Tema 3. CONJUNTOS DISJUNTOS (Relaciones de Equivalencia)

Estructuras de Datos y Algoritmos II

Grado en Ingeniería Informática

M José Polo Martín

mjpolo@usal.es

Universidad de Salamanca

curso 2022-2023

VNiVERSiDAD D SALAMANCA
CAMPUS DE EXCELENCIA INTERNACIONAL

800 AÑOS 1218~2018

DEPARTAMENTO DE iNFORMÁTiCA Y AVTOMÁTiCA

Contenido

1 Relación de Equivalencia

- Concepto matemático definido sobre un conjunto basado en la idea de representación de relaciones entre sus elementos (ciudades en una misma comunidad, colores en una imagen, ...)

Definición

Se define una **relación de equivalencia** R sobre un conjunto U si para todo par de elementos a, b pertenecientes a U, $a\ R\ b$ es verdadera o falsa

Propiedades

1. **Reflexiva.** $\forall\ a \in U,\ a\ R\ a$
2. **Simétrica.** $\forall\ a, b \in U,\ a\ R\ b$ *si y sólo si* $b\ R\ a$
3. **Transitiva.** $\forall\ a, b, c \in U,\ Si\ a\ R\ b\ y\ b\ R\ c \Rightarrow a\ R\ c$

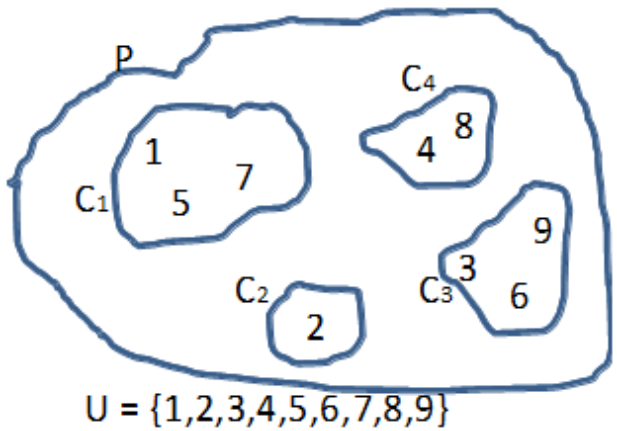
Clase de equivalencia de un elemento $a \in U$

Subconjunto de U que tiene todos los elementos relacionados con a.

Partición de U

Conjunto de todas las clases de equivalencia definidas sobre U según la relación R. Es decir, conjunto de **subconjuntos disjuntos** cuya unión es el conjunto U.

- Todo elemento de U aparece exactamente en una clase de equivalencia.
- $a\ R\ b \Rightarrow$ a y b están en la misma clase de equivalencia



Aplicaciones

- Procesamiento de imágenes digitales
  - Imagen en color o en escala de grises compuesta por una matriz de pixels
    - Clases de equivalencia: regiones continuas y del mismo color
    - Relación de equivalencia: dos pixels a y b están relacionados si tienen el mismo color y son adyacentes en la imagen
  - Operación rellenar una región de color:
    - Buscar la clase de equivalencia del punto sobre el que se aplica el rellenado
    - Después de la operación puede haber cambio en las clases de equivalencia
- **Tema 4. Grafos**
  - Algoritmo de Kruskal
  - Estudio de la conectividad

2 Nivel abstracto o de definición

Partición de U

Una estructura de datos **Partición** es la colección de conjuntos disjuntos (disjoint-set) entre sí que forman las clases de equivalencia, según alguna relación de equivalencia R, con los elementos de un cierto conjunto universal $U = \{x_1, x_2, ..., x_n\}$

$$P = \{C_1, C_2, ..., C_k\}\ \forall i, j \in \{1, 2, ..., k\} \mid i \neq j \Rightarrow C_i \bigcap C_j = \varnothing$$

Cada conjunto C_i representa una **clase de equivalencia** según alguna **relación R**

Operaciones

- Creación inicial de las clases de equivalencia (conjuntos $C_1, C_2, ..., C_n$)
- Añadir una relación de equivalencia entre dos elementos no relacionados:

  si x_i y x_j no están en la misma clase de equivalencia (pertenecen a conjuntos C_i y C_j diferentes) se combinan las dos clases de equivalencia en una clase de equivalencia nueva, preservando que todos los conjuntos de la partición son disjuntos
- Encontrar a que clase de equivalencia pertenece un elemento x_i de U

Observaciones

- No se realizan operaciones comparando los valores relativos de los elementos, solo se requiere conocer su localización, su clase de equivalencia
- La operación búsqueda devuelve el nombre de la clase de equivalencia o conjunto al que pertenece un elemento:
  - Este nombre es bastante arbitrario
  - Es importante que devuelva el mismo nombre para todos los elementos que pertenecen a la misma clase de equivalencia
  - Puede elegirse como representante un miembro cualquiera del conjunto
- La operación que crea una nueva relación de equivalencia a partir de otras dos es dinámica, pues durante el proceso los conjuntos cambian

