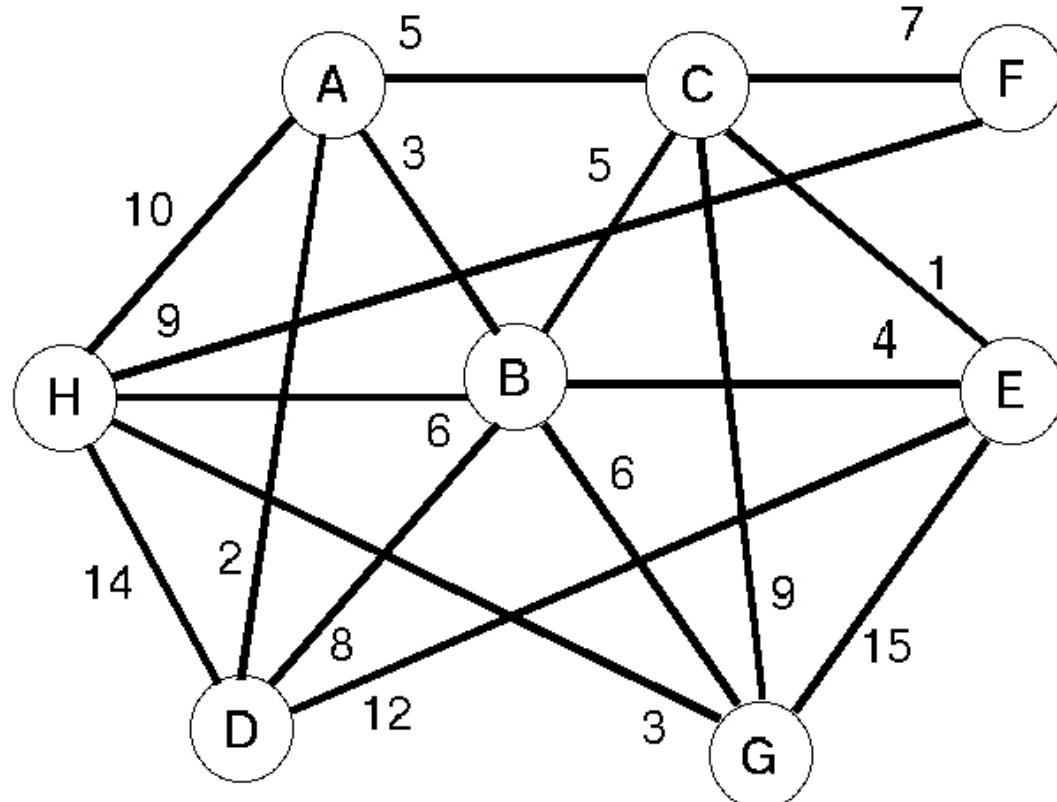


# Algoritmo de Dijkstra, parte 1

# El problema del camino más corto

Consiste en encontrar un camino entre dos nodos de tal manera que la suma de los pesos de las aristas que lo constituyen sea mínima.



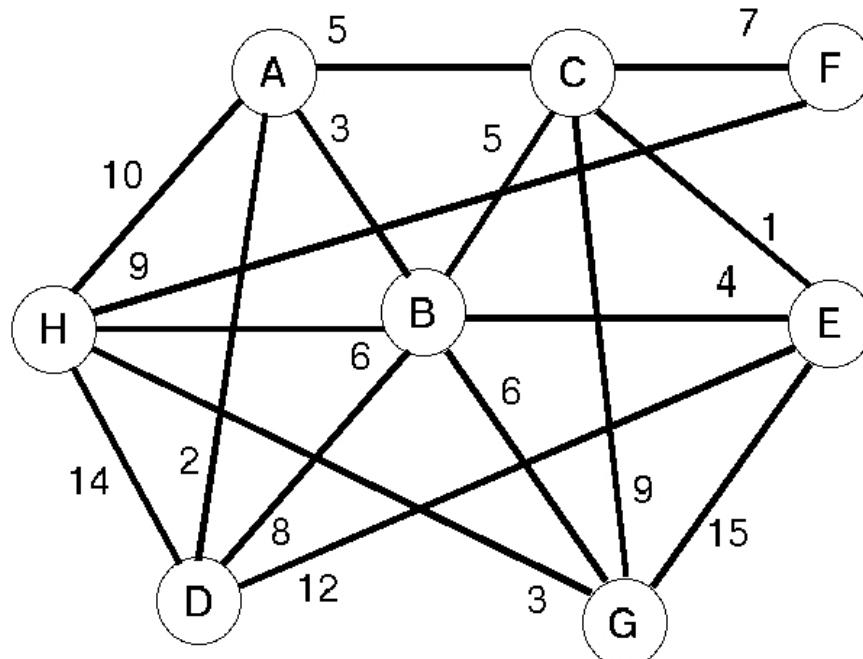
# El problema del camino más corto

**Entrada:** Un grafo dirigido  $G = (V, E)$  con aristas  $l_e$  de peso o longitud no negativas y un nodo inicial  $s$ .

**Salida:** Para todo nodo  $k$ ,  $L(k) =$  distancia del camino más corto entre  $s$  y  $k$ .

Si  $s = B$ :

$L(A)$	3
$L(B)$	0
$L(C)$	5
$L(D)$	5
$L(E)$	4
$L(F)$	12
$L(G)$	6
$L(H)$	6



# El problema del camino más corto

¿Por qué no simplemente usar BFS para calcular distancias entre nodos?

Porque dicha aproximación supone que  $l_e = 1$  para todas las aristas y por tanto determina el número de saltos, más que la longitud del camino entre nodos.

Por esta razón es necesario utilizar otro algoritmo, siendo uno de los más populares el de Dijkstra

([http://es.wikipedia.org/wiki/Edsger\\_Dijkstra](http://es.wikipedia.org/wiki/Edsger_Dijkstra))

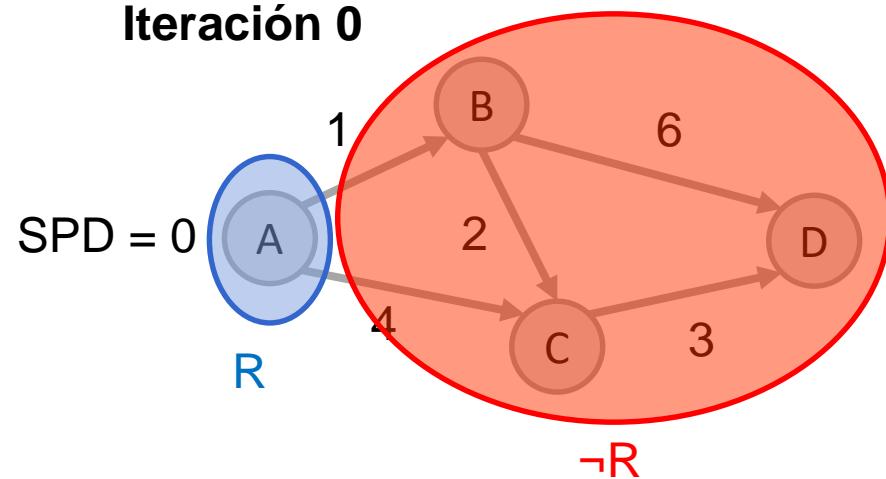
# Algoritmo de Dijkstra

```
function Dijkstra(grafo G, nodo s):
    R = {s} //Nodos ya revisados
    s.SPD = 0
    while R ≠ V: #Mientras hayan nodos por revisar
        entre todas las aristas (u, v) con u ∈ R y v ∈ ¬R:
            escoger (u, v) que minimize u.SPD + (u, v).le
        R.add(v)
        v.SPD = u.SPD + (u, v).le
```

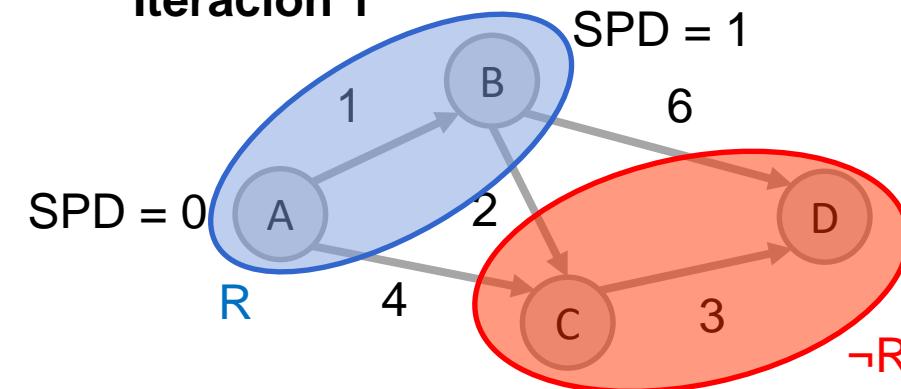
**Criterio greedy**

# Algoritmo de Dijkstra

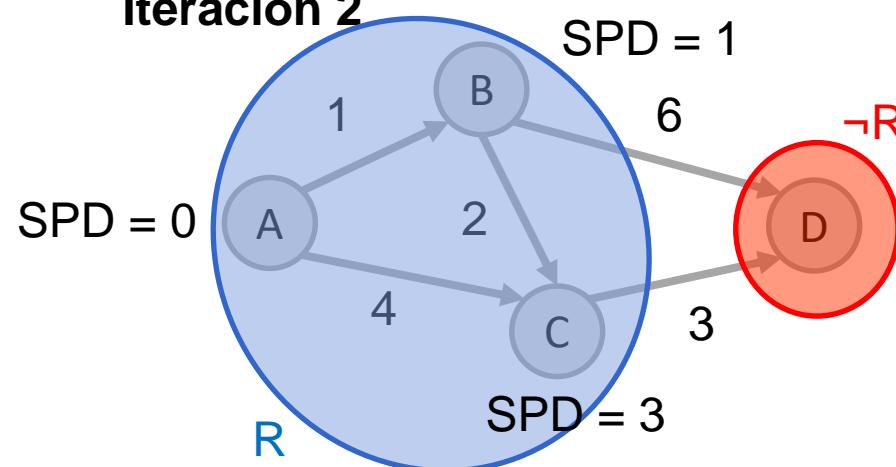
Iteración 0



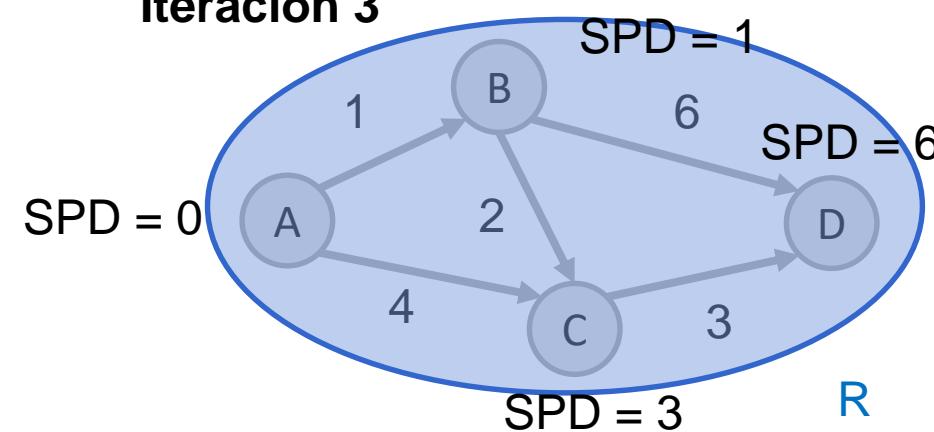
Iteración 1



Iteración 2



Iteración 3



# Algoritmo de Dijkstra, parte 2

# Algoritmo de Dijkstra

¿Cuál es la complejidad del algoritmo?

- El proceso  $\text{while } R \neq V$  realiza exactamente  $N-1$  iteraciones
- Dentro de ese proceso se revisan todas las aristas buscando aquellas que comiencen en  $R$  y terminen en  $\neg R$ , entre las que cumplan se busca un mínimo según el criterio de Dijkstra
- La eficiencia resultante por tanto es  $O(N^*M)$

¿Se puede hacer mejor?

Si, pero no cambiando el algoritmo como tal si no empleando una estructura de datos apropiada

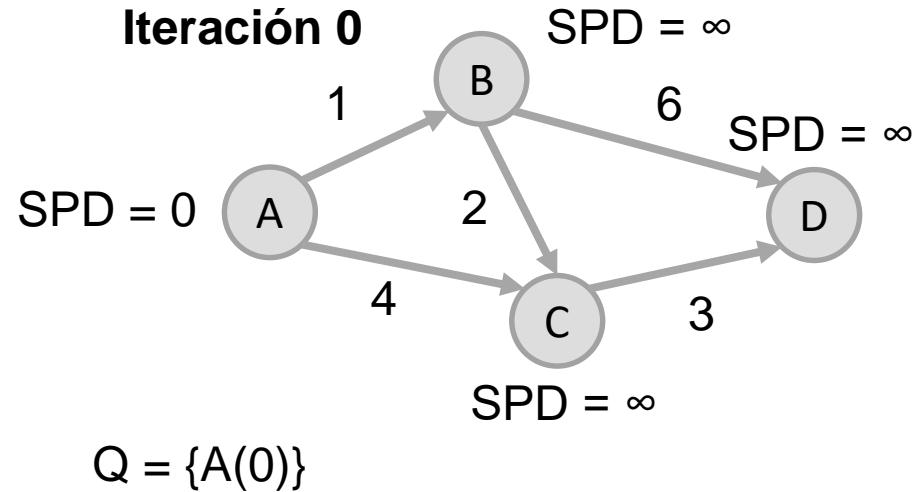
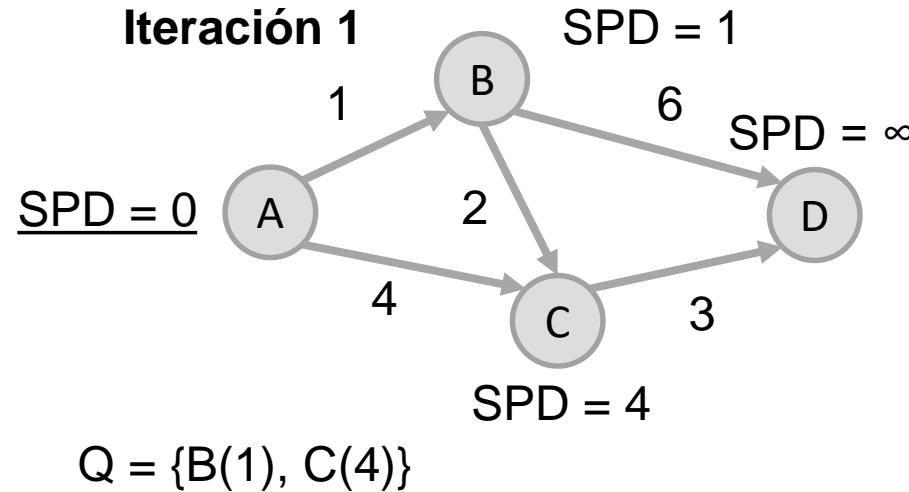
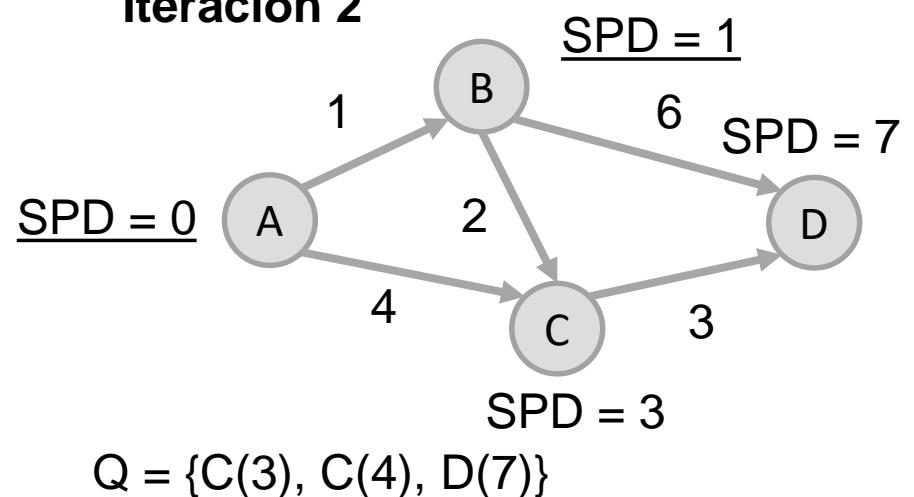
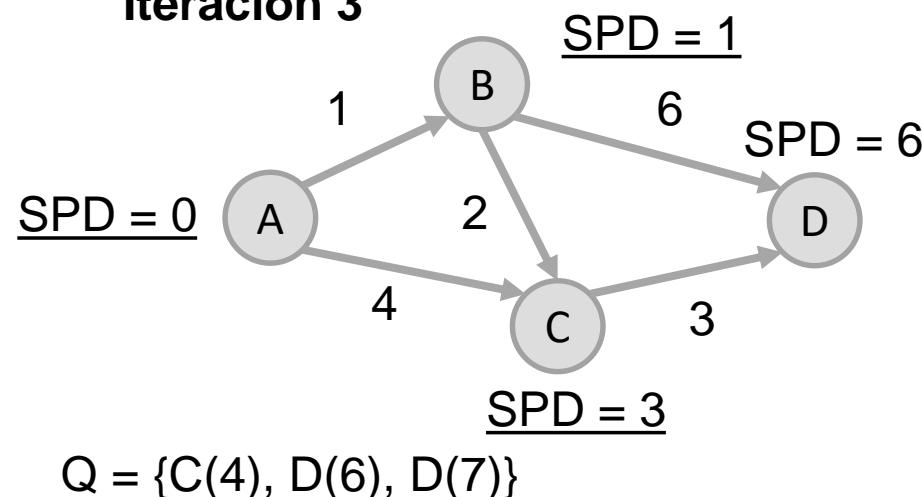
```
function Dijkstra(grafo G, nodo s):  
    R = {s} //Nodos ya revisados  
    s.SPD = 0  
    while R ≠ V: #Mientras hayan nodos por revisar  
        entre todas las aristas (u, v) con u ∈ R y v ∈ ¬R:  
            escoger (u, v) que minimize u.SPD + (u, v).le  
        R.add(v)  
        v.SPD = u.SPD + (u, v).le
```

# Algoritmo de Dijkstra

Se puede usar una cola con prioridad (implementada mediante un heap) para que nos ayude a escoger el siguiente nodo en cada iteración

```
function Dijkstra(grafo G, nodo s):
    for u ∈ V, u ≠ s: u.SPD = INF
    s.SPD = 0
    Q.add(s) //Cola con prioridad
    while Q no esté vacía:
        u = Q.pop()
        for (u, v) ∈ E:
            if u.SPD + (u, v).le < v.SPD{
                v.SPD = u.SPD + (u, v).le
                Q.add(v)
```

Para determinar la eficiencia resultante, miremos un ejemplo ...

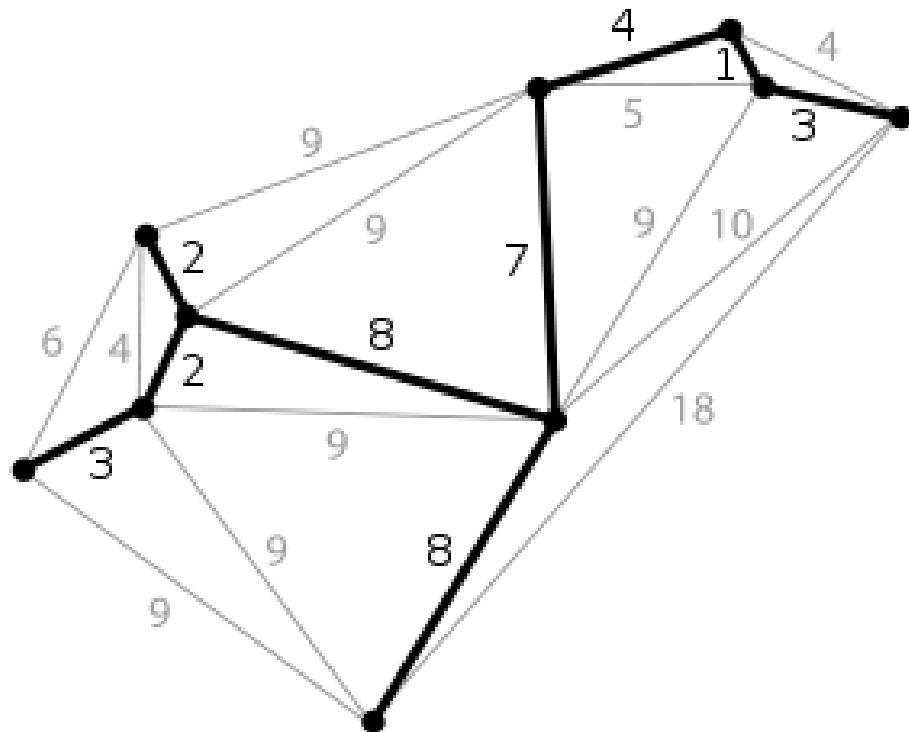
**Iteración 0****Iteración 1****Iteración 2****Iteración 3**

Complejidad resultante:  $O((M + N) * \log(N))$  y la demostración se puede encontrar en la sección 24.3 de *Introduction to Algorithms*, pag. 658

# Algoritmo de Prim

# Árboles de mínima expansión

Consiste en “conectar” todos los nodos de un grafo de la manera menos costosa posible en términos de la sumatoria de longitudes (o costos para ser mas generales) de las aristas del árbol resultante.



