

CONJUNTOS DISJUNTOS



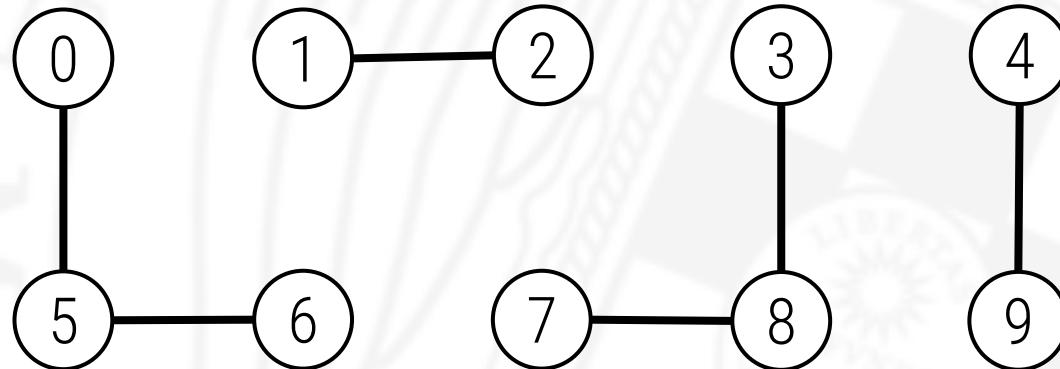
UNIVERSIDAD
COMPLUTENSE
MADRID

ALBERTO VERDEJO

Relación de equivalencia

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

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



Partición:

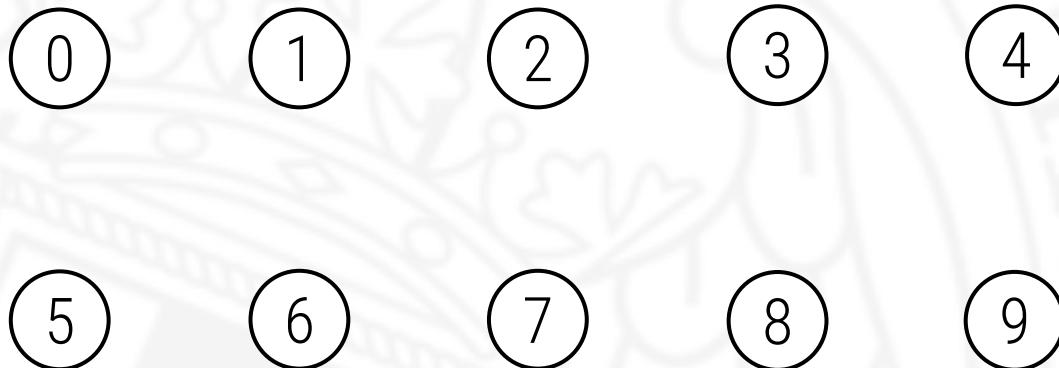
{ 0, 5, 6 }

{ 1, 2 }

{ 3, 7, 8 }

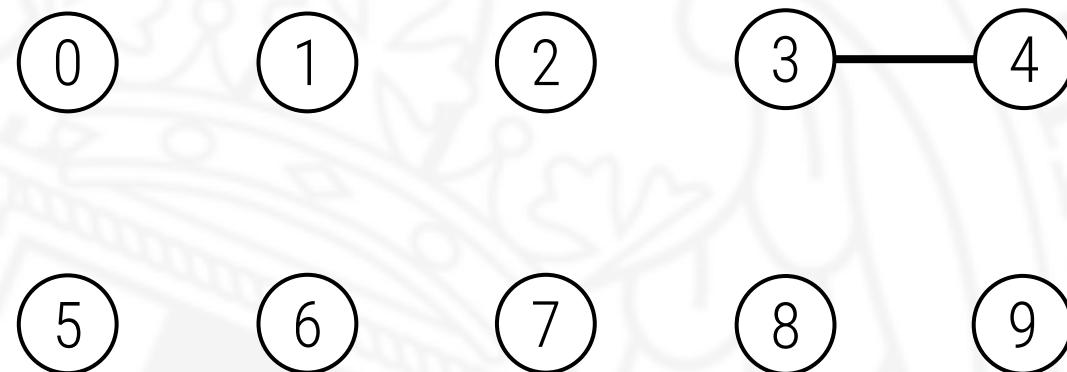
{ 4, 9 }

Relación de equivalencia dinámica



Relación de equivalencia dinámica

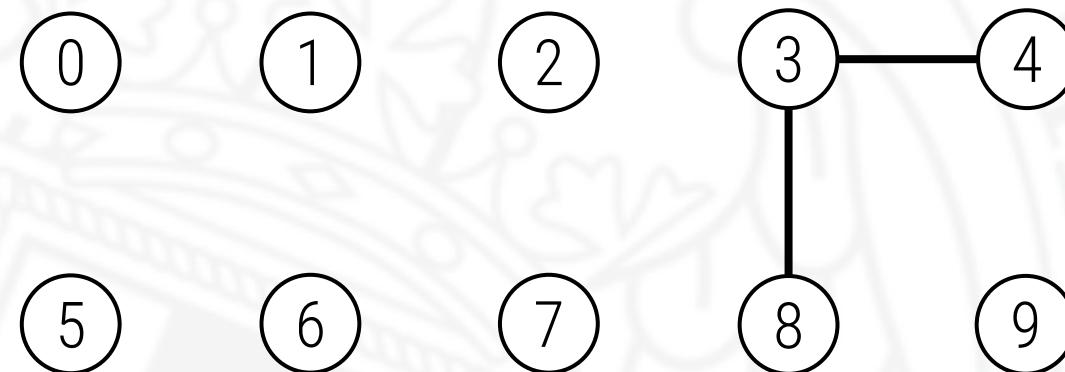
$4 R 3$



Relación de equivalencia dinámica

$4 R 3$

$3 R 8$

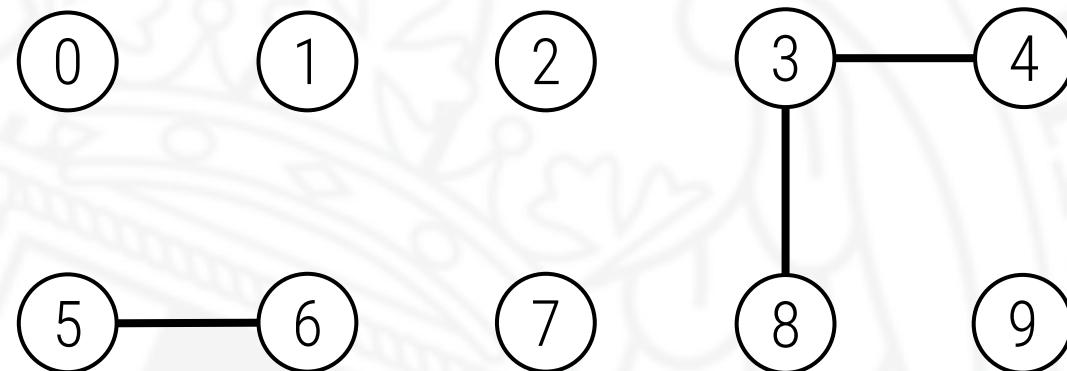


Relación de equivalencia dinámica

$4 R 3$

$3 R 8$

$6 R 5$



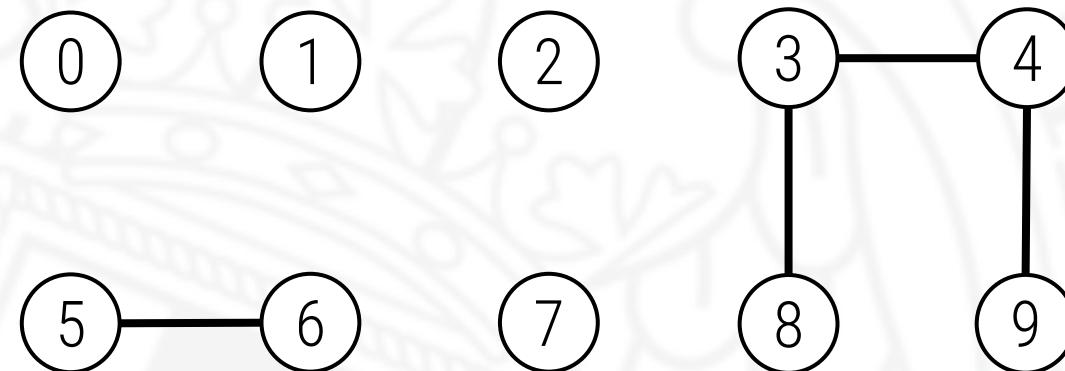
Relación de equivalencia dinámica

$4 R 3$

$3 R 8$

$6 R 5$

$9 R 4$



Relación de equivalencia dinámica

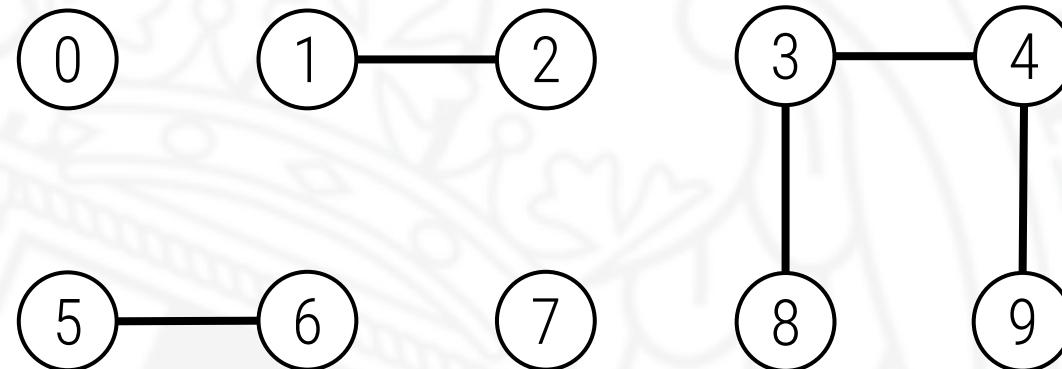
$4 R 3$

$3 R 8$

$6 R 5$

$9 R 4$

$2 R 1$



Relación de equivalencia dinámica

$4 R 3$

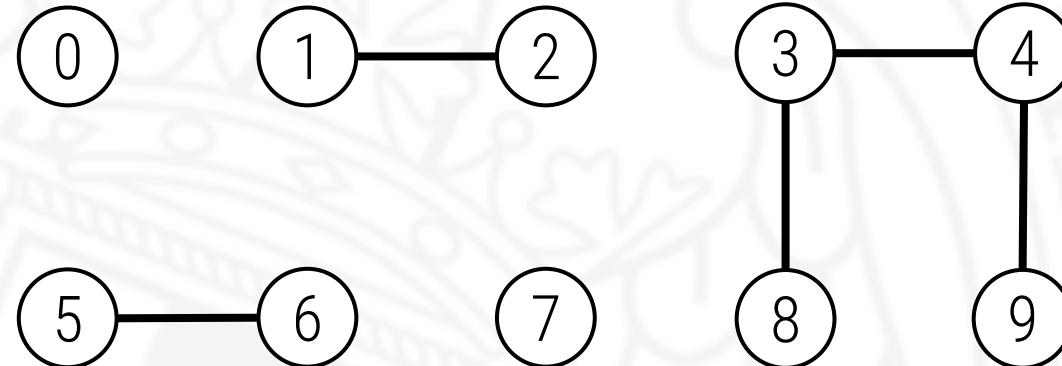
$3 R 8$

$6 R 5$

$9 R 4$

$2 R 1$

¿ $8 R 9$? ✓



Relación de equivalencia dinámica

$4 R 3$

$3 R 8$

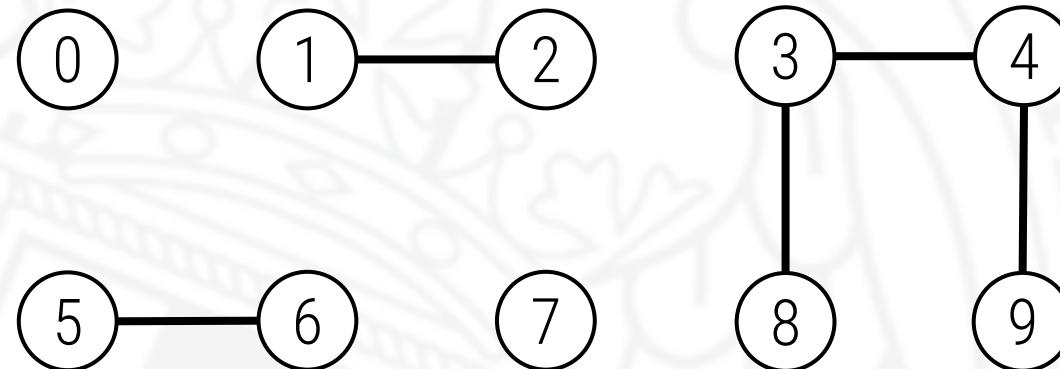
$6 R 5$

$9 R 4$

$2 R 1$

¿ $8 R 9$? ✓

¿ $5 R 7$? ✗



Relación de equivalencia dinámica

$4 R 3$

$3 R 8$

$6 R 5$

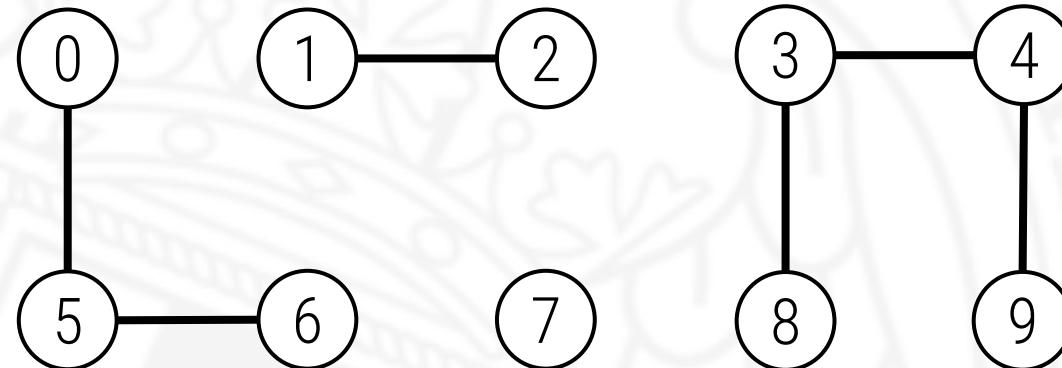
$9 R 4$

$2 R 1$

¿ $8 R 9$? ✓

¿ $5 R 7$? ✗

$5 R 0$



Relación de equivalencia dinámica

$4 R 3$

$3 R 8$

$6 R 5$

$9 R 4$

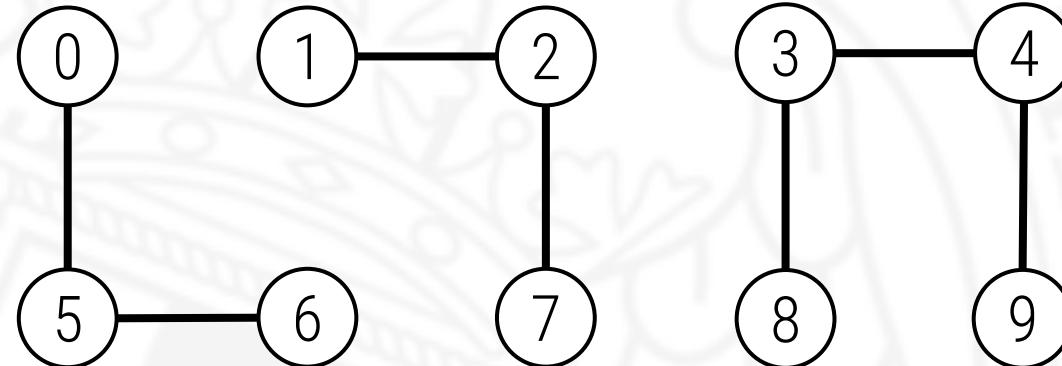
$2 R 1$

¿ $8 R 9$? ✓

¿ $5 R 7$? ✗

$5 R 0$

$7 R 2$



Relación de equivalencia dinámica

$4 R 3$

$3 R 8$

$6 R 5$

$9 R 4$

$2 R 1$

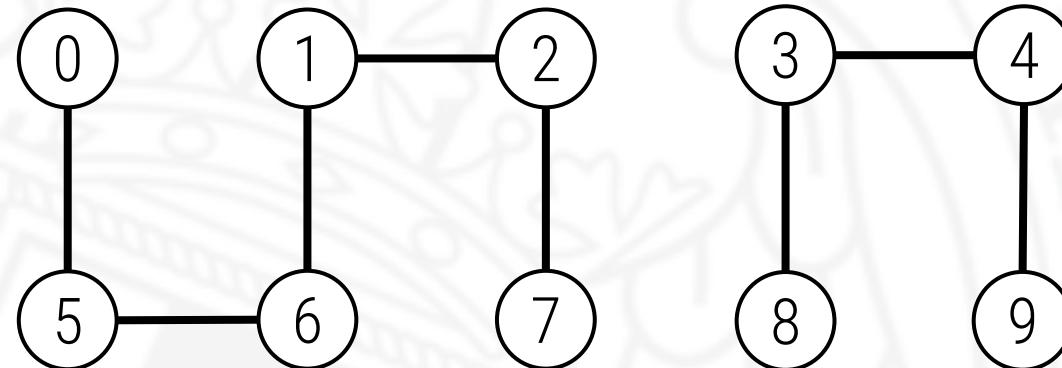
¿ $8 R 9$? ✓

¿ $5 R 7$? ✗

$5 R 0$

$7 R 2$