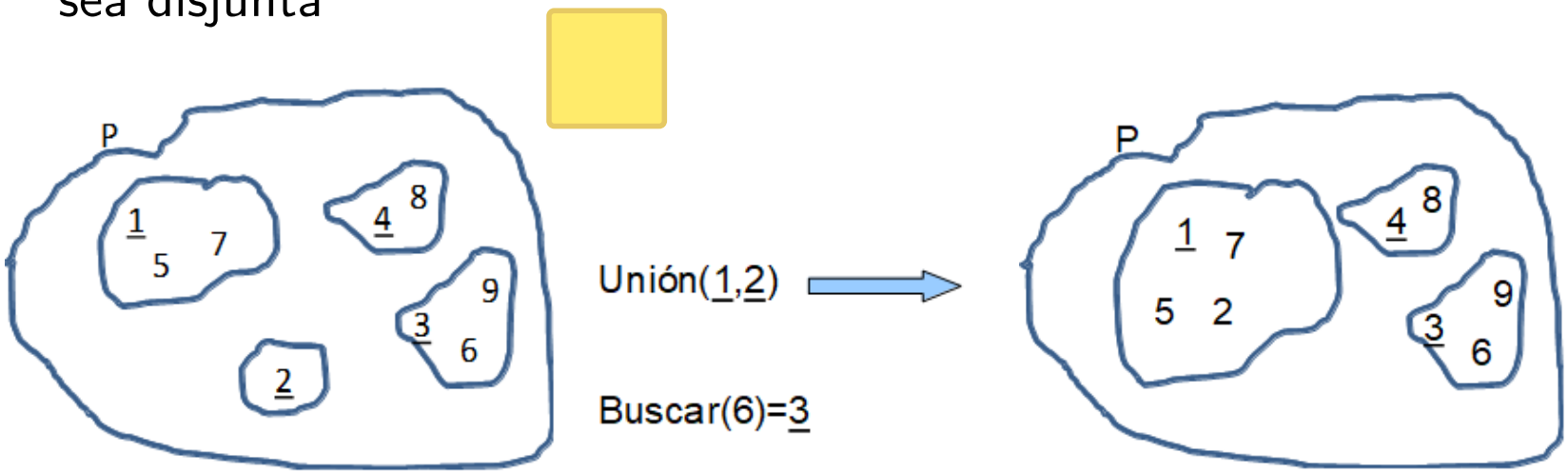
Situación inicial

Todas las relaciones son falsas, excepto las reflexivas. Colección de n conjuntos, cada uno con un elemento diferente

$$n \text{ conjuntos disjuntos: } C_i \bigcap C_j = \varnothing$$

Especificación de Operaciones

- **crear(x_i):** Crea un nuevo conjunto cuyo único miembro es x_i. Se requiere que x_i no esté en ningún otro conjunto de la estructura
- **buscar(x_i):** Devuelve la clase de equivalencia a la que pertenece x_i, es decir, el nombre del conjunto disjunto, que será uno cualquiera de sus elementos, el elegido como su representante
- **unión(x_i, x_j):** Establece una relación de equivalencia entre los elementos x_i y x_j. Es necesaria la unión de las clases de equivalencia a que pertenecen x_i y x_j, sean C_i y C_j, formando una nueva. Se requiere la eliminación de los conjuntos C_i y C_j , para que la colección sea disjunta



3 Nivel de representación o implementación

- Simplificamos el problema suponiendo que los elementos del conjunto U, sobre el que se definen las relaciones de equivalencia, están numerados de 1 a N. Si no es así habrá que definir una función biyectiva que realice la traducción entre U y el conjunto $\{1, 2, ..., N\}$

$$U = \{1, 2, ..., N\}$$

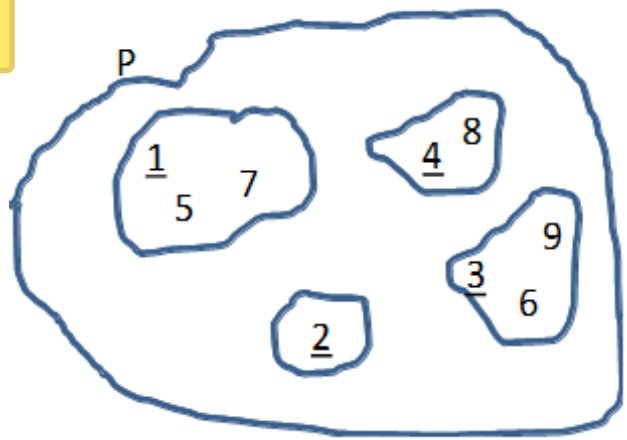
- **Nivel de Representación**
  1. Mediante matrices coste elevado
  2. Mediante listas también muy elevado comportamiento lineal
  3. Mediante árboles mejora

3.1 Representación mediante MATRICES

- La partición se representa con una matriz unidimensional de tamaño N donde: en la definición de datos todos son enteros por eso hay que especificar
  - los índices representan los elementos del conjunto U
  - cada celda almacena el nombre de la clase de equivalencia del elemento correspondiente