Fuente: [http://en.wikipedia.org/wiki/File:Minimum\\_spanning\\_tree.svg](http://en.wikipedia.org/wiki/File:Minimum_spanning_tree.svg)

# Árboles de mínima expansión

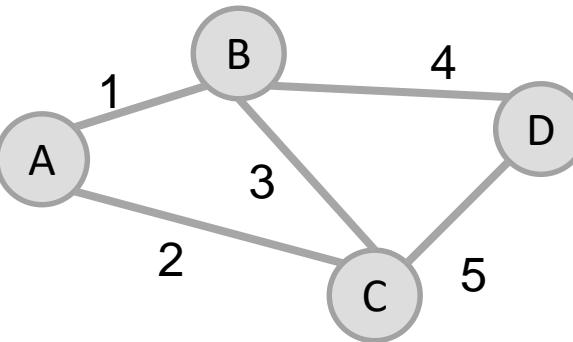
**Entrada:** Un grafo no dirigido  $G = (V, E)$  completamente conectado donde cada arista  $(u, v)$  tiene un costo  $l_e$  (incluso puede ser negativo).

**Salida:** Un árbol  $T$  de mínimo costo que “abarque” todos los nodos de  $V$ . Es decir, el sub-grafo  $(V, T)$  debe ser completamente conectado.

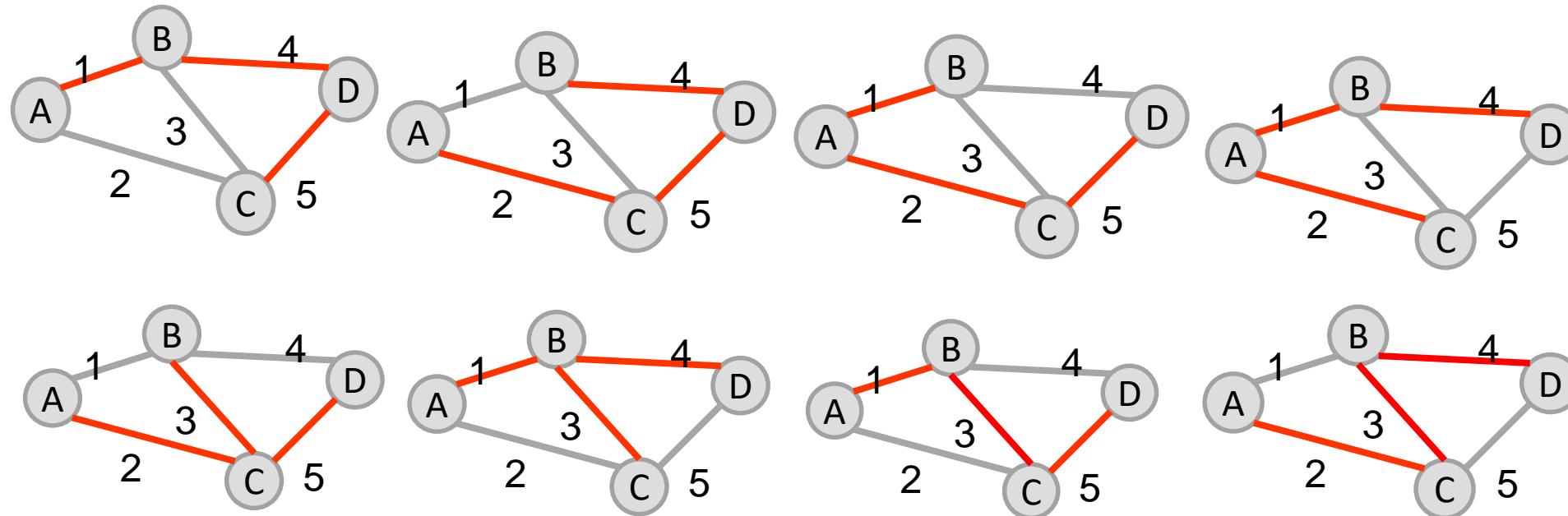
Si  $G$  no es completamente conectado, lo cual se puede verificar fácilmente usando DFS por ejemplo, el problema se puede convertir fácilmente a uno de “Bosques de mínima expansión”, que consiste en encontrar el conjunto de árboles de mínima expansión que abarcan todos los nodos.

# Árboles de mínima expansión

Ejemplo:



¿Cuántos árboles (forma de conectar los nodos sin que haya aristas redundantes) diferentes tiene este grafo?



Y en general ¿Cuántos árboles diferentes puede tener un grafo?

En un árbol conectado (todos con todos) la cantidad de árboles es tan grande como  $N^{N-2}$

([http://en.wikipedia.org/wiki/Cayley%27s\\_formula](http://en.wikipedia.org/wiki/Cayley%27s_formula))

# Algoritmo de Prim

```
function Prim(grafo G){  
    s = cualquier nodo ∈ V  
    R = {s} //Nodos ya revisados  
    T = NULL  
    while R ≠ V:  
        entre todas las aristas (u, v) con u ∈ R y v ∈ ¬R:  
            escoger (u, v) de menor costo  
        R.add(v)  
        T.add(u, v)}
```

Criterio greedy

¿Cuál es la complejidad?

Haciendo el mismo análisis que con Dijkstra llegamos a que la complejidad resultante es  $O(N * M)$ , sin duda mejor que exponencial.

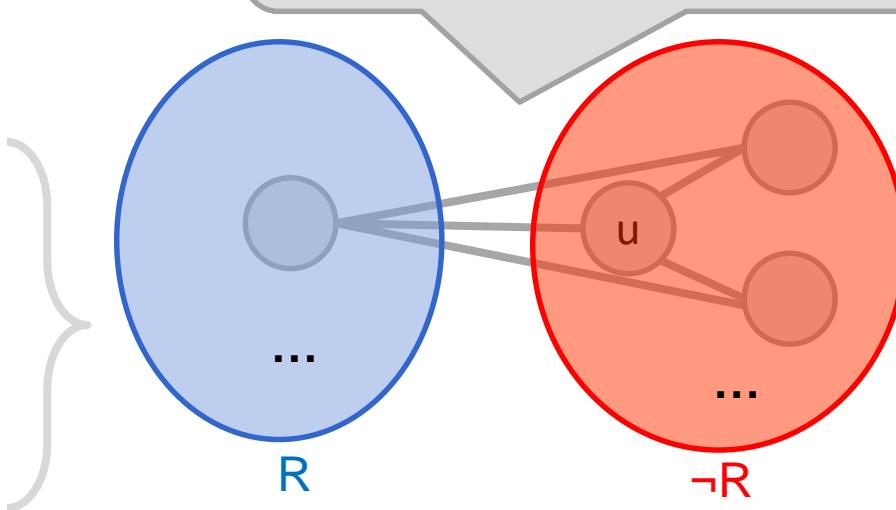
Sin embargo, igual que con Dijkstra, podemos mejorar esta eficiencia empleando una cola con prioridad implementada mediante un montículo binario.

# Algoritmo de Prim

```
function Prim(grafo G)
    s = cualquier nodo  $\in V$ 
    R = {s} //Nodos ya revisados
    T = NULL
    for  $v \in \neg R$ :
         $v.lc$  = menor costo de la arista  $(u,v)$  con  $u \in R$  ó INF si tal arista no existe
         $v.be$  =  $(u,v)$  ó NULL en los mismos casos
    Q.add( $\neg R$ ) //Cola con prioridad
    while  $R \neq V$ :
         $u = Q.pop()$ 
        if  $u \in \neg R$ :
            R.add( $u$ )
            T.add( $u.be$ )
            for  $(u, v) \in R$ :
                if  $v \in \neg R$ :
                    if  $(u,v).le < v.lc$ :
                         $v.lc = (u,v).le$ 
                         $v.be = (u,v)$ 
                        Q.add( $v$ )
```

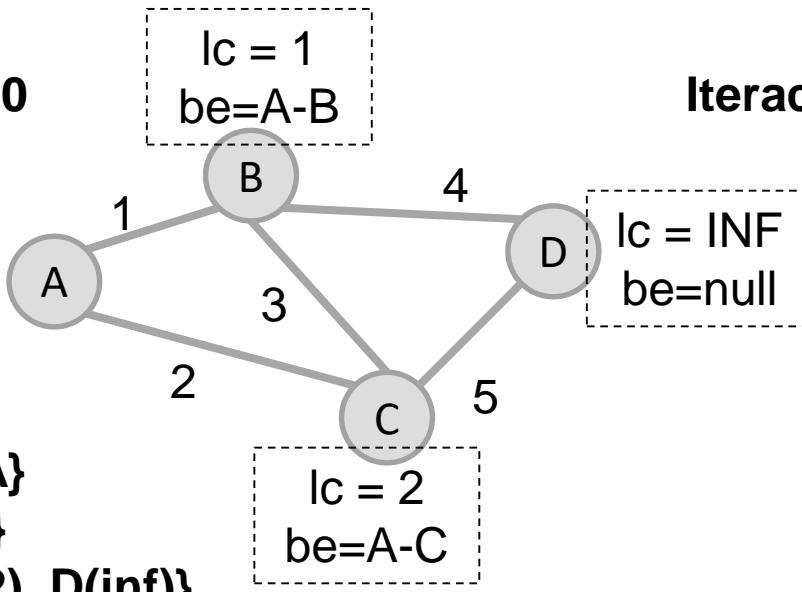
¿Cuál es la eficiencia? Igual que en Dijkstra  
 $O((N + M) * \log(N))$

Al incorporar  $u$  a  $R$ , los únicos nodos que se podrían ver afectados (en  $lc$ ) serían aquellos conectados a  $u$

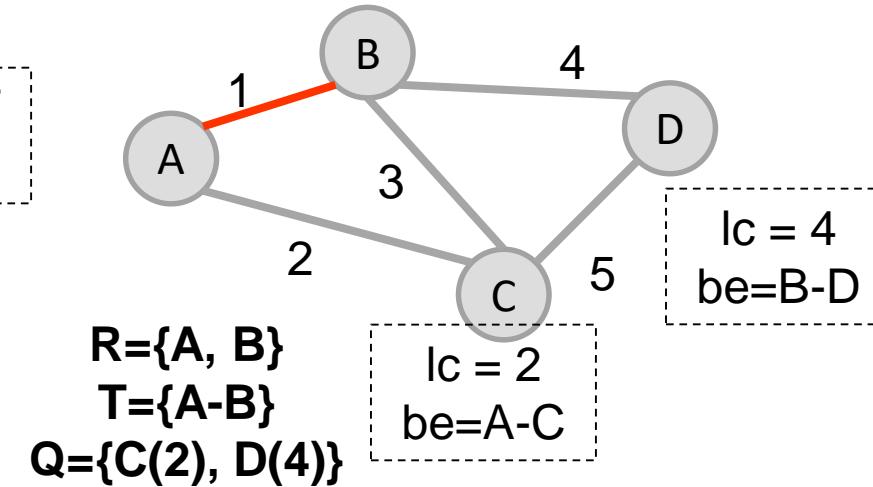


# Algoritmo de Prim

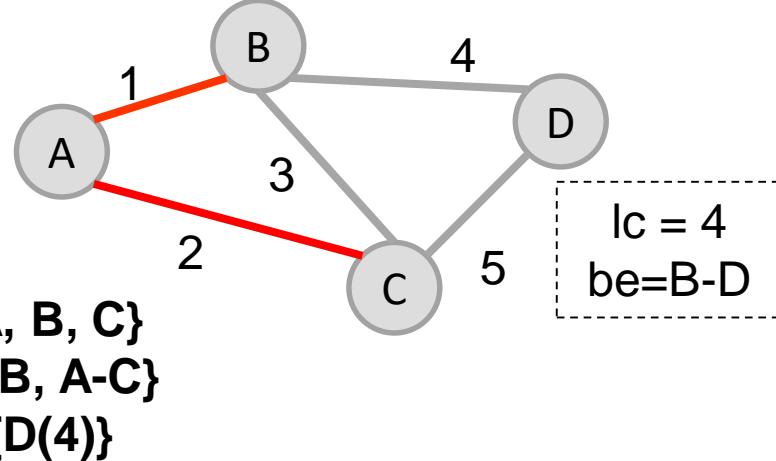
Iteración 0



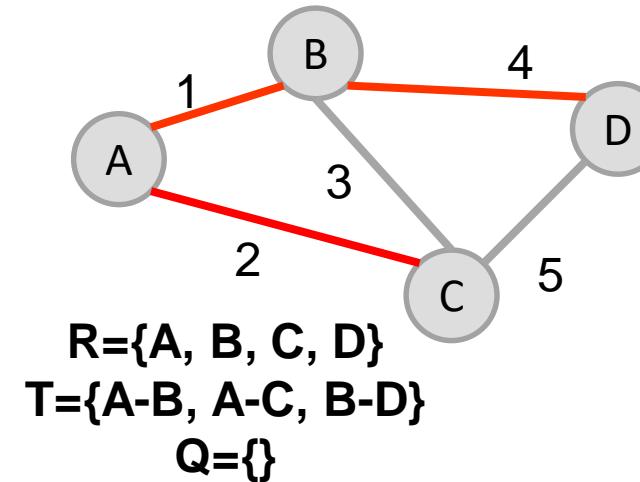
Iteración 1



Iteración 2



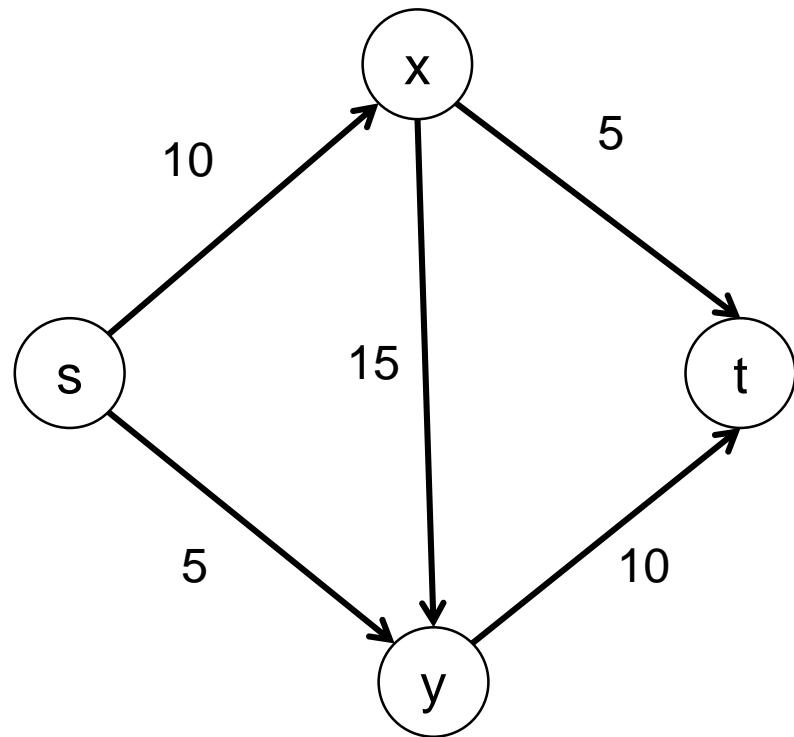
Iteración 2



# Algoritmo de Edmonds-Karp

# Flujo máximo

Dada una red con una fuente  $s$  (origen) y un sumidero  $t$  (destino) al cual se puede llegar mediante cero o más nodos intermedios se busca maximizar el flujo de  $s$  hacia  $t$



# Flujo máximo

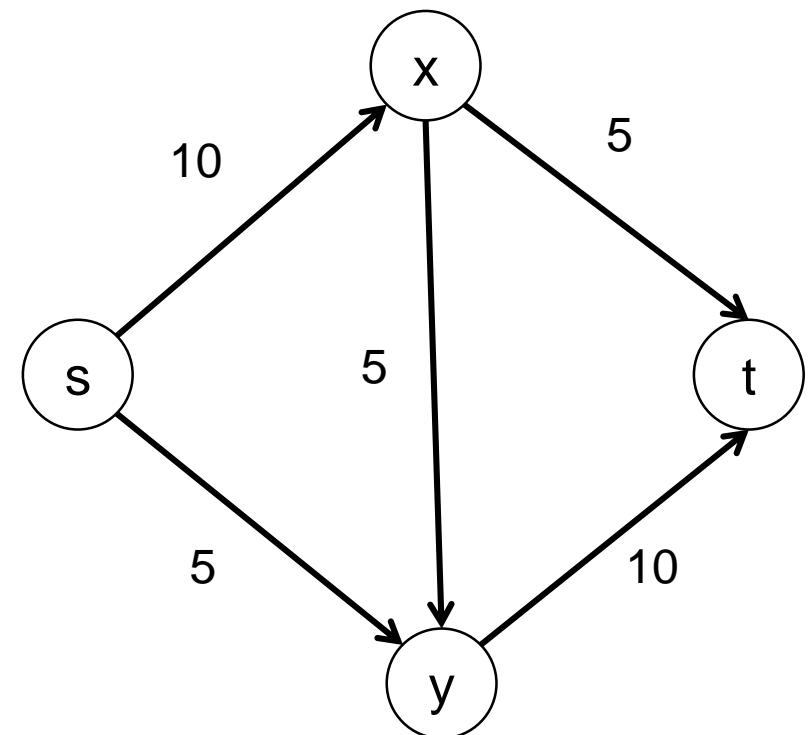
**Entrada:** Un grafo dirigido  $G = (V, E)$  con  $N$  nodos y  $M$  aristas donde cada arista  $(u, v)$  tiene una capacidad  $c$

Donde un flujo es una ruta  $f$  que satisface:

- Restricción de capacidad: el flujo de una arista no puede exceder su capacidad, es decir,  $f_{u,v} \leq c_{u,v}$
- Conservación de flujos: la suma de los flujos que entran a un nodo debe ser igual a la suma de flujos que salen del nodo, excepto para la fuente  $s$  y el sumidero  $t$

Si  $f$  corresponde al flujo de  $s$  hacia  $t$ :  $|f| = \sum_{v \in E} f_{s,v}$