$6 R 1$



Relación de equivalencia dinámica

$4 R 3$

$3 R 8$

$6 R 5$

$9 R 4$

$2 R 1$

¿ $8 R 9$? ✓

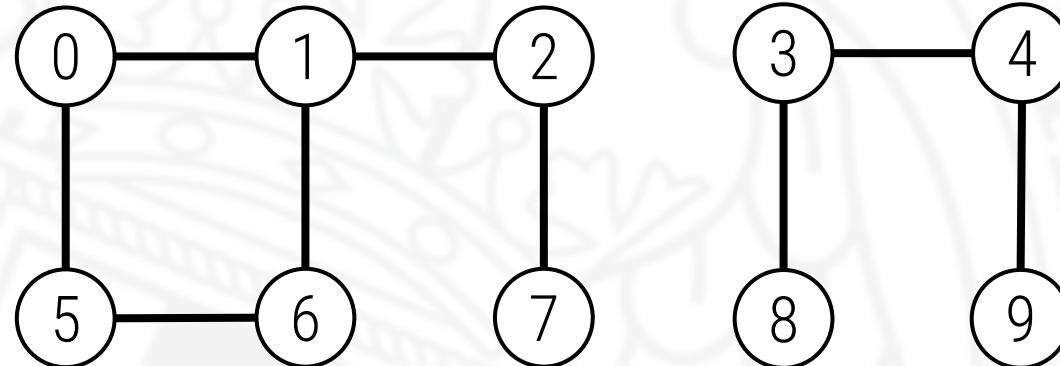
¿ $5 R 7$? ✗

$5 R 0$

$7 R 2$

$6 R 1$

$1 R 0$



Relación de equivalencia dinámica

$4 R 3$

$3 R 8$

$6 R 5$

$9 R 4$

$2 R 1$

¿ $8 R 9$? ✓

¿ $5 R 7$? ✗

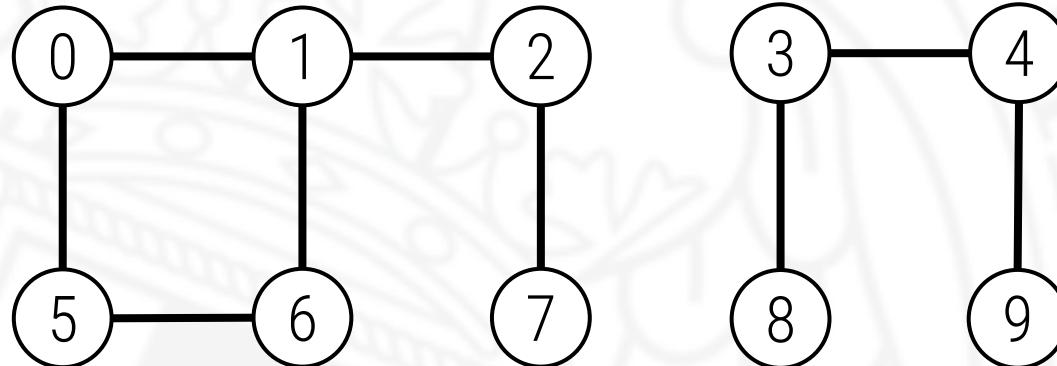
$5 R 0$

$7 R 2$

$6 R 1$

$1 R 0$

¿ $5 R 7$? ✓



TAD de conjuntos disjuntos

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`
- ▶ consultar si dos elementos pertenecen al mismo conjunto,
`bool unidos(int a, int b) const`
- ▶ consultar el cardinal de un conjunto, `int cardinal(int a) const`
- ▶ 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

{ 0, 5, 6 }

{ 1, 2, 7 }

{ 3, 4, 8, 9 }

Possible implementación: búsqueda rápida

`unir(6, 1)`

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

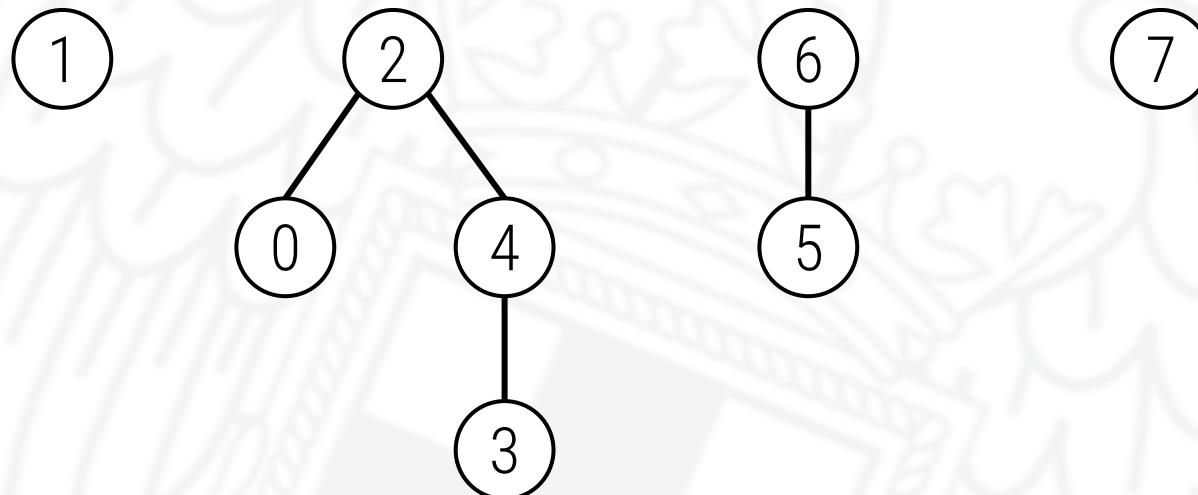
Possible implementación: búsqueda rápida

`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
	↑					↑	↑			

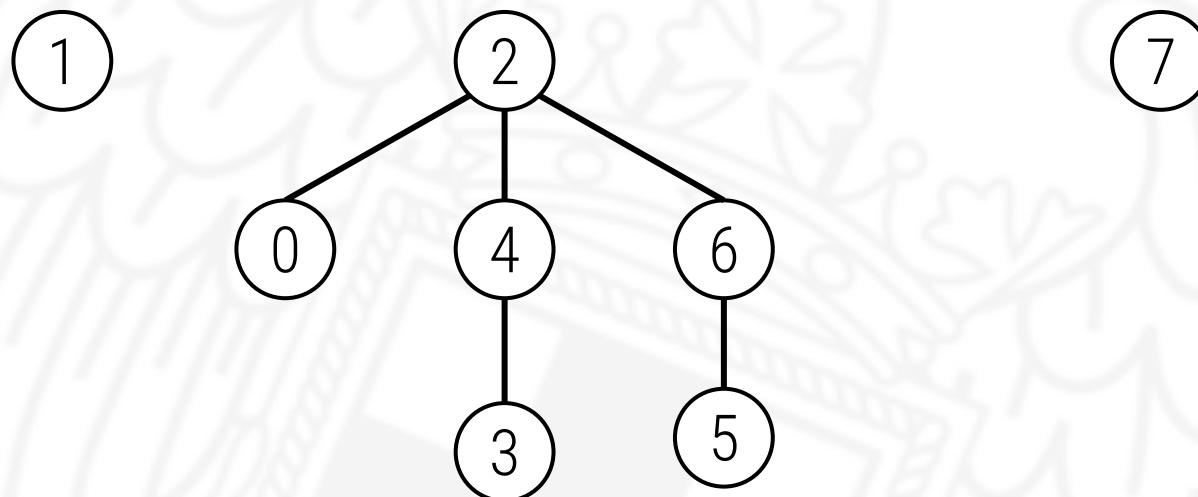
Unión rápida

- ▶ Cada conjunto se representa mediante un árbol.



Unión rápida

- ▶ Cada conjunto se representa mediante un árbol.

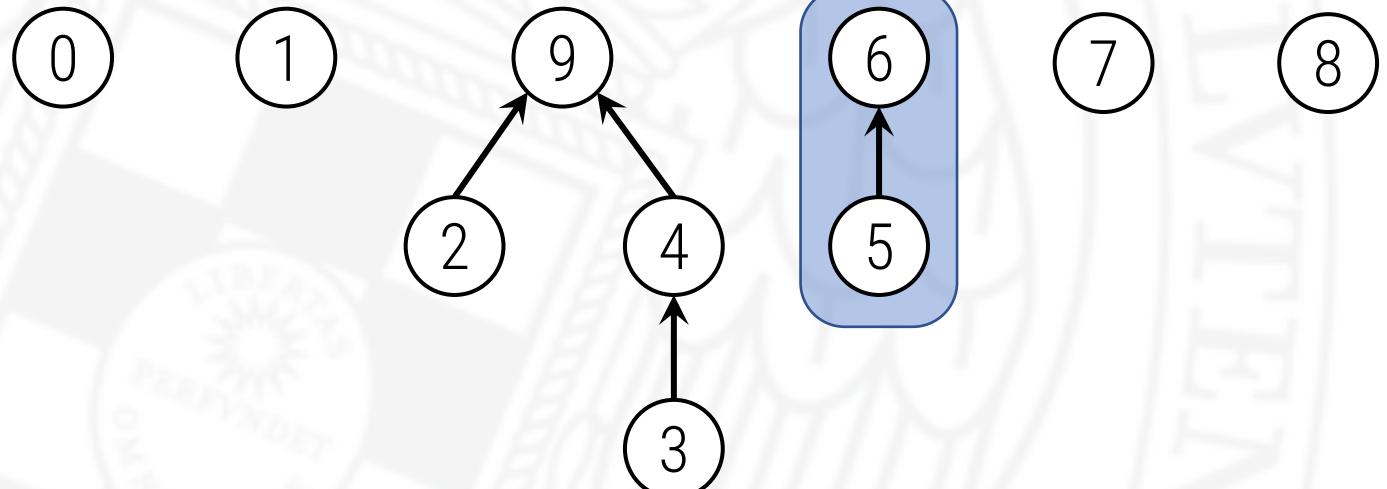


Unión rápida

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

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

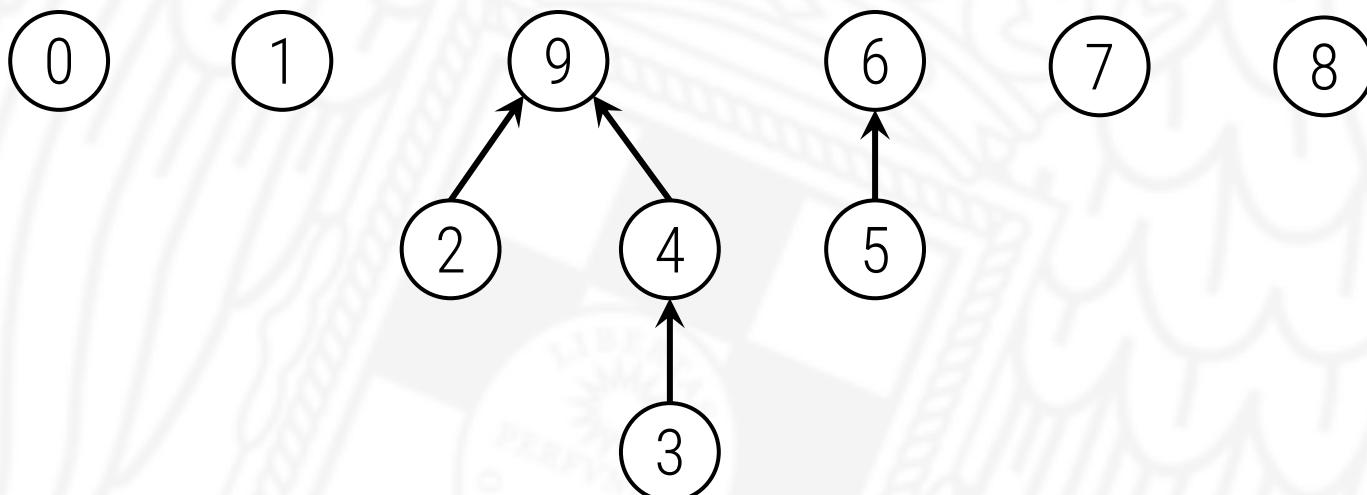
{ 0 }
{ 1 }
{ 2, 3, 4, 9 }
{ 5, 6 }
{ 7, 8 }