Algorithm declaraciones básicas

```
1: constantes
2:   N = 100
3: tipos
4:   tipoElemento = entero
5:   tipoConjunto = entero
6:   tipoPartición = matriz[1, . . . , N] de tipoConjunto
7: tipos
```



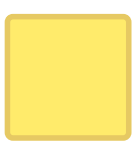
Variable P de tipo partición

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

Operaciones

- La operación **crear(x)** que inicia la estructura a una situación inicial donde todas las relaciones son falsas excepto las reflexivas toma un tiempo de O(N), pero solo se aplicará una vez
- La operación **buscar(x)** devuelve el contenido de la celda **x** del array, está claramente en un $O(1)$
- La operación **unión(x,y)** es $O(N)$: si **x** está en clase C_i e **y** en clase C_j, se recorre la matriz cambiado todo i a j. Esta operación es previsible que aparezca con bastante frecuencia y el coste puede ser excesivo para muchas aplicaciones

La **implementación** de todas estas operaciones es **inmediata** y se deja como **ejercicio**



Análisis de una secuencia arbitraria de operaciones búsqueda/unión

- No sabemos el orden en que se van a presentar estas operaciones, pero no habrá más de $N-1$ uniones, ya que entonces todos los objetos están en el mismo conjunto
- Una secuencia de $N-1$ uniones puede requerir un tiempo en $O(N^2)$
- Si hay n operaciones de búsqueda requerirán un tiempo $O(n)$
- Si n y N son comparables (ocurre en muchas aplicaciones), la secuencia de operaciones requiere un tiempo que se encuentra en $O(N^2)$
- Procesamiento imágenes:
  - Se aplicará unión si un píxel tiene al lado otro del mismo color (bastante frecuente)
  - Aplicar unión para cada píxel de una imagen de $800x600 \Rightarrow (800 * 600)^2 \approx$ ¡ 230 mil millones de comparaciones!
- Coste excesivo para muchas aplicaciones

3.2 Representación mediante LISTAS

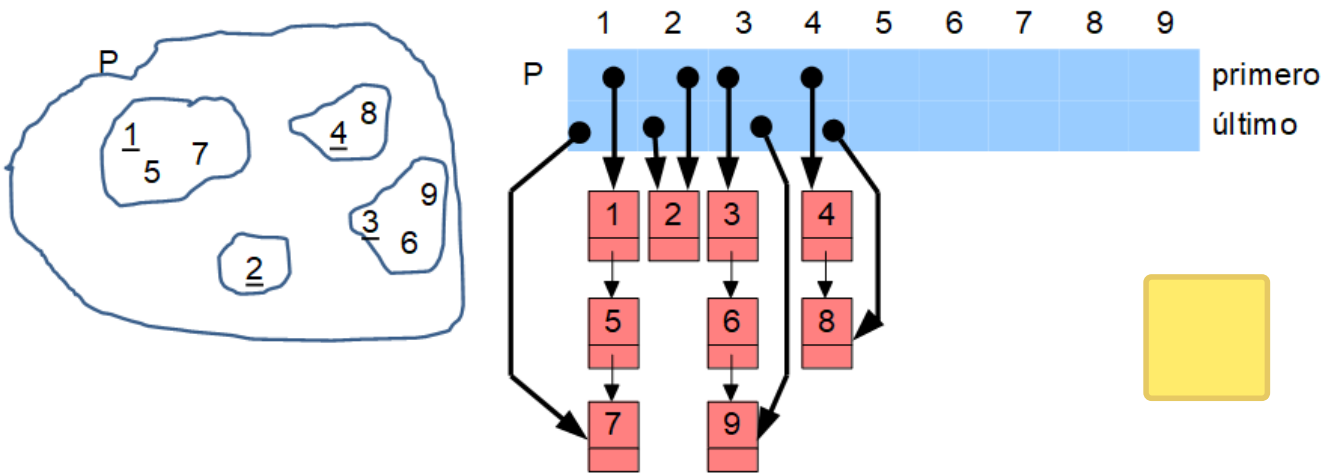
- Cada clase de equivalencia se representa mediante una lista que contiene sus elementos
- Para la unión necesitamos una estructura que permita concatenar dos listas de forma rápida busco operación unión que sea cte
  1. Mantener en la cabecera de las listas dos punteros, uno al primer elemento y otro al último vector dos celdas: primero y último
  2. Listas circulares circulares y doblemente enlazadas

orden da igual, depende del orden de unión

Algorithm declaraciones básicas

```
1: constantes
2:   N = 100
3: tipos
4:   tipoElemento = entero
5:   tipoConjunto = entero
6:   tipoPartición = matriz[1, . . . , N] de tipoLista
7: fin tipos
```

