6.6: Estructura de Partición: UF-Set

1. Introducción
2. Representación
3. Mejoras en la eficiencia
   - Combinación por rango
   - Compresión de caminos
4. Implementación

Bibliografía

Michael T. Goodrich and Roberto Tamassia. "*Data Structures & Algorithms in Java*" (4th edition), John Wiley & Sons, 2005

(apartado 6 del capítulo 11)

1. Introducción

Relaciones de equivalencia

- Una relación *R* definida en un conjunto *C* es un subconjunto del producto cartesiano $C \times C$ de manera que *aRb* denota que (a, b) ∈ R
- *R* es una **relación de equivalencia** si cumple las siguientes propiedades:
  - ***Reflexiva***: *aRa* para todo $a \in C$.
  - ***Simétrica***: *aRb* si y solo si *bRa*, para todo $a, b \in C$.
  - ***Transitiva***: *aRb* y *bRc* implica *aRc*, para todo $a, b, c \in C$.
- Un conjunto de elementos se puede particionar en clases de equivalencia a partir de la definición de una relación de equivalencia

1. Introducción

UF-Sets

- Los UF-Sets (*Union-Find Set*) son unas estructuras eficientes para determinar las posibles **particiones** de un conjunto
  - Los elementos están organizados en subconjuntos disjuntos
  - El número de elementos es fijo (no se añaden ni se borran)
- Sus **operaciones** características son:
  - Unión (*Union*): unión de dos conjuntos disjuntos
  - Encuentra (*Find*): dado un elemento debe determinar a que conjunto pertenece

1. Introducción

Aplicaciones de UF-Sets

Cada subconjunto se puede identificar por uno de sus miembros

- **Aplicaciones**:
  - Obtención del árbol de recubrimiento de mínimo peso en un grafo no dirigido (Kruskal)
  - Componentes conexas de un grafo no dirigido
  - Equivalencia entre autómatas finitos

2. Representación de UF-Sets

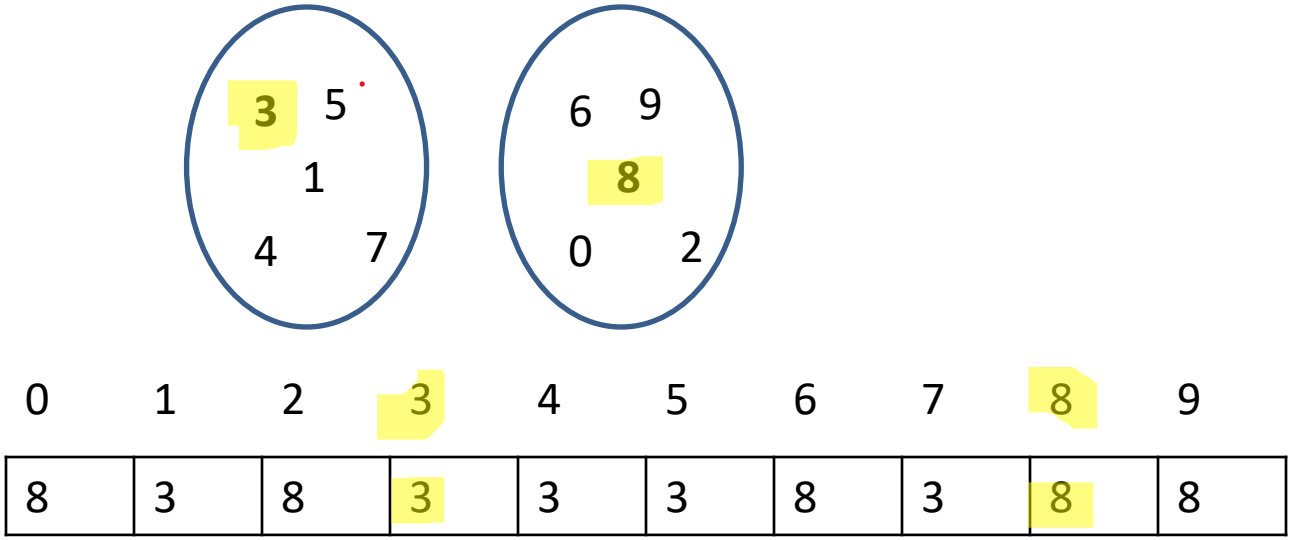
Operaciones sobre UF-Sets

```
public interface UFSet
{
  /** Devuelve el identificador del conjunto
   *  al que pertence el elemento x
   */
  int find(int x);

  /** Une dos conjuntos Identificados por
   *  x e y
   */
  void union(int x, int y);
}
```

2. Representación de *UF-Sets*

OJO! Una mala representación

Se usa un *array* donde el índice representa al elemento del conjunto, y el contenido es el identificador de la clase a la que pertenece.

