

CONJUNTOS DISJUNTOS

Representan relaciones de equivalencia entre elementos y resuelven eficientemente el problema de saber si dos elementos están relacionados o establecen nuevas relaciones entre ellos



UNIVERSIDAD
COMPLUTENSE
MADRID

ALBERTO VERDEJO

Relación de equivalencia

Conjuntos disjuntos, a pesar de que aquí los representamos como grafos, son más abstractos.

- Queremos representar una **relación de equivalencia R** :

- Reflexiva: $a R a$
- Simétrica: $a R b \Rightarrow b R a$
- Transitiva: $a R b \wedge b R c \Rightarrow a R c$

Relación de equivalencia implica que A se relaciona con A. Si B está relacionado con A, A lo está con B. Si A está relacionado con B, y B con C entonces relación de EQUIVALENCIA



Partición:

{ 0, 5, 6 }

{ 1, 2 }

{ 3, 7, 8 }

{ 4, 9 }

Las clases de equivalencia es similar a las componentes conexas de un grafo.

Cada elemento pertenece a un conjunto. Estos se llaman clases de equivalencia.

Tener el mismo color de pelo, moreno rubio o rojo es relación de equivalencia entre personas.

Relación de equivalencia dinámica

Las llamamos dinámicas porque van apareciendo nuevas relaciones entre pares de elementos a la vez que nos preguntamos si dos elementos están relacionados

Aquí todavía no hay relaciones.

0

1

2

3

4

5

6

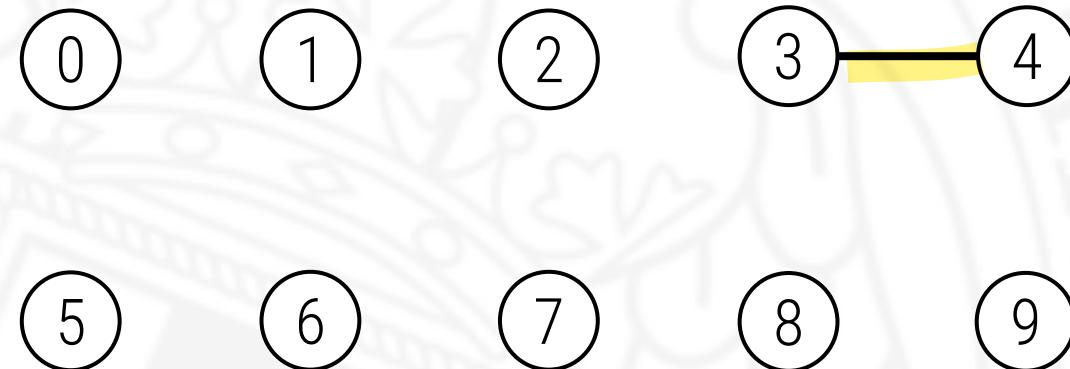
7

8

9

Relación de equivalencia dinámica

$4 R 3$

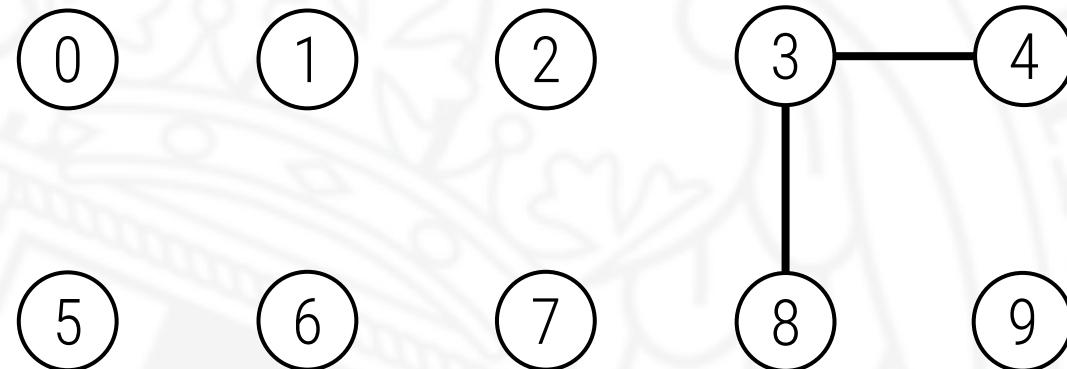


Relación de equivalencia dinámica

$$4 R 3$$

$$3 R 8$$

Por tanto, el 4 y el 8 también lo están

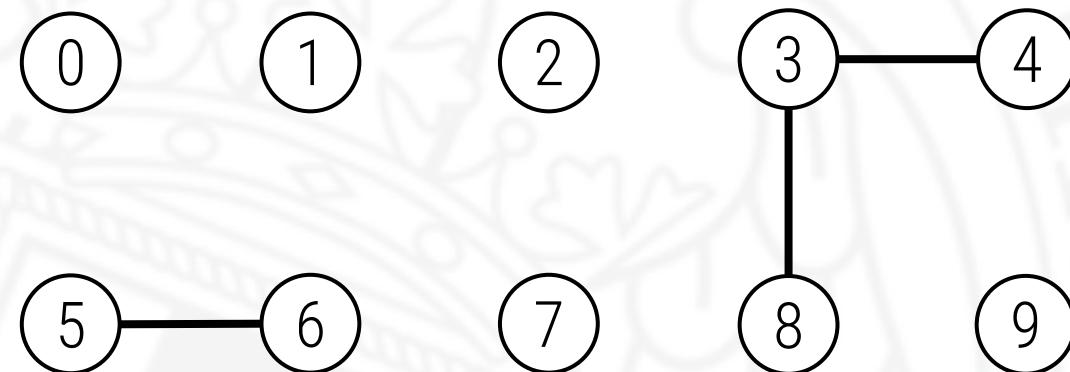


Relación de equivalencia dinámica

$4 R 3$

$3 R 8$

$6 R 5$



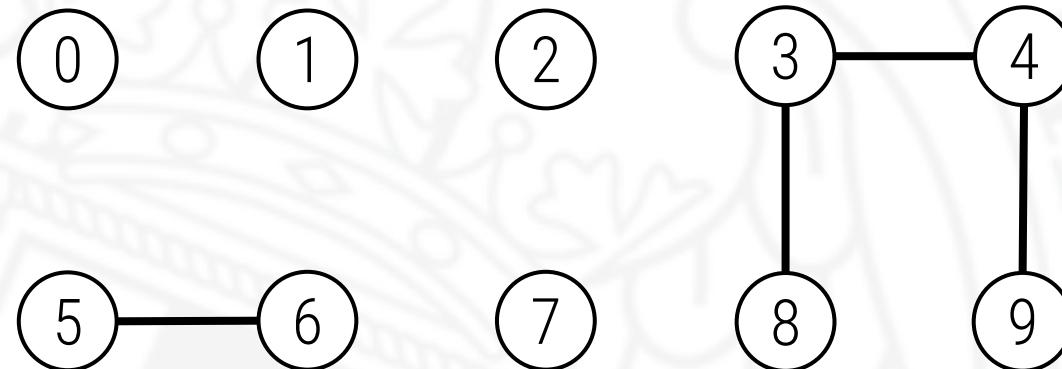
Relación de equivalencia dinámica

$4 R 3$

$3 R 8$

$6 R 5$

$9 R 4$



Relación de equivalencia dinámica

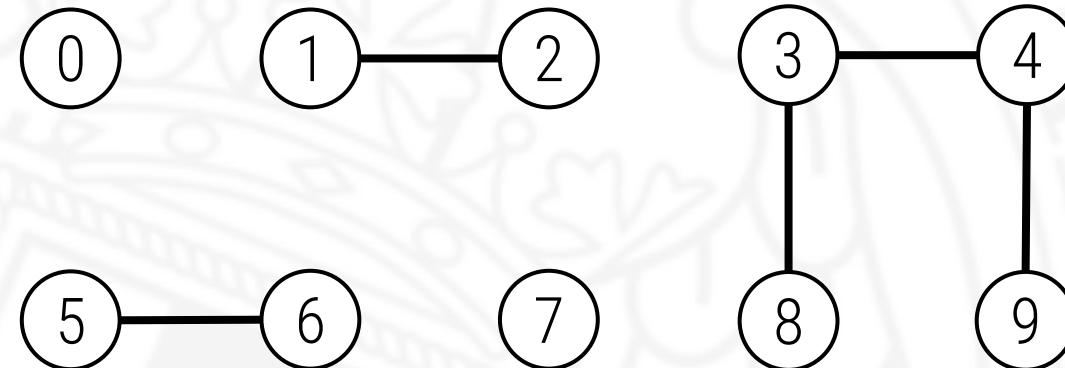
$4 R 3$

$3 R 8$

$6 R 5$

$9 R 4$

$2 R 1$



Relación de equivalencia dinámica

$4 R 3$

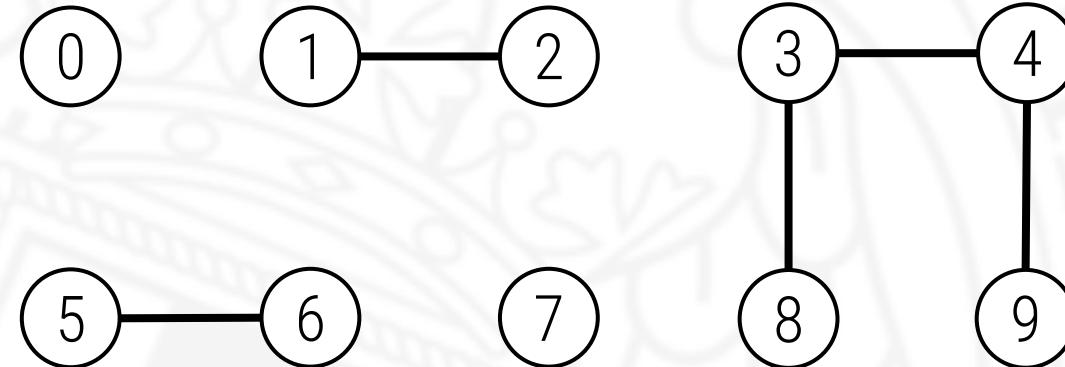
$3 R 8$

$6 R 5$

$9 R 4$

$2 R 1$

¿ $8 R 9$? ✓



Relación de equivalencia dinámica

$4 R 3$

$3 R 8$

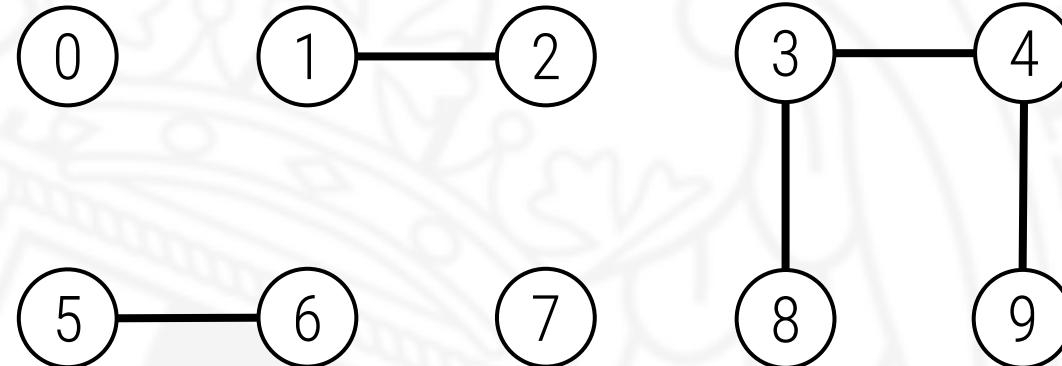
$6 R 5$

$9 R 4$

$2 R 1$

¿ $8 R 9$? 