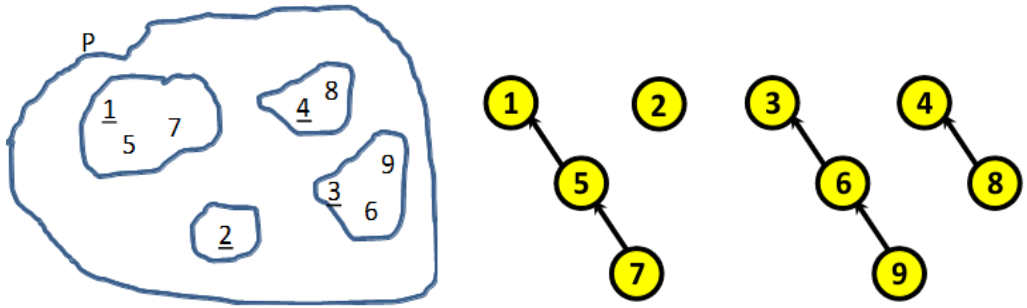
Ejemplo



- La operación unión puede conseguirse ahora en un tiempo constante
- La operación buscar, sin embargo, debe recorrer todas las listas hasta encontrar el elemento en una de ellas. En el peor de los casos todos los elementos de todas las listas. En promedio la mitad de las listas, siendo de $O(N)$
- Si este comportamiento no era bueno para la unión en la representación anterior, puede ser aún peor para la búsqueda, si se necesita frecuentemente
  La **implementación** de todas estas operaciones es **inmediata** y se deja como **ejercicio**

3.3 Representación mediante ÁRBOLES

- Las dos representaciones anteriores utilizan estructuras lineales dando lugar a tiempos lineales, bien para la búsqueda bien para la unión. Alternativa, utilizar estructuras no lineales, árboles
- **Cada clase de equivalencia se representa por un árbol utilizando su raíz para nombrar al conjunto**
- La unión puede conseguirse en un tiempo constante, colocando un árbol como subárbol del otro
- Para la búsqueda es necesario, dado un elemento del árbol, encontrar cual es la raíz: **representación de árboles con punteros al nodo padre**
- Se puede seguir utilizando la representación mediante una matriz, donde cada entrada almacena el padre del elemento i
- La estructura no almacena un sólo árbol, sino un conjunto de árboles: **bosque** de relaciones de equivalencia



Algorithm declaraciones básicas

```
1: constantes
2:   N = 100
3: tipos
4:   tipoElemento = entero
5:   tipoConjunto = entero
6:   tipoPartición = matriz[1, . . . , N] de tipoConjunto
7: tipos
```

Variable P de tipo partición

1	2	3	4	5	6	7	8	9
?	?	?	?	1	3	5	4	6

Convenio para la representación

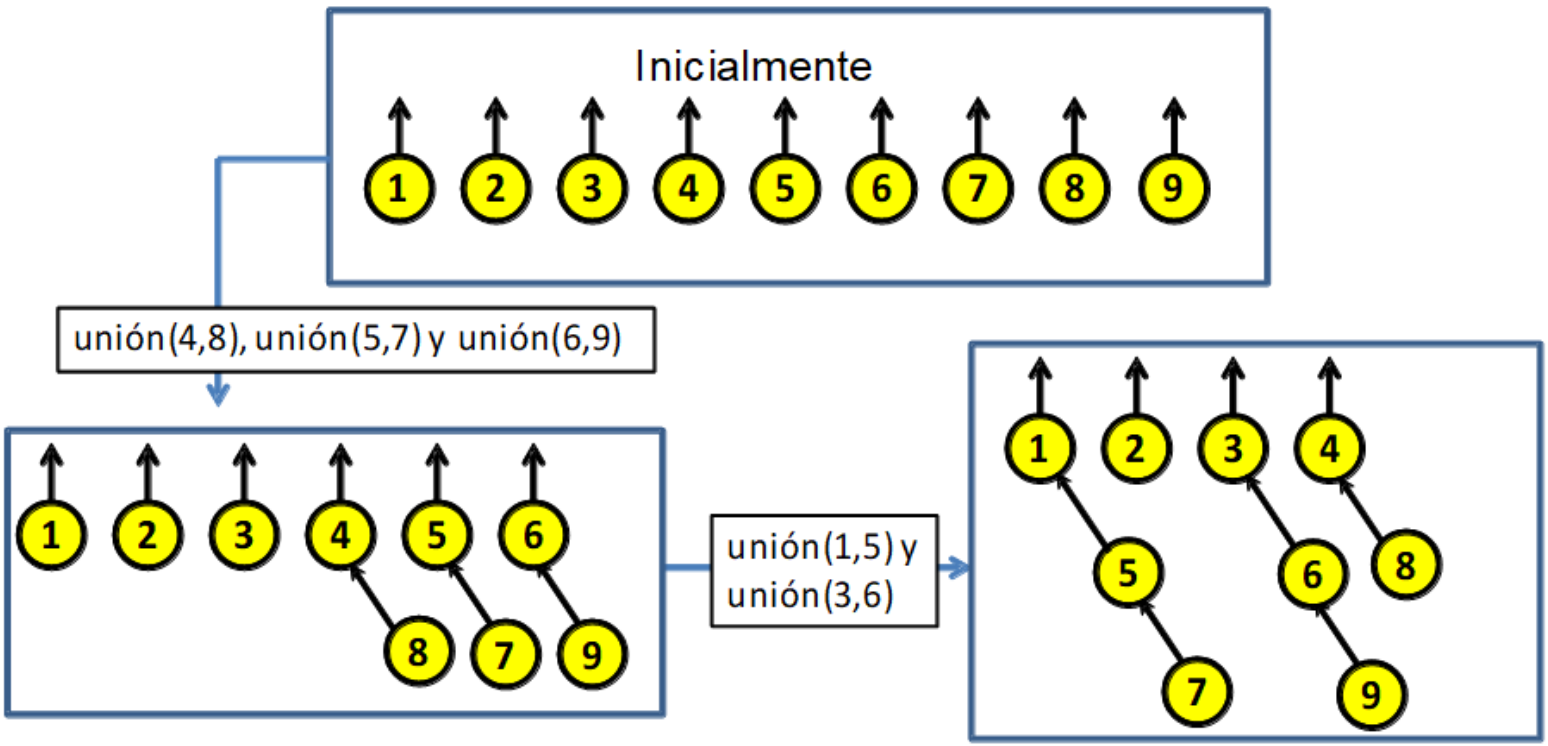
- Si C[i]=0 entonces i es a la vez el nombre del conjunto y su raíz
- Si C[i]=j $\neq$0 entonces j es el padre de i en algún árbol