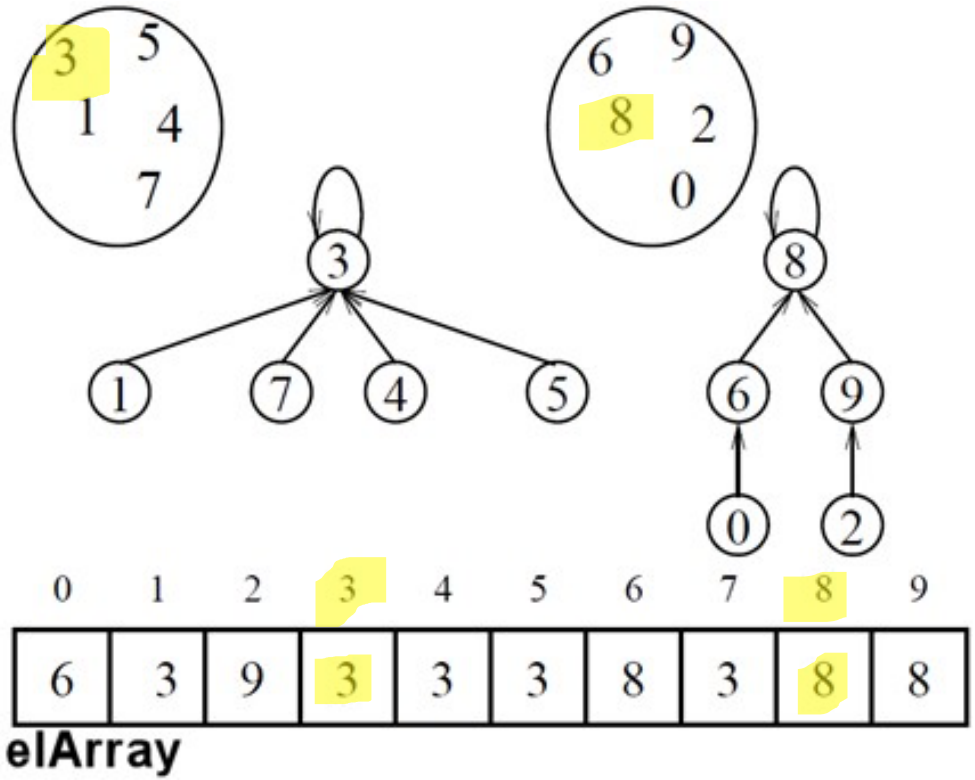


La operación Find es sencilla find(i)= elArray[i], pero la operación Union es muy costosa porque hay que recorrer todo el array cambiando la clase de los elementos que vamos a asignar al nuevo conjunto resultado de la unión.

2. Representación de UF-Sets

Representación en bosque

- Cada subconjunto se guarda como un árbol:
  - Los nodos del árbol son los elementos del subconjunto
  - En cada nodo guardamos una referencia al padre
  - El elemento raíz del árbol se usa para representar el subconjunto

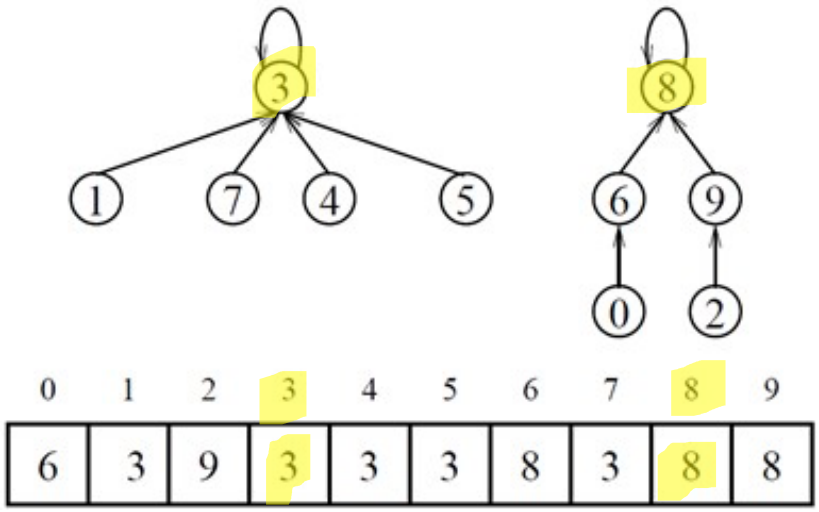


Se usa un *array* donde el índice representa al elemento del conjunto y su contenido es:

- *elArray*[*i*] es el padre del elemento *i*
- Si *elArray*[*i*]=*i*, *i* es la raíz de un árbol

2. Representación de UF-Sets

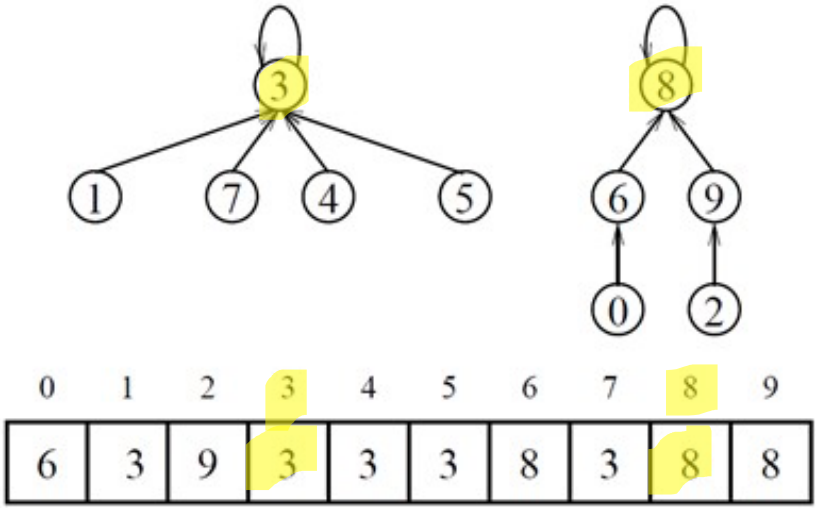
Operaciones sobre UF-Sets



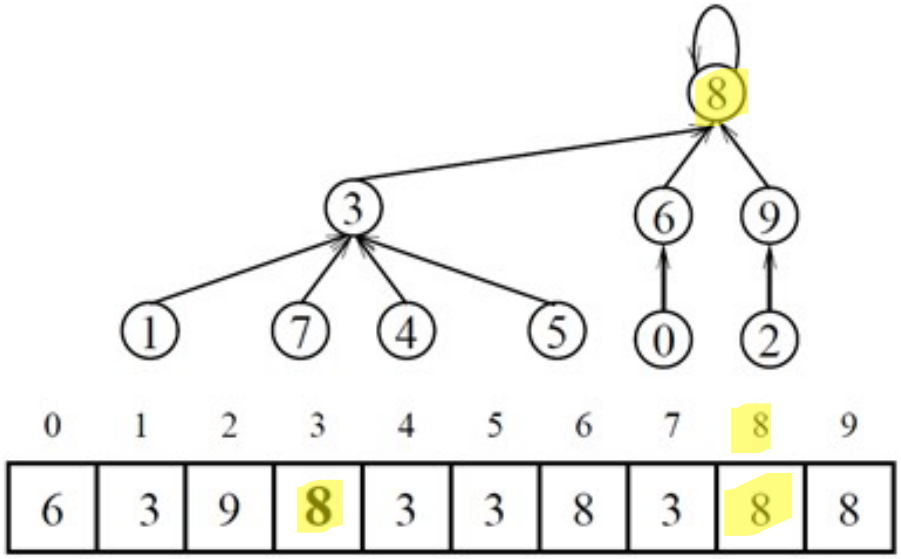
find(0) = 8
find(9) = 8
find(4) = 3
find(3) = 3

2. Representación de UF-Sets

Operaciones sobre UF-Sets



union(3, 8)