¿ $5 R 7$? 



Relación de equivalencia dinámica

$4 R 3$

$3 R 8$

$6 R 5$

$9 R 4$

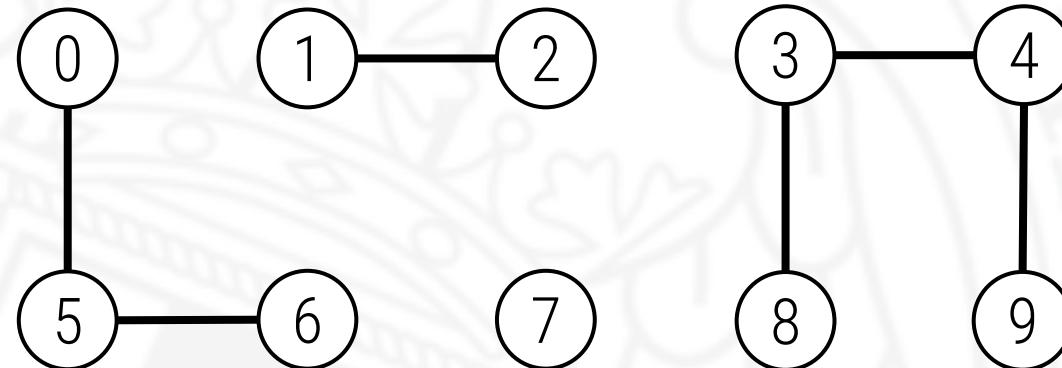
$2 R 1$

\checkmark

\checkmark

$5 R 7 ?$

X



Relación de equivalencia dinámica

$4 R 3$

$3 R 8$

$6 R 5$

$9 R 4$

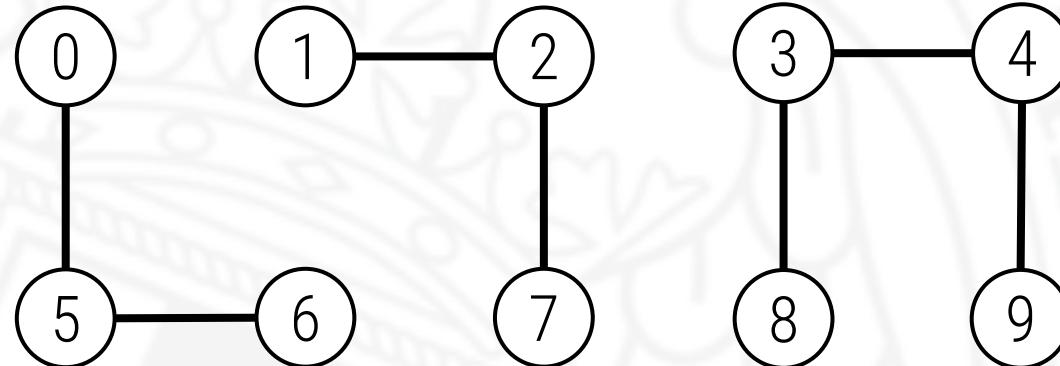
$2 R 1$

¿ $8 R 9$? ✓

¿ $5 R 7$? ✗

$5 R 0$

$7 R 2$



Relación de equivalencia dinámica

$4 R 3$

$3 R 8$

$6 R 5$

$9 R 4$

$2 R 1$

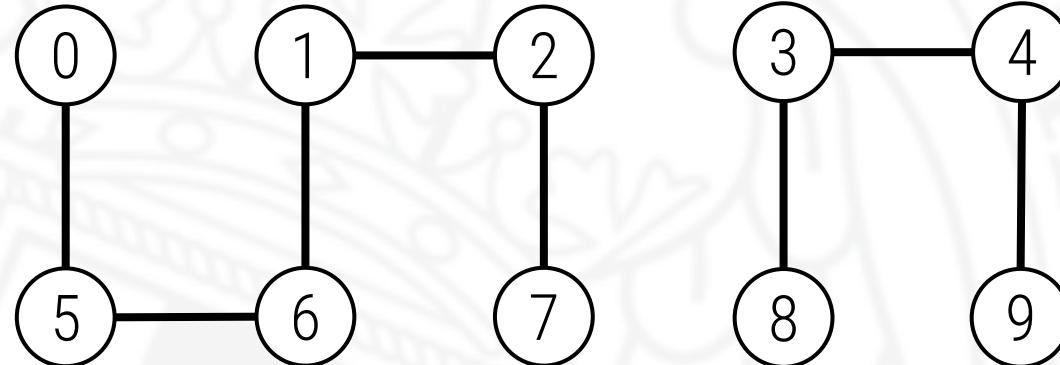
¿ $8 R 9$? ✓

¿ $5 R 7$? ✗

$5 R 0$

$7 R 2$

$6 R 1$



Relación de equivalencia dinámica

$4 R 3$

$3 R 8$

$6 R 5$

$9 R 4$

$2 R 1$

¿ $8 R 9$? ✓

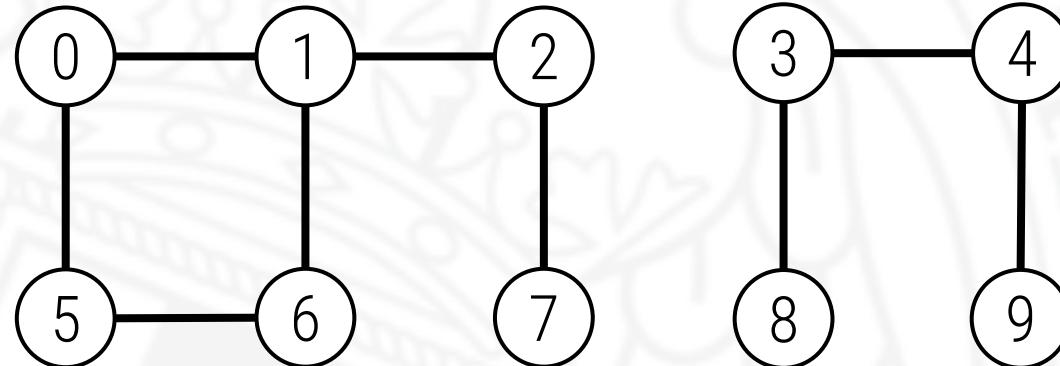
¿ $5 R 7$? ✗

$5 R 0$

$7 R 2$

$6 R 1$

$1 R 0$



Relación de equivalencia dinámica

$4 R 3$

$3 R 8$

$6 R 5$

$9 R 4$

$2 R 1$

¿ $8 R 9$? 

¿ $5 R 7$? 

$5 R 0$

$7 R 2$

Ahora si están relacionados. Es una relación dinámica.

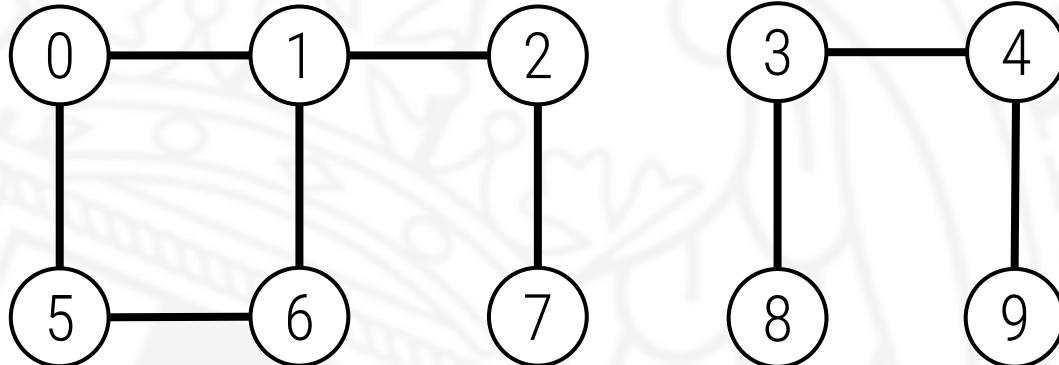
$6 R 1$

$1 R 0$



Esto ya lo sabíamos nosotros.

¿ $5 R 7$? 



Uso típico que vamos a querer hacer de las relaciones de equivalencia.

TAD de conjuntos disjuntos

Son las más importantes.

El TAD de los conjuntos disjuntos cuenta con las siguientes operaciones:

- ▶ crear una partición unitaria, **ConjuntosDisjuntos(int N)**
- ▶ unir dos conjuntos, **void unir(int a, int b)**
- ▶ buscar un elemento, **int buscar(int a) const**

Habrá tantos conjuntos como elementos, cada elemento está relacionado consigo mismo.

Si ya estaban relacionados, no tiene efecto.

Une dos conjuntos.

Cuando 2 cjos se unen, el repre de la unión es uno de los repres de esas dos clases.

busca al representante del conjunto a, que puede ser cualquier elemento del conjunto, pero no cambiará mientras no cambie la clase.

- ▶ consultar si dos elementos pertenecen al mismo conjunto,

bool unidos(int a, int b) const

Si pertenecen a la misma clase de equivalencia (están relacionados o no).

- ▶ consultar el cardinal de un conjunto, **int cardinal(int a) const**

Cardinal del cjo con el elemento a.

- ▶ consultar el número de conjuntos, **int num_cjtos() const**

Possible implementación: búsqueda rápida

	0	1	2	3	4	5	6	7	8	9
id[]	0	1	1	8	8	0	0	1	8	8

Guardamos para cada elemento el representante del conjunto en el que está.

$$\{ \color{red}{0}, 5, 6 \} \quad \{ \color{red}{1}, 2, 7 \} \quad \{ 3, 4, \color{red}{8}, 9 \}$$

Representantes.

Sabemos si dos elementos están relacionados si tienen el mismo representante.

Possible implementación: búsqueda rápida

`unir(6, 1)`

El coste en todos los casos sería lineal.

	0	1	2	3	4	5	6	7	8	9
id[]	0	1	1	8	8	0	0	1	8	8
	↑			↑	↑					

O cambiamos los 0 por 1's o los 1's por 0's

Hay que recorrer siempre todas las posiciones del vector comprobando para cada posición si tiene un 0 cambiarlo a 1.

