Estructura de Datos para conjuntos disjuntos.

Queremos una estructura que principalmente nos deje realizar 3 operaciones las cuales son:

- Unir dos Conjuntos
- Encontrar en que conjunto esta un elemento
- Verificar si dos elementos están en el mismo conjunto.

Para esto ocuparemos la estructura Union-Find Disjoint Sets las cuales soporta estas operaciones y las hace eficiente por lo cual usaremos Union por rango y compresión de caminos, ahora veremos como haremos cada una de estas operaciones.

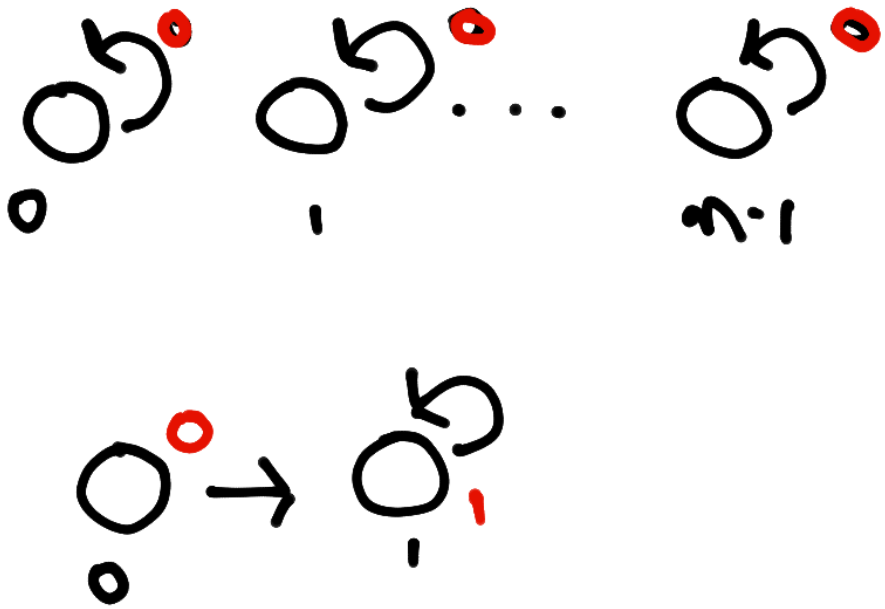
UFDS (Union-Find Disjoint Sets) inicializa primero `n` conjuntos con un solo elemento, él mismo, es decir inicialmente tenemos `n` conjuntos $\{0\}$, $\{1\}$, ... ,$\{i\}$ ... $\{n-1\}$ Si los conjuntos que ocupas son de elementos que no son números puedes mapearlos de tal forma sean números.

Ahora ya que tenemos una idea de como se ven nuestros conjuntos veamos como realizar esas tres operaciones que estan arriba.

Pero para esto necesitamos definir un par de cosas,

1. A cada conjunto se le identificará por un *representante*, inicialmente `representante[i] = i` a dicho representante lo llamaremos padre, pues tomará forma de arbol por lo que solo podrá haber un representante (la raiz), entonces `p[i] = i` al iniciar.
2. A cada conjunto tendrá definido su rank, el cual nos dirá la profundidad de ese conjunto, esto con proposito de eficiencia.

En la siguiente imagen en la parte superior tenemos la inicialización de los conjuntos con ellos mismo sus padres y con rank cero.