Variable P de tipo partición

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

Operación Unión

- Se combinan dos árboles haciendo que la raíz de uno apunte a la raíz del otro
- Convenio (arbitrario) en unión(x,y) la nueva raíz es x



Operaciones

Algorithm crear(**ref** P: tipoPartición)

Entrada: partición P con valores desconocidos (no representa nada)
Salida: partición P con valores que representan la situación inicial

```
1: i : tipoElemento
2: para i ← 1 hasta N hacer
3:     P[i] ← 0
4: fin para
```

Algorithm buscar(x:tipoElemento,P: tipoPartición):tipoConjunto

Entrada: partición P y elemento x
Salida: clase de equivalencia a la que pertenece x

```
1: si P[x] = 0 entonces
2:     devolver x
3: si no
4:     devolver buscar(P[x], P)
5: fin si
```

Algorithm union(**raíz1,raíz2**:tipoConjunto, **ref** P: tipoPartición) **¡¡¡ ojo !!!**

Entrada: representantes de los elementos que se quieren relacionar y partición P que representa las clases de equivalencia actuales
Salida: partición P con las nuevas clases de equivalencia

```
1: P[raíz2] ← raíz1
```

Análisis de una secuencia arbitraria de operaciones búsqueda/unión

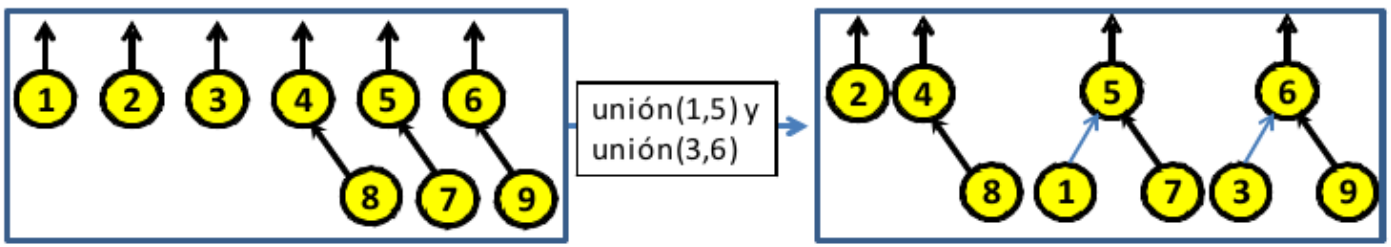
- La operación unión ahora toma un tiempo constante($O(1)$)
- La operación búsqueda devuelve la raíz del árbol del elemento buscado
  - El tiempo de ejecución es proporcional a la profundidad del nodo
  - Con la estrategia utilizada en la unión puede crearse un árbol de profundidad $N-1$. Por tanto, en el peor de los casos el tiempo de búsqueda es $O(N)$
- Una secuencia arbitraria de n operaciones de búsqueda y $N-1$ operaciones unión, en el caso peor, requerirá un tiempo en $O(nN)$, que será $O(N^2)$ si n es comparable a N
- ¡No hemos ganado nada con respecto a la representaciones anteriores!

Mejora

Eliminar la arbitrariedad de la operación unión haciendo siempre que el subárbol menor sea un subárbol del mayor

Mejoras en la operación unión

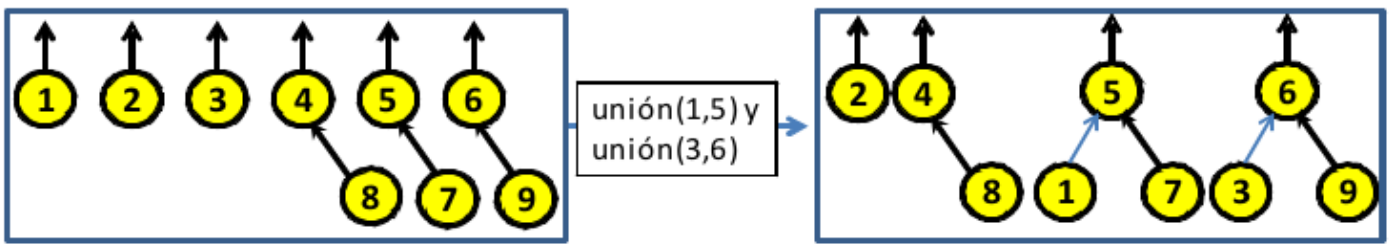
- Dos alternativas:
  - Unión por tamaño: el árbol de menor tamaño se hace subárbol del mayor
  - Unión por altura: el árbol de menor altura se hace subárbol del más alto