Possible implementación: búsqueda rápida

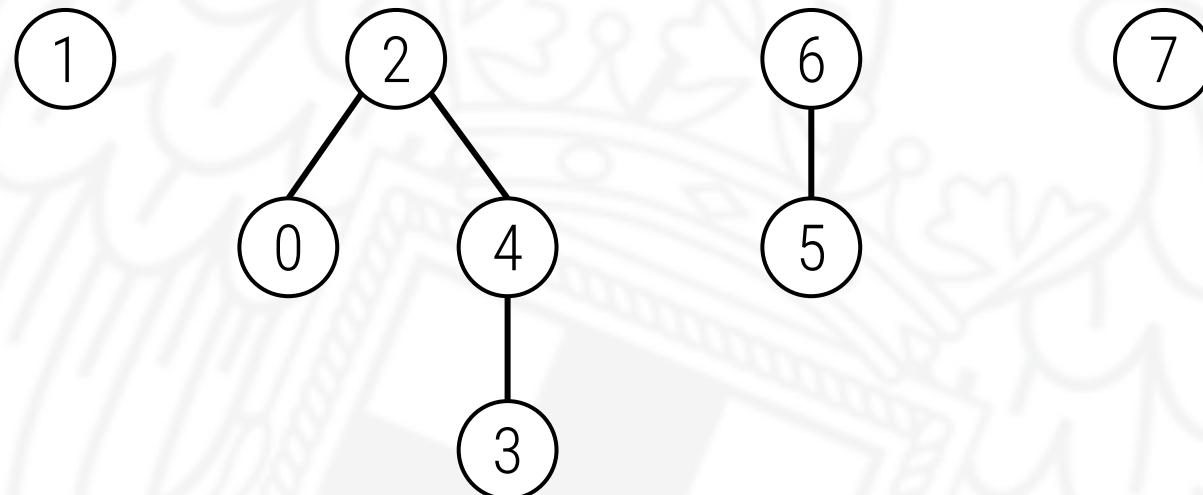
`unir(6, 1)`

	0	1	2	3	4	5	6	7	8	9
<code>id[]</code>	1	1	1	8	8	1	1	1	8	8
	↑					↑	↑			

Unión rápida

- ▶ Cada conjunto se representa mediante un árbol.

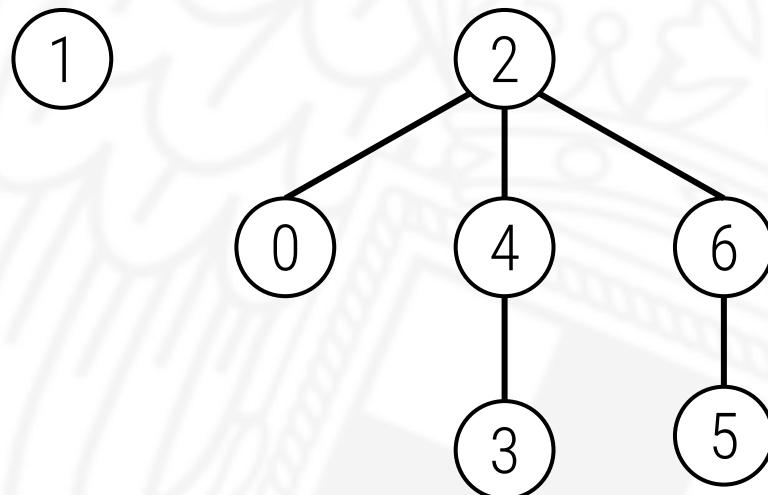
Para unirlos es muy fácil. basta poner a uno de ellos como hijo del otro.



Unión rápida

- ▶ Cada conjunto se representa mediante un árbol.

Usamos la RAÍZ como el representante de la clase de equivalencia.



Unión rápida

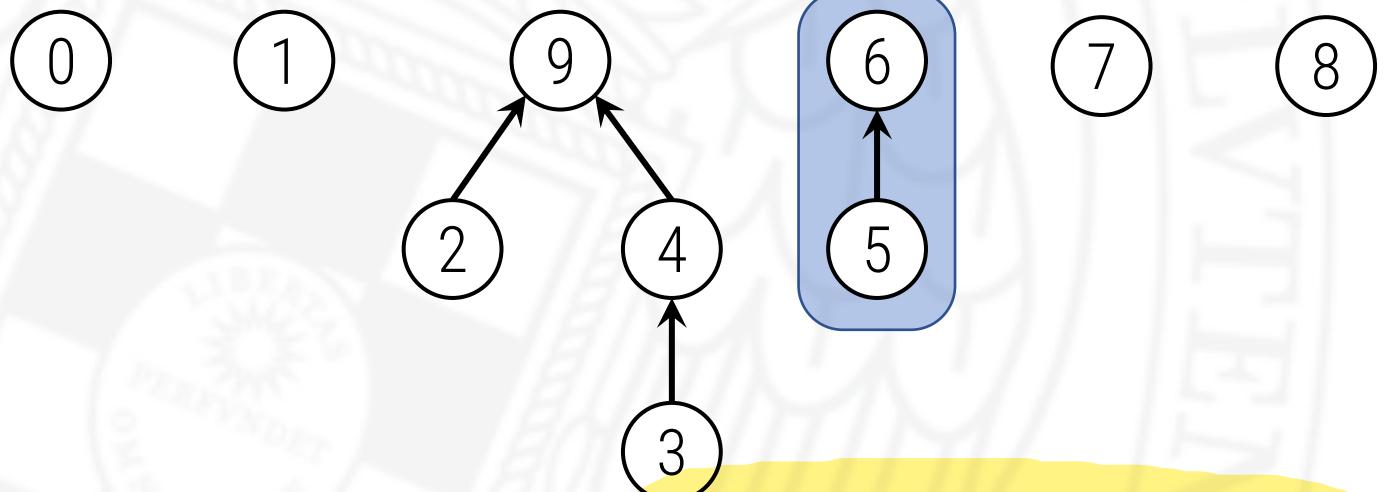
- ▶ Cada conjunto se representa mediante un árbol. $p[i]$ es el padre de i

Para representar todos los árboles hacemos uso de un vector. Donde para cada elemento guardamos cual es su padre en el árbol en el que se encuentra.

	0	1	2	3	4	5	6	7	8	9
$p[]$	0	1	9	4	9	6	6	7	8	9

9 es raíz.

{ 0 }
{ 1 }
{ 2, 3, 4, 9 }
{ 5, 6 }
{ 7, 8 }

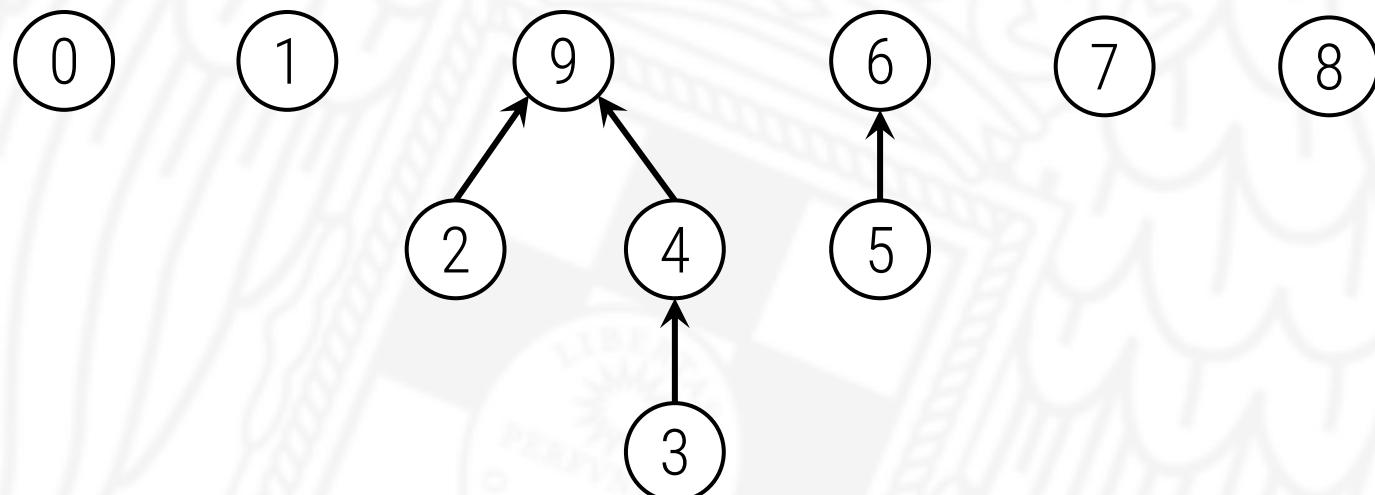


Pasamos de padre en padre hasta alcanzar a la raíz.