**Salida:**  $f_{max}$



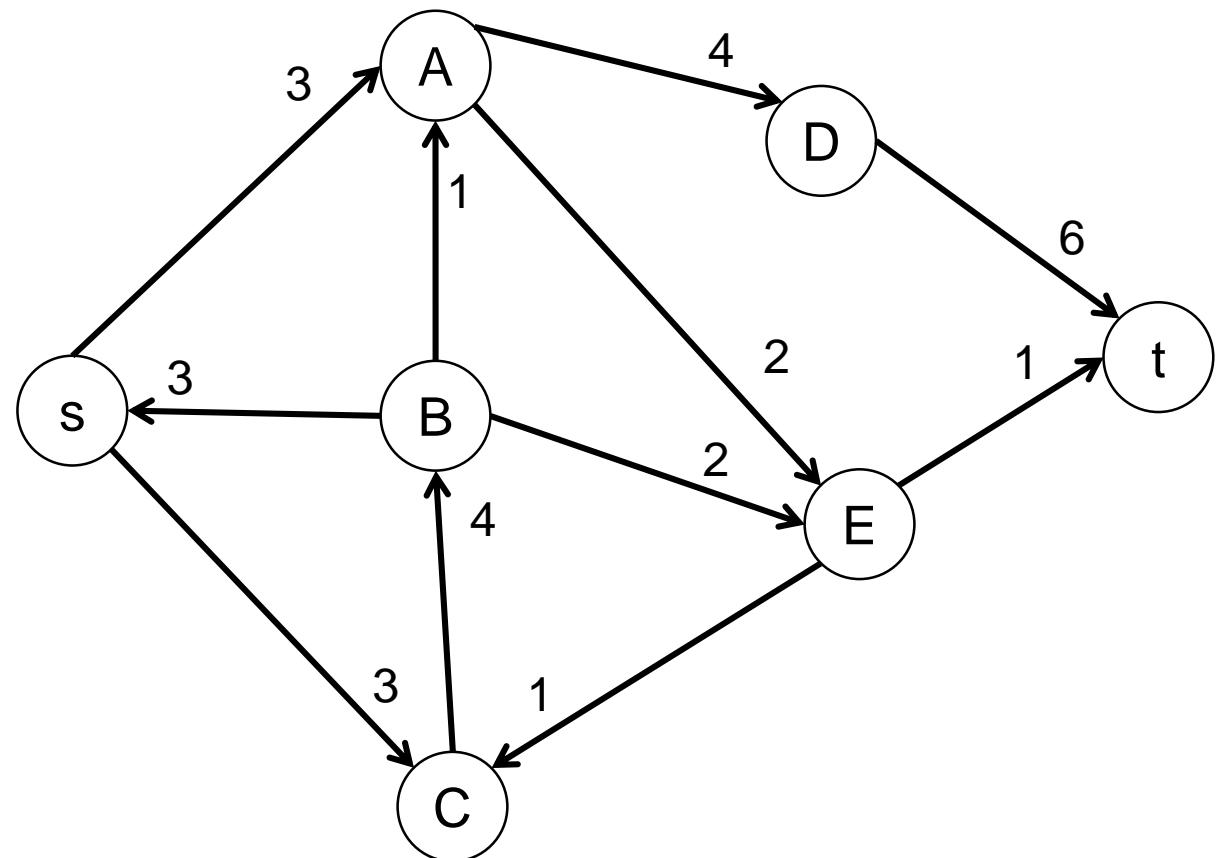
# Método de Ford-Fulkerson

1.  $R_{u,v} = c_{u,v}$  para toda arista  $u,v$  en  $E$
2.  $F_{u,v} = 0$  para toda arista  $u,v$  en  $E$
3.  $mf = 0$
4. Mientras haya un camino  $P$  desde  $s$  hacia  $t$  tal que no haya ciclos y  $R_{u,v} > 0$  para todo  $u,v$  en  $P$ :
5.  $B(p) = \min(R_{u,v})$  para todo  $u,v$  en  $P$  (cuello de botella)
6. Para todo  $u,v$  en  $P$ :
7.  $R_{u,v} = R_{u,v} - B(p)$
8.  $F_{u,v} = F_{u,v} + B(p)$
9.  $mf = mf + B(p)$

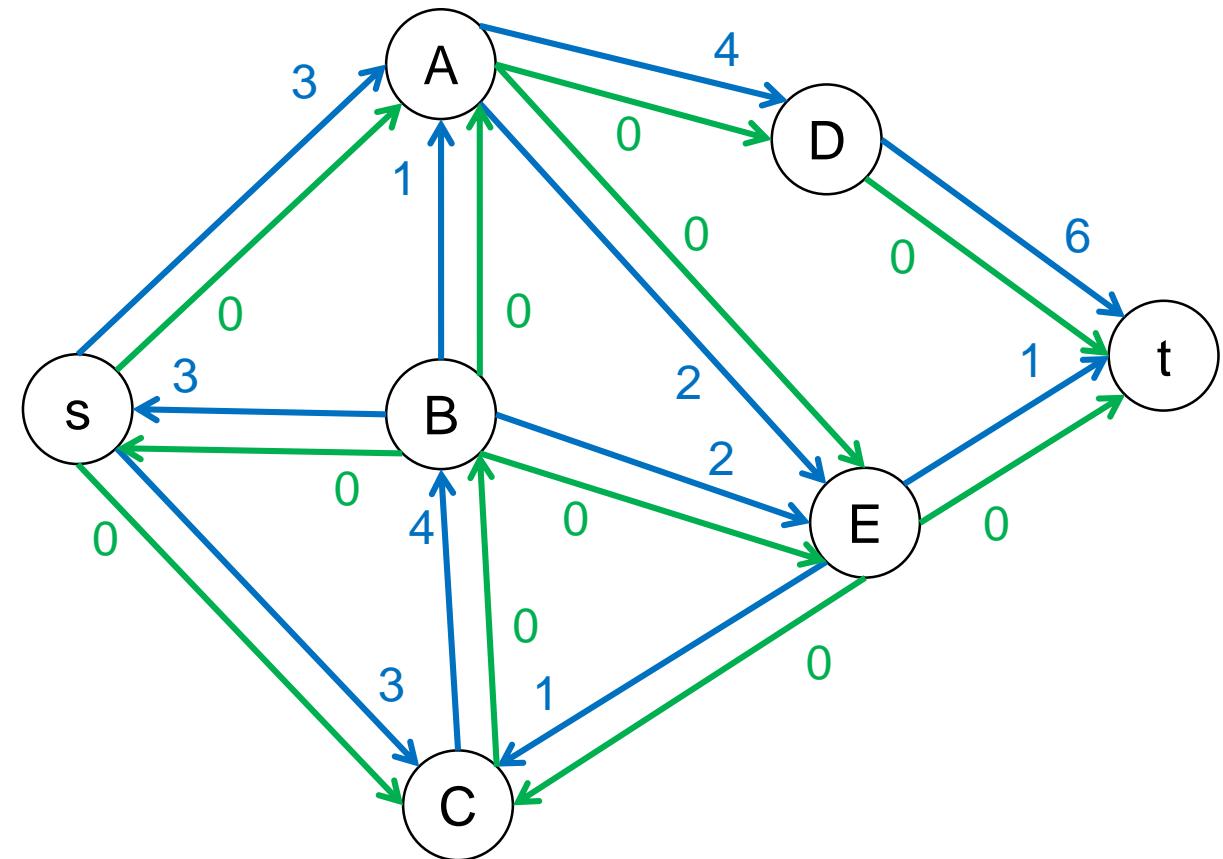
**Criterio greedy**

Cuando los caminos  $P$  dentro de la instrucción 4 se encuentran mediante mínima cantidad de saltos por BFS, el método de Ford-Fulkerson se convierte en el algoritmo de Edmonds-Karp

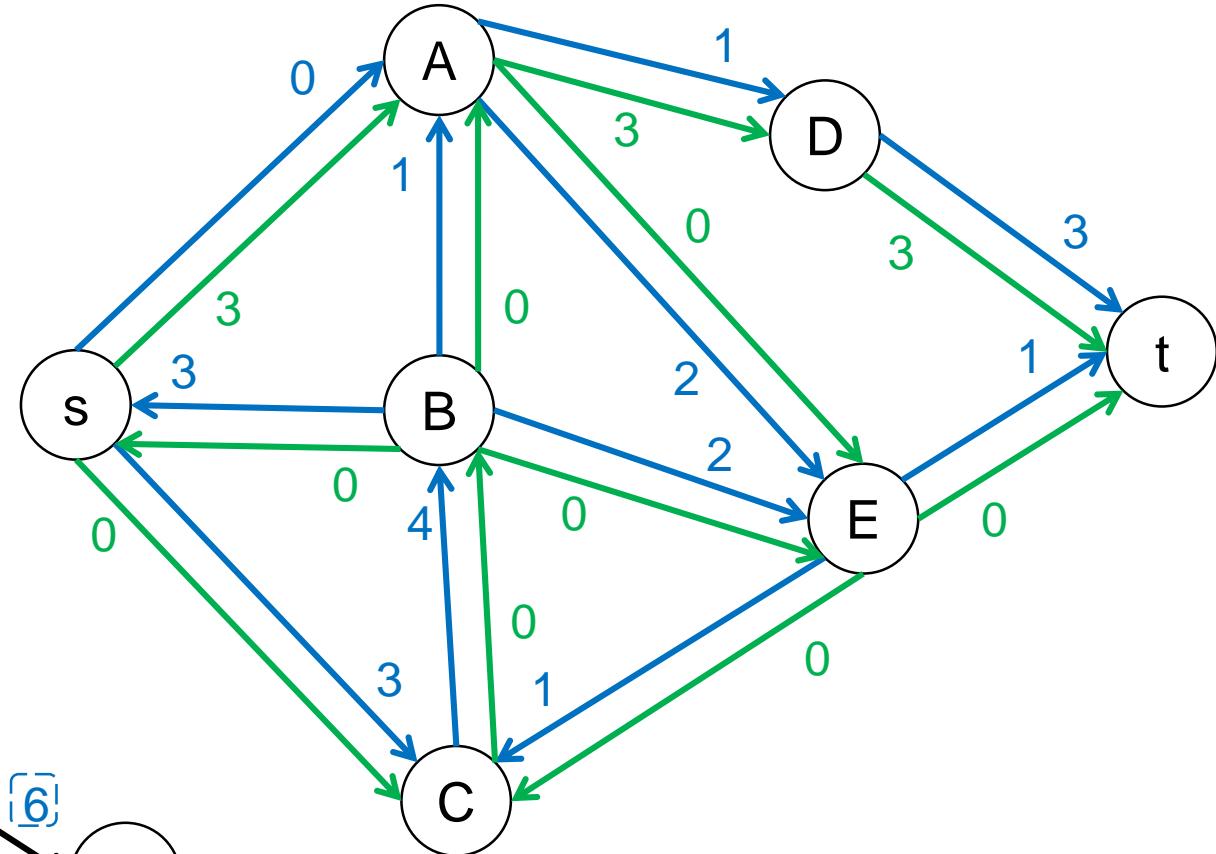
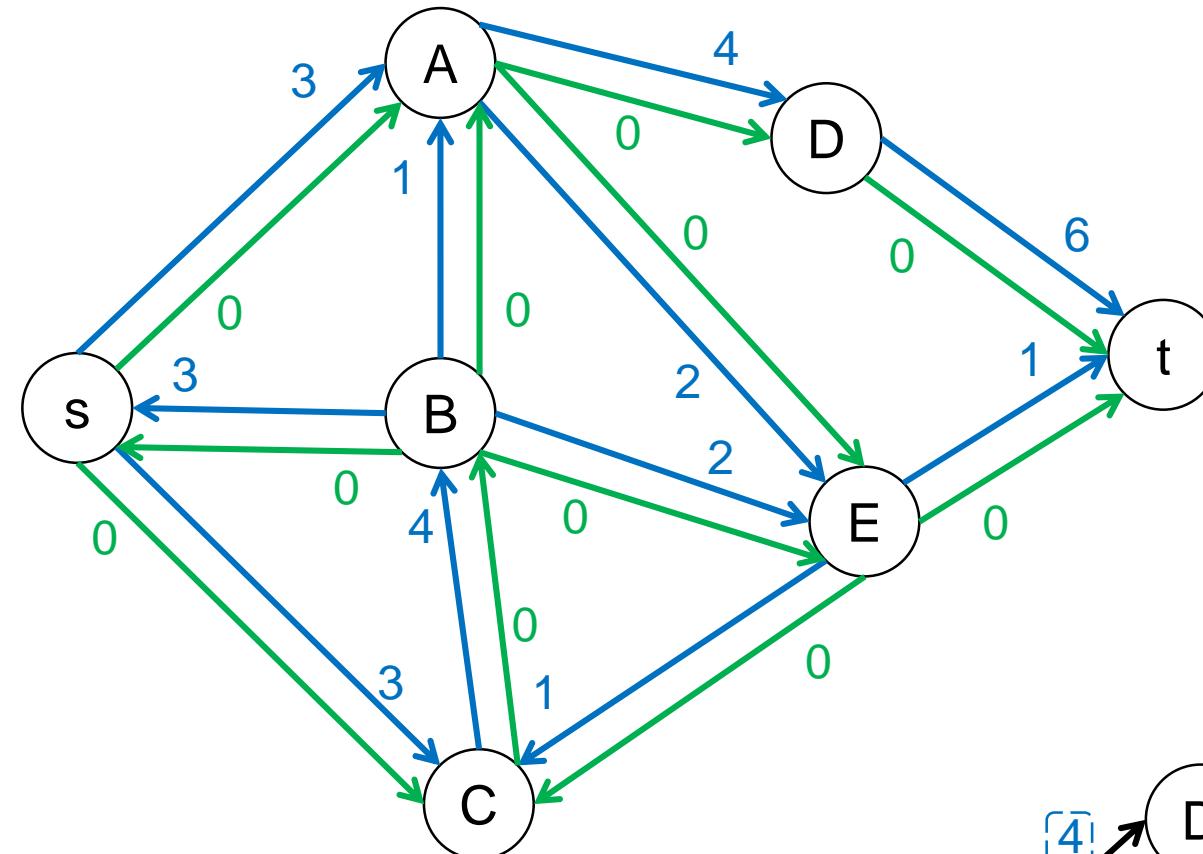
# Algoritmo de Edmonds-Karp: estado inicial



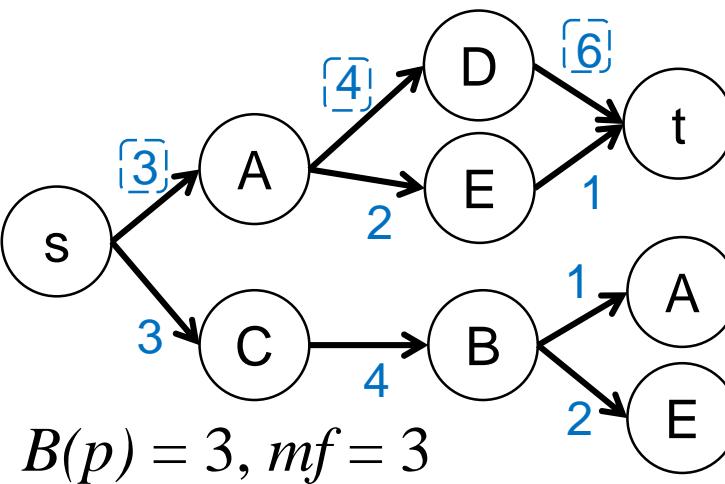
Grafo residual:  $R, F$



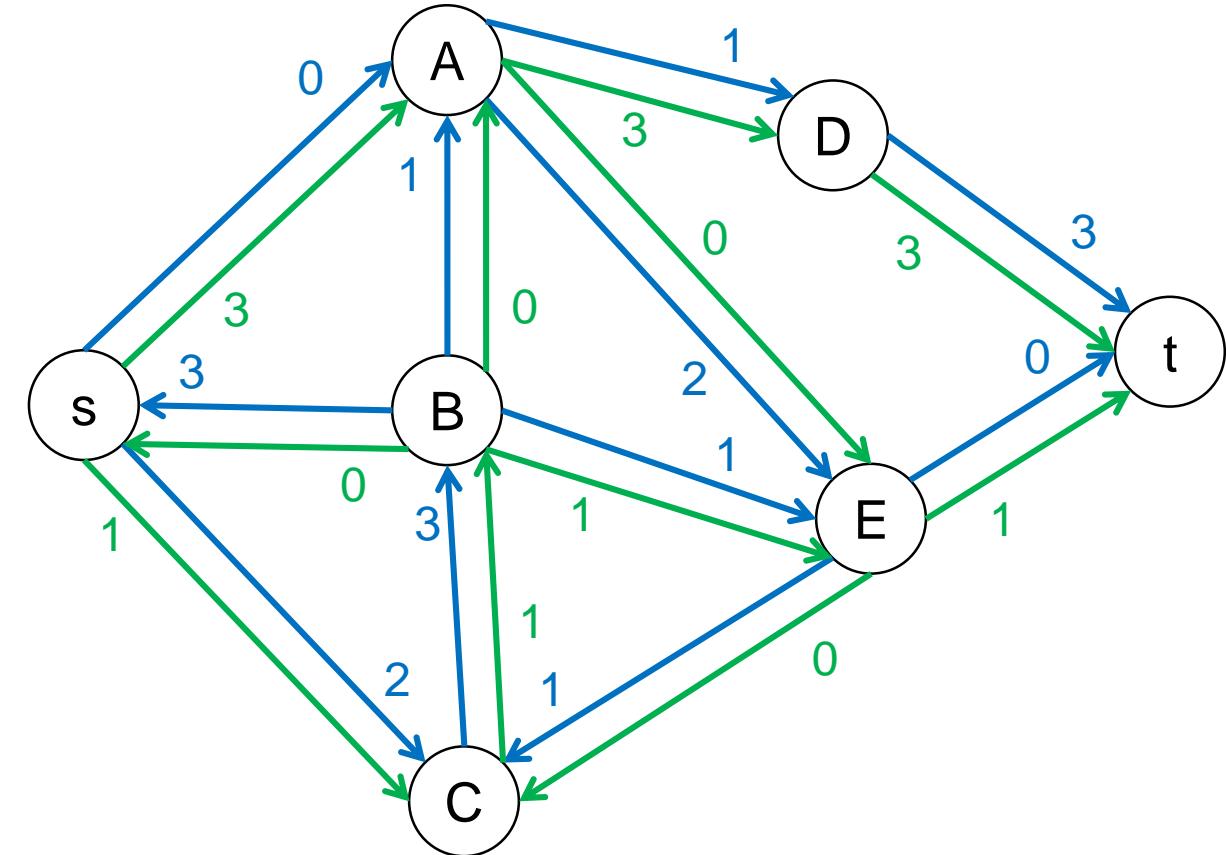
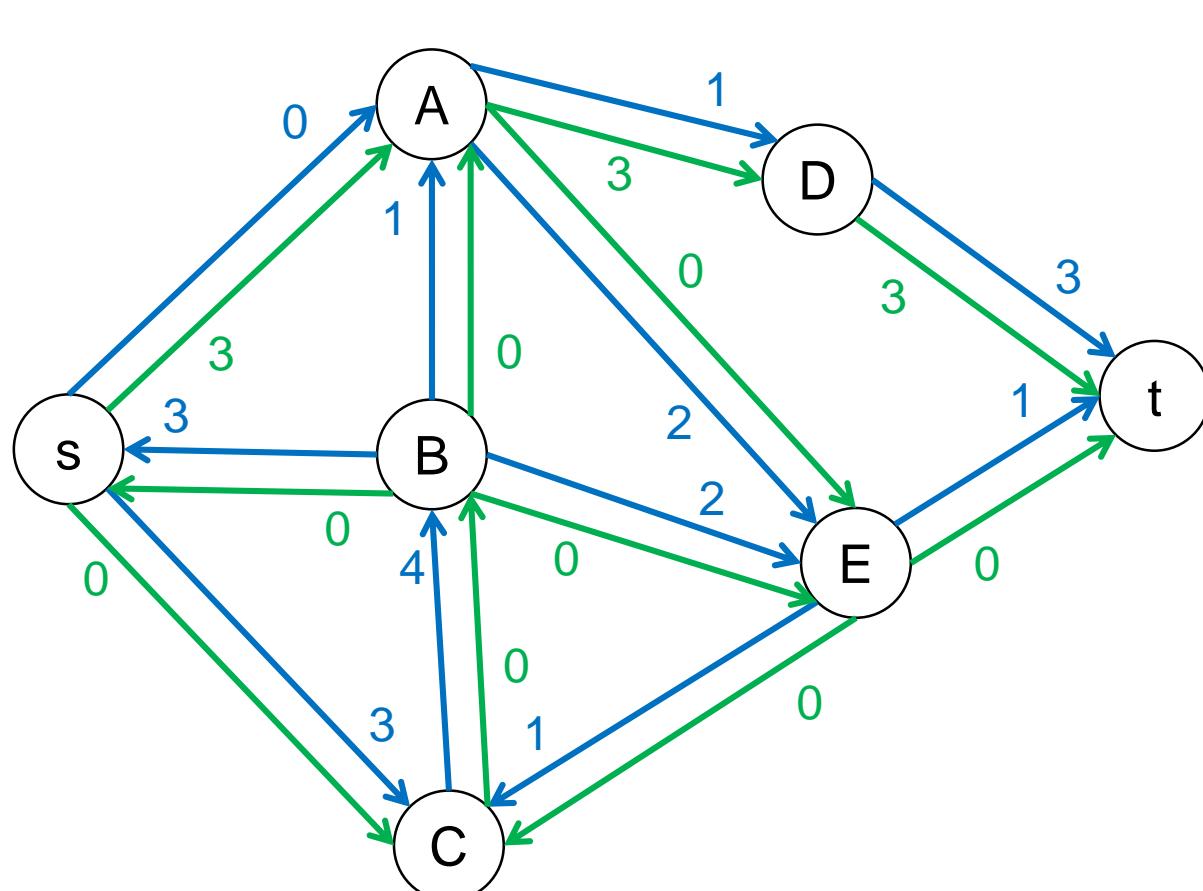
# Algoritmo de Edmonds-Karp: iteración 1