Unión por tamaño

Convenio para la representación

- Si i es la raíz del árbol C[i] contiene el tamaño del árbol que representa en negativo
- Si C[i]=j $>$ 0 entonces j es el padre de i en algún árbol



Variable P de tipo partición

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

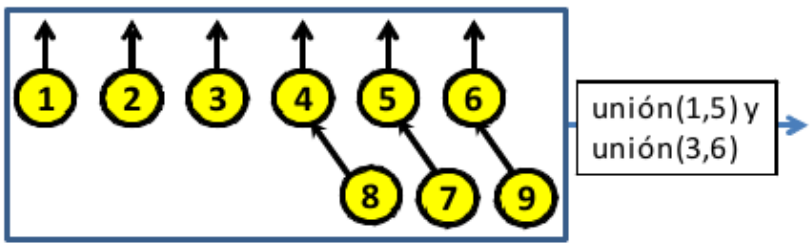
Variable P de tipo partición

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

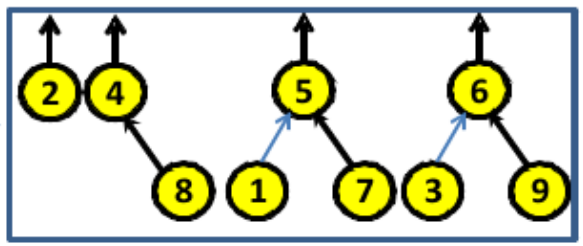
Unión por altura

Convenio para la representación

- Si i es la raíz del árbol C[i] contiene la altura del del árbol que representa en negativo
- Si C[i]=j $>$ 0 entonces j es el padre de i en algún árbol



unión(1,5) y
unión(3,6)



Variable P de tipo partición

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

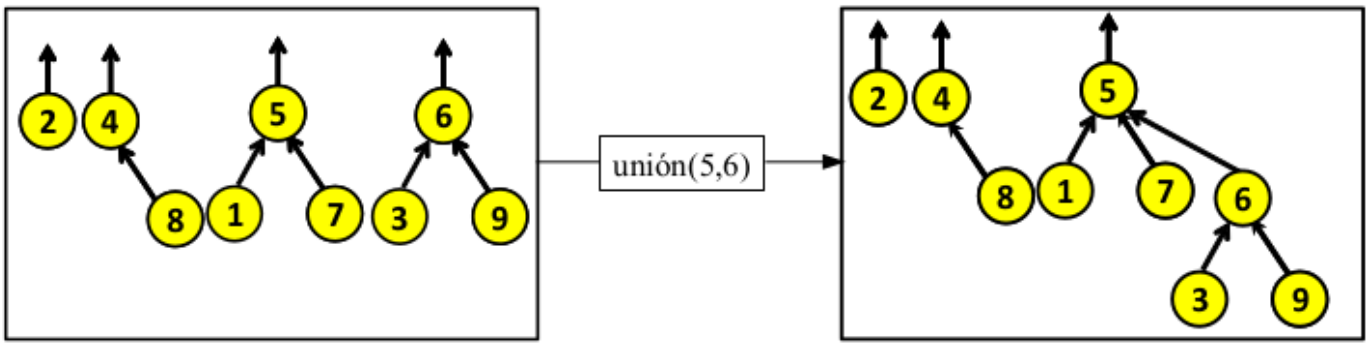
Variable P de tipo partición

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

Análisis unión por altura

- Si se fusionan dos árboles de alturas respectivas h_1 y h_2, la altura del árbol resultante será:

$$h = \begin{cases} max(h_1, h_2) & si \quad h_1 \neq h_2 \\ h_1 + 1 & si \quad h_1 = h_2 \end{cases}$$



Teorema

Empleando la técnica de unión por altura, al cabo de una secuencia arbitraria de operaciones unión, que comienzan en la situación inicial, un árbol de k nodos tiene una altura que es como máximo $[\lg k]$

Demostración por inducción:

1. Verdadero para el caso base $\Rightarrow k = 1 \Rightarrow h = 0 \leq [\lg 1]$
2. Hipótesis de inducción. Teorema cierto para todo m menor que k

$$m \diagup 1 \leq m < k \Rightarrow h \leq [\lg m]$$

3. Demostrar que es cierto para k. Un árbol de k nodos se obtiene de otros dos mas pequeños con a y b nodos, donde:
   - sin pérdida de generalidad suponemos $a \leq b$
   - $a \geq 1$ (partimos de la situación inicial)
   - $k = a + b$