Unión rápida

`unir(3, 5)`

	0	1	2	3	4	5	6	7	8	9
p[]	0	1	9	4	9	6	6	7	8	9

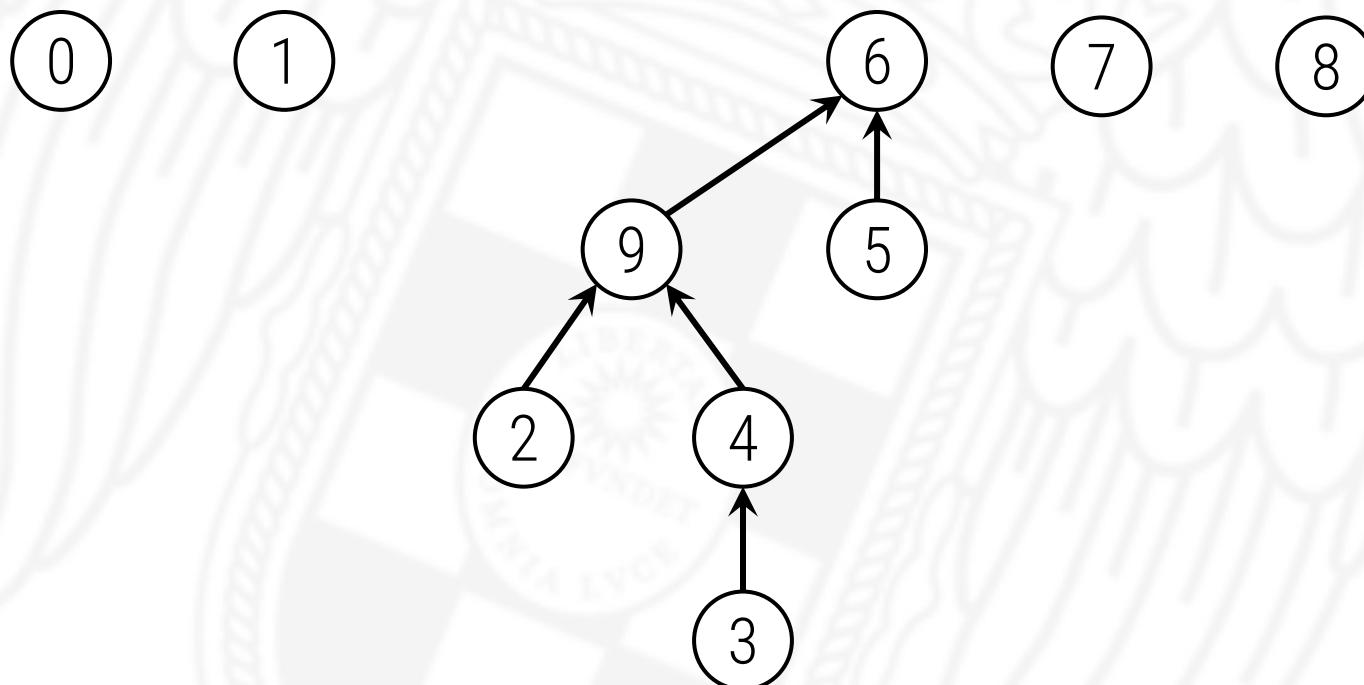


Unión rápida

`unir(3, 5)`

	0	1	2	3	4	5	6	7	8	9
p[]	0	1	9	4	9	6	6	7	8	6

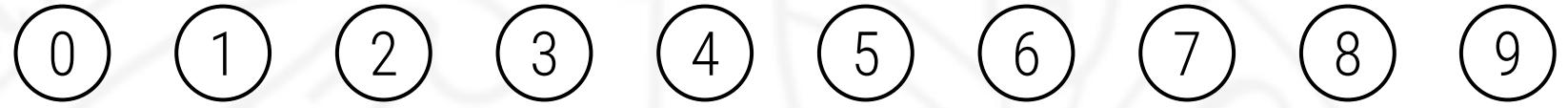
Simplemente cambias el padre de la raíz de uno de los dos árboles.



Unión rápida

Haciéndolo de esta manera podemos tener un problema.

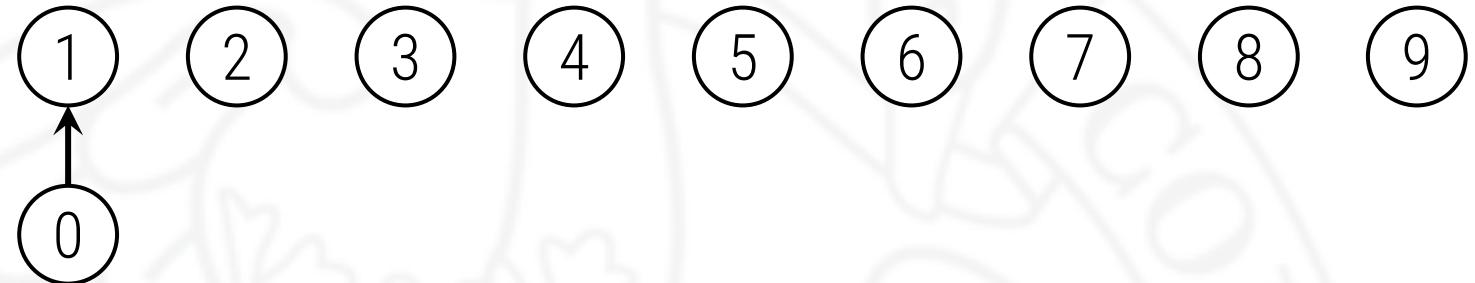
`unir(0, 1)`



Unión rápida

`unir(0, 1)`

`unir(0, 2)`



Unión rápida

`unir(0, 1)`

`unir(0, 2)`

`unir(0, 3)`



Unión rápida

`unir(0, 1)`

`unir(0, 2)`

`unir(0, 3)`

`unir(0, 4)`



Unión rápida, ¿búsqueda lenta?

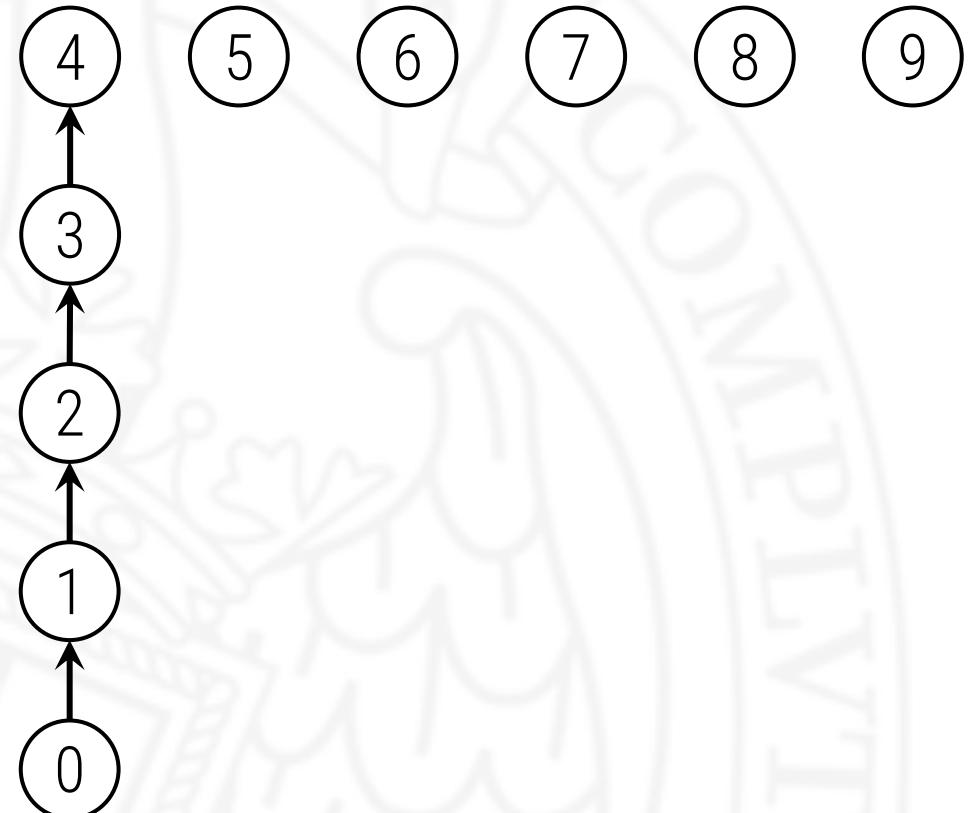
unir(0, 1)

unir(0, 2)

unir(0, 3)

unir(0, 4)

Esto puede ser lineal porque siempre buscamos en el árbol más grande la raíz, teniendo que pasar por cada uno de sus nodos.



Unión por tamaños

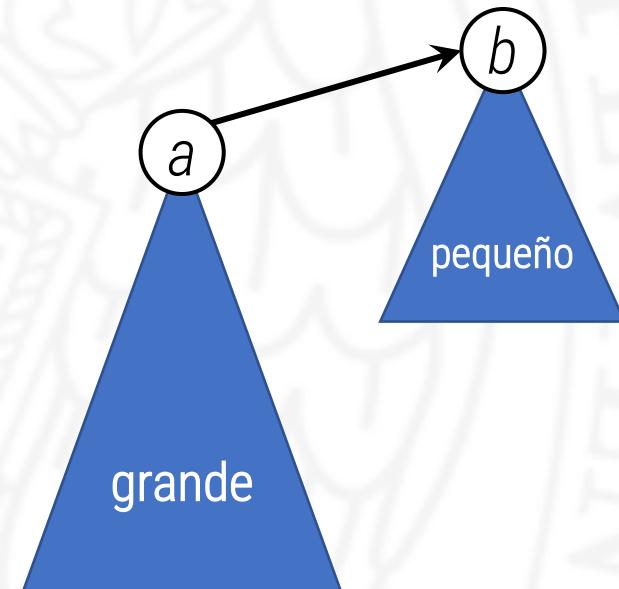
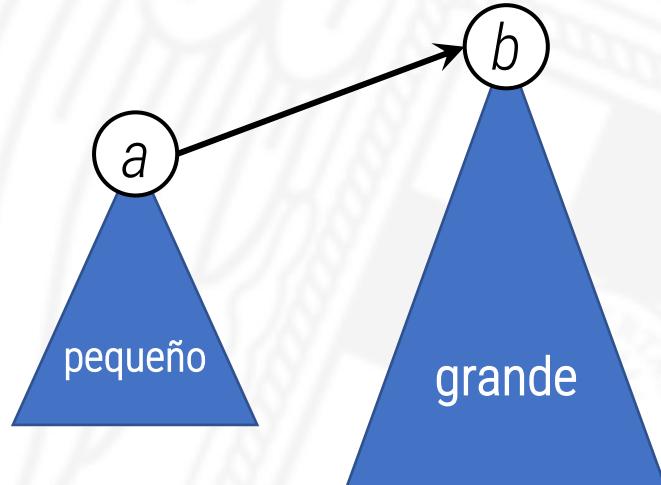
Al unir, el árbol más pequeño pasa a ser hijo del árbol más grande.

- ▶ Vamos a mantener los tamaños de los árboles (número de elementos)
- ▶ Al unir, el árbol más pequeño pasa a ser hijo del árbol mayor

Unión rápida

En la unión rápida a veces el grande es el que apunta al pequeño, lo que puede hacer que el coste sea mayor.

`unir(a, b)`



Unión por tamaños

- ▶ Vamos a mantener los tamaños de los árboles (número de elementos)
- ▶ Al unir, el árbol más pequeño pasa a ser hijo del árbol mayor

Unión rápida por tamaños

`unir(a, b)`

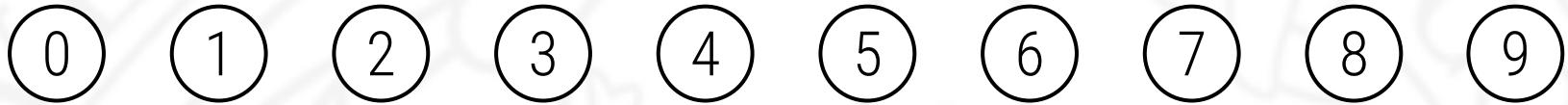
De esta manera la altura crece menos veces. Siempre el pequeño apunta al grande.



Esto hace que sea logarítmica con respecto al número de elementos en la partición.