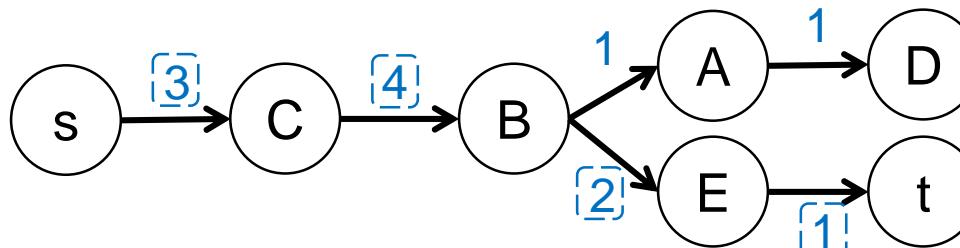
Caminos mediante BFS



# Algoritmo de Edmonds-Karp: iteración 2

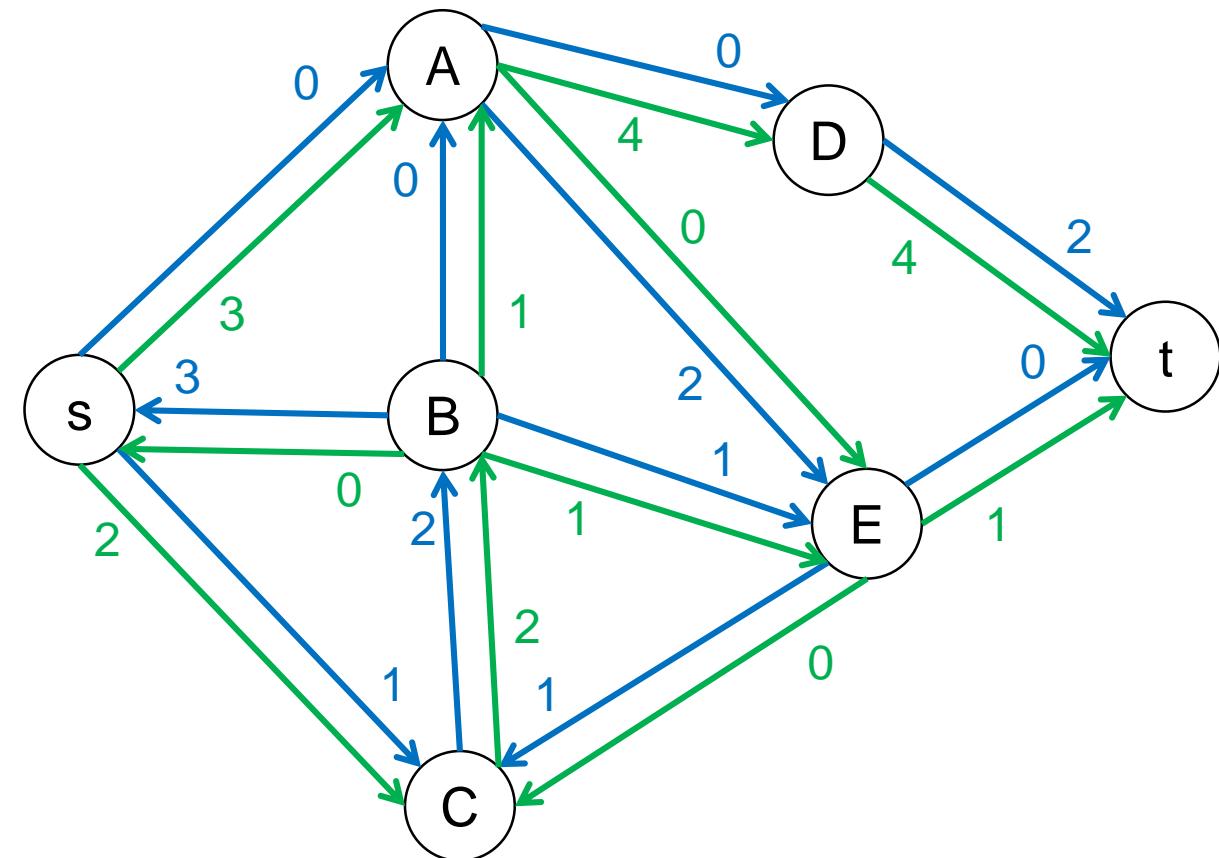
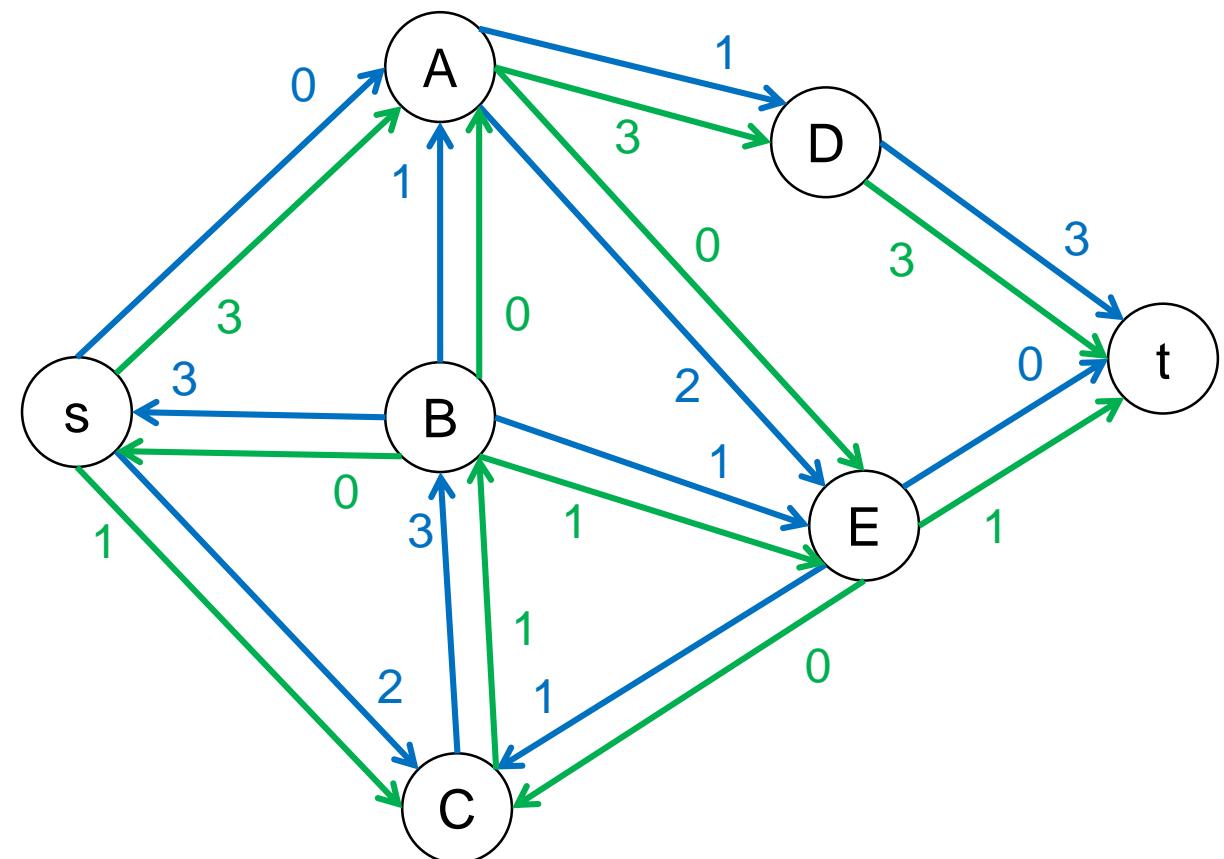


Caminos de aumento

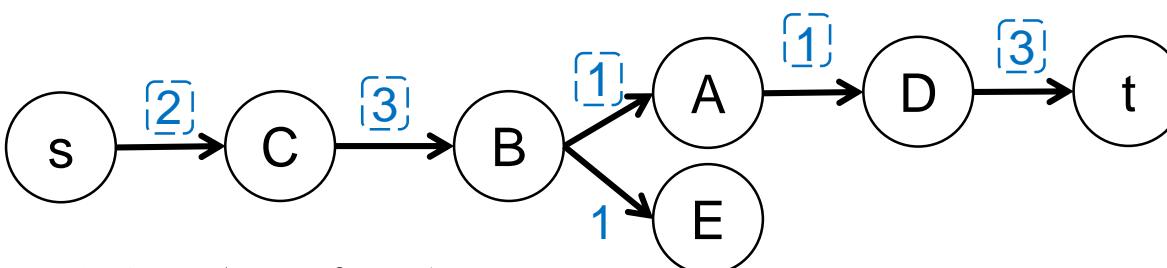


$$B(p) = 1, mf = 4$$

# Algoritmo de Edmonds-Karp: iteración 3

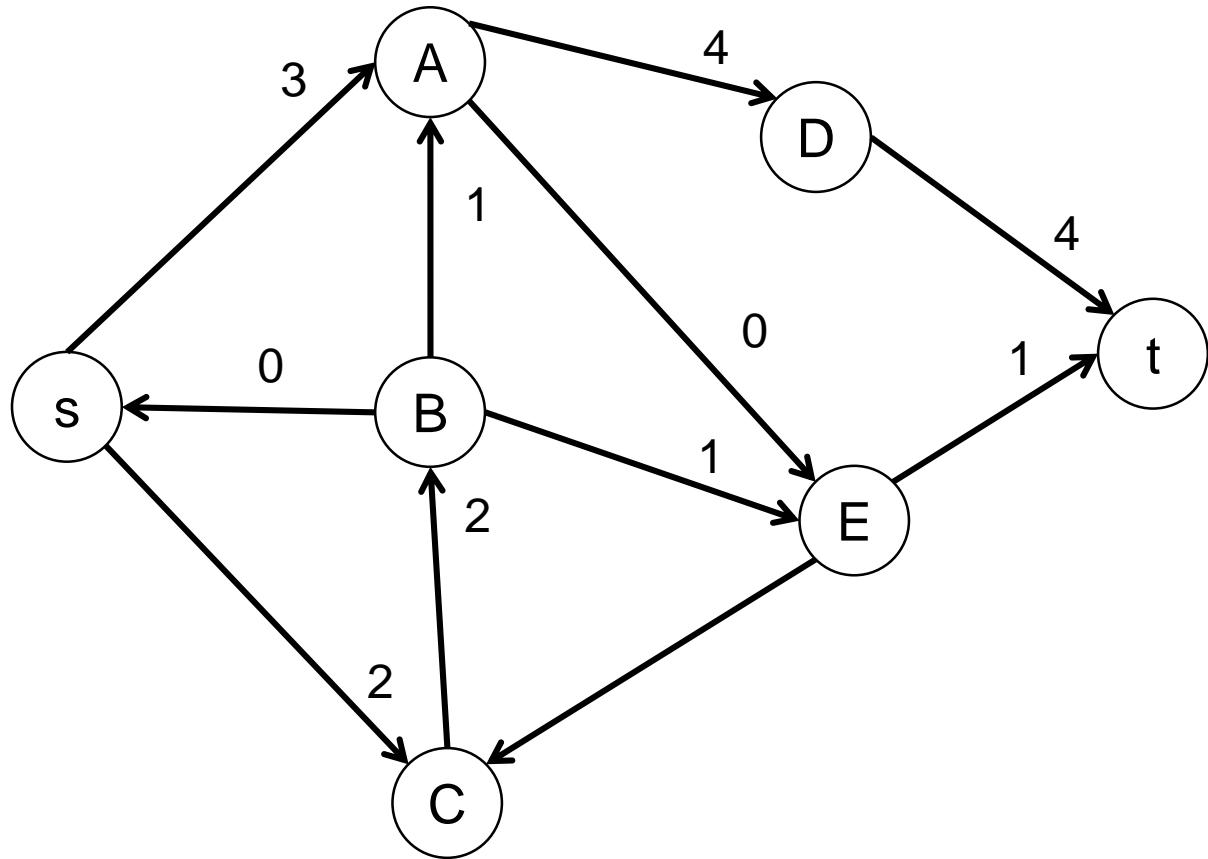
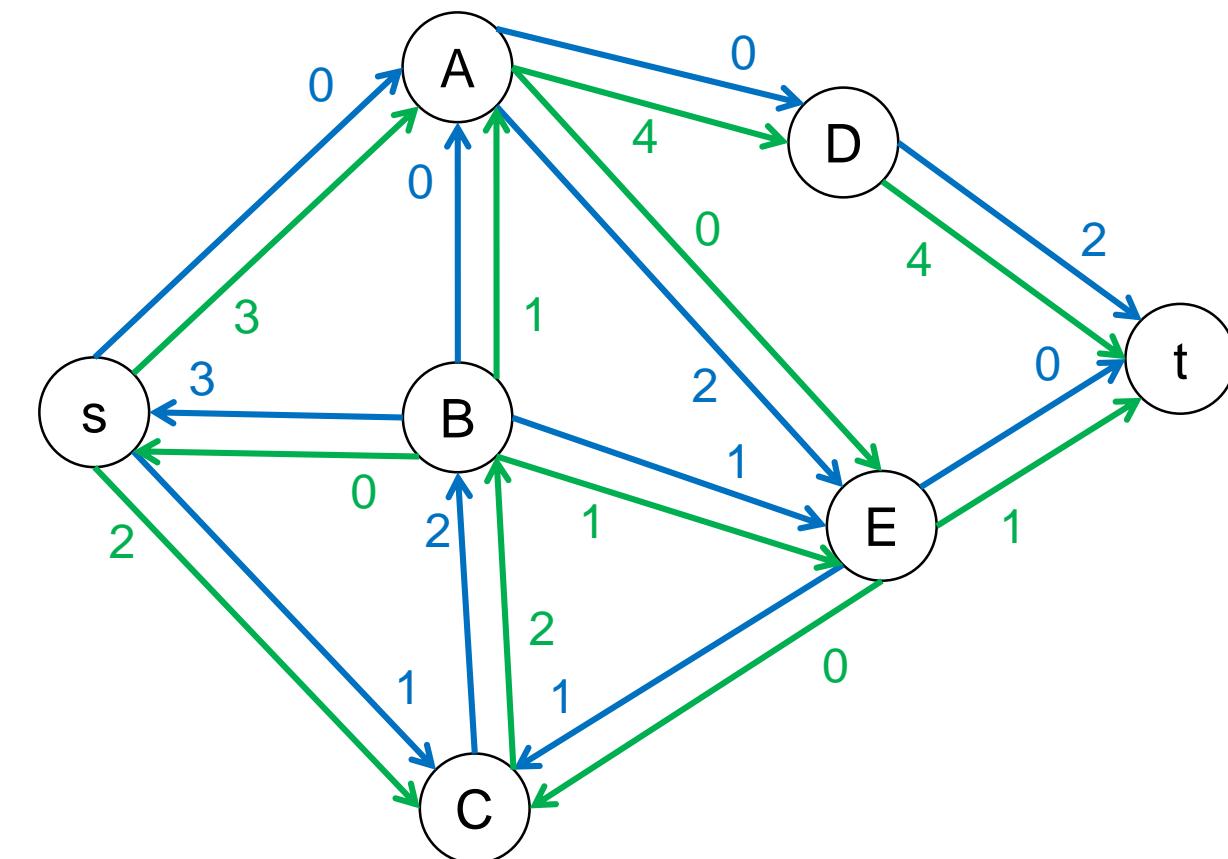


Caminos de aumento

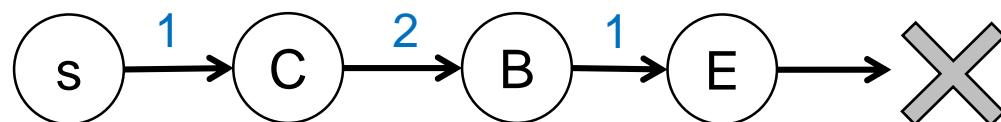


$$B(p) = 1, mf = 5$$

# Algoritmo de Edmonds-Karp: iteración 4



Caminos de aumento

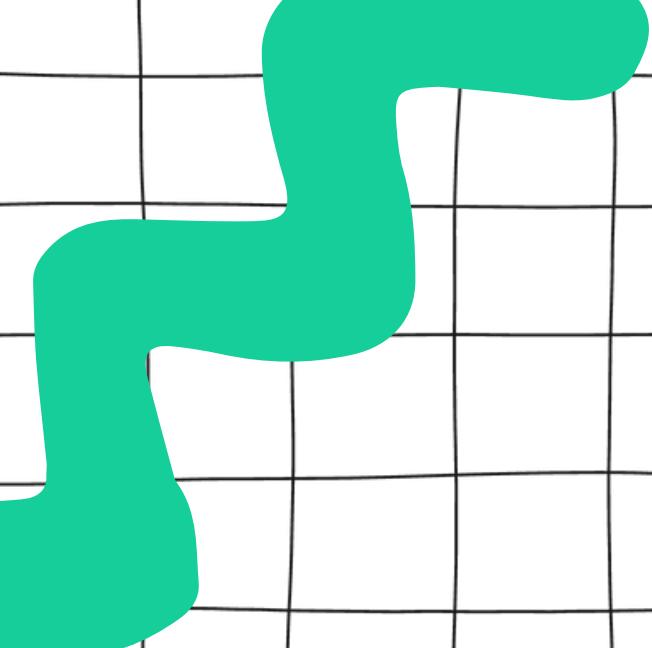


$$mf = 5$$

La complejidad del algoritmo es  $O(NM^2)$  (ver teorema 26.8 y la prueba correspondiente en *Introduction to algorithms*, p. 727)

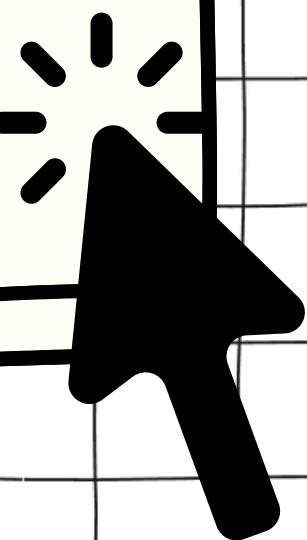
Greedy III



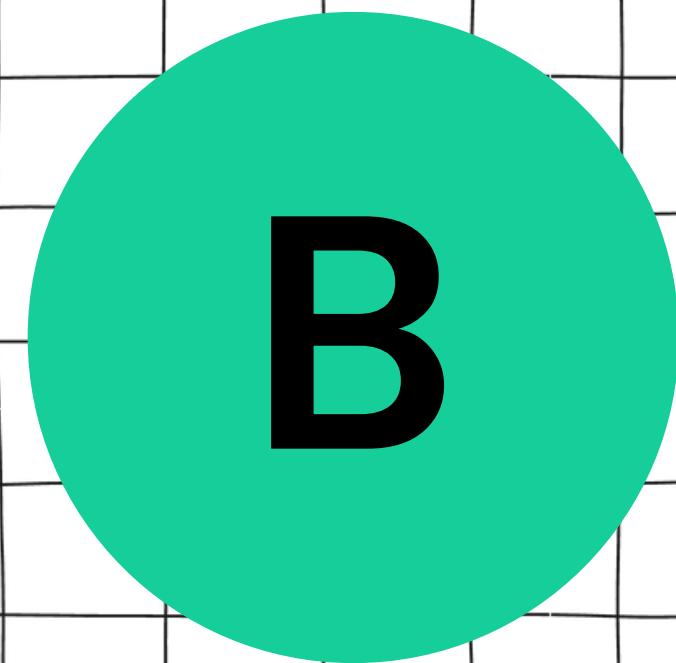
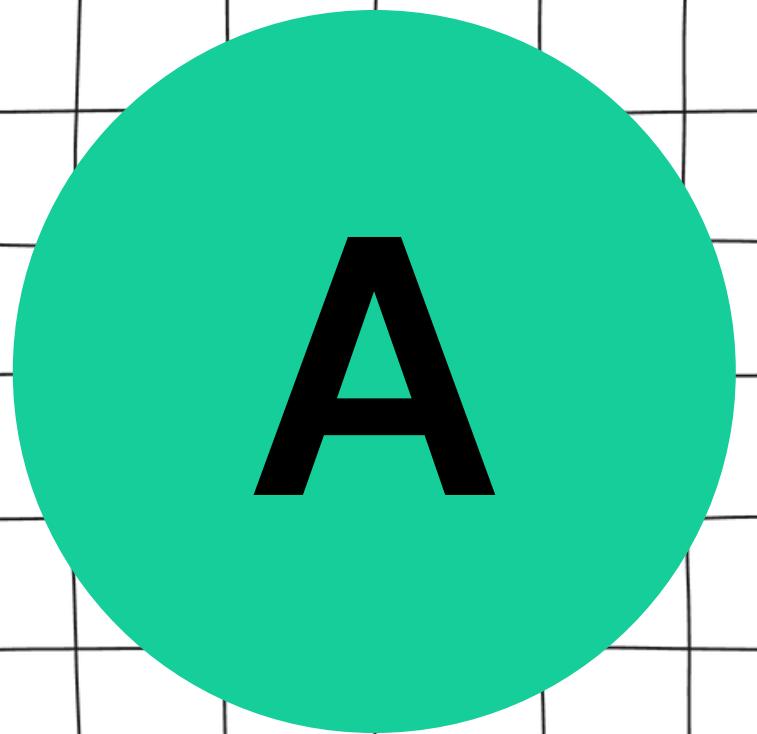