Demostración teorema

$$\left.\begin{array}{l} a \leq b \\ a+b=k \end{array}\right\} \Rightarrow a \leq k-a \Rightarrow a \leq \frac{k}{2} \Rightarrow \underbrace{a<k}_{k>1 \Rightarrow \frac{k}{2}<k} \overbrace{\Rightarrow}^{\text{Hipótesis de inducción}} h_a \leq [\lg a]$$

$$\left.\begin{array}{l} a \geq 1 \\ a+b=k \end{array}\right\} \Rightarrow b \leq k-1 \Rightarrow \underbrace{b<k}_{k>1 \Rightarrow k-1<k} \overbrace{\Rightarrow}^{\text{Hipótesis de inducción}} h_b \leq [\lg b]$$

Sea h_k la altura del árbol resultado de la unión $\Rightarrow h_k = \begin{cases} max(h_a, h_b) & si \quad h_a \neq h_b \\ h_a + 1 & si \quad h_a = h_b \end{cases}$

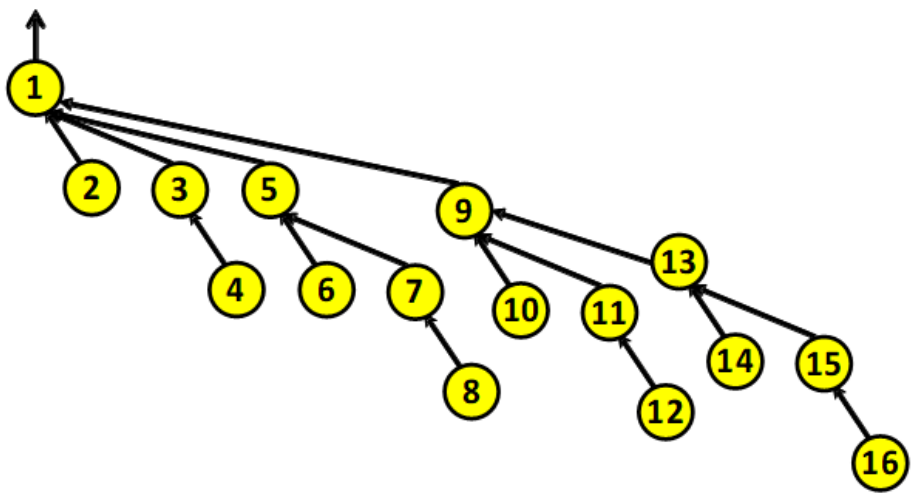
¿$h_k \leq [\lg k]$?

Se pueden dar dos casos

1. Si $h_a \neq h_b \Rightarrow h_k = max(h_a, h_b) \leq max([\lg a], [\lg b]) \leq [\lg k]$
2. Si $h_a = h_b \Rightarrow h_k = h_a + 1 \leq [\lg a] + 1 \leq [\lg \frac{k}{2}] + 1) \leq [\lg k] - 1 + 1 \leq [\lg k]$

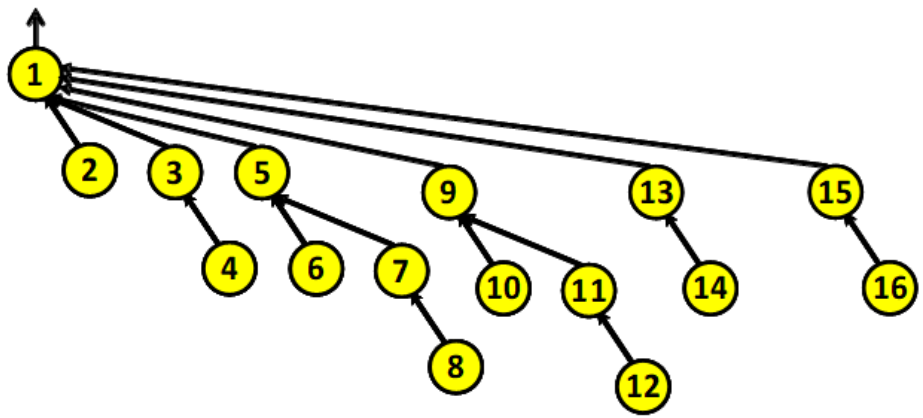
Análisis

- Si cada consulta o modificación de un elemento de la matriz cuenta como operación elemental, la operación unión sigue tomando tiempo constante
- La operación búsqueda, ahora es $O(\lg N)$
- Una secuencia arbitraria de n operaciones de búsqueda y $N-1$ operaciones unión, a partir de la situación inicial, en el caso peor requerirá un tiempo en $O(N + n \lg N)$, que será $O(n \lg n)$ si n es comparable a N
- Árbol del peor caso para $N = 16$



4 COMPRESIÓN DE CAMINOS

- Técnica que se aplica a la operación de búsqueda para tratar de conseguir que las operaciones sean más rápidas
- Para determinar que conjunto contiene a cierto elemento **x**, la operación de búsqueda sube desde el nodo que contiene a **x** hasta la raíz del árbol
- La **compresión de caminos**, una vez que se conoce la raíz, vuelve a recorrer este camino, modificando el padre de cada nodo del camino, para que sea directamente la raíz
- Compresión de caminos después de **buscar(15)**