Unión rápida

`unir(3, 5)`

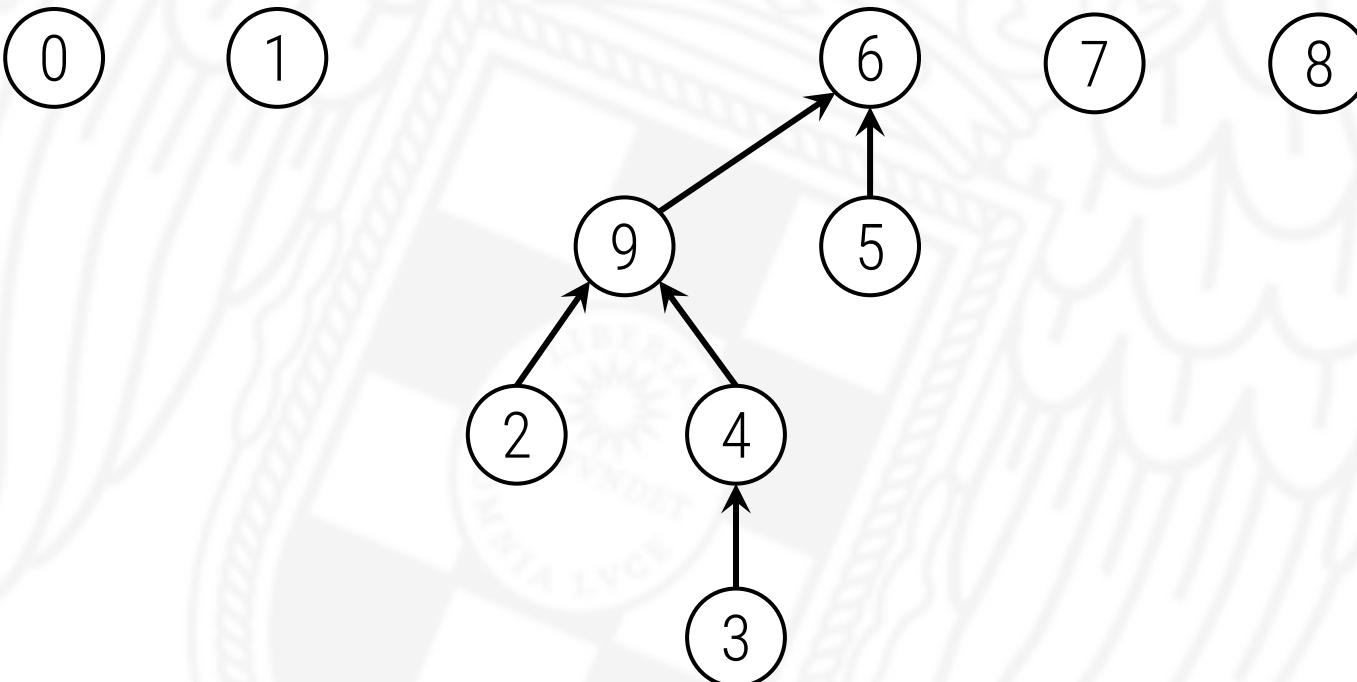
	0	1	2	3	4	5	6	7	8	9
<code>p[]</code>	0	1	9	4	9	6	6	7	8	9



Unión rápida

`unir(3, 5)`

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



Unión rápida

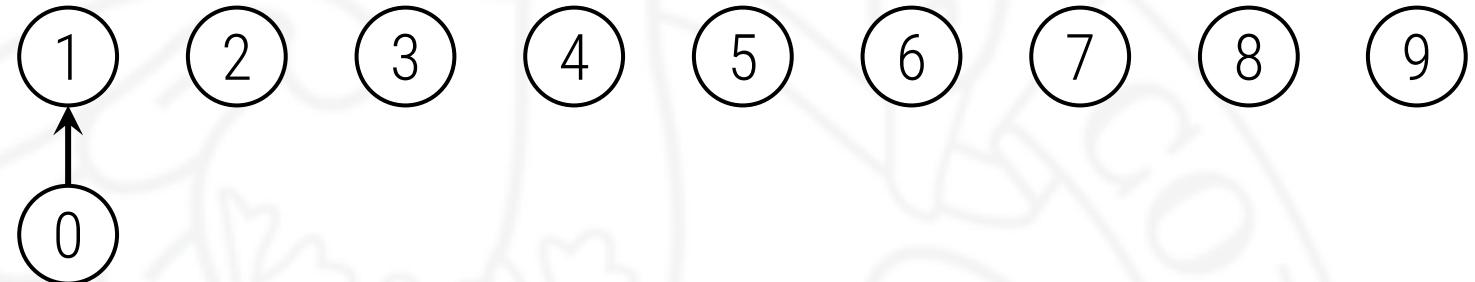
unir(0, 1)

- 0
- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9

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)`



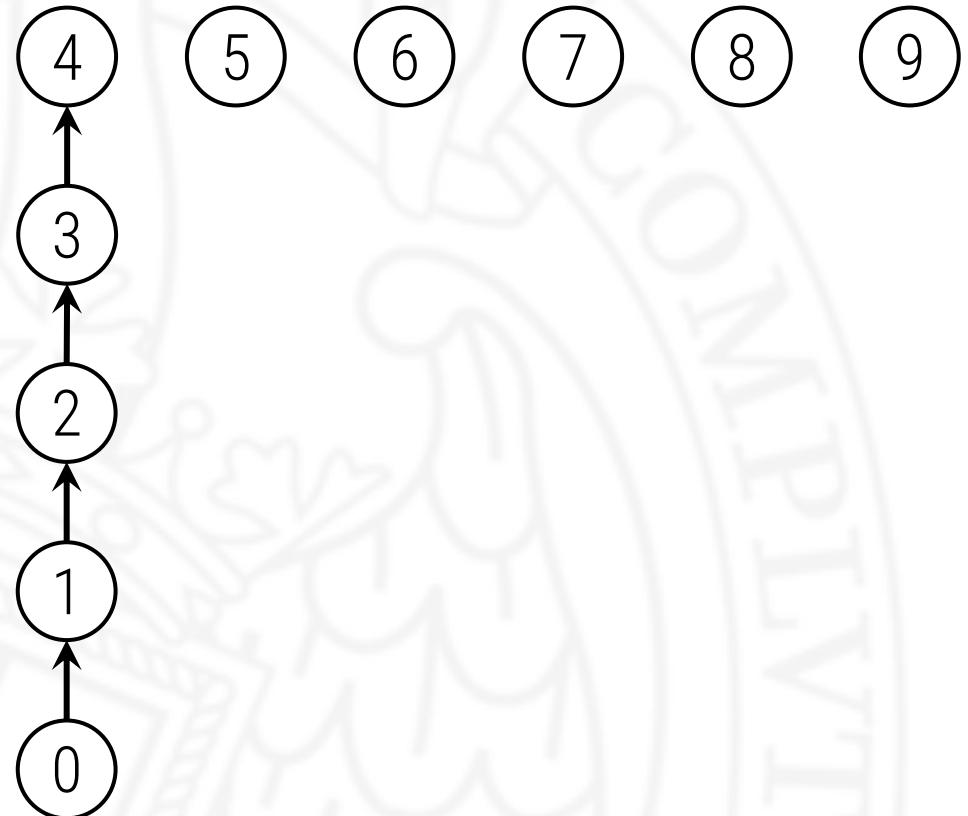
Unión rápida, ¿búsqueda lenta?