**DSU**

# **Disjoint Set Union**



# Conjuntos disjuntos



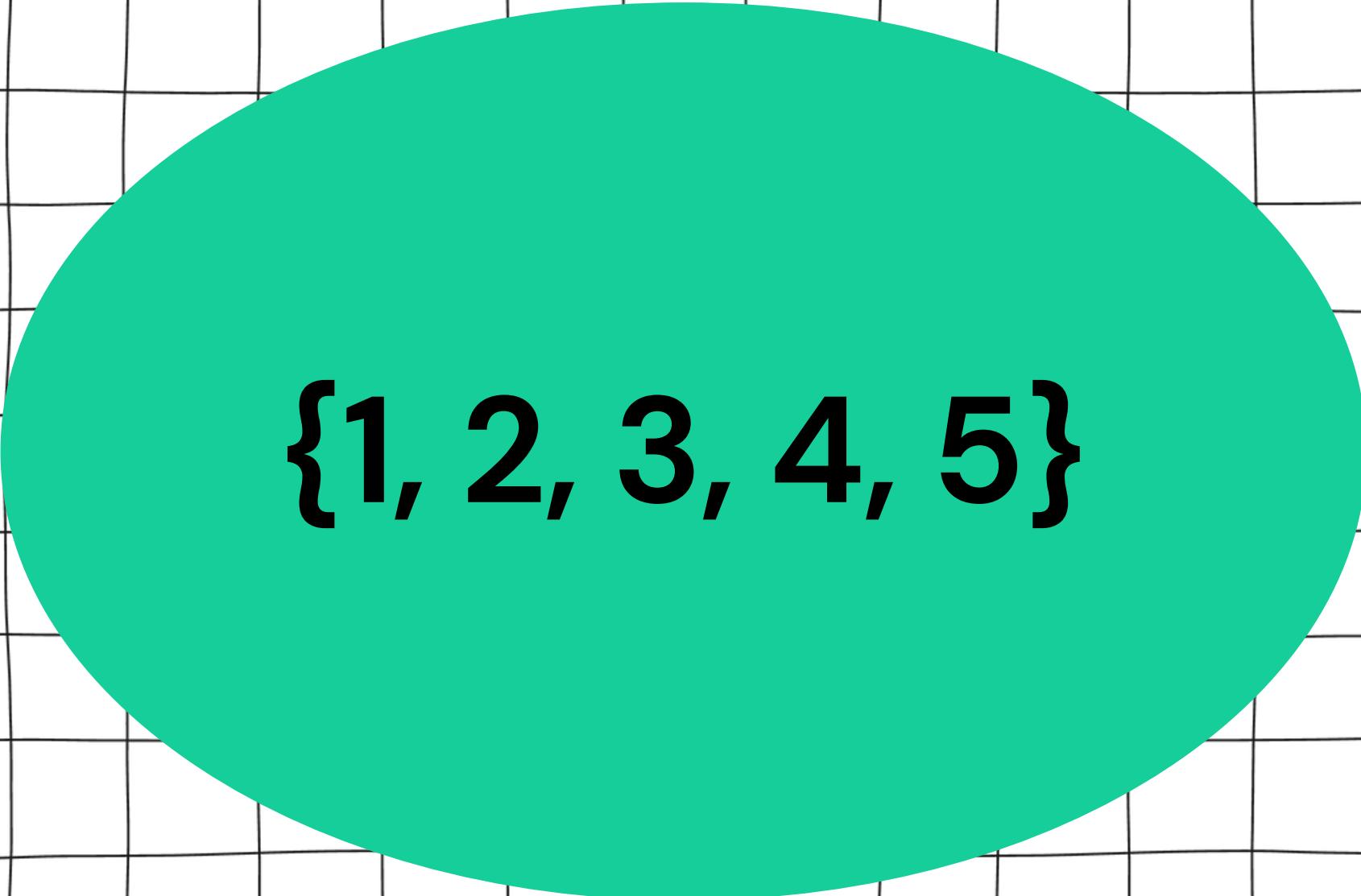
$$A \cap B = \{\}$$

# Operaciones del DSU

- **get(a)**
  - **Retorna a que conjunto pertenece a**
- **union(a, b)**
  - **Une los conjuntos a los que pertenece a y b**

# **Representante del conjunto**

**El representante en DSU es el nodo raíz que identifica a un conjunto, usado para determinar si dos elementos pertenecen al mismo conjunto.**



**{1, 2, 3, 4, 5}**

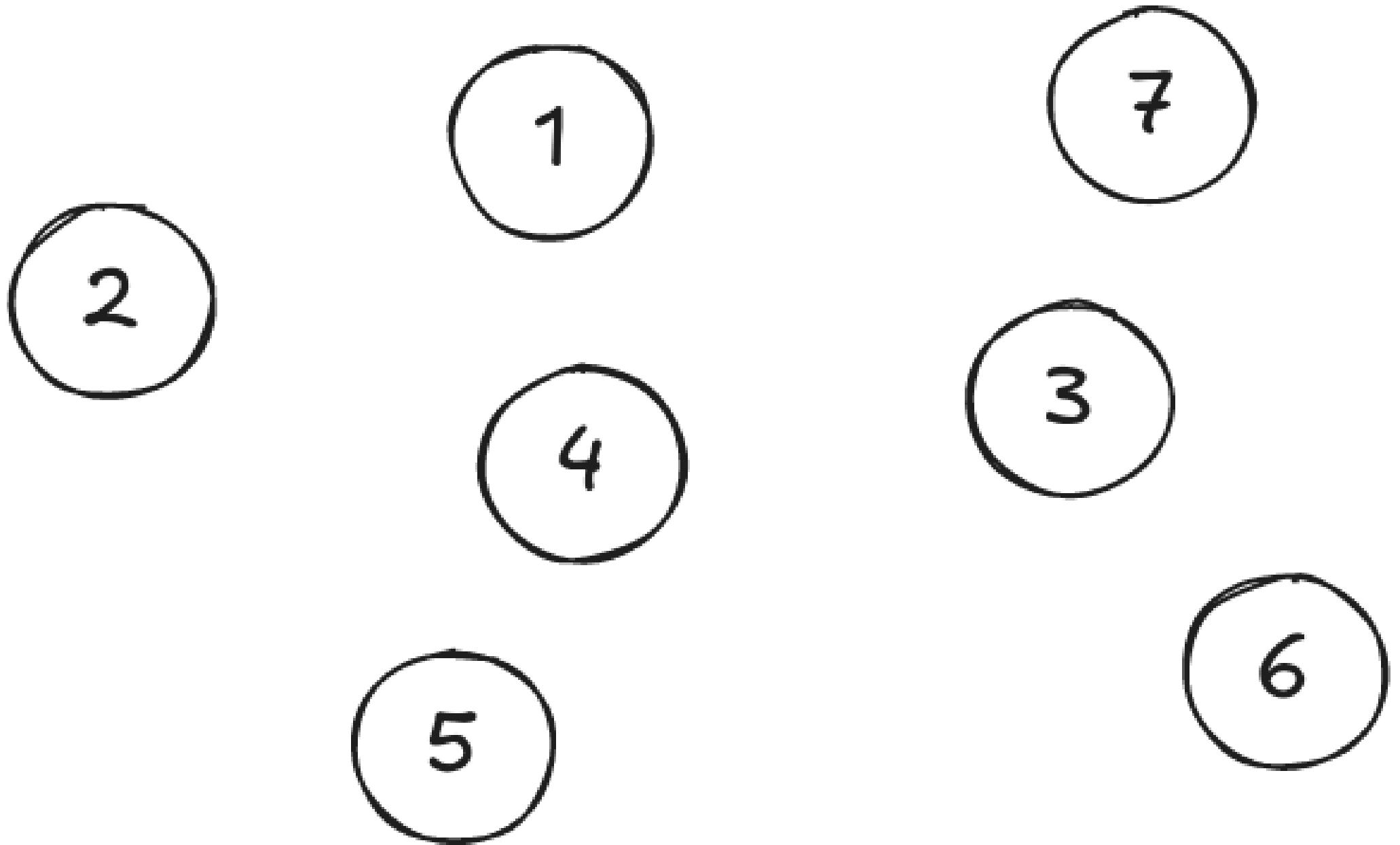
- Cualquier elemento del conjunto puede ser tomado como el representante.

# 3

{1, 2, 3, 4, 5}

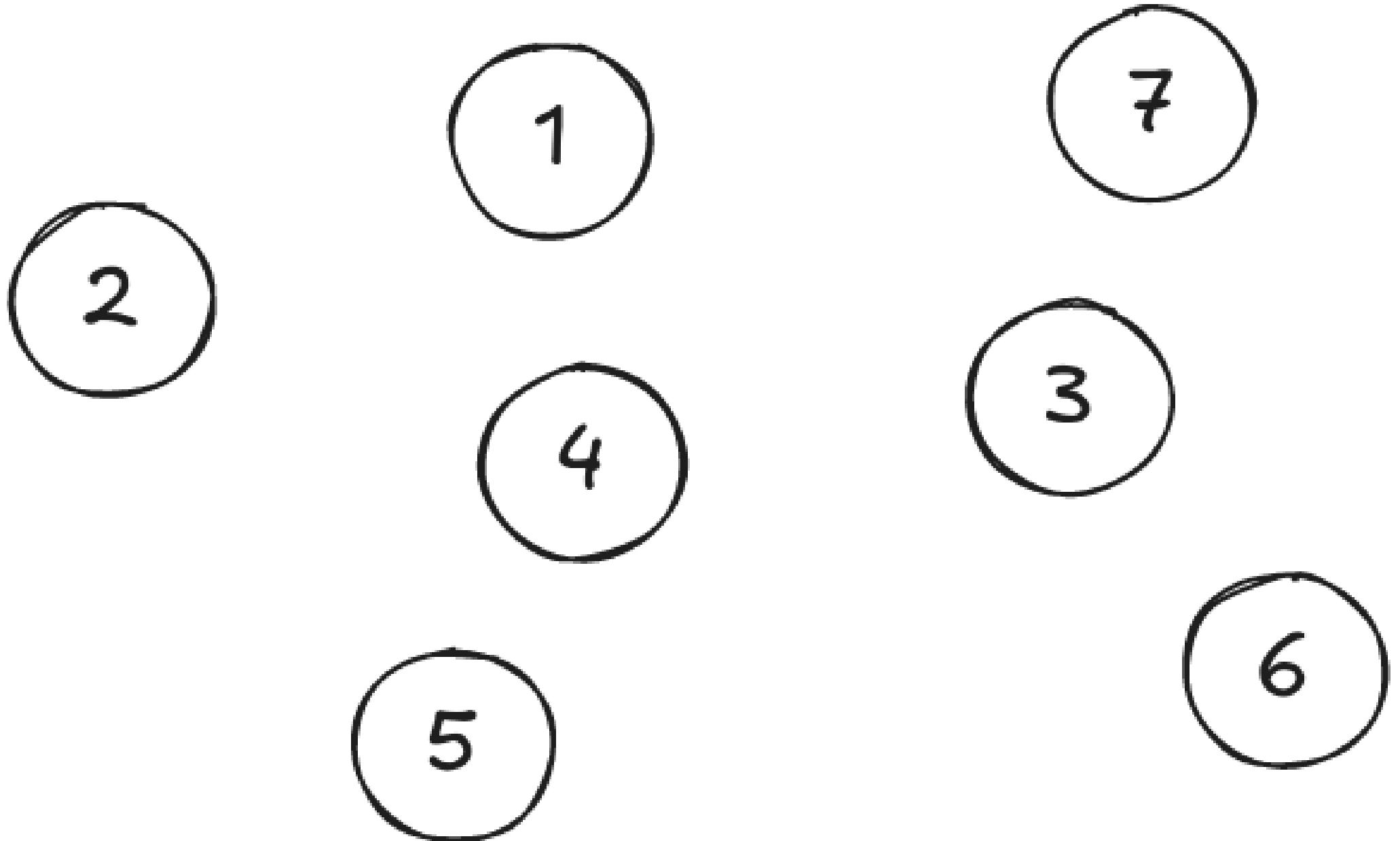
- Cualquier elemento del conjunto puede ser tomado como el representante.
- Ej: 3