Estrategia

- Se ejecuta durante la operación **buscar(x)** y es independiente de la estrategia utilizada en la unión
- Todo nodo en el camino de **x** a la raíz cambia su padre por la raíz
- Accesos futuros más rápidos a cambio de algunos movimientos de punteros adicionales

---

Algorithm buscar(x:tipoElemento, **ref** P: tipoPartición):tipoConjunto

---

Entrada: partición P y elemento x
Salida: clase de equivalencia a la que pertenece x modificada, el padre de cada nodo del camino en el retorno pasa a ser directamente la raíz

```
1: si P[x] <= 0 entonces
2:     devolver x
3: si no
4:     P[x] ← buscar(P[x], P)
5:     devolver P[x]
6: fin si
```

Observaciones

- La implantación de la compresión de caminos se hace con un cambio trivial en el algoritmo básico de búsqueda: asignación recursiva a P[x] del valor que devuelve buscar
- La compresión de caminos tiende a reducir la altura del árbol y por tanto a conseguir que las operaciones buscar subsiguientes sean más rápidas
- La nueva operación de búsqueda, sin embargo, recorre dos veces el camino que va desde el nodo en cuestión hasta la raíz. Requiere aproximadamente el doble de tiempo
- Si pocas operaciones de búsqueda puede que no merezca la pena
- Si operaciones de búsqueda frecuentes, al cabo de un tiempo, todos los nodos implicados quedan asociados con su raíz, y las operaciones buscar subsiguientes tomarán un tiempo constante
- La unión perturbará ligeramente esta situación y no durante mucho tiempo
- La mayor parte del tiempo, tanto las operaciones de búsqueda como unión requerirán un tiempo constante

Observaciones

- La compresión de caminos es perfectamente compatible con la unión por tamaños, y así ambas rutinas pueden implantarse a la vez
- La compresión de caminos no es del todo compatible con la unión por altura, porque la compresión puede cambiar las alturas de los árboles:
  - No está muy claro como volver a calcularlas con eficiencia
  - Solución, no se calculan
  - La alturas que se almacenan para cada árbol se convierten en alturas estimadas o **rangos**
  - La unión por altura se denomina entonces **unión por rango**
- El tiempo promedio de la operación buscar incluyendo compresión de caminos y unión por tamaño o altura, resulta difícil de calcular
- Intuitivamente puede observarse que será un tiempo mucho menor que logarítmico: será difícil obtener un árbol de profundidad 5 teniendo en cuenta que se requiere la unión de dos árboles de profundidad 4

Coste utilizando unión por rango y compresión de caminos

- Utilizando ambas heurísticas, el coste en el caso peor para una secuencia de $m(n + N - 1)$ operaciones es $O(m\alpha(m, N))$, donde $\alpha(m, N)$ es una función (inversa de la función de Ackerman) que crece muy despacio (*)
- Tan despacio que en cualquier situación práctica que podamos imaginar se tiene que $\alpha(m, n) \leq 4 \Rightarrow$ coste casi lineal
- La función de Ackerman, definida para enteros $i, j \geq 1$:

$$\begin{aligned} A(i, j) &= 2^j && \text{para } j \geq 1 \\ A(i, 1) &= A(i - 1, 2) && \text{para } i \geq 2 \\ A(i, j) &= A(i - 1, A(i, j - 1)) && \text{para } i, j \geq 2 \end{aligned}$$

- La función inversa $\alpha(m, n)$:

$$\alpha(m, n) = min\{i \geq 1 | A(i, [m/N]) > \log N\}$$

(*) Análisis muy complejo demostrado por Robert E. Tarjan en 1975 (versión más simple en 1983)

La función de Ackerman y su inversa $\alpha(m, n)$

- Algunos valores de la función de Ackerman

$$A(1,j) = 2^j, \text{ para } j \geq 1$$
$$A(i,1) = A(i-1,2), \text{ para } i \geq 2$$
$$A(i,j) = A(i-1,A(i,j-1)), \text{ para } i,j \geq 2$$

	$j=1$	$j=2$	$j=3$	$j=4$
$i=1$	2^1	2^2	2^3	2^4
$i=2$	2^2	2^{2^2}	$2^{2^{2^2}}$	$2^{2^{2^{2^2}}}$
$i=3$	2^{2^2}	$2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\}16$	$2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\}2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\}16$	$2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\}2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\}2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\}16$

- Función "inversa" en cuanto a que crece tan despacio como deprisa lo hace la función de Ackerman

$$\alpha(m,N) = \min\{i \geq 1 | A(i, \lfloor m/N \rfloor) > \log N\}$$

$$m \geq N \Rightarrow \lceil m/N \rceil \geq 1 \Rightarrow A(i, \lceil m/N \rceil) \geq A(i,1) \ \ para \ \ i \geq 1$$

$$A(4, \lceil m/N \rceil) \geq A(4,1) = A(3,2) = 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\}16 veces$$

- Solo para valores enormes de N ocurrirá $A(4,1) > \log N$. Por tanto, para valores razonables de m y N, $\alpha(m,n) \geq 4$