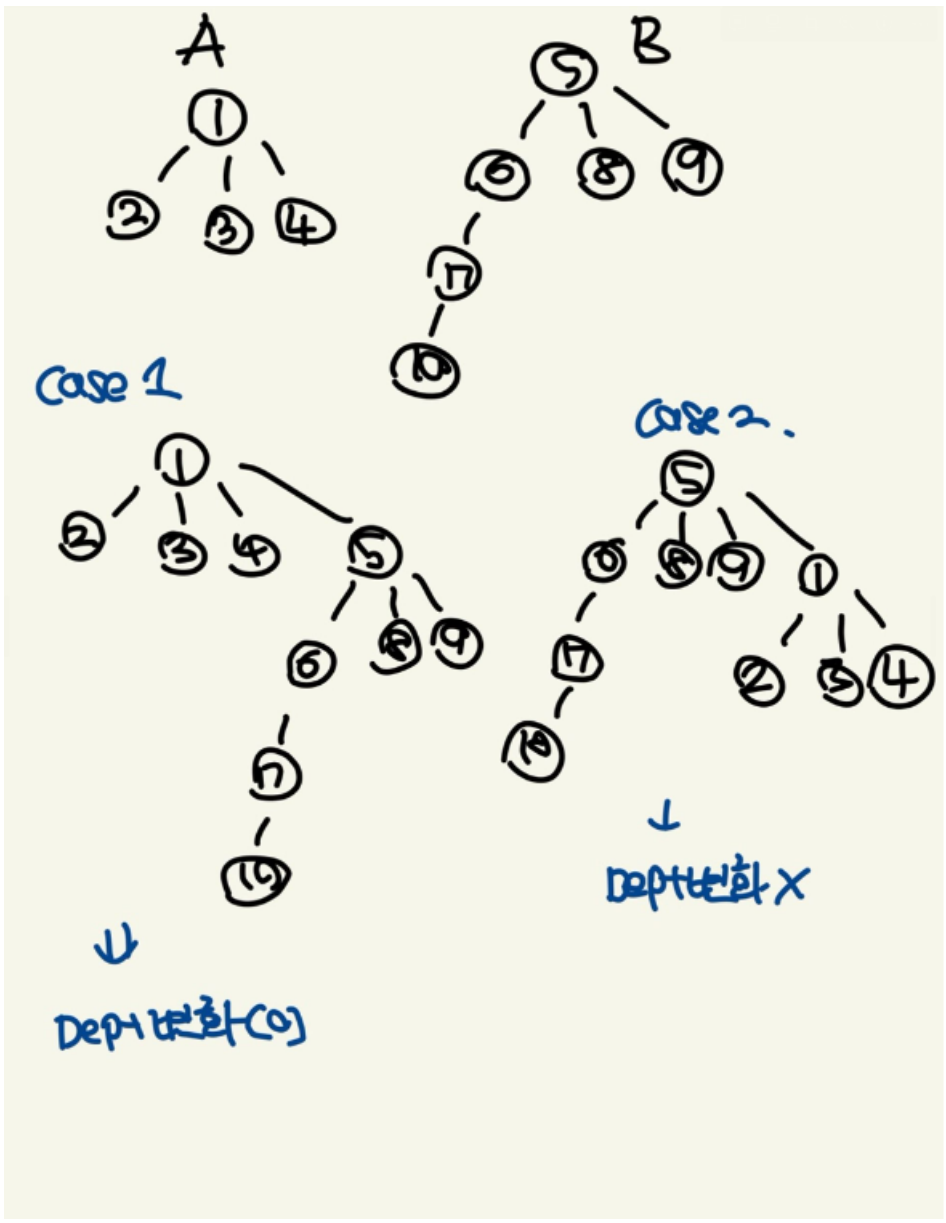
$\Theta(n + f \cdot (1 + \log_{2+f/n} n))$.

Union 연산의 최적화

Union-by size

자 이번에는 union을 최적화 하는 방법입니다. 트리의 weight를 먼저 정의해야하는대요, size란 tree에 몇개의 node가 달려있는 지를 알려주는 값입니다.

예를 들어서



위의 예시를 들자면 A의 size는 4가 되고, B의 size는 6이 됩니다. 즉 작은 크기(weight)의 tree를 더 높은 루트에 연결 시켜서 최종 tree의 깊이의 증가를 최소화 하는 방법입니다.