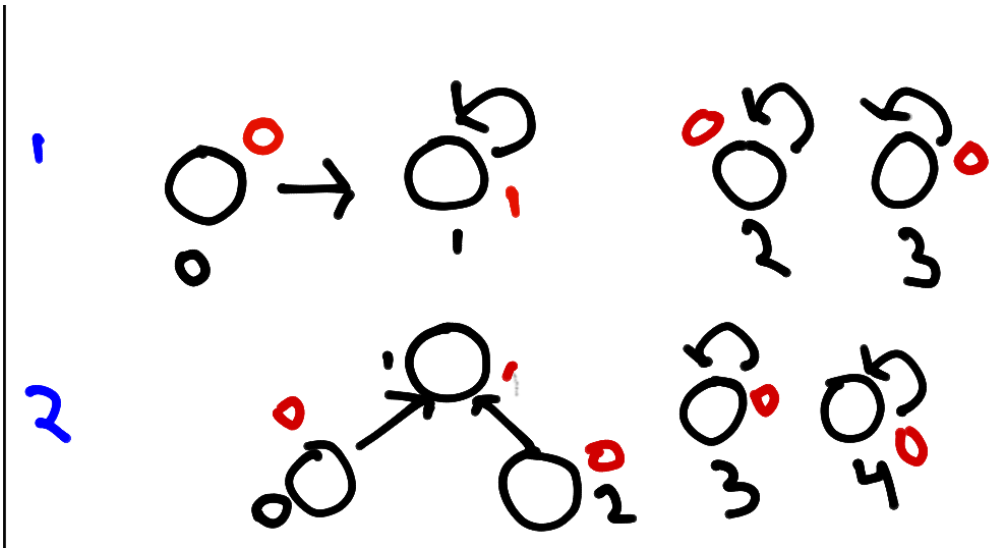
Veamos como se implementan las operaciones :

- **UNION**:
  - En la imagen parte inferior tenemos la operación de unir dos conjuntos, en este caso unimos $\{0\}$ y $\{1\}$ lo que crea $\{0,1\}$ lo cual en la forma que lo estamos pensando simplemente es mover al padre de `0` que se vuelva el que tenga mas rank como tienen lo mismo no importa, así el padre de `0` se vuelve `1` y el rank de `1` aumenta en uno si tenían el mismo rank.
  - Por lo que generalizando tendremos que para unir dos conjuntos basta mover el padre del que tenga menos rango a que ahora sea padre del que tenga más rango y aumentar el rango del padre.
- **FindSet**:
  - Como inicialmente los padres apuntan así mismos para encontrar el elemento significativo de un elemento basta con seguir al apuntador del padre hasta que `p[i] = i` así sabemos que `i` es el elemento siginificativo.
- **SameSet**:
  - Para saber si dos elementos pertenecen al mismo conjunto basta con verificar que su elemento significativo sea el mismo i,e `findSet(i) == findSet(j)`

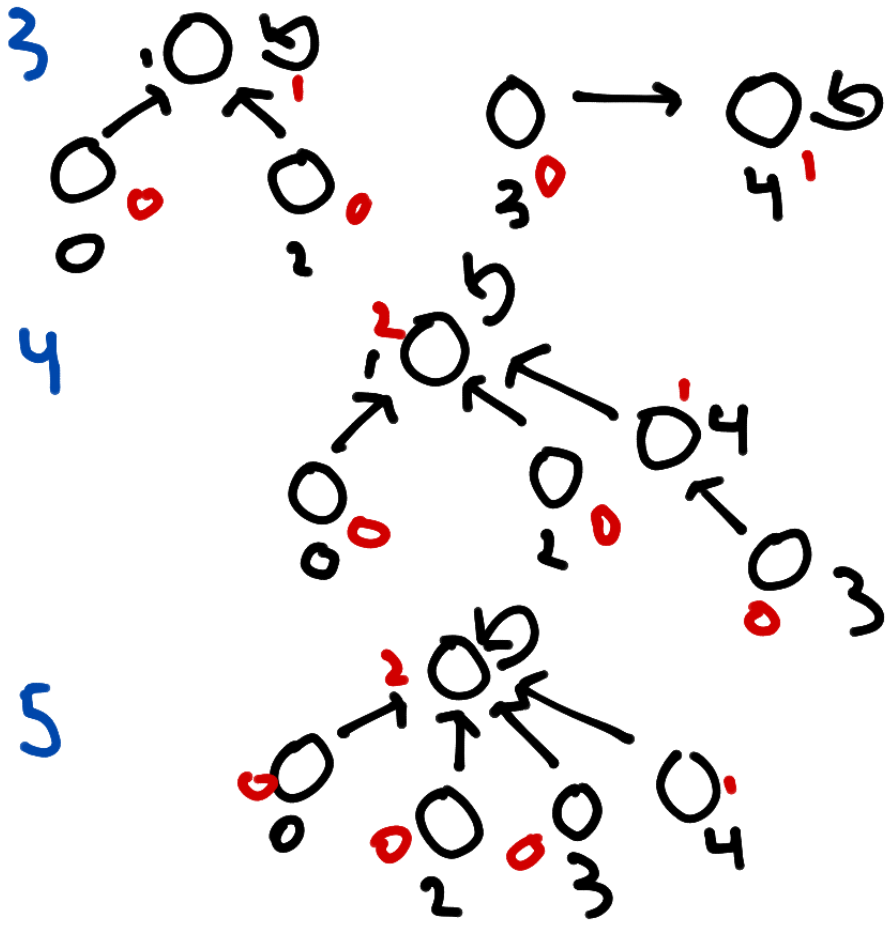
Veamos una ilustración más.