Unión por tamaños

unir(4, 3)

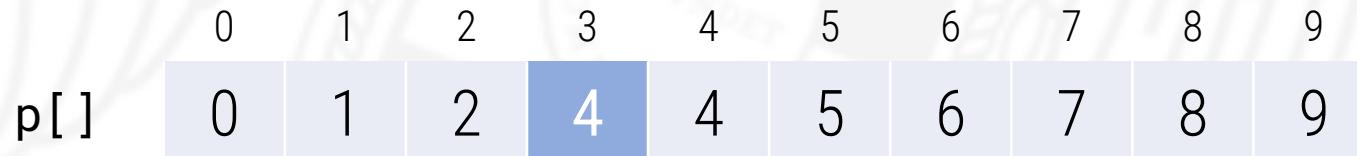
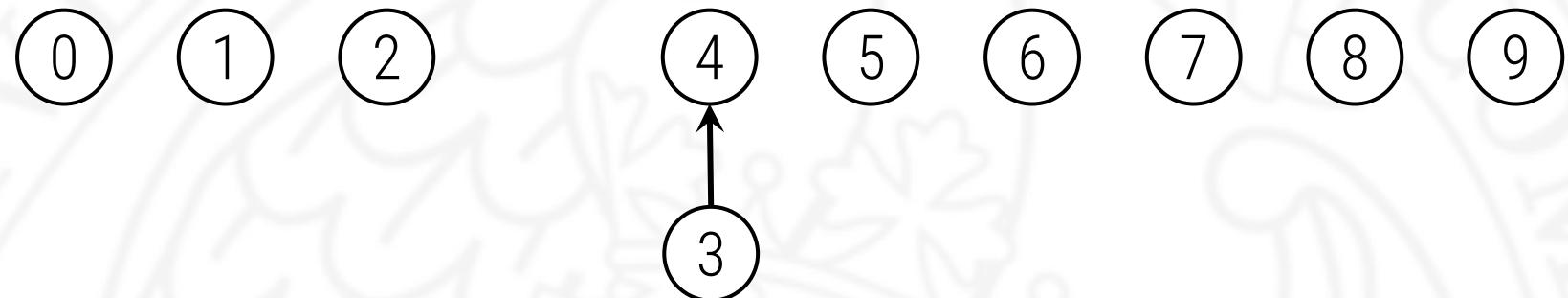


Como ahora mismo el tamaño coincide hacemos que el primero sea el que se queda como raíz. A igualdad, el de la derecha se COMPORTA como si fuera el pequeño.

	0	1	2	3	4	5	6	7	8	9
p[]	0	1	2	3	4	5	6	7	8	9

Unión por tamaños

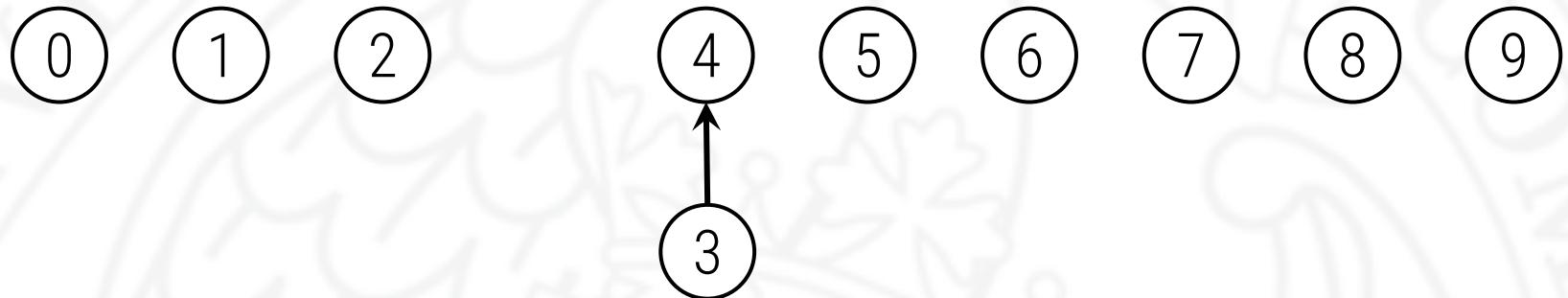
`unir(4, 3)`



Unión por tamaños

unir(3, 8)

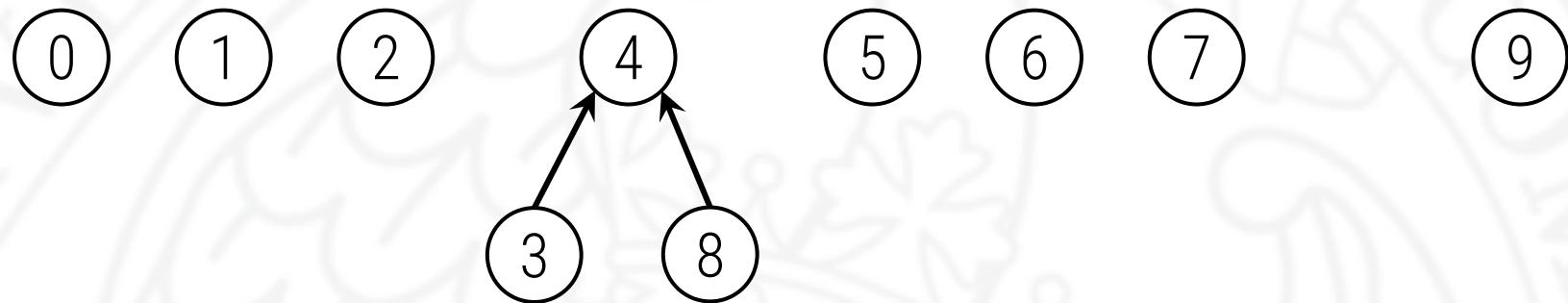
Apuntará a la raíz del 3 que tiene mayor cardinal.



	0	1	2	3	4	4	5	6	7	8	9
p[]	0	1	2	4	4	5	6	7	8	9	

Unión por tamaños

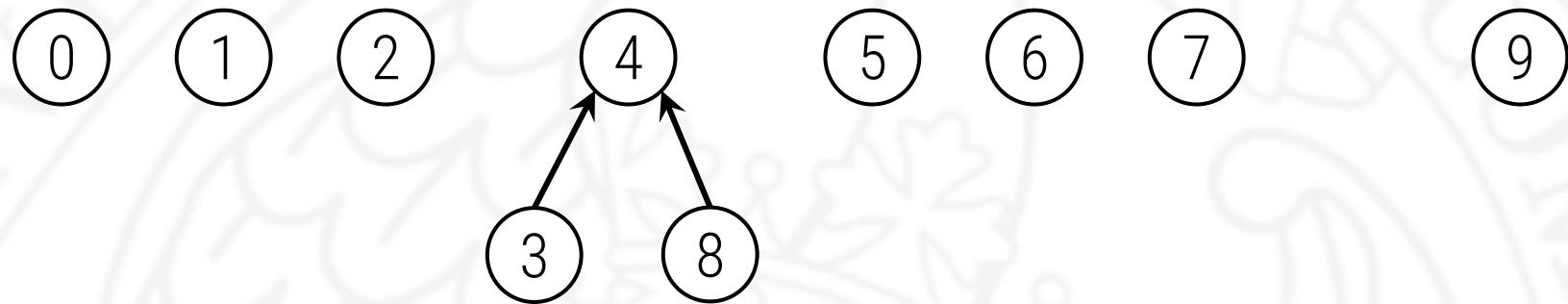
`unir(3, 8)`



0	1	2	3	4	4	5	6	7	8	9
p[]	0	1	2	4	4	5	6	7	4	9

Unión por tamaños

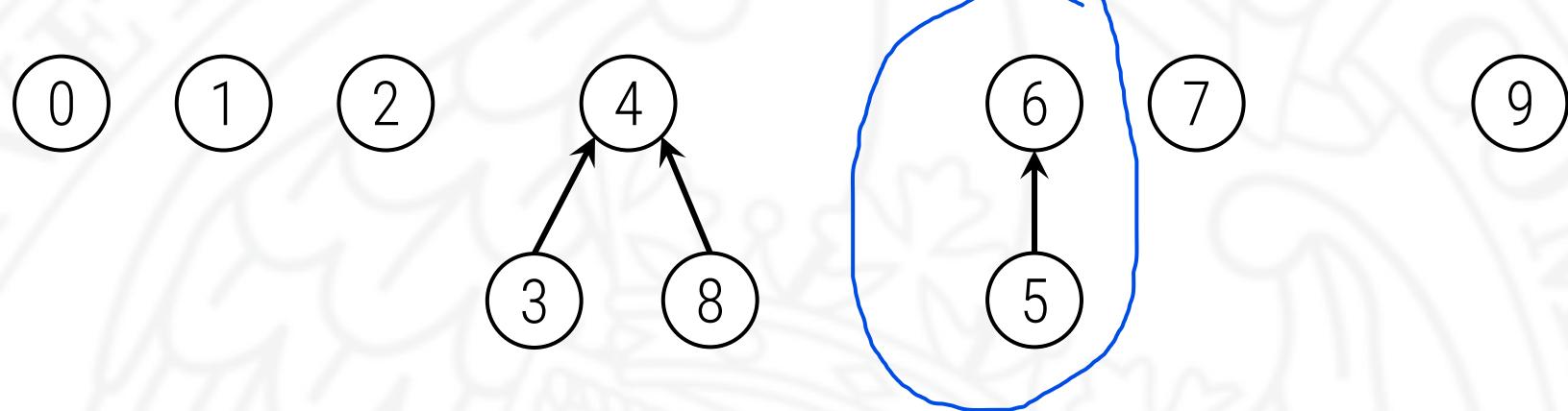
`unir(6, 5)`



	0	1	2	3	4	5	6	7	8	9
<code>p[]</code>	0	1	2	4	4	5	6	7	4	9

Unión por tamaños

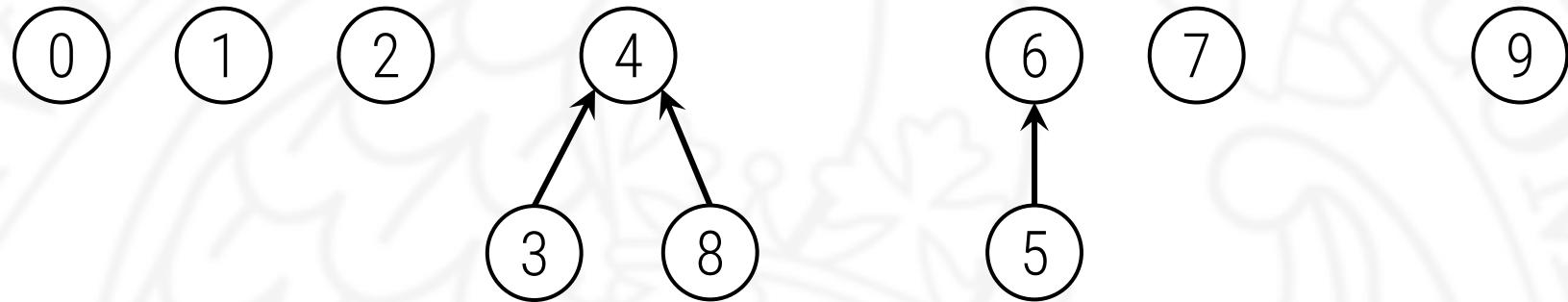
unir(6, 5)