# Ejemplo



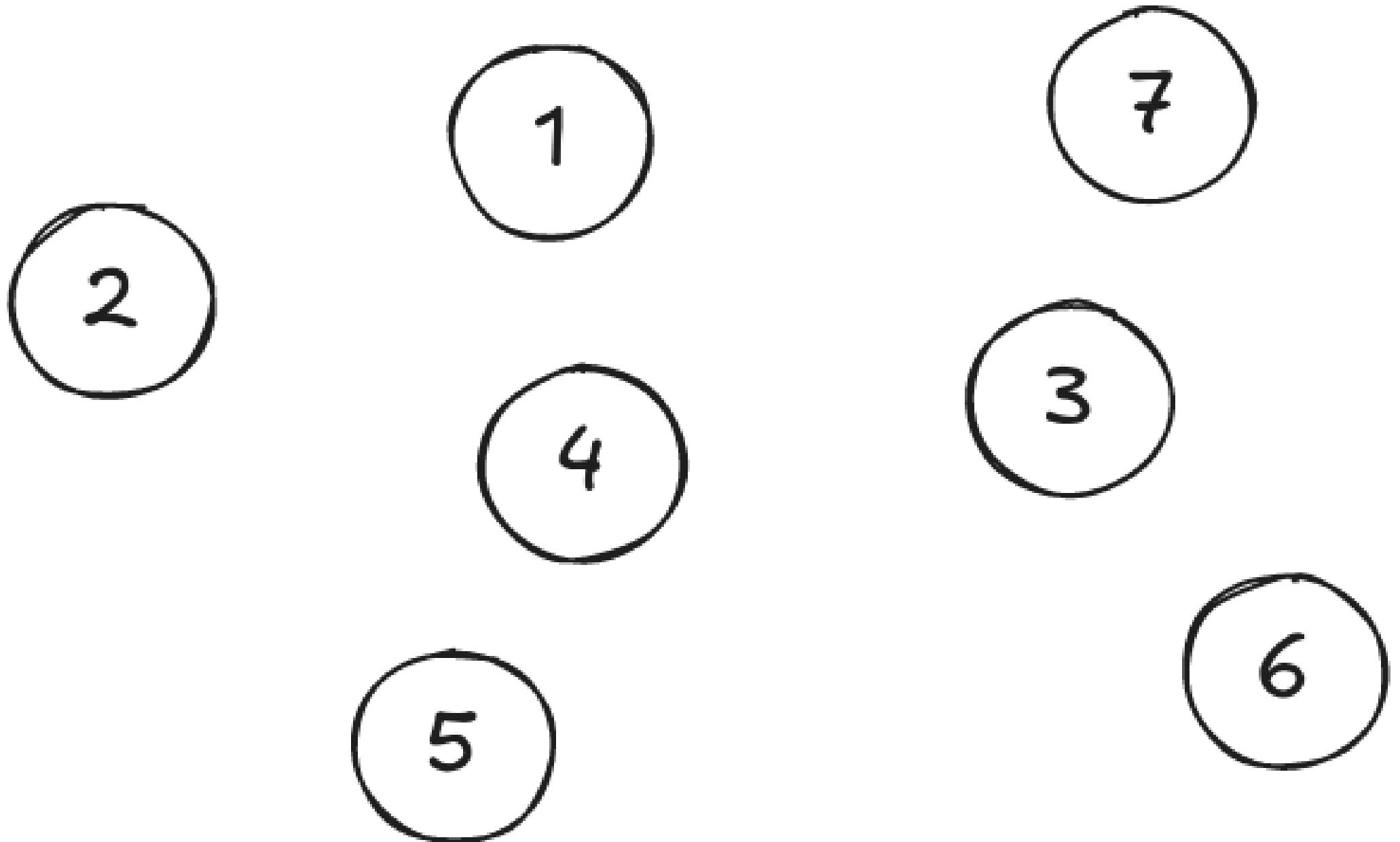
## Ejemplo

- **find(2)**



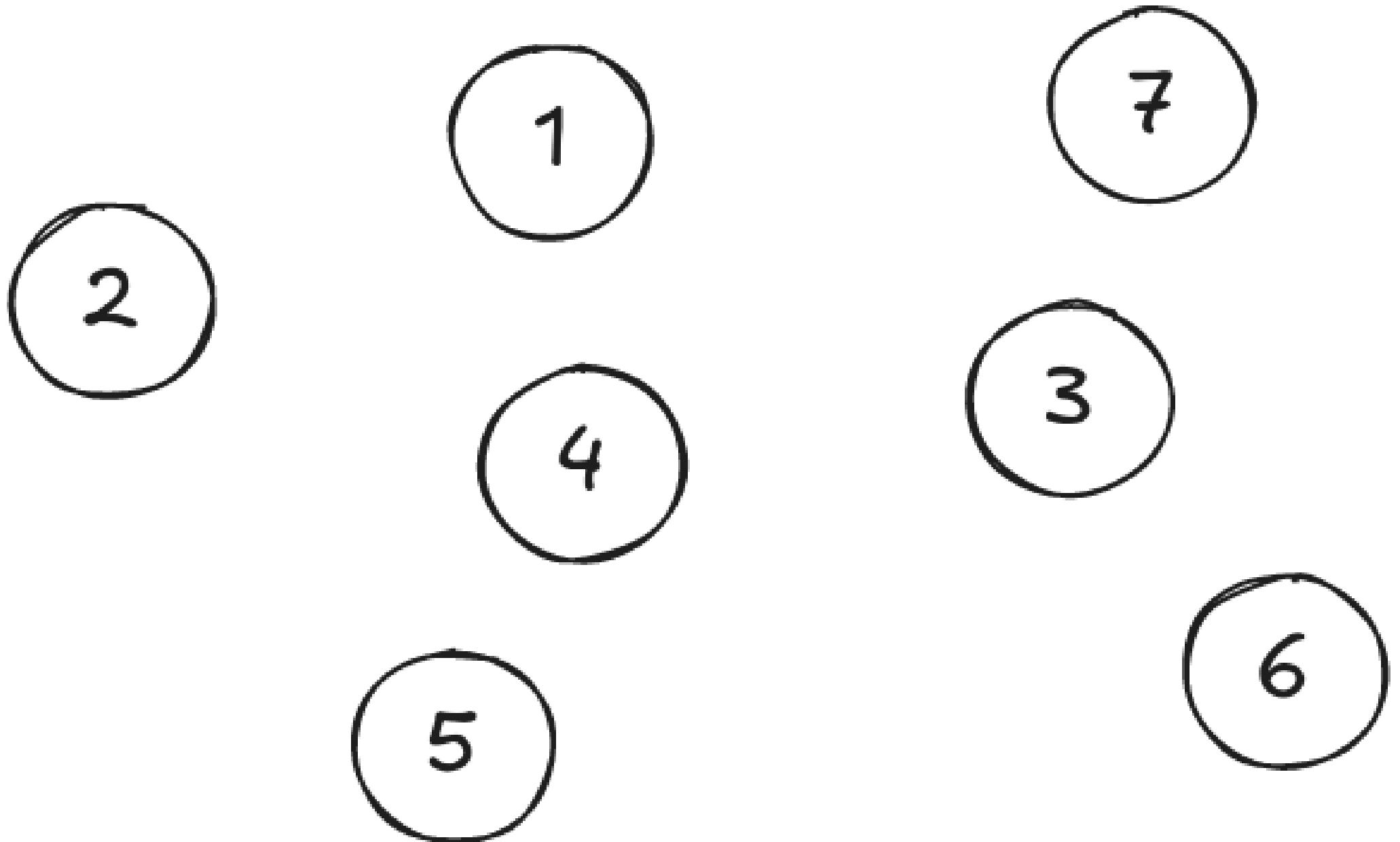
## Ejemplo

- **find(2)**
  - 2



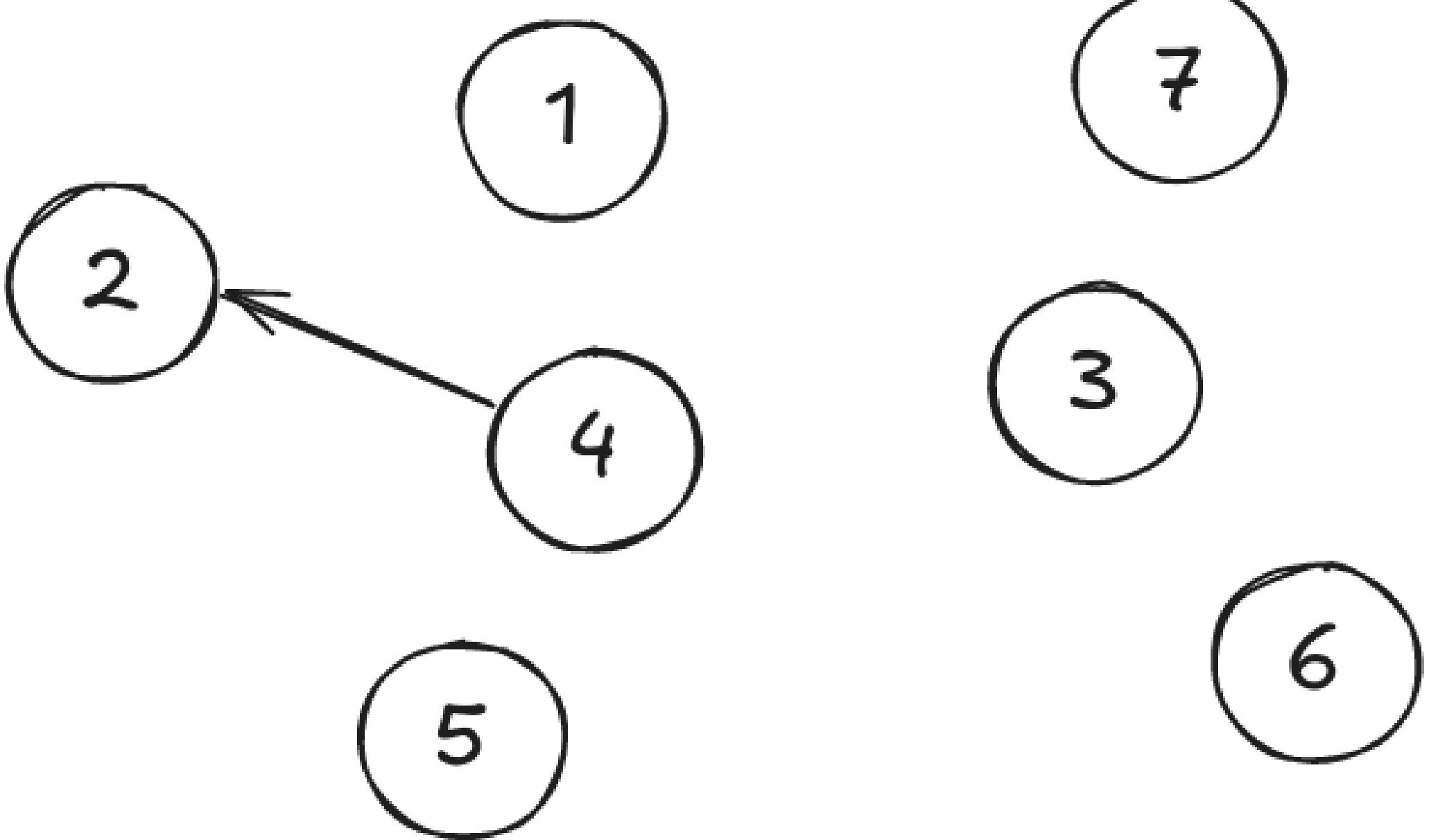
## Ejemplo

- **find(2)**
  - 2
- **union(2,4)**



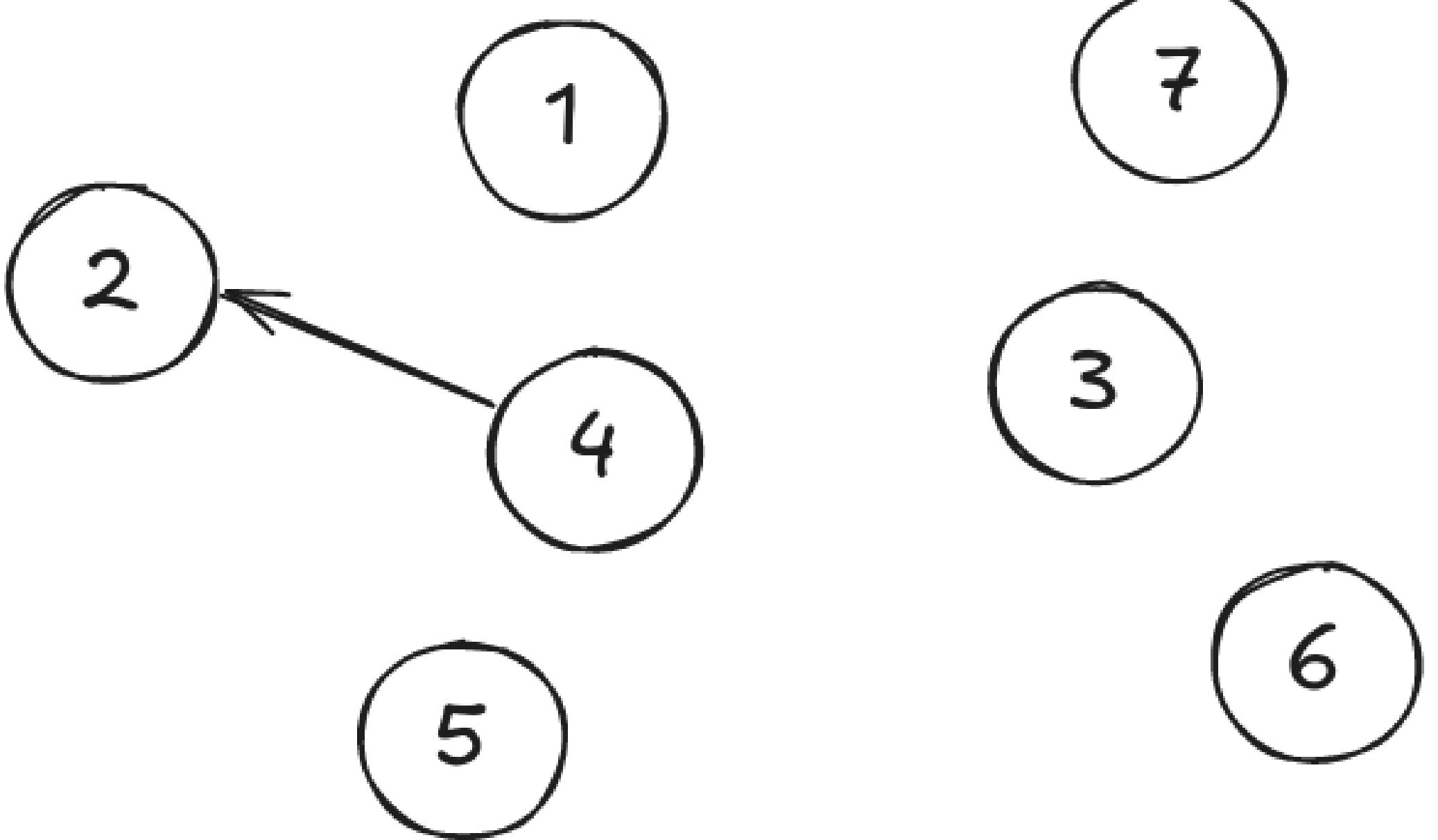
## Ejemplo

- **find(2)**
  - 2
- **union(2,4)**



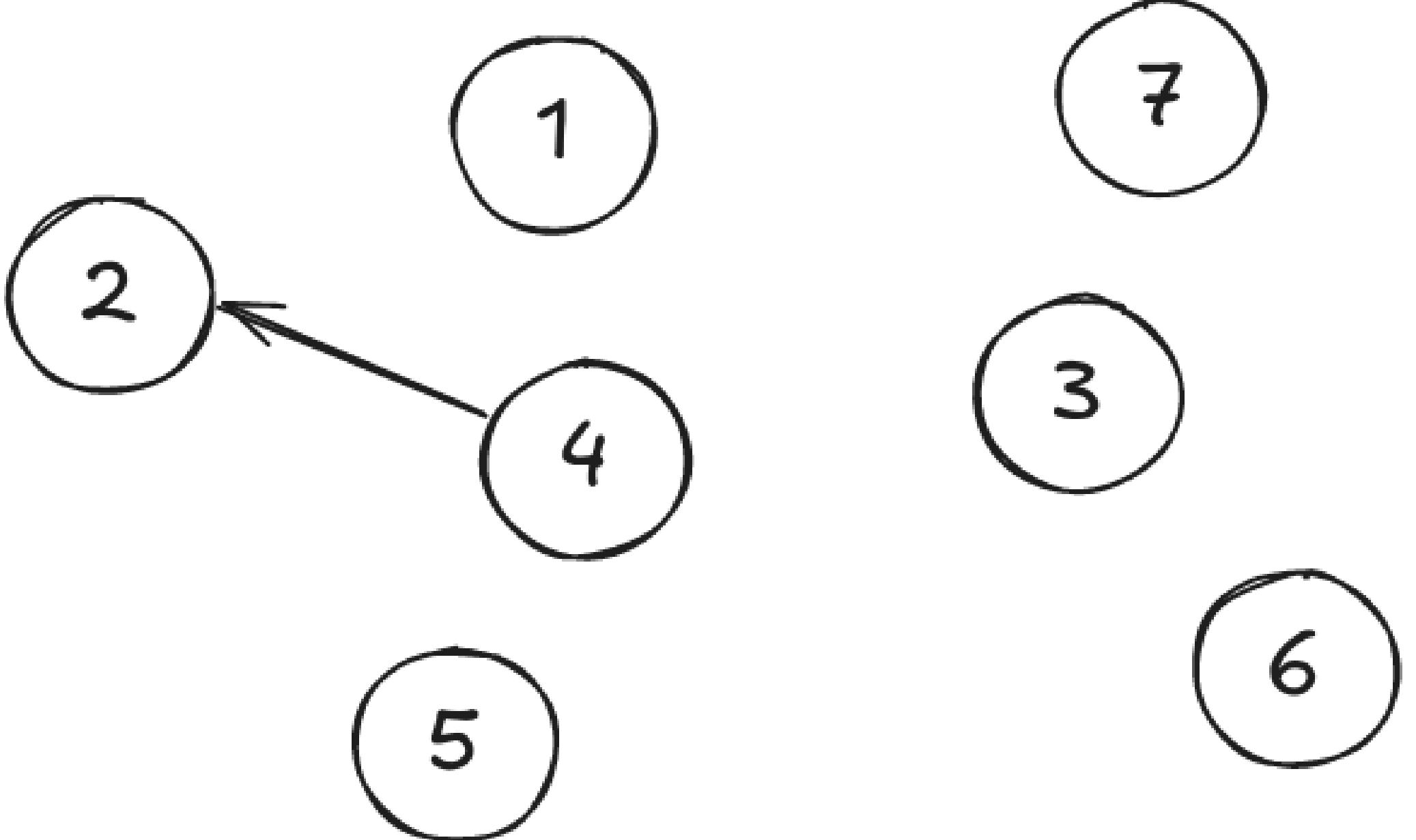
## Ejemplo

- **find(2)**
  - 2
- **union(2,4)**
- **find(4)**



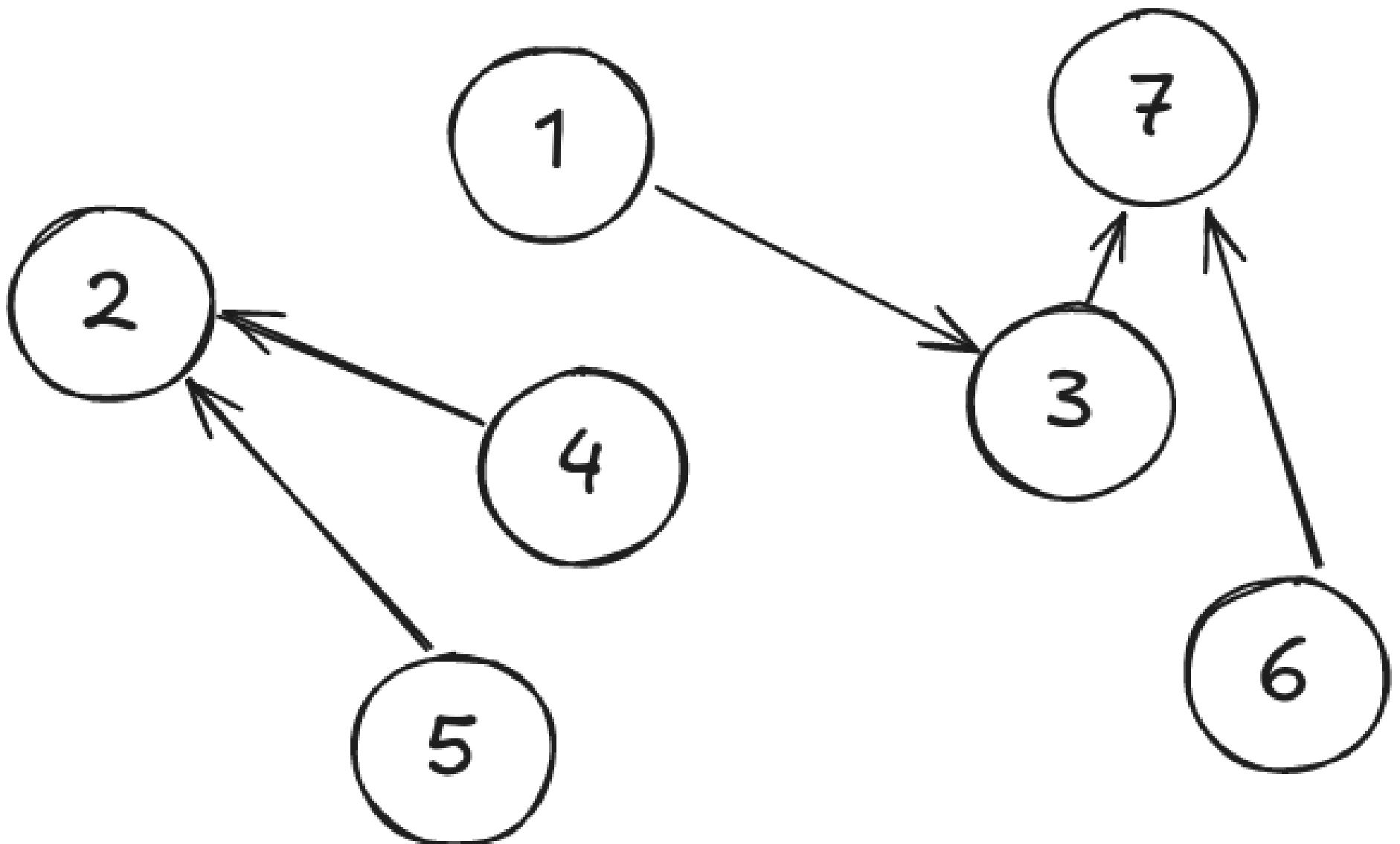
## Ejemplo

- **find(2)**
  - 2
- **union(2,4)**
- **find(4)**
  - 2



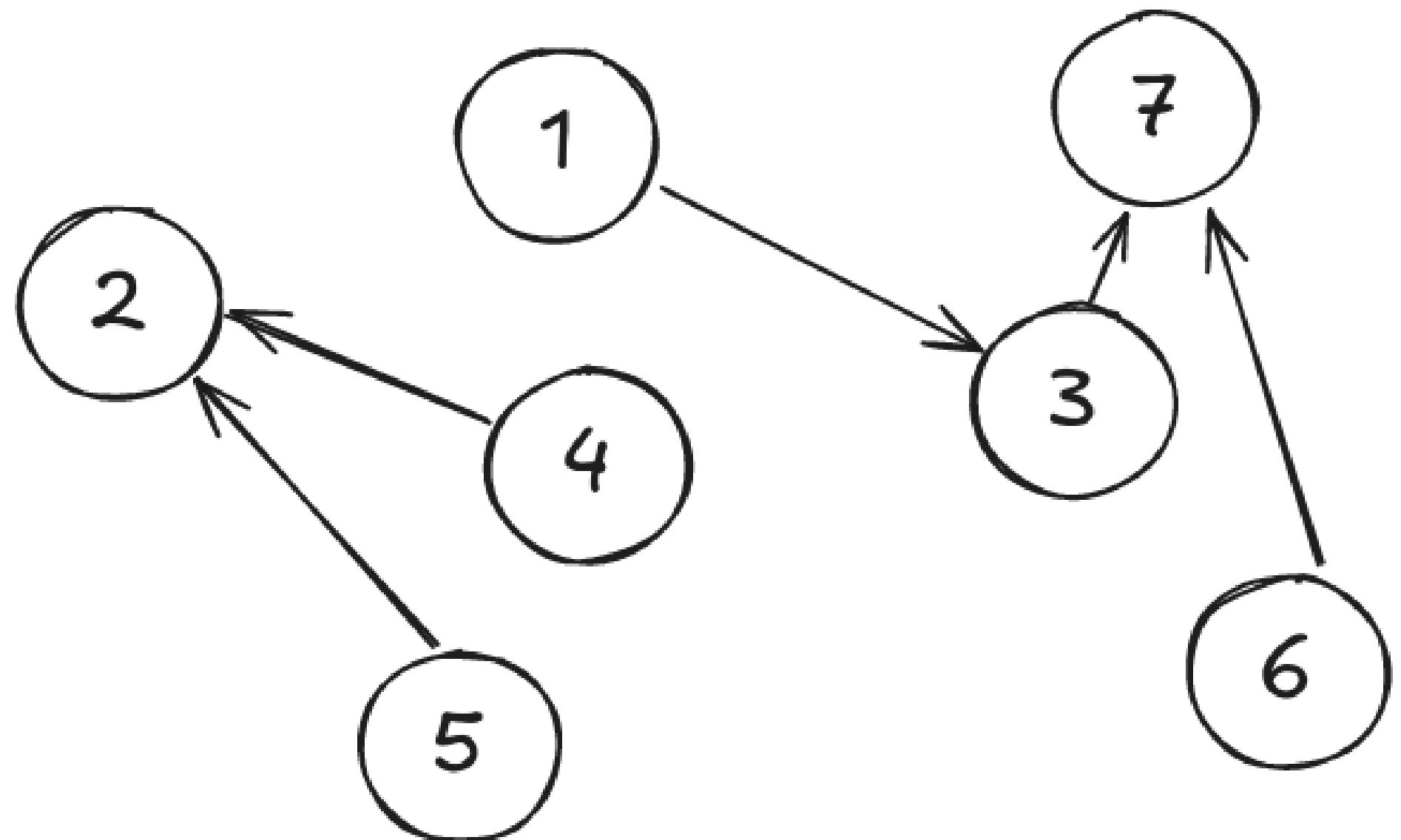
## Ejemplo

- **find(2)**
  - 2
- **union(2,4)**
- **find(4)**
  - 2
- ...