`unir(0, 1)`

`unir(0, 2)`

`unir(0, 3)`

`unir(0, 4)`

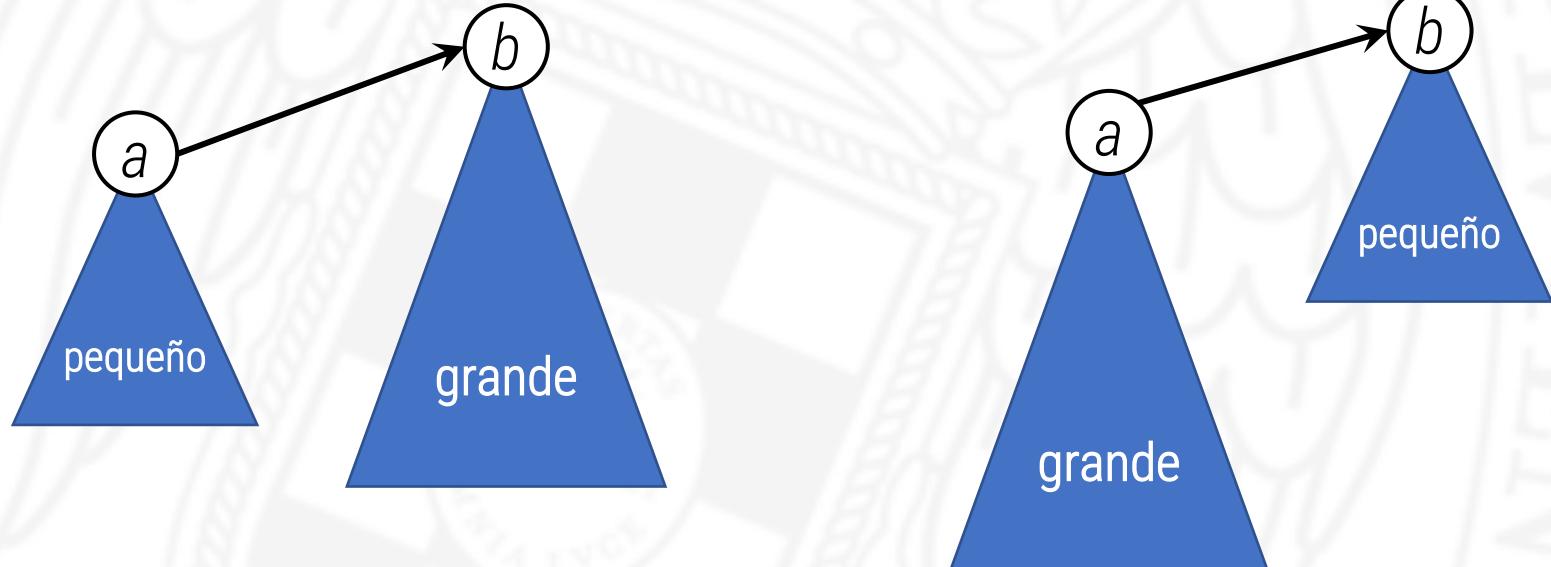


Unión por tamaños

- ▶ 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

`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

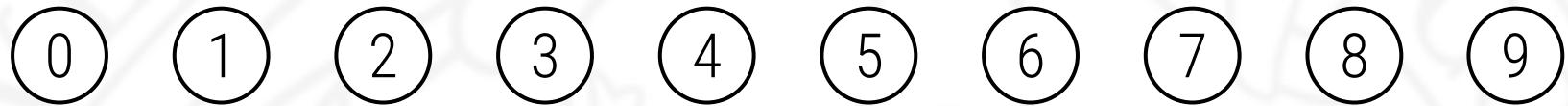
Unión rápida por tamaños

`unir(a, b)`



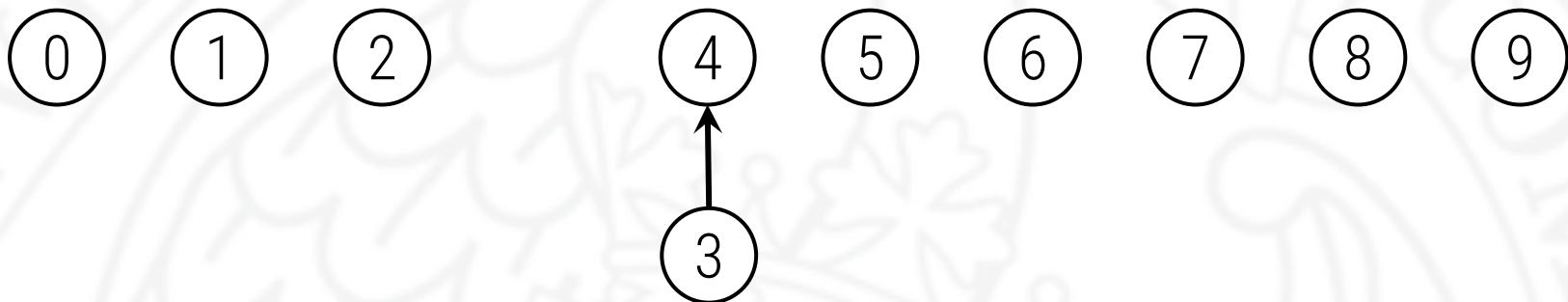
Unión por tamaños

`unir(4, 3)`



Unión por tamaños

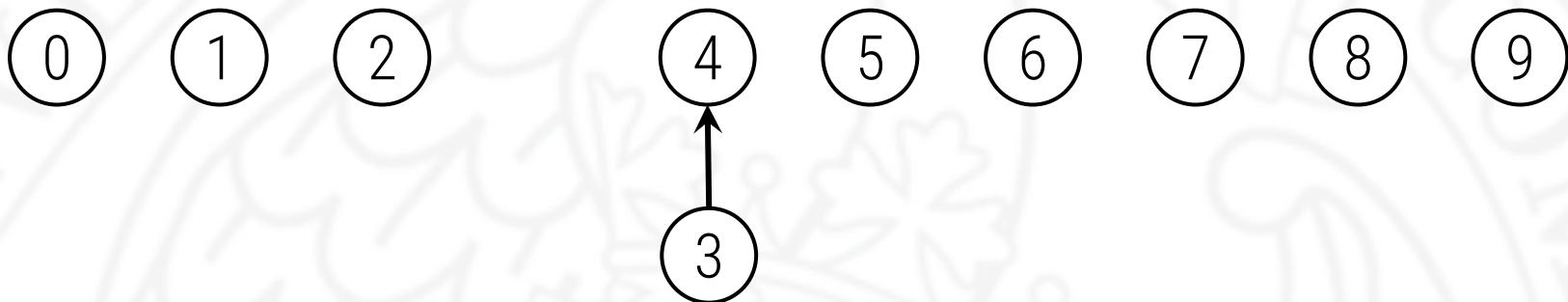
`unir(4, 3)`



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

Unión por tamaños

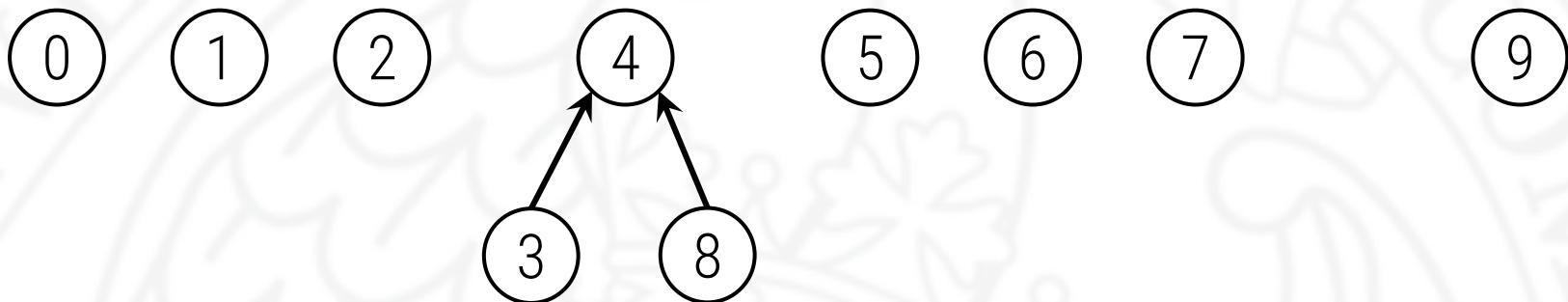
`unir(3, 8)`



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

Unión por tamaños

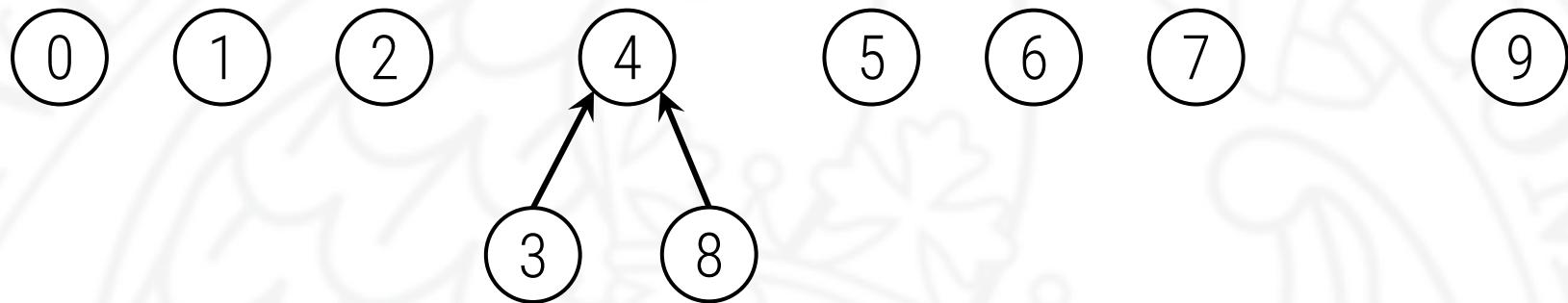
`unir(3, 8)`



	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

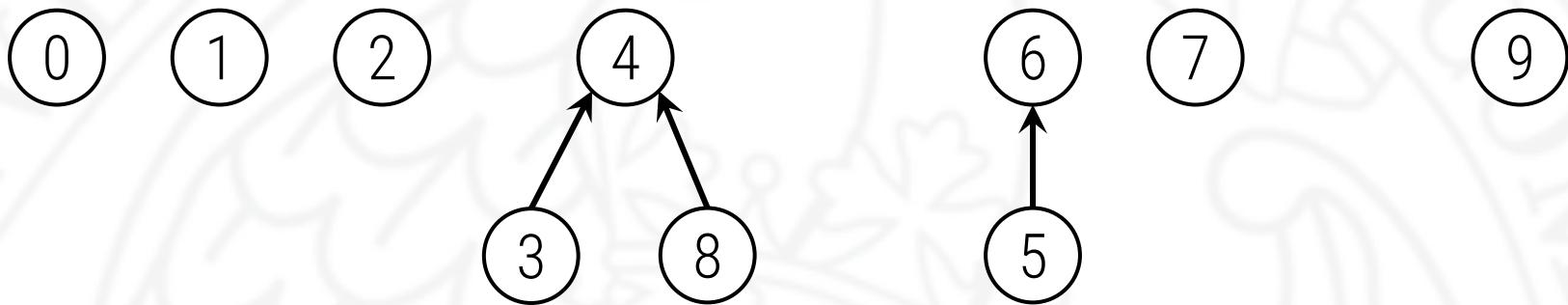
`unir(6, 5)`



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

Unión por tamaños

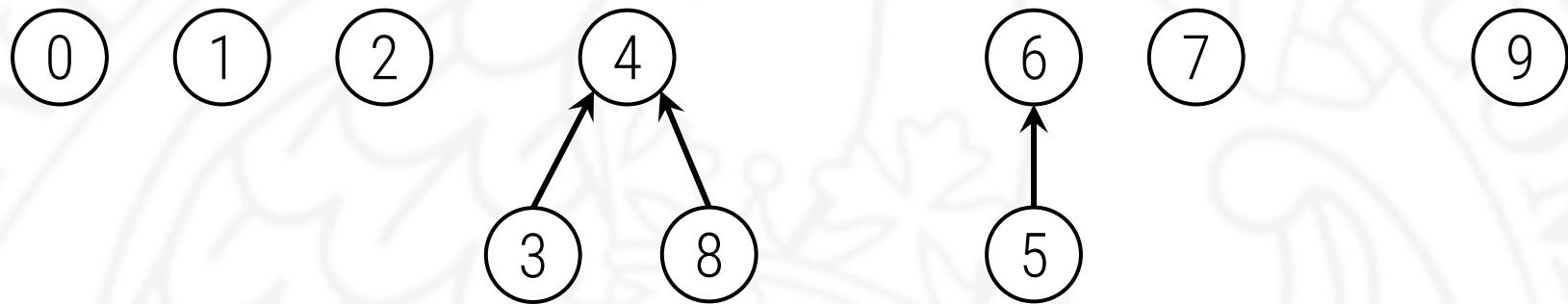
`unir(6, 5)`