0	1	2	3	4	4	5	6	7	8	9
p[]	0	1	2	4	4	6	6	7	4	9

Unión por tamaños

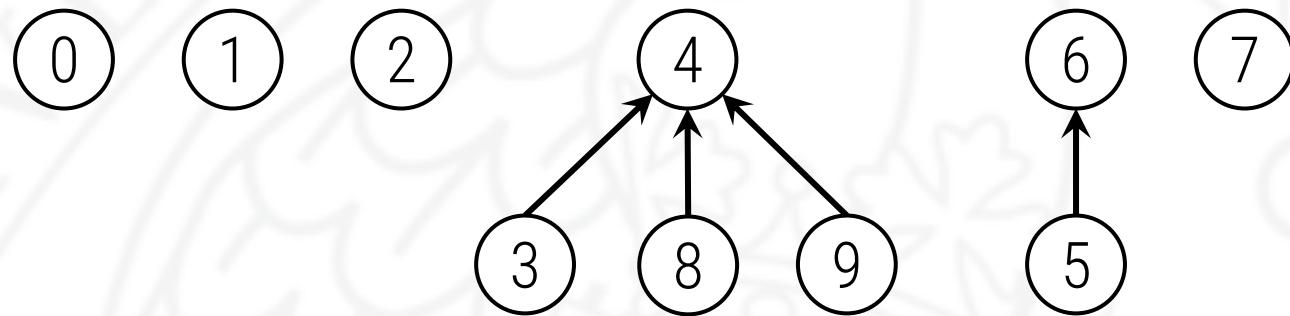
unir(9, 4)



0	1	2	3	4	4	6	6	7	4	9
p[]	0	1	2	4	4	6	6	7	4	9

Unión por tamaños

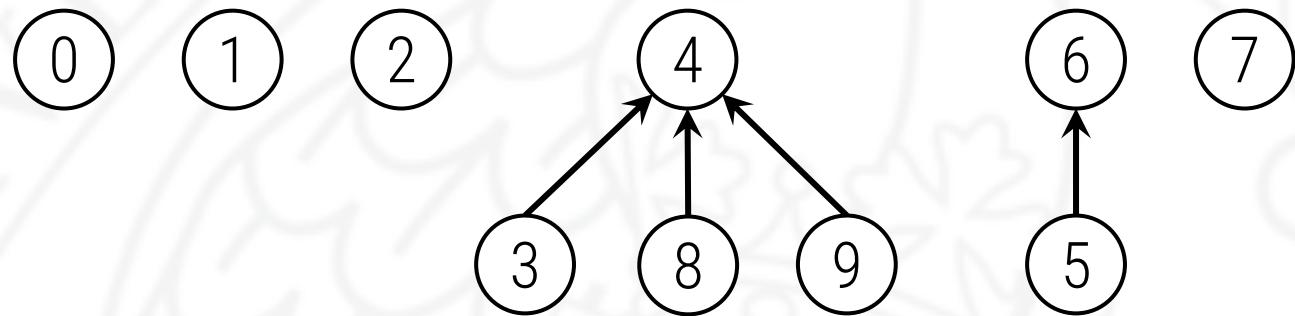
`unir(9, 4)`



0	1	2	3	4	4	6	6	7	4	4
p[]	0	1	2	4	4	6	6	7	4	4

Unión por tamaños

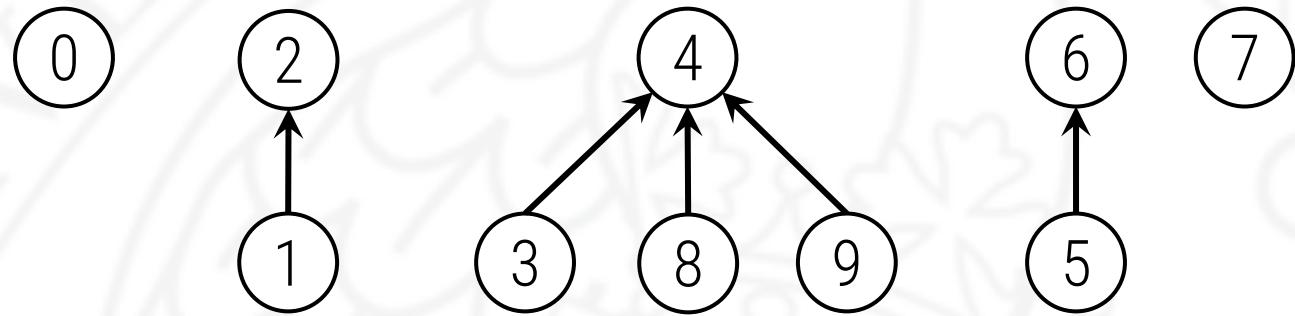
`unir(2, 1)`



0	1	2	3	4	4	6	6	7	4	4
p[]	0	1	2	4	4	6	6	7	4	4

Unión por tamaños

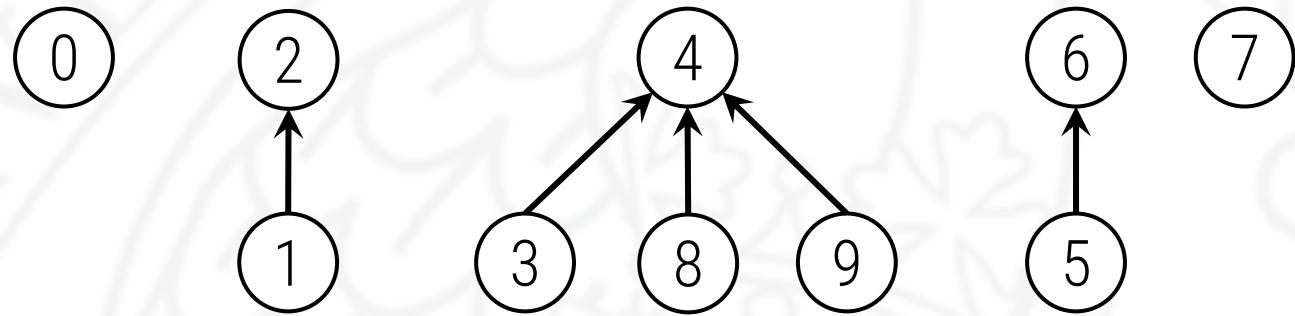
`unir(2, 1)`



0	1	2	3	4	5	6	7	8	9	
p[]	0	2	2	4	4	6	6	7	4	4

Unión por tamaños

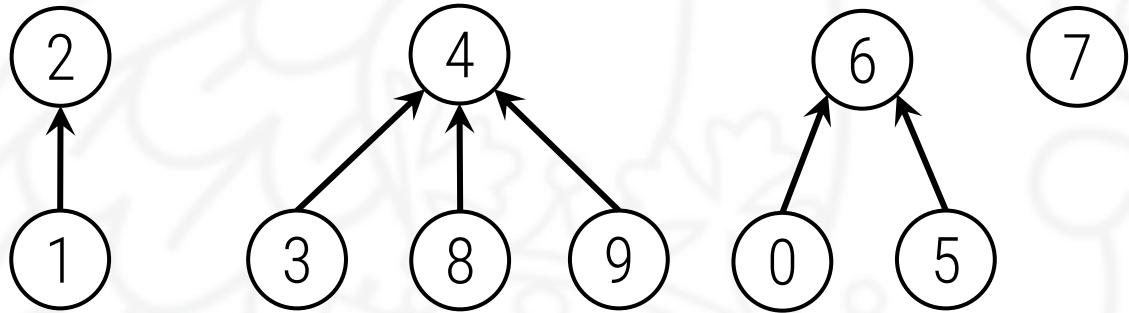
unir(5, 0)



0	1	2	3	4	5	6	7	8	9	
p[]	0	2	2	4	4	6	6	7	4	4

Unión por tamaños

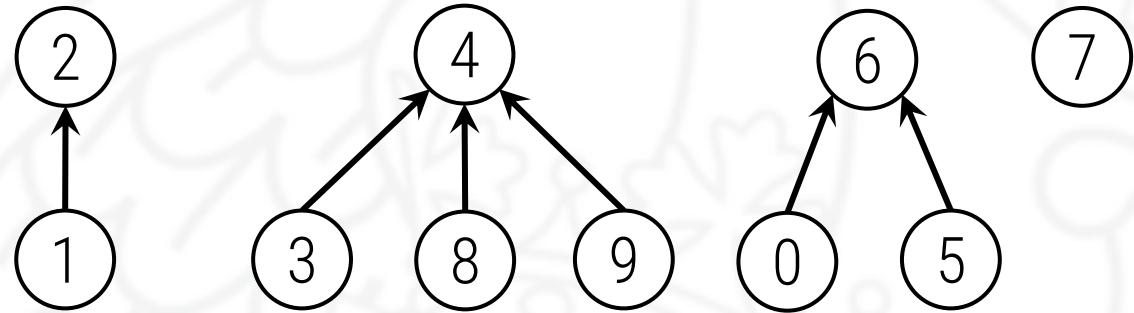
`unir(5, 0)`



0	1	2	3	4	5	6	7	8	9	
p[]	6	2	2	4	4	6	6	7	4	4

Unión por tamaños

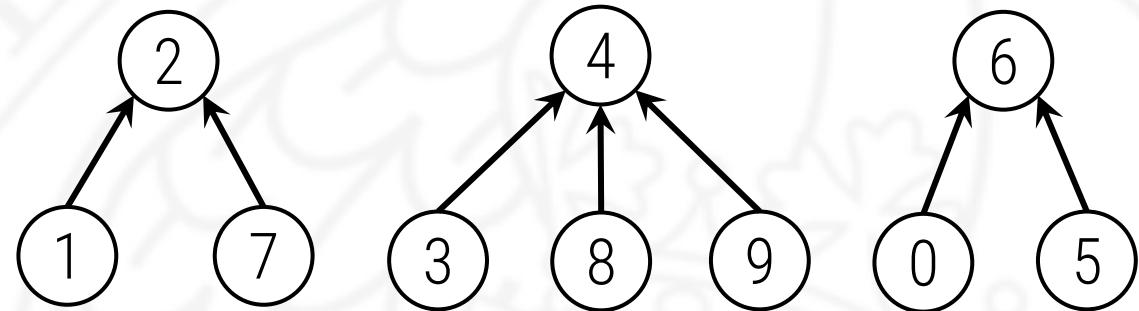
unir(7, 2)



0	1	2	3	4	5	6	7	8	9	
p[]	6	2	2	4	4	6	6	7	4	4

Unión por tamaños

unir(7, 2)