3. Mejoras en la eficiencia

Introducción

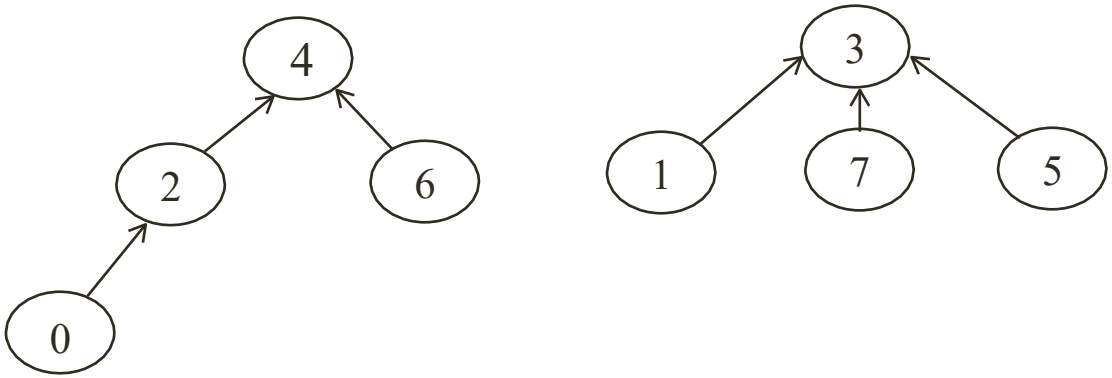
- El método *find* puede tener un coste lineal en función del número de nodos si los árboles están desequilibrados (árboles como listas)
- Las mejoras en el coste se basan en reducir la altura de los árboles:
  - Combinar por rango
  - Compresión de caminos
- Con estas mejoras el coste amortizado de las operaciones es prácticamente constante
  - El coste es una inversa de la función de Ackermann, que crece muy lentamente (ejemplo: $(2^{65536}) = 5$)

3. Mejoras en la eficiencia

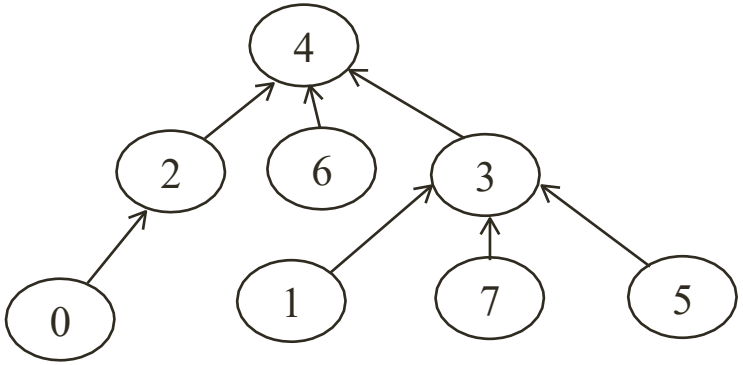
Combinación por altura

- En la operación *union* hacemos que la raíz del árbol de menor altura apunte a la raíz del de mayor altura
  - Si las alturas de los dos árboles a unir son distintas, la altura del árbol resultante será la del árbol de mayor altura
  - Si ambos árboles tienen la misma altura, la altura del árbol resultante será una unidad mayor que la de los árboles a unir
- Para ello es necesario guardar la altura de cada árbol
  - Este valor se puede mantener en el propio vector, en el nodo asociado a la raíz de cada árbol, pero con signo negativo
  - Entonces, si elArray[i]<0, i es la raíz de un árbol y, además $|elArray[i]| - 1$ será la altura de dicho árbol

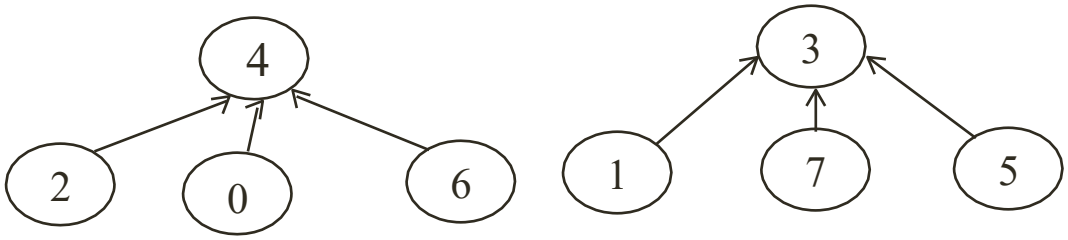
Si queremos unir estos dos árboles: unión(4,3)…..



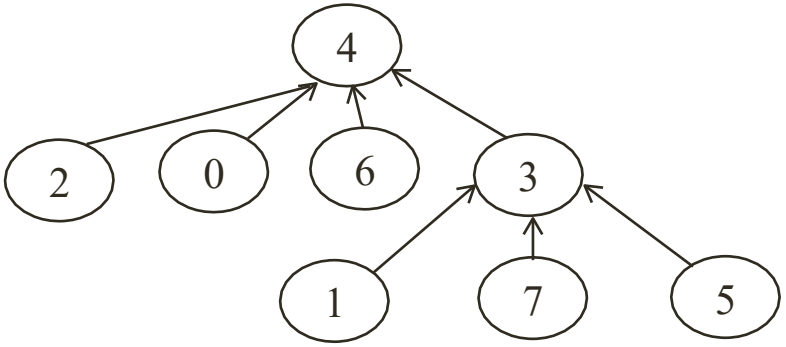
…hacemos que el árbol de menor altura cuelgue del de mayor altura.