## Ejemplo

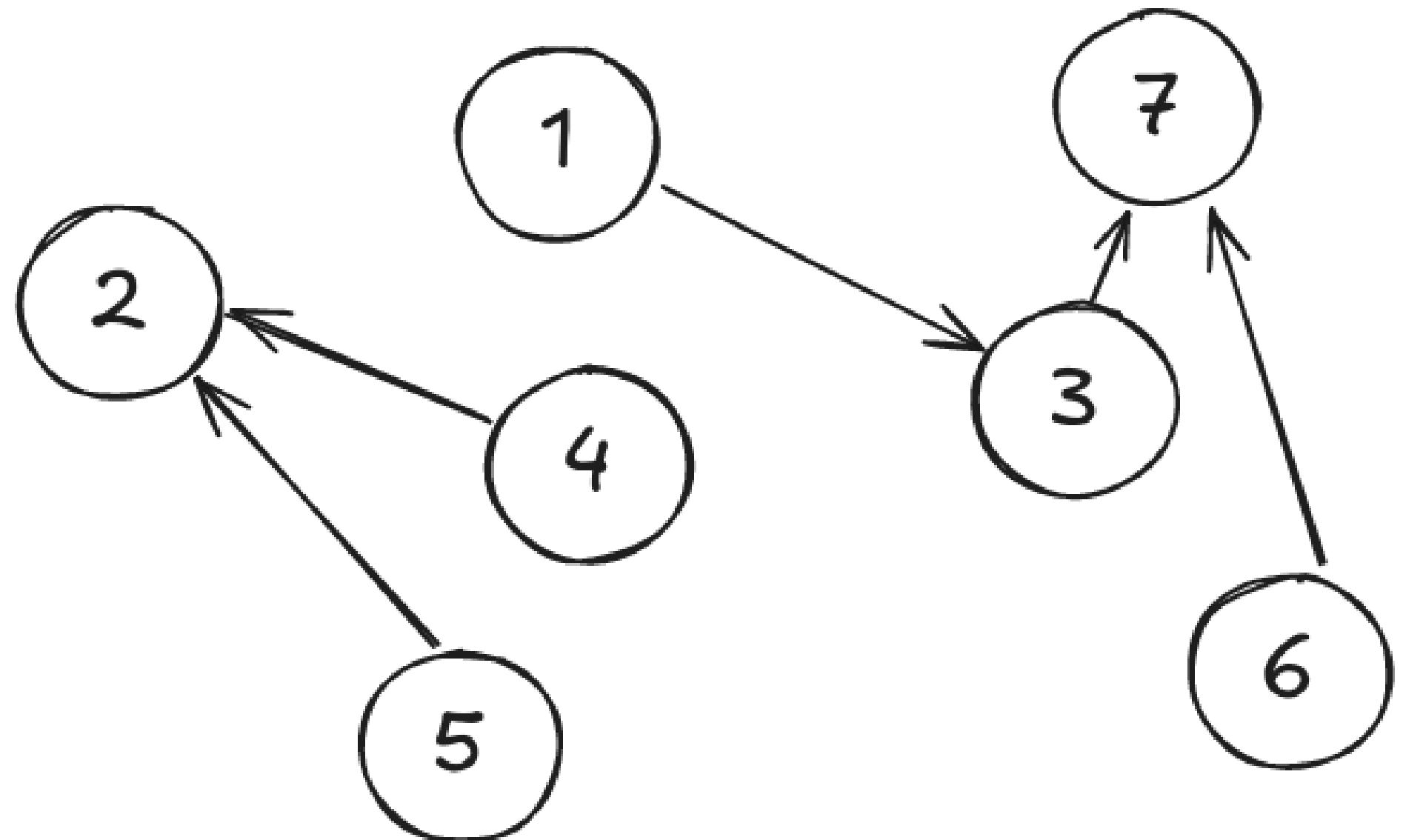
- **find(1)**



## Ejemplo

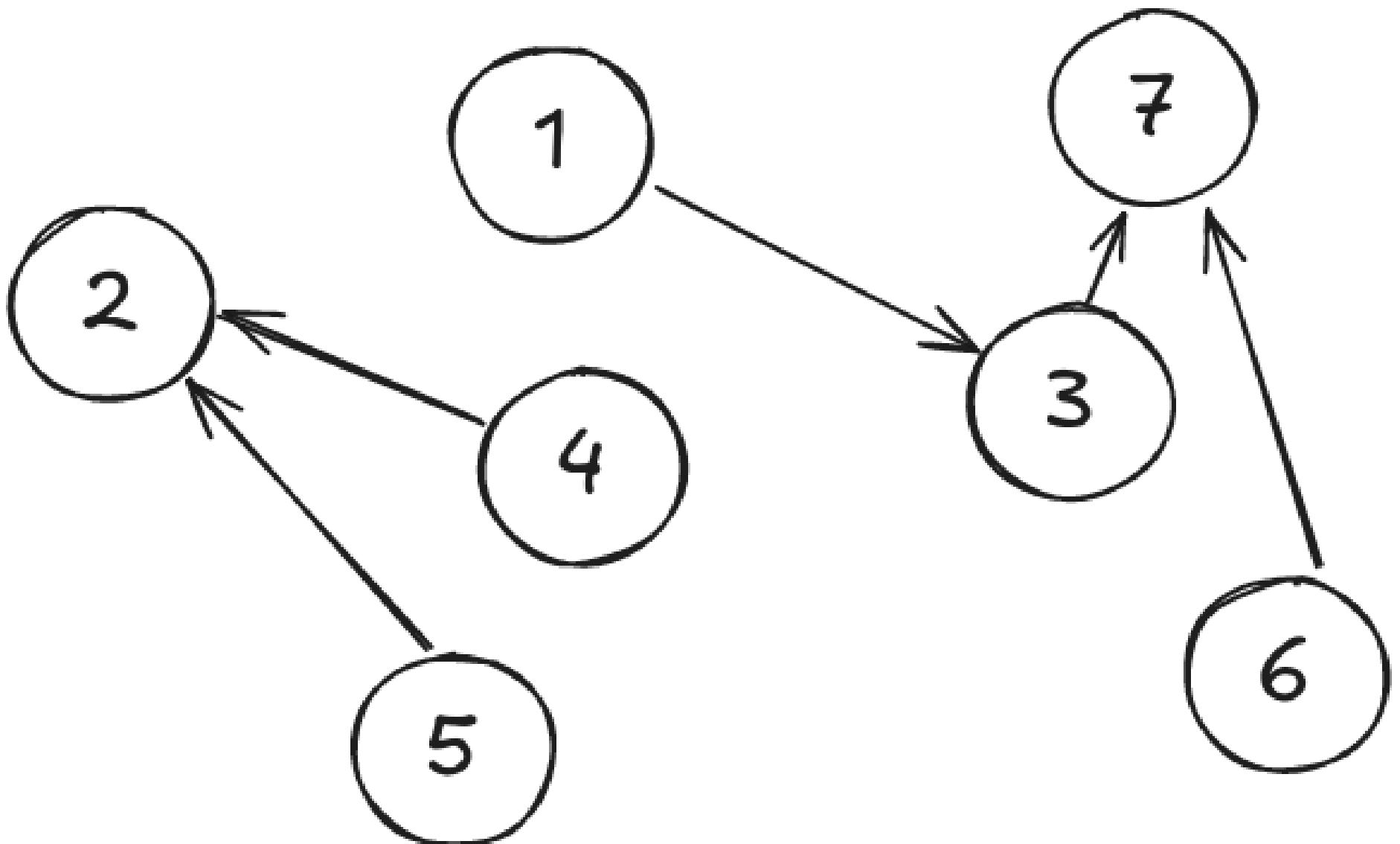
- **find(1)**

- 7



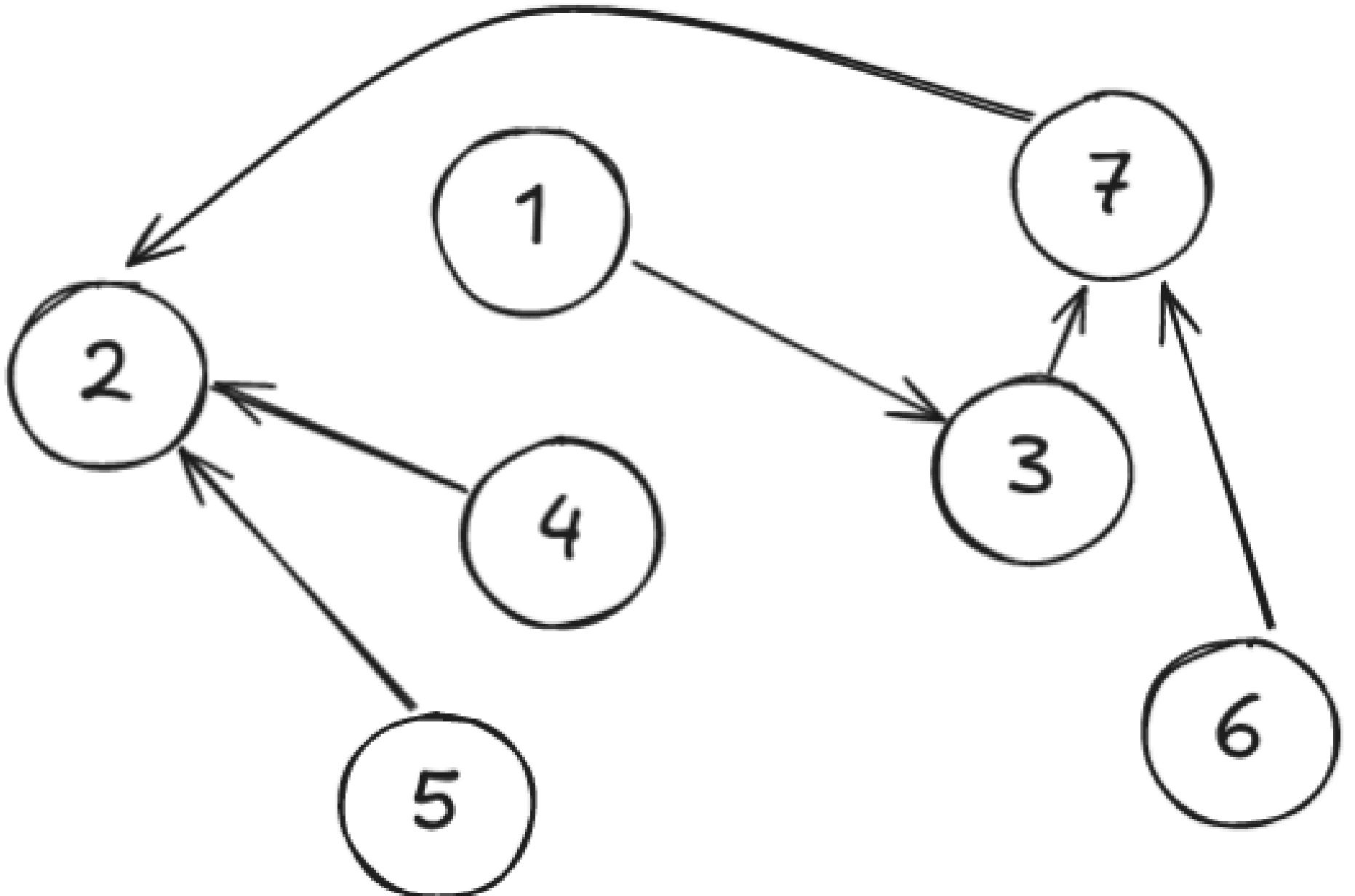
## Ejemplo

- **find(1)**
  - 7
- **union(3,2)**



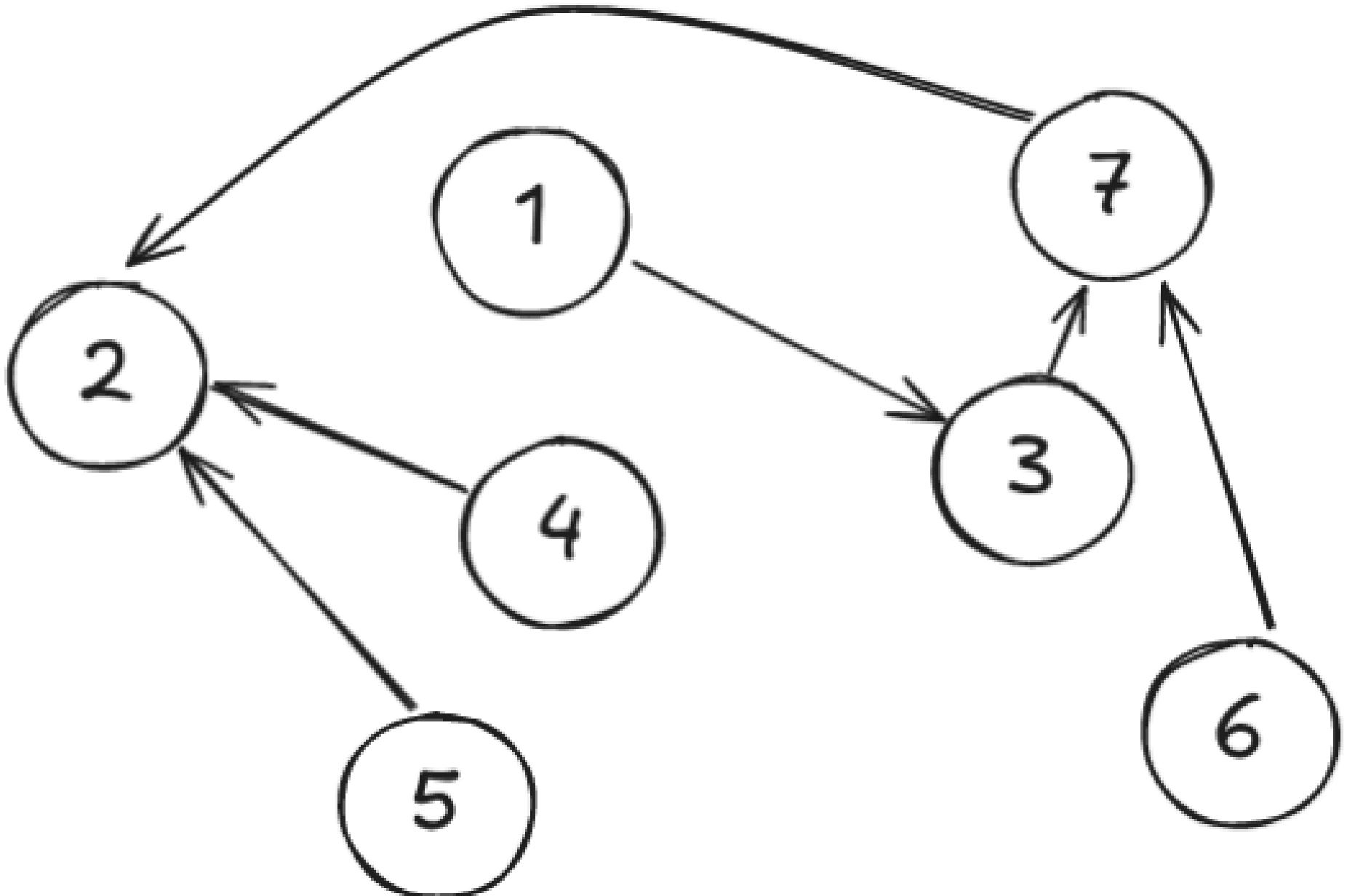
## Ejemplo

- **find(1)**
  - 7
- **union(3,2)**



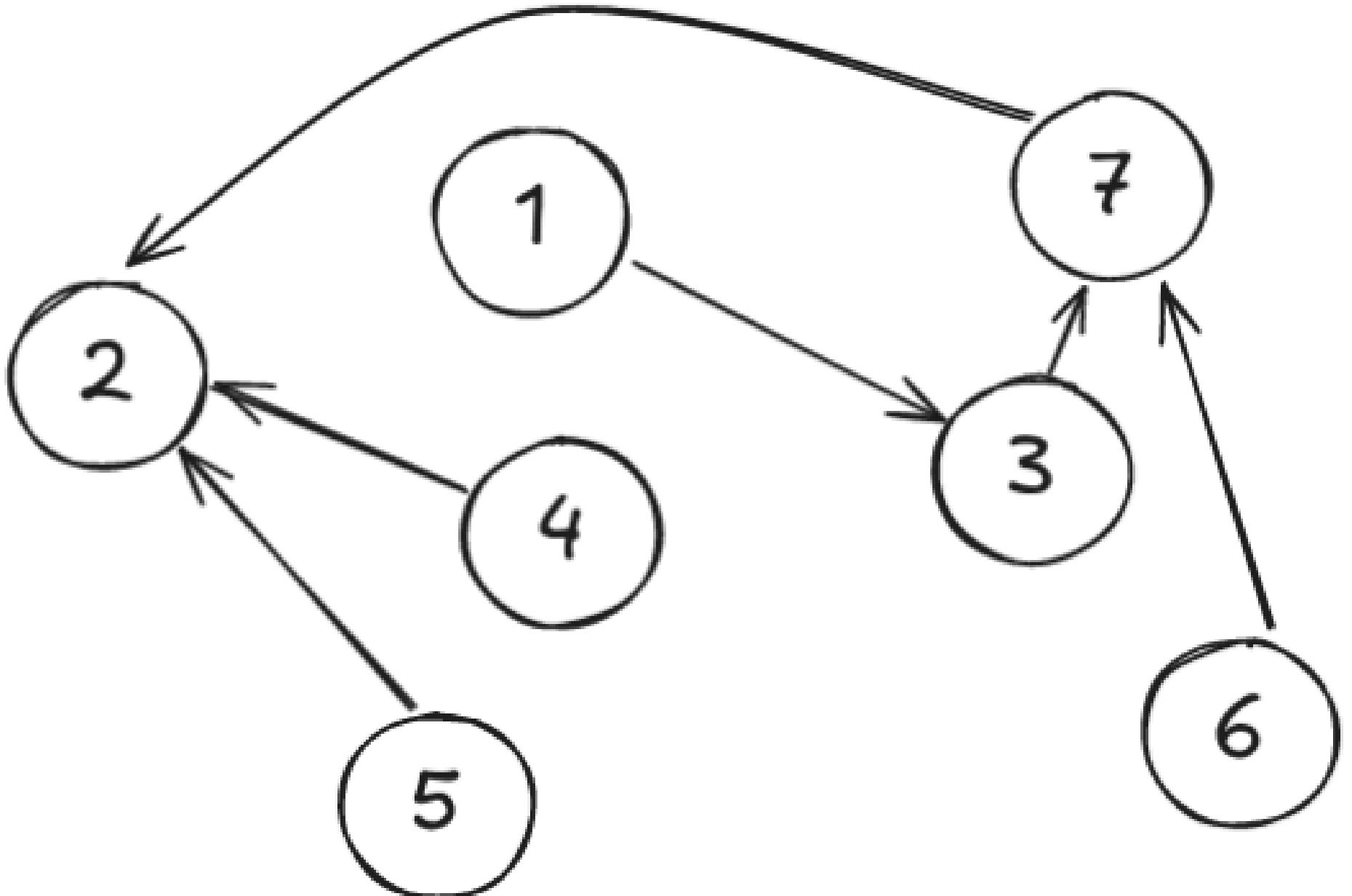
## Ejemplo

- **find(1)**
  - 7
- **union(3,2)**
- **find(1)**



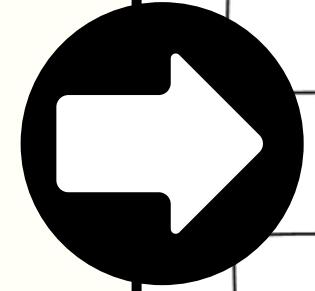
## Ejemplo

- **find(1)**
  - 7
- **union(3,2)**
- **find(1)**
  - 2



# Implementación

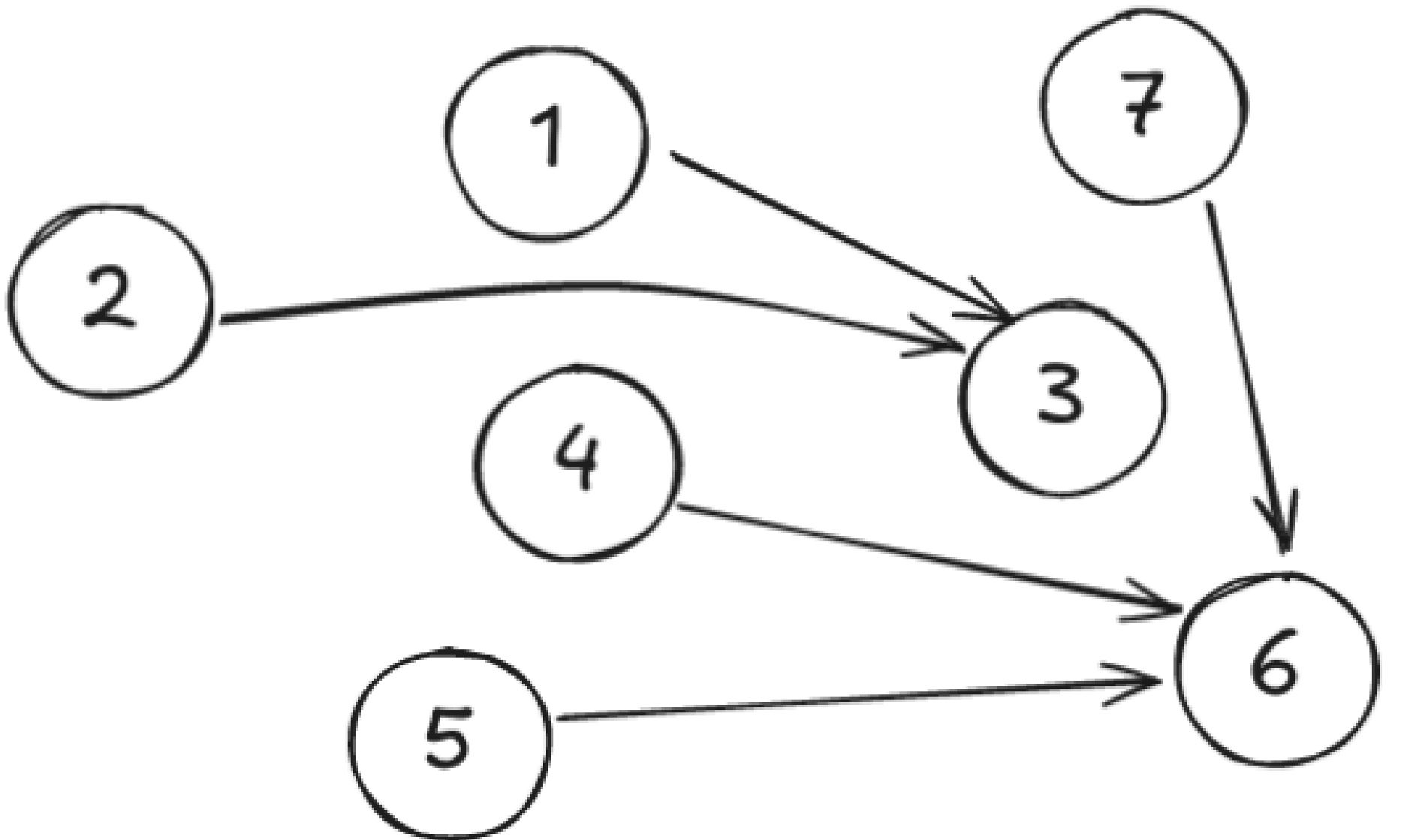
- $O(?)$
- ¿Se puede hacer mas eficiente?
- ¿Se puede implementar “mas fácil”?



```
init():  
    p = new int[n]  
    for i in 1..n:  
        p[i] = i  
  
get(a):  
    return p[a]  
  
union(a, b):  
    a = p[a]  
    b = p[b]  
    for i in 1..n:  
        if p[i] == a:  
            p[i] = b
```