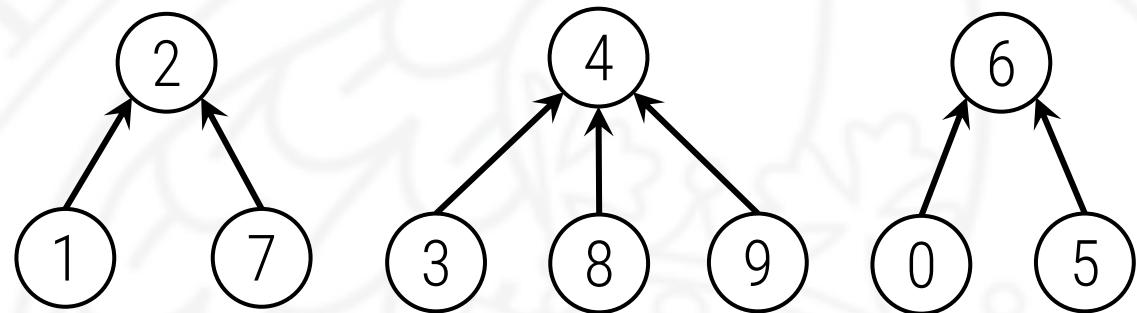


0	1	2	3	4	5	6	7	8	9
p[]	6	2	2	4	4	6	6	2	4

Unión por tamaños

unir(6, 1)

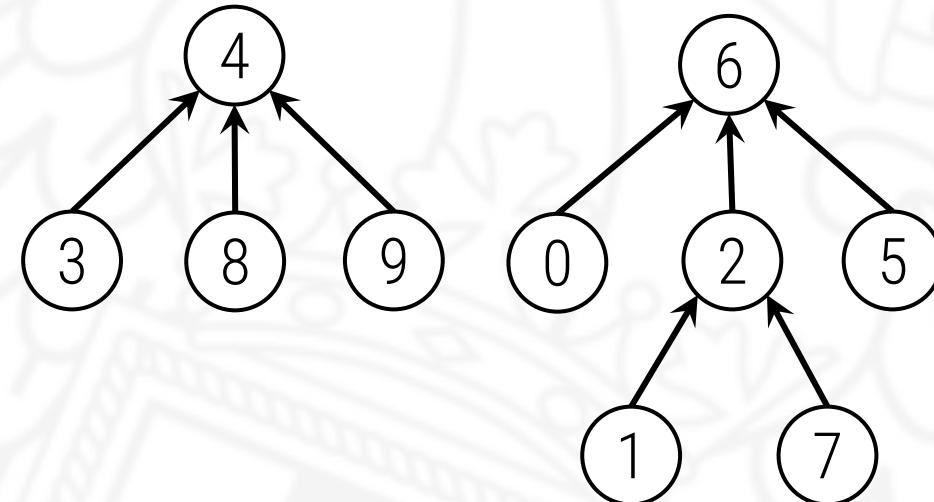
Su raíz se unirá al 6



0	1	2	3	4	5	6	7	8	9	
p[]	6	2	2	4	4	6	6	2	4	4

Unión por tamaños

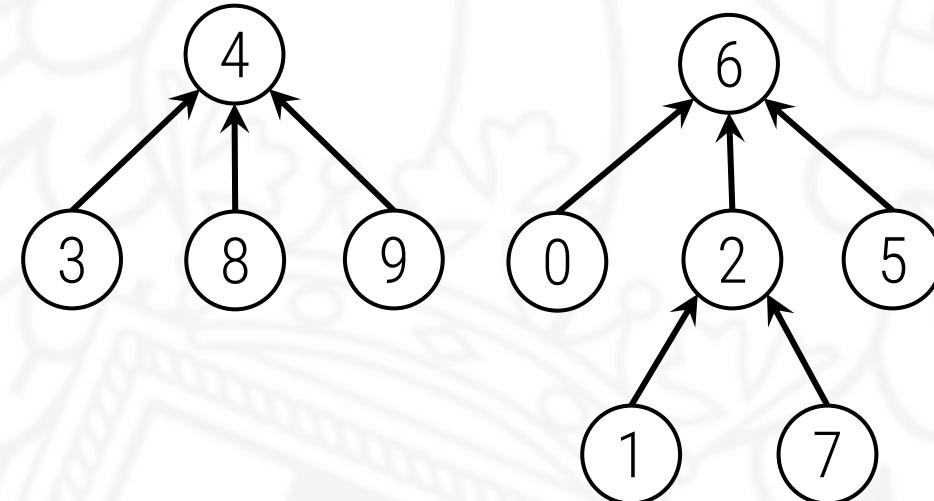
`unir(6, 1)`



0	1	2	3	4	5	6	7	8	9	
p[]	6	2	6	4	4	6	6	2	4	4

Unión por tamaños

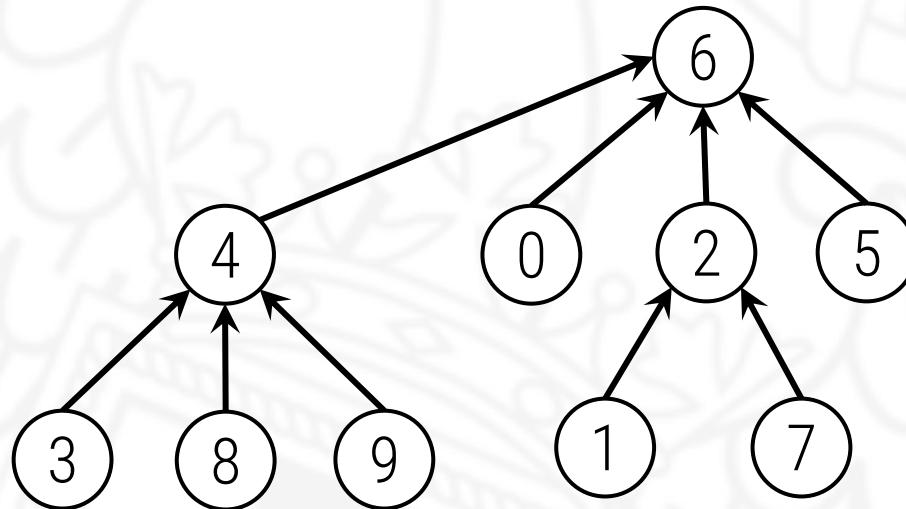
`unir(7, 3)`



0	1	2	3	4	5	6	7	8	9	
p[]	6	2	6	4	4	6	6	2	4	4

Unión por tamaños

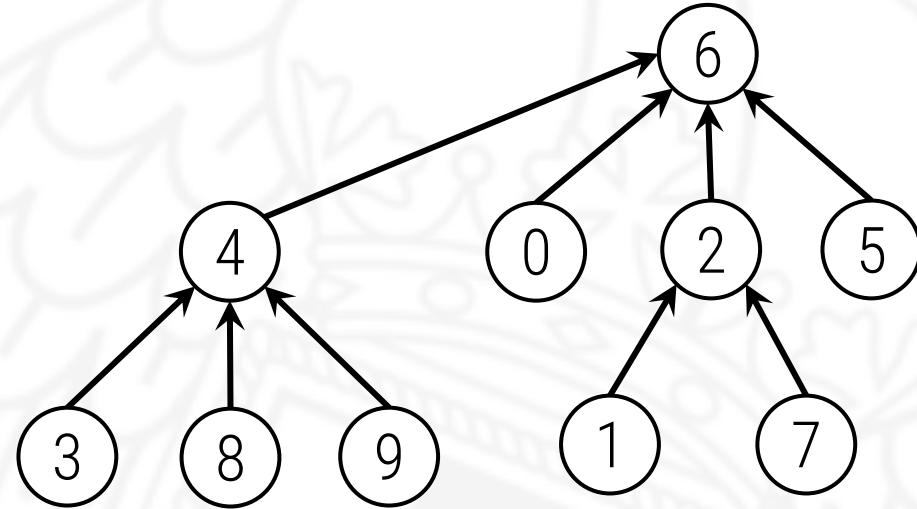
`unir(7, 3)`



0	1	2	3	4	5	6	7	8	9	
p[]	6	2	6	4	6	6	6	2	4	4

Unión por tamaños

unir(7, 3)



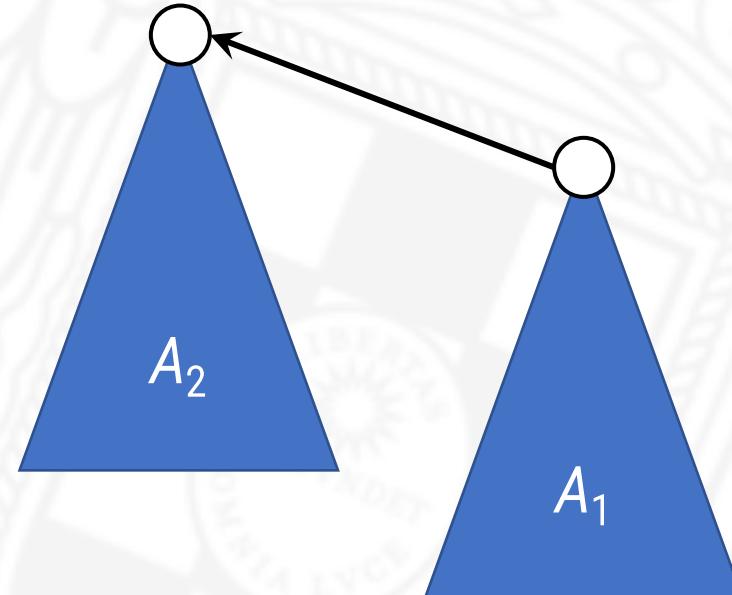
Si solo tenemos un árbol NO es posible hacer más uniones que modifiquen la estructura.

0	1	2	3	4	5	6	7	8	9	
p[]	6	2	6	4	6	6	6	2	4	4

Unión por tamaños, análisis

- ▶ La unión de raíces es constante.
- ▶ La búsqueda es proporcional a la profundidad del elemento buscado.
- ▶ Al hacer la unión por tamaños, la profundidad está acotada por $O(\log N)$.

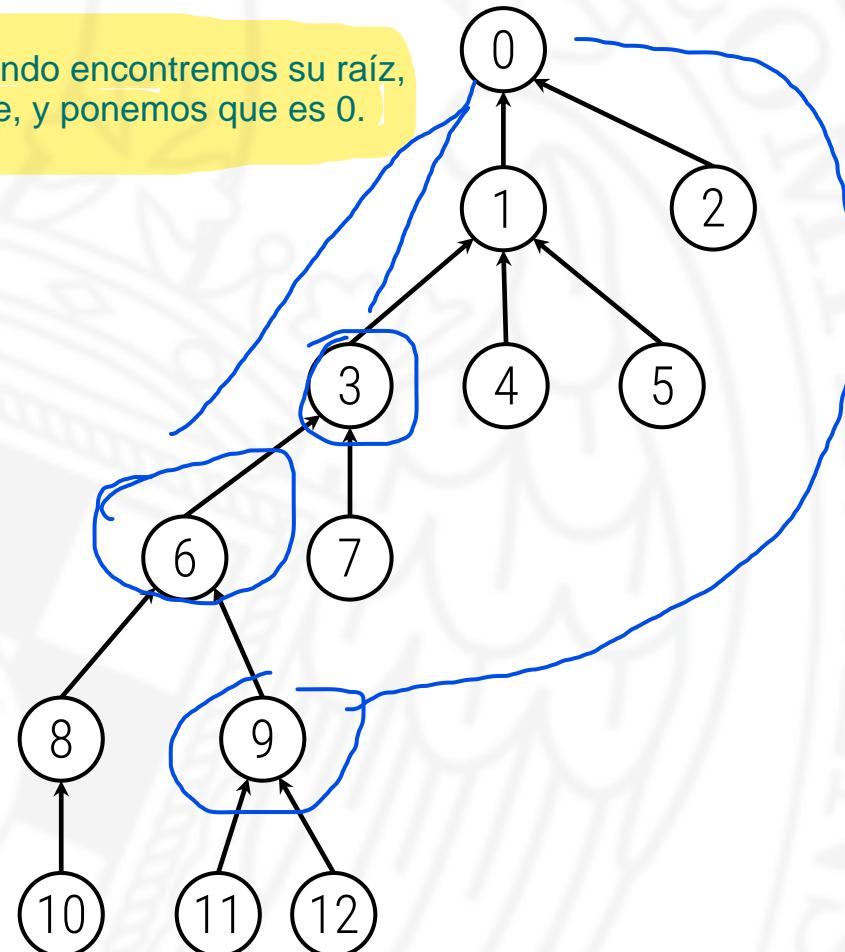
Es logarítmico con respecto al número de nodos en la partición.