Si los árboles a unir tienen la misma altura: unión(4,3)…..

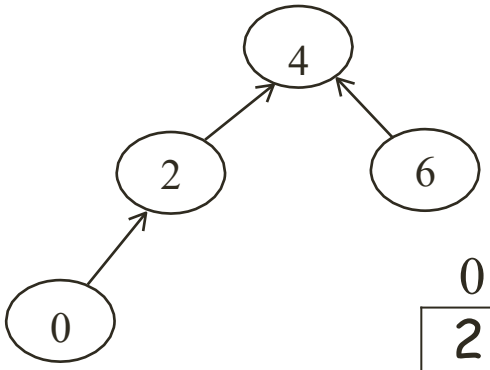


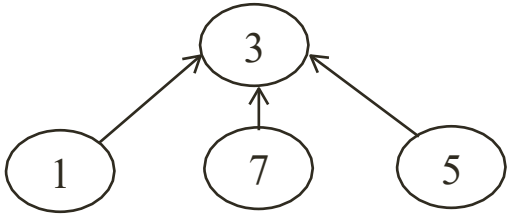
… la altura del árbol resultante será una unidad mayor que la de los árboles a unir.



3. Mejoras en la eficiencia

Combinación por altura. Ejemplo:

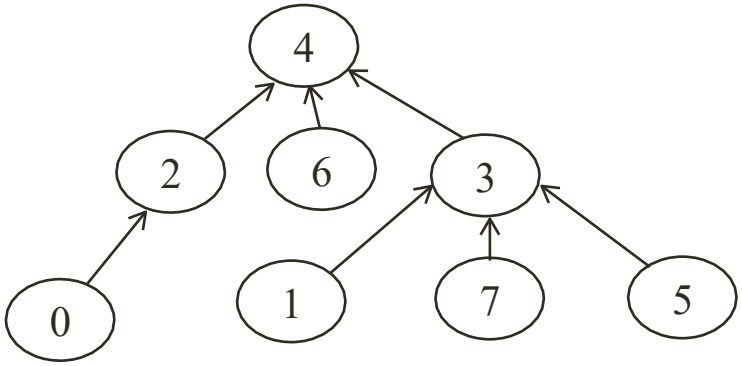




0	1	2	3	4	5	6	7
2	3	4	-2	-3	3	4	3

El 3 es la raíz de un árbol de altura 1

El 4 es la raíz de un árbol de altura 2



- Al unir ambos árboles, colgamos el de menor altura (el 3) del más alto (el 4)