이를 위해서는 위에서 구현했던 코드에서 size를 나타내는 배열을 하나 더 초기화 해야합니다.

```
int N;
vector<int> parents;
vector<int> weight; // 노드의 weight를 나타내는 배열



    Union_Find(const int& n_input) : N(n_input), parents(N+1), weight(N+1, 1) {
    for(int i = 1; i <= N; ++i)
        parents[i] = i;
} -> weight를 1로 초기화를 진행한다.
```

```
void Weight_Union(int a, int b) // 2개의 집합을 연결하는 함수
{
    a = Find(a); b = Find(b);

    if (a == b) return; // 같은 부모

    if (weight[a] <= weight[b])
        swap(a,b);

    parents[b] = a;
    weight[a] += weight[b];
}
```

위의 코드를 보면, weight를 모두 1로 초기화를 진행합니다.

그리고 Weight_Union을 확인해보면 크게 로직은 3개 입니다.

1. 같은 부모인 경우에는 return을 한다.
2. 만약 b의 무게가 더 크다면 → a의 부모는 b가 된다, b의 weight에 a의 weight를 더한다.
3. 만약 a의 무게가 더 크다면 → b의 부모는 a가 된다, a의 weight에 b의 weight를 더한다.

union-by-rank

위의 union-by-rank의 문제점은 자식의 개수가 많다고 해서, tree의 높이가 더 높다라는 것을 보장할 수 없다는 것입니다.

이에 트리의 깊이를 배열로 초기화 하여 배열의 깊이가 더 높은 곳에 깊이나 낮은 tree를 연결하는 **union-by-rank** 방식이 있습니다.

코드 작성이 더 복잡하기는 하지만, 더 효과적인 방법이긴 합니다.

```
    int N;
    vector<int> parents;
    vector<int> rank; // Changed from weight to rank
```

```
    Union_Find(const int& n_input) : N(n_input), parents(N+1), rank(N+1, 0) {
        for(int i = 1; i <= N; ++i)
            parents[i] = i;
    }
```

```
    void Union(int a, int b) // 2개의 집합을 연결하는 함수
    {
        a = Find(a);
        b = Find(b);

        if (a == b) return; // 같은 부모

        // Union by rank
        if (rank[a] < rank[b])
            swap(a, b);

        // Make b a child of a
        parents[b] = a;

        // If ranks are the same, increment the rank of a
        if (rank[a] == rank[b])
            ++rank[a];
    }
```

참고

- Union-by-rank, union-by-size둘 중 아무것이나 사용해도 됨

시간복잡도

경로 압축을 사용하지 않고 Weight-Union(union-by rank, union-by-size)만을 사용한 union-find의 시간복잡도는 : **O(Mlg(N))**

최종 시간복잡도

지금까지 Union-Find 알고리즘에 대해 알아보았습니다.

이 알고리즘의 주요 연산은 **Union** 연산과 **Find** 연산이 있으며, 시간 복잡도에 가장 큰 영향을 미치는 연산은 **Find** 연산입니다.

Find 연산의 효율성은 트리의 깊이와 밀접한 관계가 있습니다. 따라서 트리의 깊이를 줄이는 것이 중요합니다. 이를 위해 **경로 압축**(Path Compression)과 **Weight-Union**(또는 union-by-rank, union-by-size) 방식을 사용합니다.

그렇다면 최종 시간복잡도는 어떻게 될까요?

최종 시간복잡도는 **O(M)**

즉, 상수 시간에 **근접하게** 떨어진다고 볼 수 있습니다.

왜 상수 시간이 아니고 **근접하게** 라는 말을 사용할까요?? 이는 *아크만 함수*와 관련이 있습니다. 원래는 아크만 함수의 역함수가 M과 곱해져 있습니다. 그러나 아크만 함수의 역함수는 매우매우매우 느리게 상승하는 함수이기 때문에 상수 시간복잡도라고 생각하시면 됩니다.

→ 이론적으로는 초선형(superlinear)

엄밀히 따진다면 M * F(a)의 시간복잡도를 가집니다. <F(a)를 아크만 함수의 역함수라고 가정>

참고 자료

1. Introducing to Algorithm
2. https://blog.naver.com/PostView.naver?blogId=kks227&logNo=220791837179&parentCategoryNo=&categoryNo=299&viewDate=&isShowPopularPosts=false&from=postList
3. https://ko.wikipedia.org/wiki/아커만_함수