Tomando los conjuntos anteriores hasta el 4. De tal forma que lo tenemos como en la parte 1.

- Ahora realizamos la operacion `Union(0,2)` y nos queda la imagen en la parte 2.



- Ahora realizamos la operación de `Union(3,4)` lo cual obtenemos la parte 3 de la siguiente imagen.
- Ahora si unimos `Union(3,2)` nos queda la imagen 4, notemos algo importante por eficiencia, para encontrar el elemento significativo de 3 tomariamos 2 operaciones, pero si hacemos algo que se llama path compression, que es, mienstras busco su elemento lo actualizado ie, `return p[i] == i ? i : p[i] = findSet(p[i])` lo actualizamos mientras lo buscamos.
- Después de la compresión del camino nos queda la imagen 5, ya en este caso encotnrar el elemento significativo se hace en $\mathbb{O}(1)$



Ahora implemenetemos esta idea en C++, usaremos un poco de la idea de la programación orientada a objetos que en concursos nos podría salvar tiempo si es que necesitamos usar distintos grupo de conjuntos, pero las variables declaradas en la estructuras podrían ser declaradas globales y las funciones también y funcionarían igual.

```
struct UnionFind{
        vector<int> p;
        vector<int> rank;
        int num_conj; // cantidad de elementos
        UnionFind(int n){// constructor inicializa mis elementos
                p.assign(n, 0);
                for(int i=0; i<n; i++) p[i] = i;
                rank.assign(n,0);
                num_conj = n;
        }
        int findSet(int i){
                return p[i] == i ? i : (p[i] = findSet(p[i]));
        }
        bool isSameSet(int i, int j){
                return findSet(i) == findSet(j);
        }
        void UnionSet(int i, int j){
                if(!isSameSet(i,j)){
                        num_conj--;
                        int px = findSet(i);
                        int py = findSet(j);
                        if(rank[px] > rank[py]){
                                p[py] = px;
                        }else{
                                p[px] = py;
                                if(rank[px] == rank[py]) rank[py]++;
                        }
                }
        }
};
```

Añadimos algo más, añadimos la variable `num_conj` la cual nos dice cuantos conjuntos hay en el momento, claramente cada que unimos dos conjuntos distintos un elemento desaparece, e inicialmente tenemos n, por lo que de esa forma podemos saber cuantos conjuntos hay en cada momento.

Complejidad de las operaciones, para propositos de programación competitiva podemos tomarlo como $\approx \mathbb{O}(1)$ pues la real, tomando `M` operaciones de UFDS es en $\mathbb{O}(M\alpha(n))$ tomamos a $\alpha(n)$ como menor de 5 cuando $n \leq 1M$