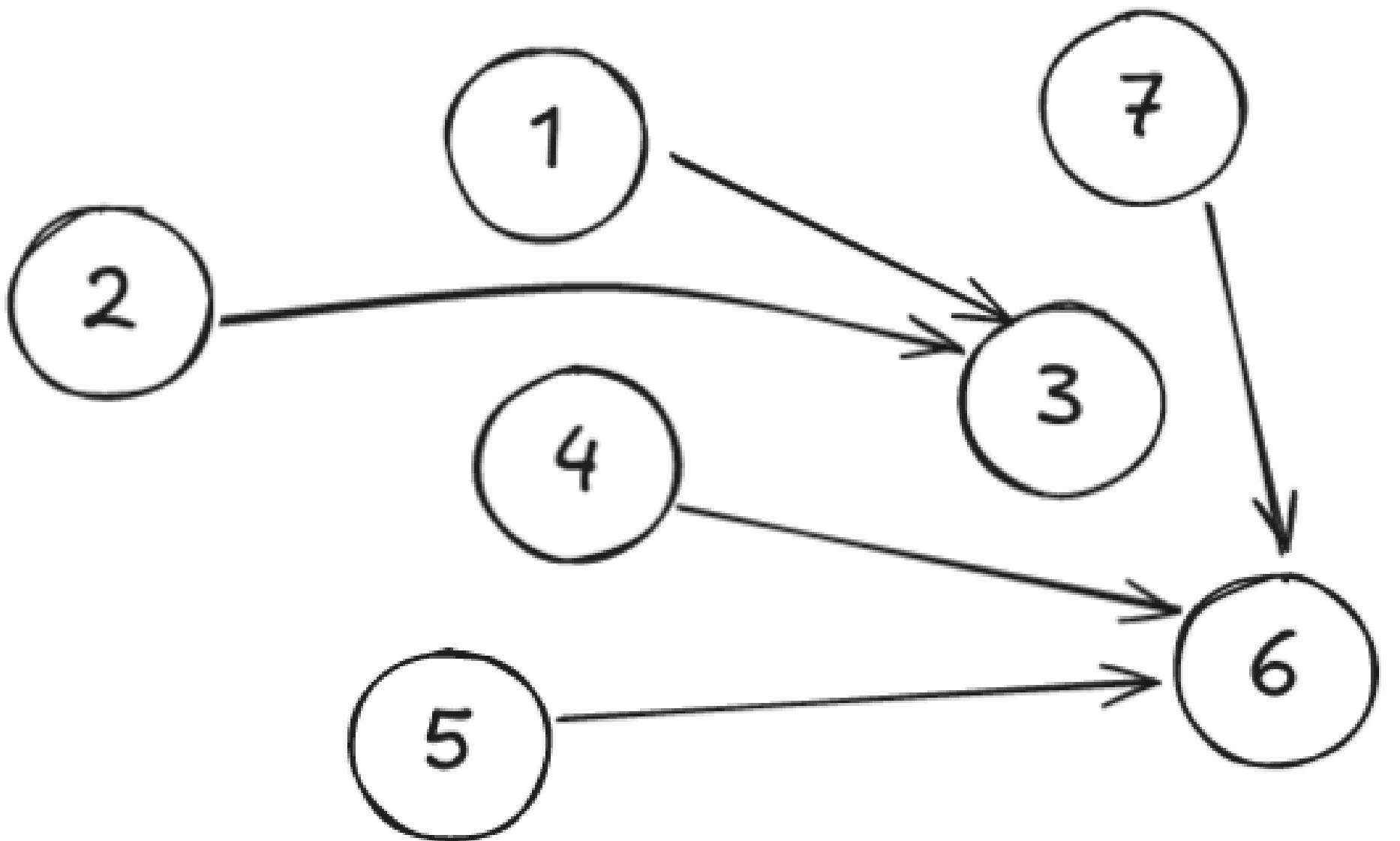
# Problema

- ¿ get(4) ?



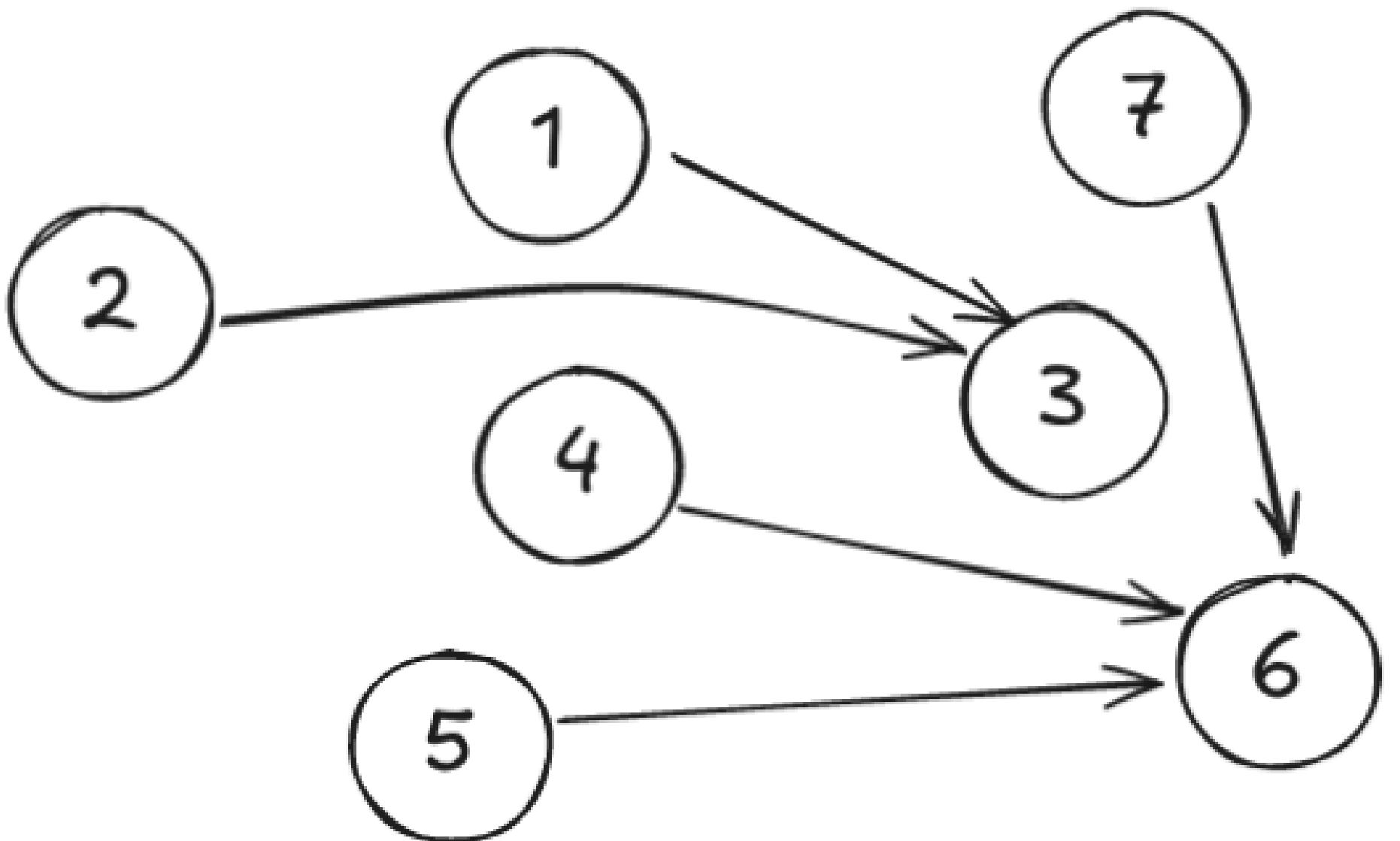
# Problema

- ¿  $\text{get}(4)$  ?
  - $O(1)$



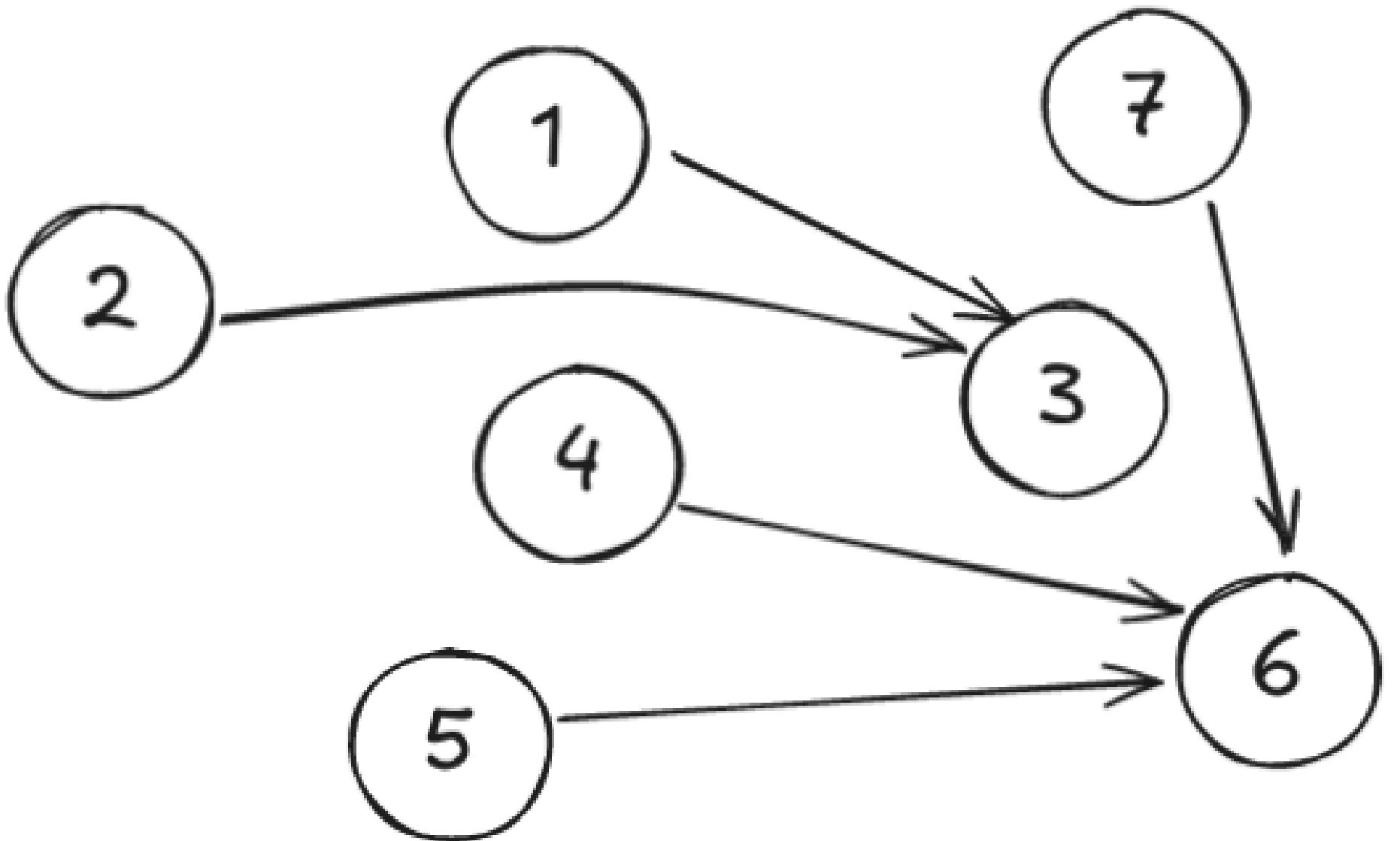
# Problema

- ¿  $\text{get}(4)$  ?
  - $O(1)$
- ¿  $\text{unión}(1,4)$  ?



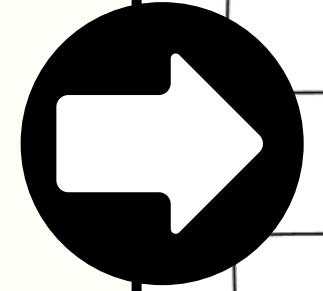
# Problema

- ¿  $\text{get}(4)$  ?
  - $O(1)$
- ¿  $\text{unión}(1,4)$  ?
  - $O(n)$



# Implementación

- O(?)
- ¿Se puede hacer mas eficiente?
- ¿Se puede implementar “mas fácil”?



```
init():
    p = new int[n]
    l = new List[n]
    for i in 1..n:
        p[i] = i
        l[i] = { i }

get(a):
    return p[a]

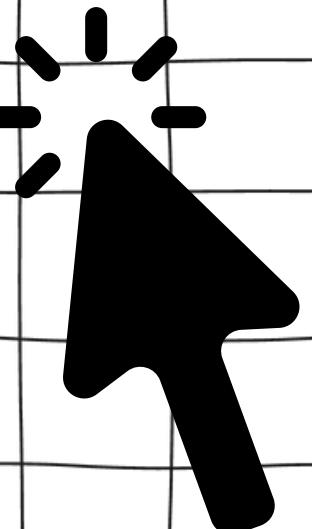
union(a, b):
    a = p[a]
    b = p[b]
    for x in l[a]:
        p[x] = b
    l[b].append(l[a])
```

# La magia



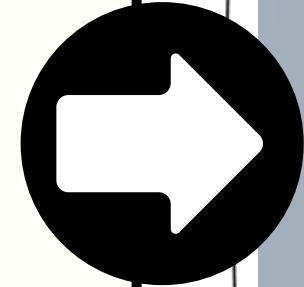
## link-small-to-large

Unir los dos conjuntos con el criterio de cual es mas grande para amortizar la cantidad de elementos que debo cambiar de representante



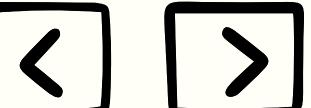
# Implementación

- O(?)
- ¿Se puede hacer mas eficiente?
- ¿Se puede implementar “mas fácil”?



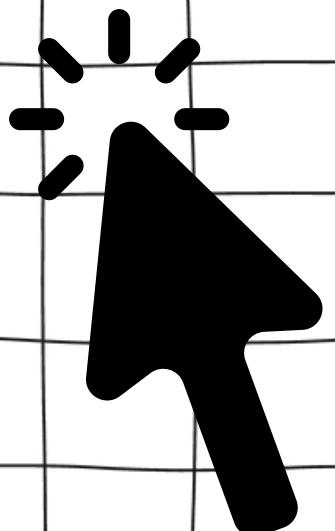
```
union(a, b):  
    a = p[a]  
    b = p[b]  
    if size(l[a]) > size(l[b]):  
        swap(a, b)  
        for x in l[a]:  
            p[x] = b  
        l[b].append(l[a])
```

## La magia



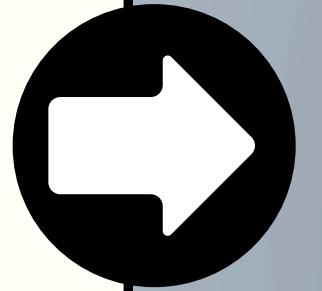
### Path-compression

- Optimiza la operación Find haciendo que los nodos apunten directamente al representante
- Reduce la profundidad del árbol y acelera futuras consultas



# Implementación

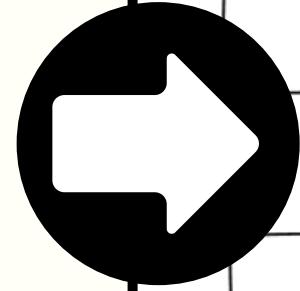
- O(?)
- ¿Se puede hacer mas eficiente?
- ¿Se puede implementar “mas fácil”?



```
get(a):  
    while a != p[a]:  
        a = p[a]  
    return a
```

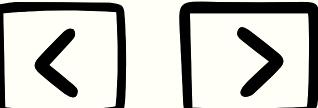
# Implementación

- O(?)
- ¿Se puede hacer mas eficiente?
- ¿Se puede implementar “mas fácil”?



```
get(a):  
    if p[a] != a:  
        p[a] = get(p[a])  
    return p[a]  
  
union(a, b):  
    a = get(a)  
    b = get(b)  
    if size[a]>size[b]:  
        swap(a, b)  
    p[a] = b  
    size[b] += size[a]
```

# Eficiencia



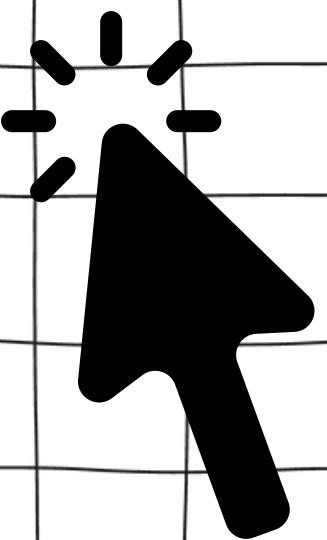
## Efficiency of a Good But Not Linear Set Union Algorithm

ROBERT ENDRE TARJAN

*University of California, Berkeley, California*

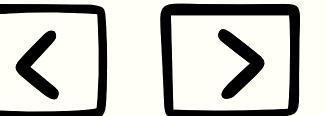
[Tutorial] Proving the inverse Ackermann complexity of Union-Find

By [-is-this-fft-](#), 3 years ago,

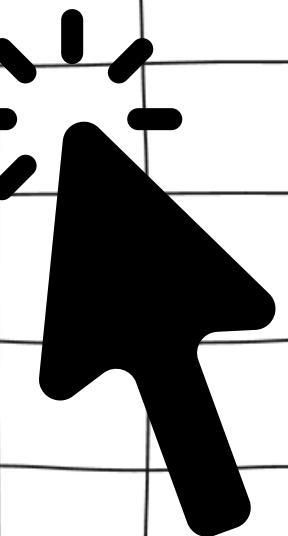


**A disfrutar del poder**

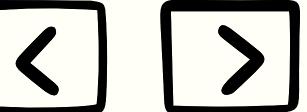
MST



?



# MST



**MST-KRUSKAL( $G, w$ )**

```
1  $A = \emptyset$ 
2 for each vertex  $v \in G.V$ 
3   MAKE-SET( $v$ )
4   sort the edges of  $G.E$  into nondecreasing order by weight  $w$ 
5   for each edge  $(u, v) \in G.E$ , taken in nondecreasing order by weight
6     if FIND-SET( $u$ )  $\neq$  FIND-SET( $v$ )
7        $A = A \cup \{(u, v)\}$ 
8       UNION( $u, v$ )
9   return  $A$ 
```