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

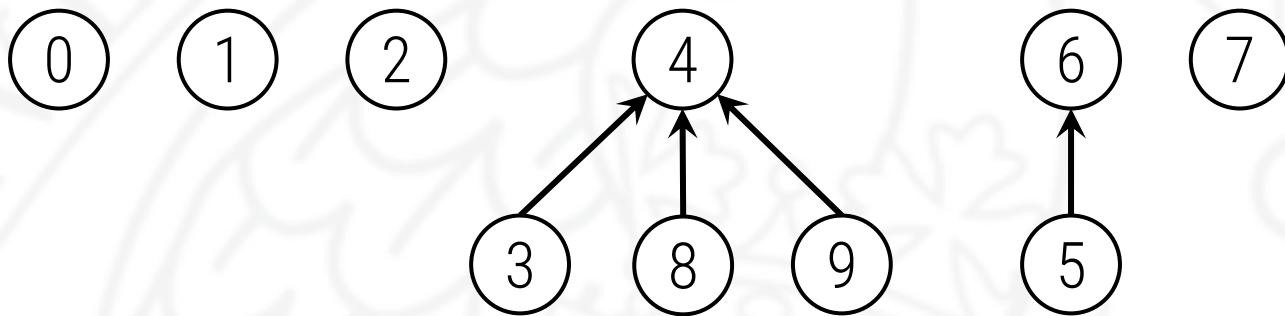
`unir(9, 4)`



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

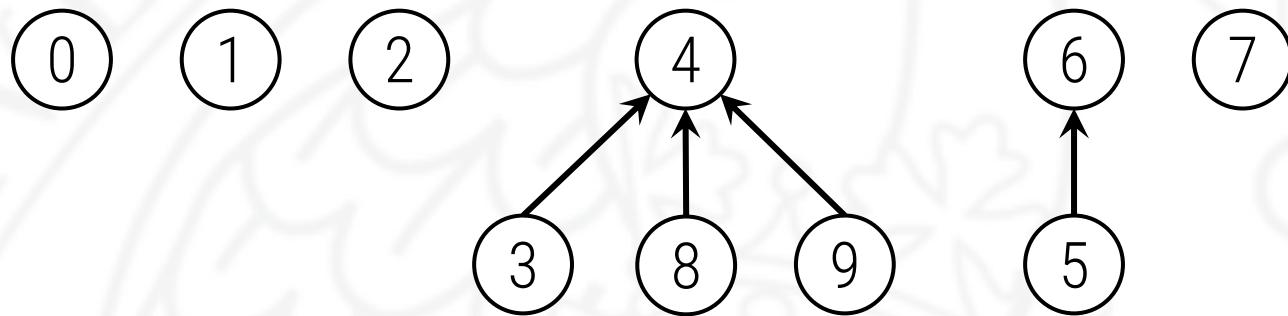
`unir(9, 4)`



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

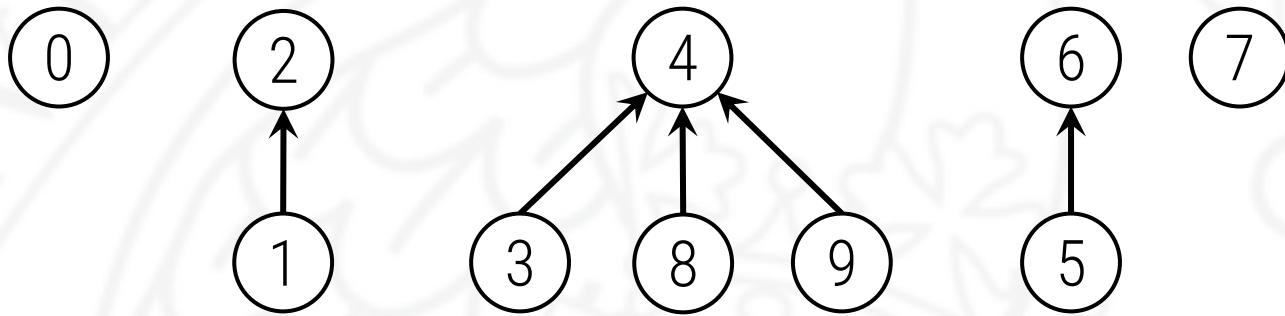
`unir(2, 1)`



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

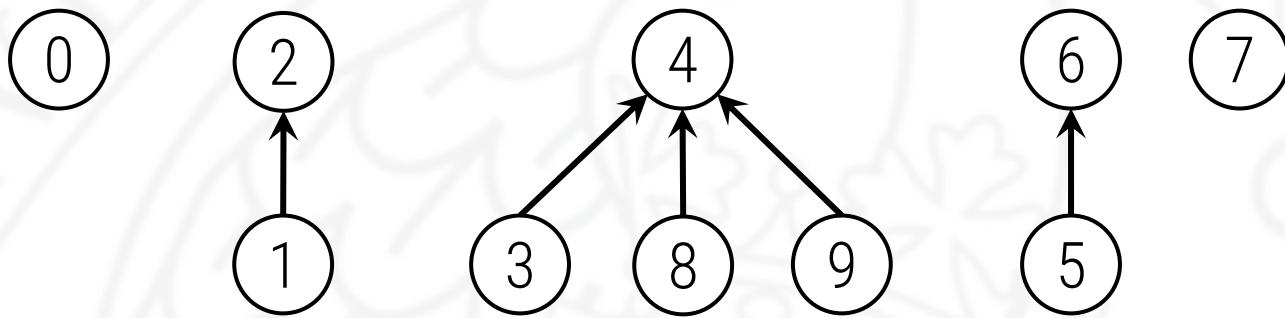
`unir(2, 1)`



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

Unión por tamaños

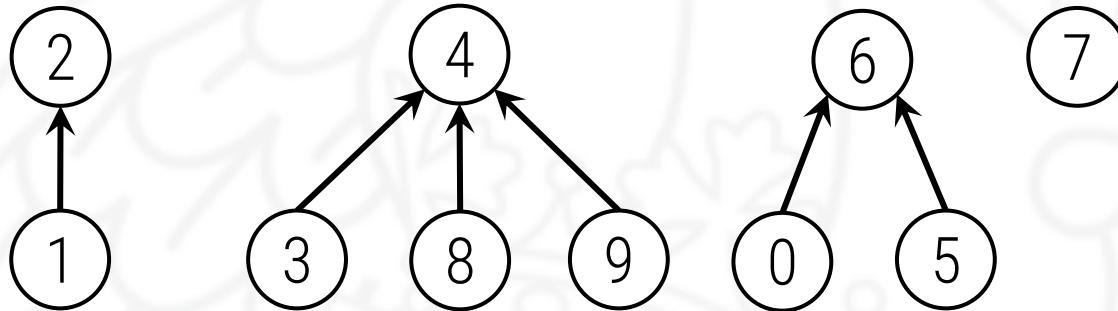
`unir(5, 0)`



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

Unión por tamaños

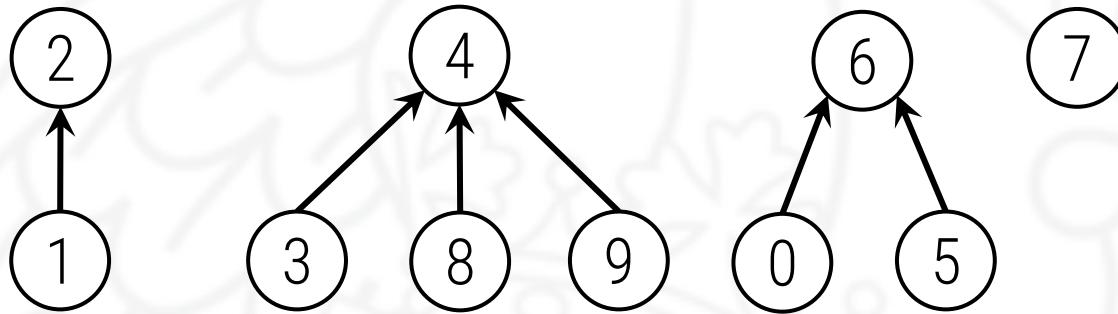
`unir(5, 0)`



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

Unión por tamaños

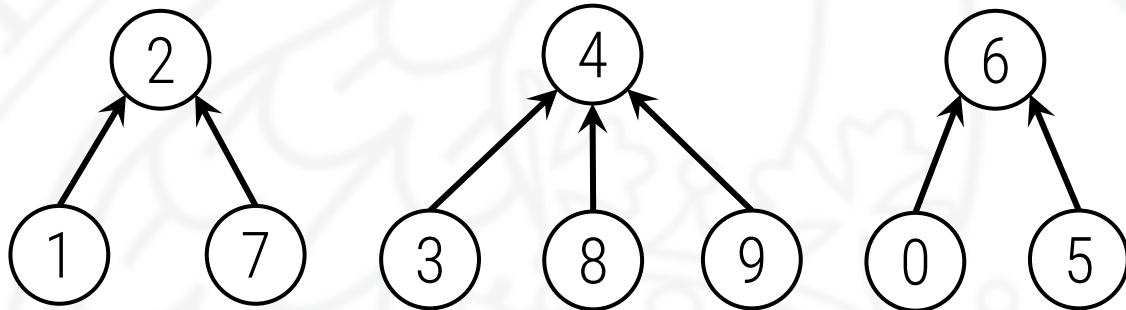
`unir(7, 2)`



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

Unión por tamaños

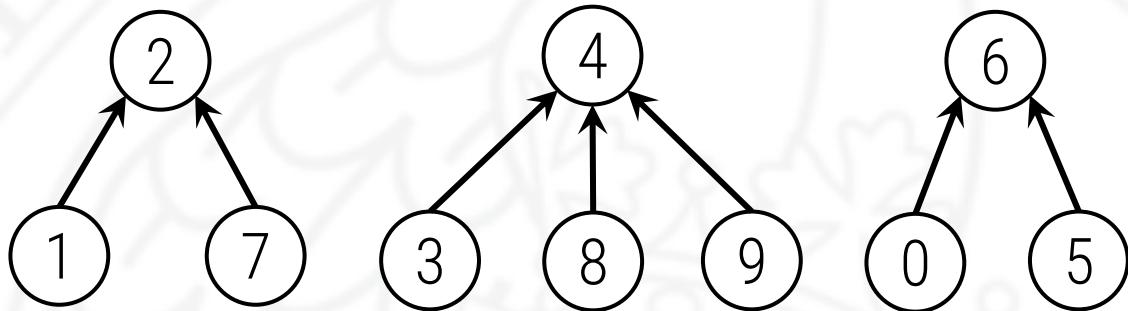
`unir(7, 2)`