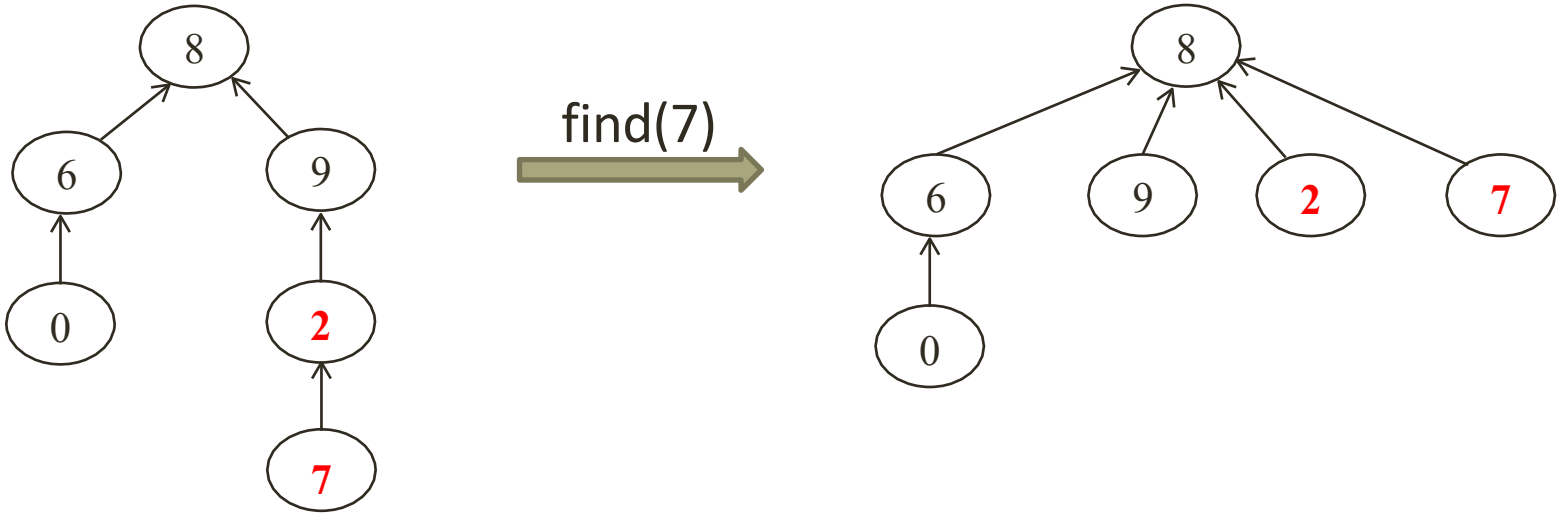
0	1	2	3	4	5	6	7
2	3	4	4	-3	3	4	3

3. Mejoras en la eficiencia

Compresión de caminos

- En las operaciones de búsqueda (*find*) hacemos que cada nodo por el que pasemos apunte directamente a la raíz del árbol.

Ejemplo:



3. Mejoras en la eficiencia

Combinación de las dos estrategias

- La compresión de caminos no es totalmente compatible con la combinación por altura, ya que la compresión de caminos puede reducir la altura del árbol.
- Por ello se usa la Unión por Rango que se basa en una estimación de la altura. El algoritmo Union es el mismo, pero el valor que queda almacenado como altura no tiene por qué ser exactamente la altura, es una estimación (que es cota superior de la altura)

4. Implementación. *Atributos y constructor*

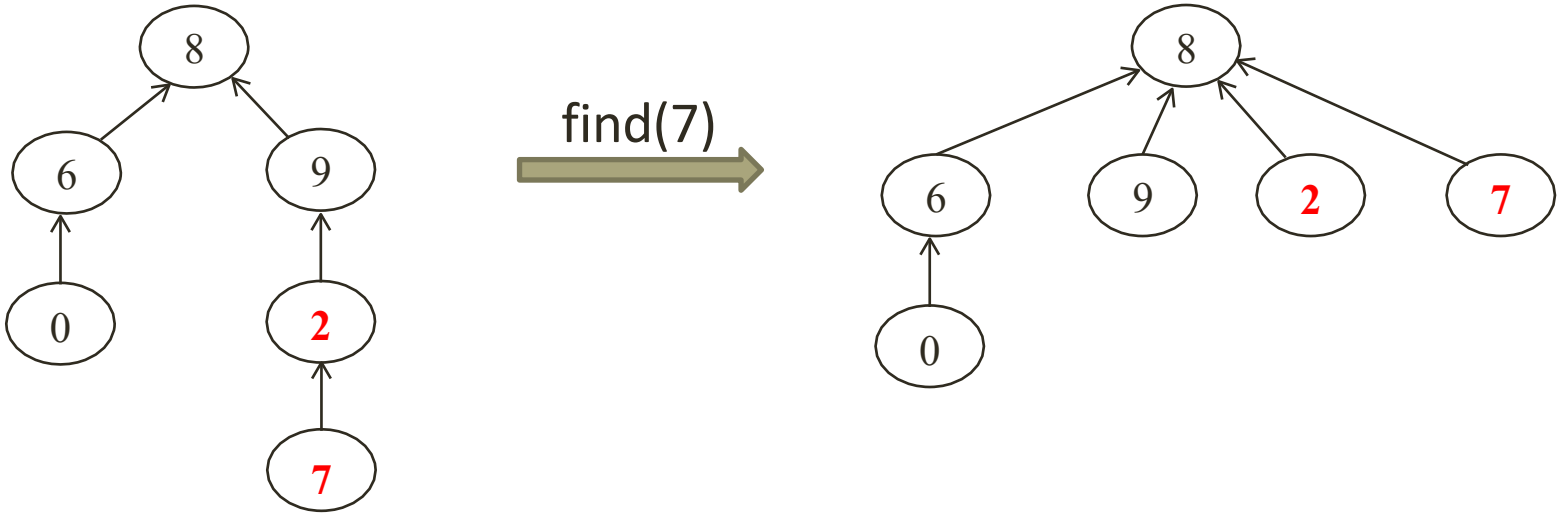
```
public class ForestUFSet implements UFSet {
  // Array de int que representa un bosque de
  // arboles, de forma que
  // si elArray[i] > 0 : referencia al padre
  // si elArray[i] < 0 : i es el identificador de una clase
  // (la raiz del arbol que representa a la clase) y
  // |elArray[i]| - 1 es su altura (rango)

  protected int talla; // Núm. Elementos

  protected int elArray[];


  // Crea un UFSet de talla n. Al principio se crean n
  // árboles distintos de un solo elemento (altura 0)
  public ForestUFSet(int n) {
    talla = n;
    elArray = new int[talla];
    for (int i = 0; i < talla; i++)
      elArray[i] = -1; // Altura 0
  }
```