Unión por tamaños y compresión de caminos

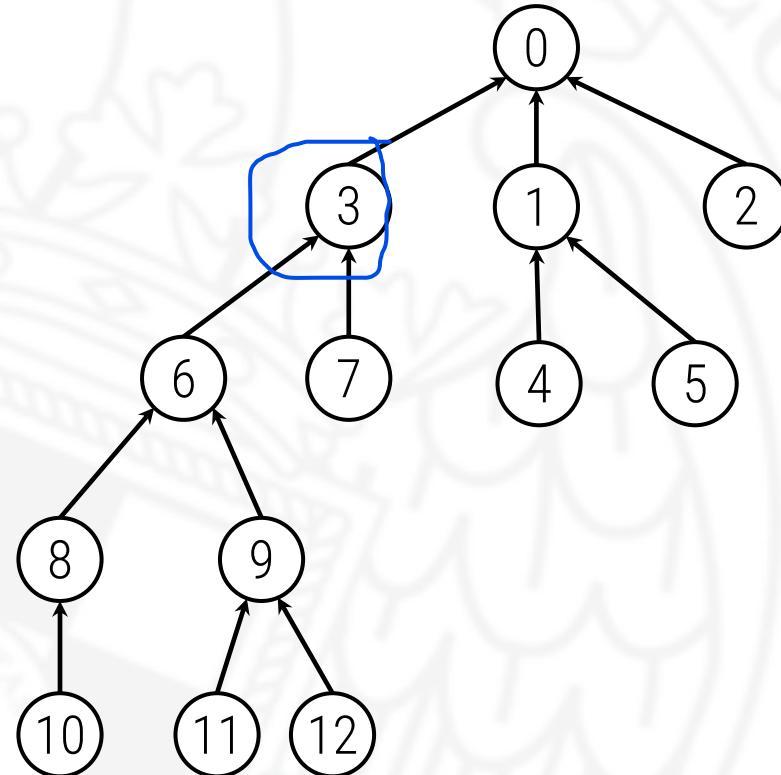
- Después de buscar la raíz del árbol donde se encuentra x, cambiar su padre para que sea esa raíz.

Es decir, cuando buscamos la raíz del 9, que viendo el árbol es 0, cuando encontramos su raíz, cambiamos para todos los que forman parte de esa búsqueda el padre, y ponemos que es 0. Aquí pondríamos al 9, al 6 y al 3.



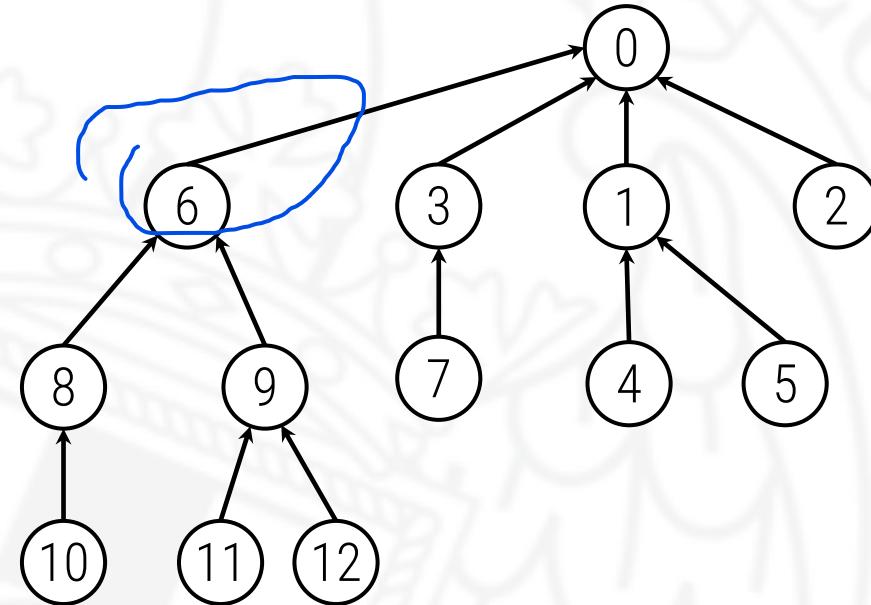
Unión por tamaños y compresión de caminos

- ▶ Después de buscar la raíz del árbol donde se encuentra x , cambiar su padre para que sea esa raíz.



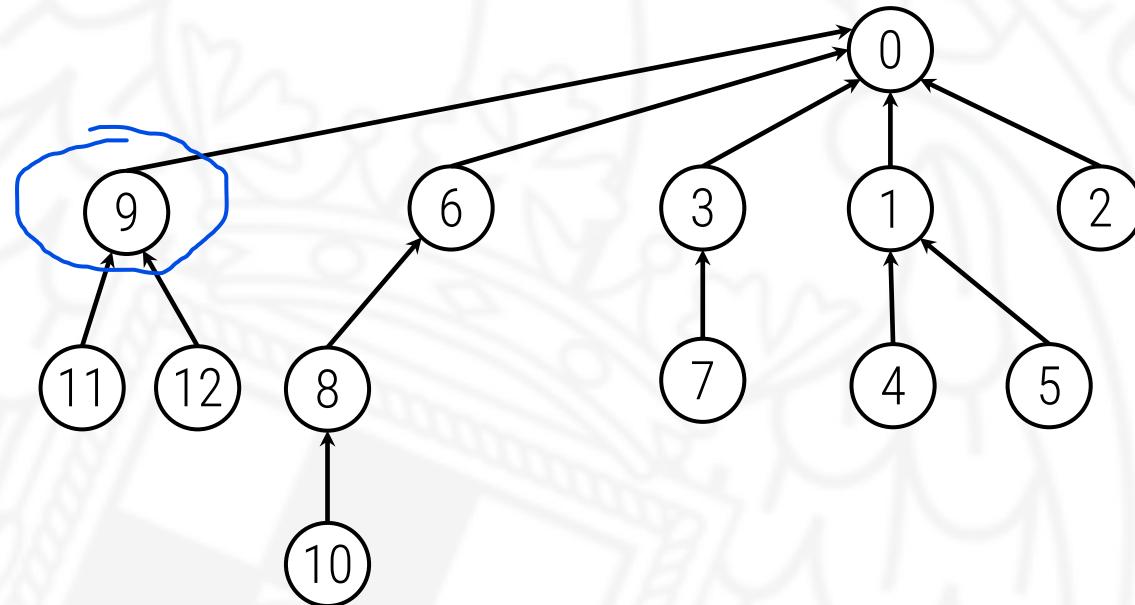
Unión por tamaños y compresión de caminos

- Después de buscar la raíz del árbol donde se encuentra x, cambiar su padre para que sea esa raíz.



Unión por tamaños y compresión de caminos

- Después de buscar la raíz del árbol donde se encuentra x , cambiar su padre para que sea esa raíz.



Unión por tamaños y compresión de caminos, análisis

- Comenzando por una partición unitaria de N elementos, cualquier secuencia de M llamadas a **unir** y **buscar** tiene un coste en $O(N + M \lg^* N)$.

$$\lg^* N = \begin{cases} 0 & \text{si } N \leq 1 \\ 1 + \lg^*(\log_2 N) & \text{si } N > 1 \end{cases}$$

Log * es una función que crece muy muy lentamente.

N	$\lg^* N$
1	0
2	1
4	2
16	3
65536	4
2^{65536}	5

No es constante pero casi.

Resumen de costes

Implementación	complejidad en el caso peor
búsqueda rápida	$N M$
unión rápida	$N M$
unión rápida por tamaños	$N + M \log N$
unión rápida con compresión de caminos	$N + M \log N$
unión rápida por tamaños y con compresión de caminos	$N + M \lg^* N$

M llamadas a `unir` y `buscar` sobre una partición con N elementos

Implementación



ConjuntosDisjuntos.h

```
class ConjuntosDisjuntos {  
protected:  
    int ncjtos; // número de conjuntos disjuntos  
    mutable std::vector<int> p; // enlace al padre  
    std::vector<int> tam; // tamaño de los árboles  
  
public:  
    // partición unitaria de N elementos  
    ConjuntosDisjuntos(int N) : ncjtos(N), p(N), tam(N, 1) {  
        for (int i = 0; i < N; ++i)  
            p[i] = i; // La raíz de cada árbol es el mismo, porque no hay ninguno relacionado.  
    }  
}
```

Luego veremos por qué está declarado como mutable.

Para cada nodo indica cual es su parente.

tamaño de todos es 1.

Implementación



ConjuntosDisjuntos.h

```
void unir(int a, int b) {
    int i = buscar(a); {
    int j = buscar(b); } Buscamos sus raíces.
    if (i == j) return; Si coinciden están en el mismo conjunto. No se hace nada más-
    if (tam[i] >= tam[j]) { // i es la raíz del árbol más grande
        tam[i] += tam[j]; p[j] = i; se comparan tamaños de las raíces.
    } else { i es el padre de j
        tam[j] += tam[i]; p[i] = j;
    } j es el padre de i
    --ncjtos; habrá un conjunto menos.
}
```

Implementación



ConjuntosDisjuntos.h

```
int buscar(int a) const {
    if (p.at(a) == a) // es una raíz
        return a;
    else
        return p[a] = buscar(p[a]);
```

Buscamos para el padre hasta llegar a la raíz.

}

Cambiamos el parente de a lo devuelto a la llamada recursiva, es por eso que p es mutable.

Implementación



ConjuntosDisjuntos.h

```
bool unidos(int a, int b) const {
    return buscar(a) == buscar(b);
}

int cardinal(int a) const {
    return tam[buscar(a)];
}

int num_cjtos() const {
    return ncjtos;
}

};
```

log * N