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

Unión por tamaños

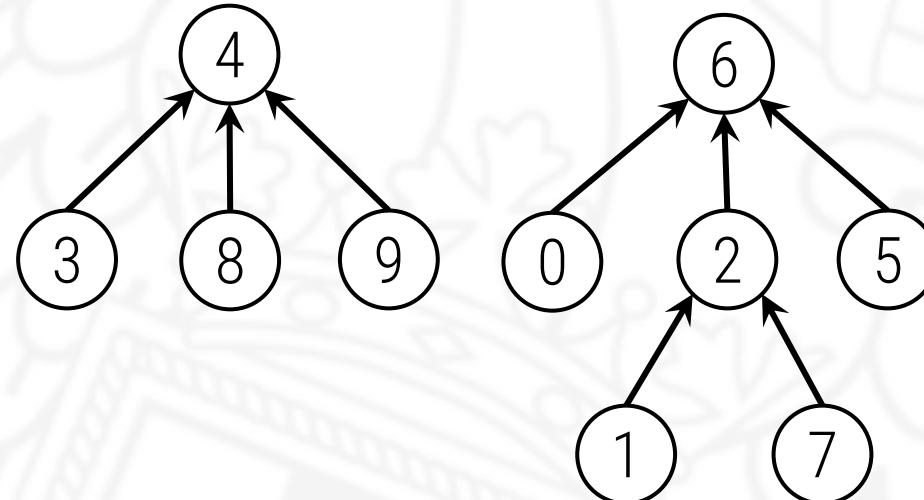
`unir(6, 1)`



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

Unión por tamaños

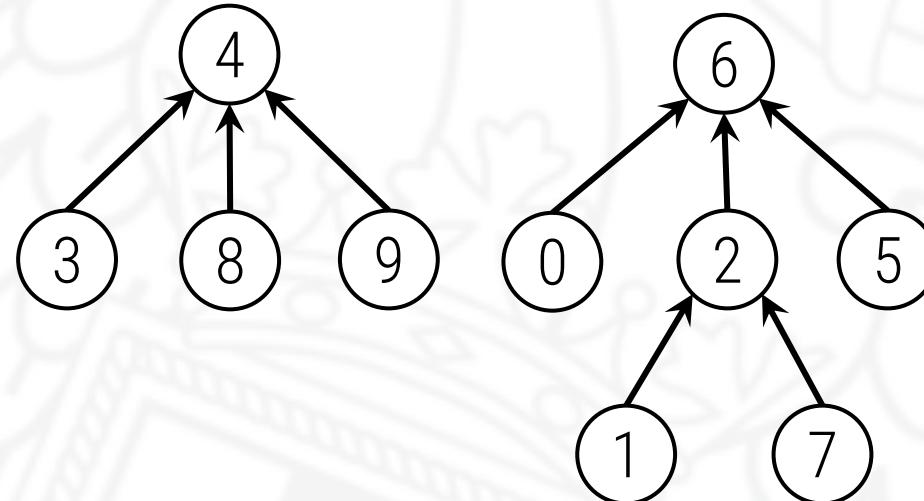
`unir(6, 1)`



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

Unión por tamaños

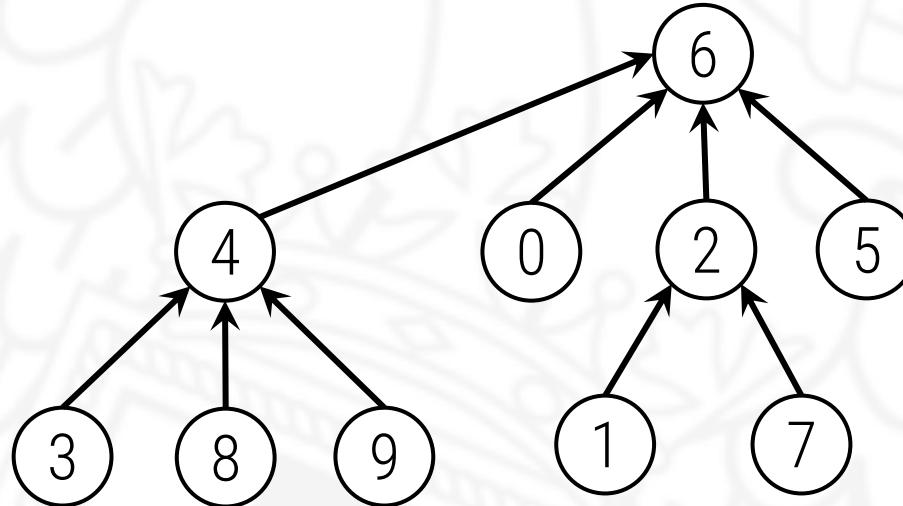
`unir(7, 3)`



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

Unión por tamaños

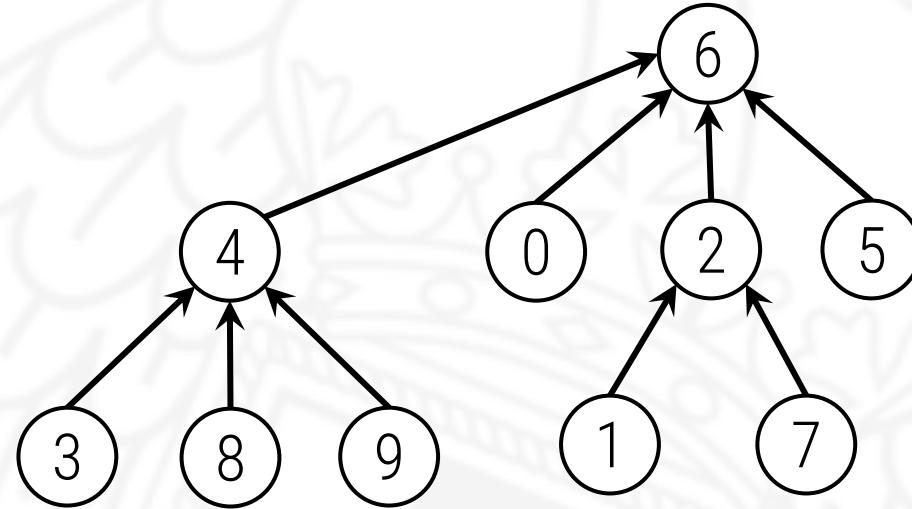
`unir(7, 3)`



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

Unión por tamaños

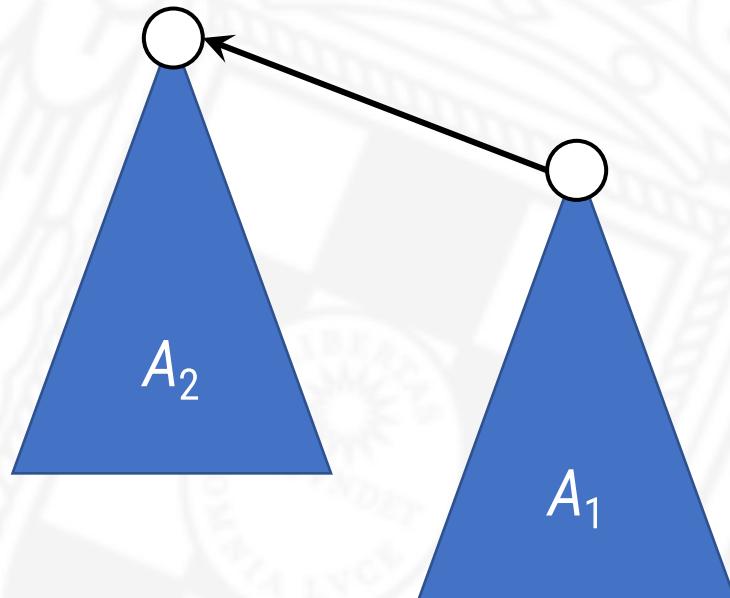
`unir(7, 3)`



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

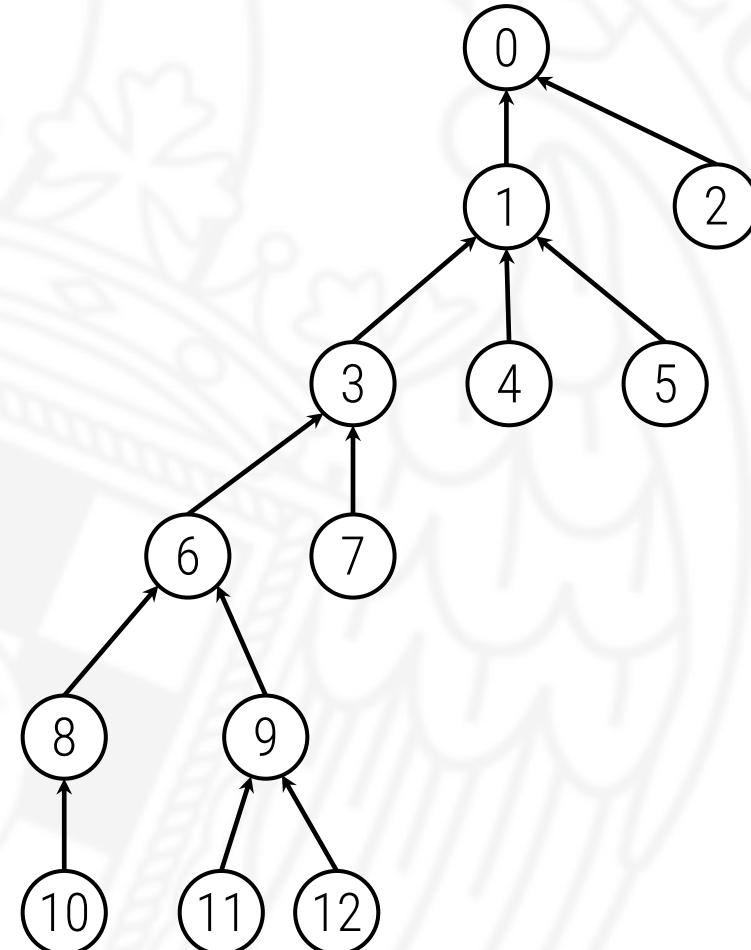
Unión por tamaños, análisis

- ▶ 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 $O(\log N)$.



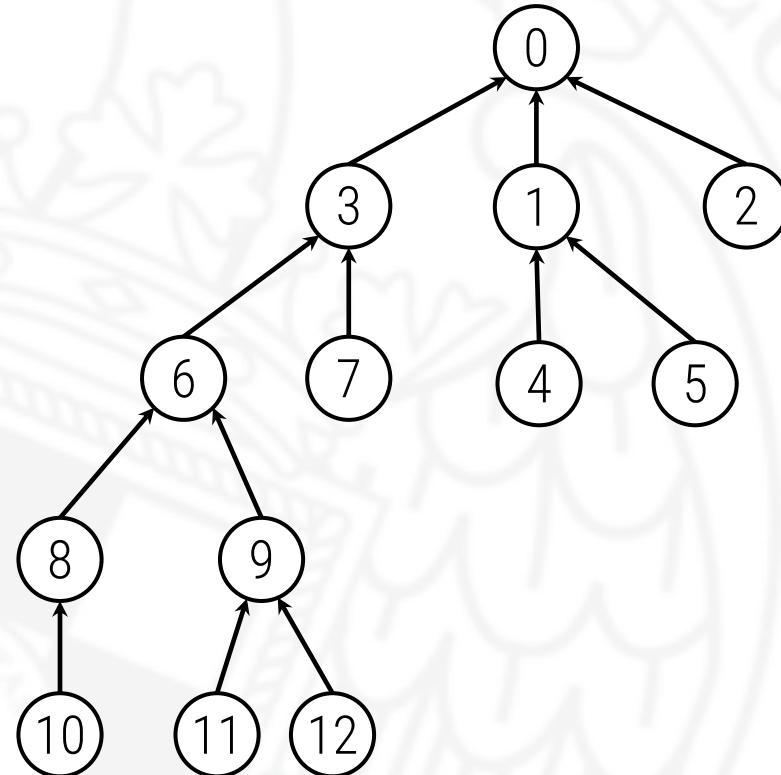
Unión por tamaños y compresión de caminos

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