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

4. Implementación

Búsqueda de elementos

```
/** Devuelve el identificador del conjunto al que pertence
  * el elemento x, ademas de enlazar todo los elementos
  * del camino visitado directamente con la raíz */
  public int find(int x) {
      if (elArray[x]<0) return x; // Raíz del conjunto
      // Compresión del camino
      else return elArray[x]=find(elArray[x]);
      }
```



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

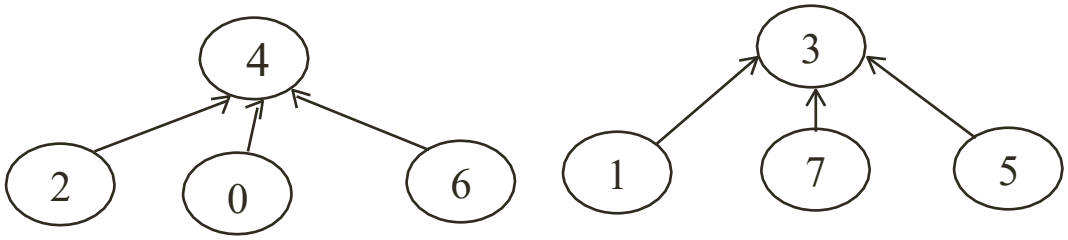
0	1	2	3	4	5	6	7	8	9
6	-1	8	-1	-1	-1	8	8	-4	8

4. Implementación

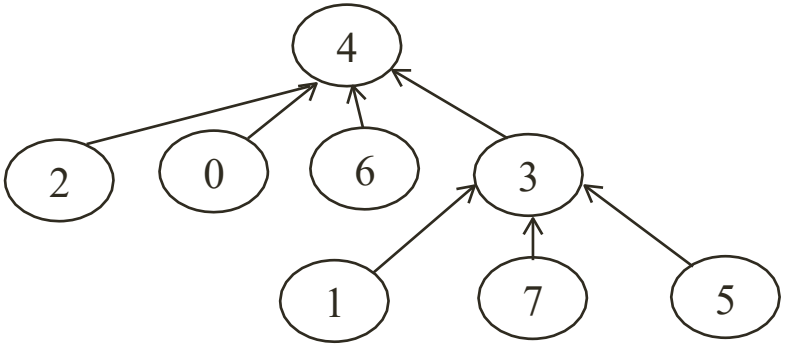
Unión por Rango de conjuntos

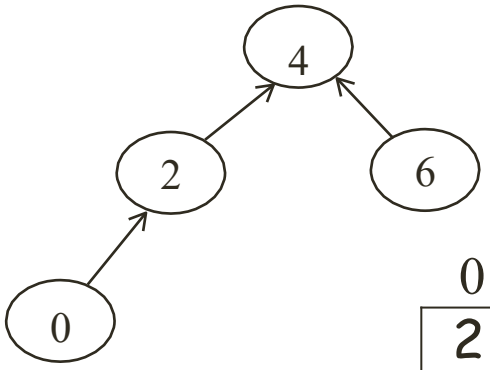
```
/** Une los conjuntos identificados por x e y. Los elementos x e
y han de ser identificadores de sus respectivas clases.
*/
public void union (int x, int y) {

  if (elArray[x]==elArray[y]){          // Alturas iguales
     elArray[x] = y;                    // Colgamos x de y
     elArray[y]--;          // Incrementamos la altura de y
  } else if (elArray[x] < elArray[y]) {
     elArray[y] = x;                    // Colgamos y de x
  } else {
     elArray[x] = y;                    // Colgamos x de y
  }
  }
```

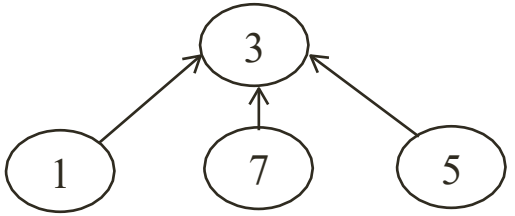
Si los árboles a unir tienen la misma altura: unión(4,3)…..



… la altura del árbol resultante será una unidad mayor que la de los árboles a unir.



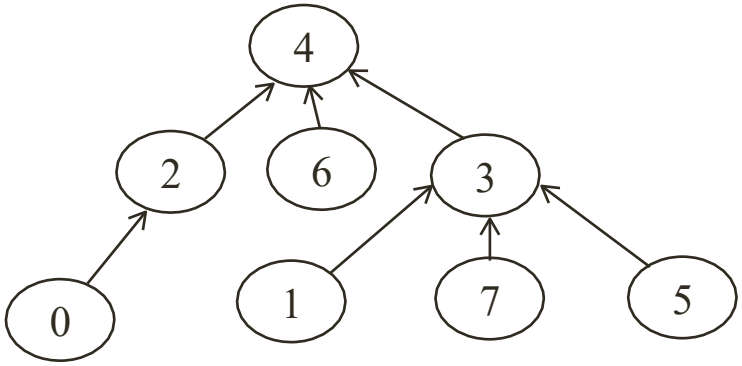




0	1	2	3	4	5	6	7
2	3	4	-2	-3	3	4	3

El 3 es la raíz de un árbol de altura 1

El 4 es la raíz de un árbol de altura 2



- Al unir ambos árboles, colgamos el de menor altura (el 3) del más alto (el 4)

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

Ejercicio

Sea la siguiente representación de un UF-Set:

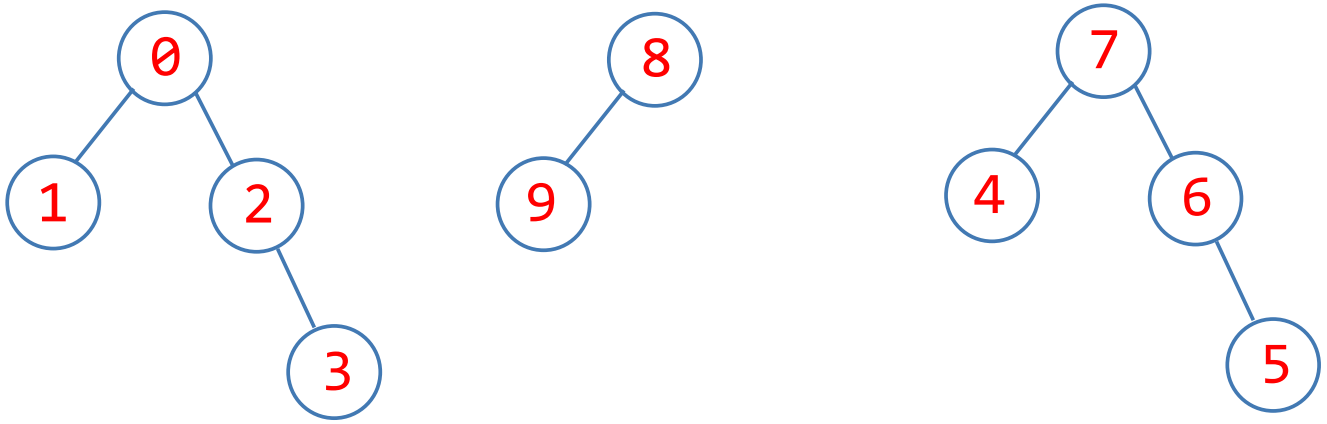
0	1	2	3	4	5	6	7	8	9
-3	0	0	2	7	6	7	-3	-2	8

a) Dibujar el bosque de árboles que contiene.

b) Teniendo en cuenta que se implementa la fusión por rango y la compresión de caminos, indicar cómo irá evolucionando dicha representación tras la ejecución de las instrucciones siguientes. **Nota:** al unir dos árboles con la misma altura (o rango), el primer árbol deberá colgar del segundo.

find(3)
find(5)
union(0,7)
union(7,8)

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



	0	1	2	3	4	5	6	7	8	9
find(3)	-3	0	0	**0**	7	6	7	-3	-2	8
find(5)	-3	0	0	0	7	**7**	7	-3	-2	8
union(0, 7)	**7**	0	0	0	7	7	7	**-4**	-2	8
union(7, 8)	7	0	0	0	7	7	7	-4	**7**	8