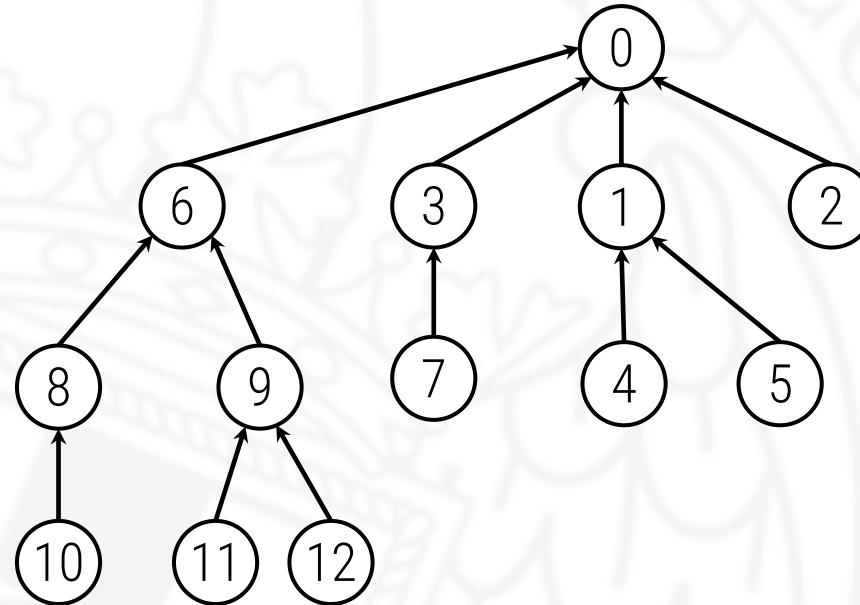
Unión por tamaños y compresión de caminos

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



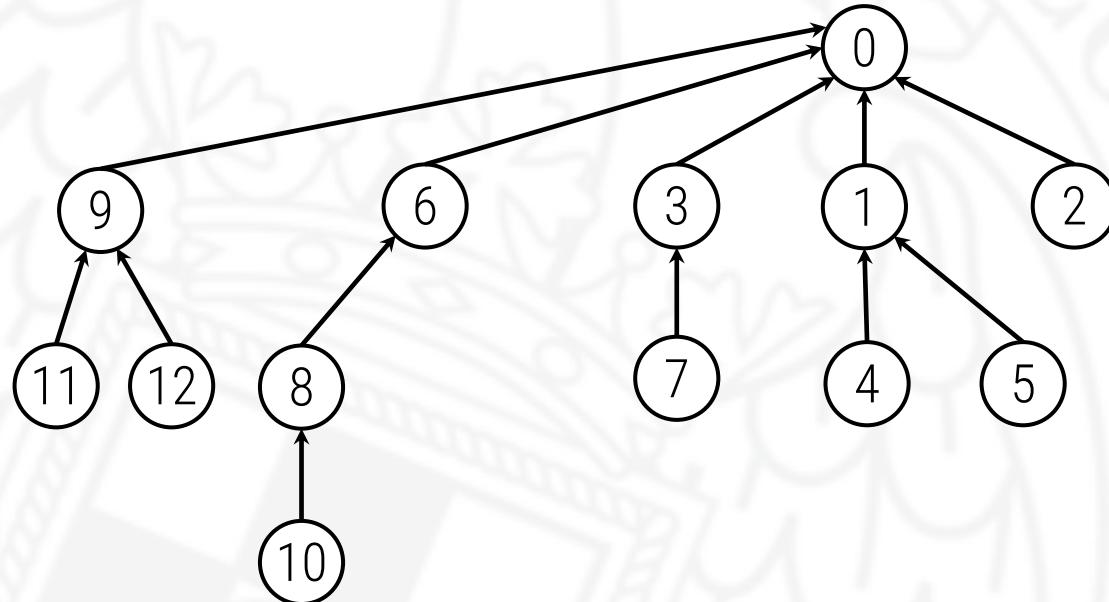
Unión por tamaños y compresión de caminos

- ▶ 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

- ▶ 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 $O(N + M \lg^* N)$.

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

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

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;  
    }  
}
```

Implementación



ConjuntosDisjuntos.h

```
void unir(int a, int b) {
    int i = buscar(a);
    int j = buscar(b);
    if (i == j) return;
    if (tam[i] >= tam[j]) { // i es la raíz del árbol más grande
        tam[i] += tam[j]; p[j] = i;
    } else {
        tam[j] += tam[i]; p[i] = j;
    }
    --ncjtos;
}
```

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]);
}
```

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;
}

};
```