3TADDS.pdf

Anónimo

Algoritmia y Estructuras de Datos Avanzadas

2º Grado en Ingeniería Informática

Escuela Politécnica Superior
Universidad Autónoma de Madrid

3️⃣

Tipos Abstractos de Datos y Estructuras de Datos

1. Tipos de datos abstractos y estructuras

Los algoritmos trabajan con datos → En algoritmos complejos se utilizan **Estructuras de datos** para almacenar los datos de forma más eficiente. → Mejores algoritmos

- Estructuras de datos simples: **strings**, **lists**, **array**
- + Avanzadas: **Diccionarios**, **Sets**
- Estructuras Avanzadas: **listas enlazadas**, **árboles**, **grafos**

TADs

Tipos Abstractos de Datos (TADs): Conjunto de elementos + primitivas actuando en ellos.

Ejemplo: pilas → `ini`, `pop`, `push`.

Ejemplos TADs vs ED (Estructuras de datos)

- TADs → pilas, colas, colas con prioridad
- EDs → arrays, listas enlazadas, árboles binarios de busqueda, heaps, grafos, ...

2. Colas de prioridad (PQ) vs. Min Heaps

2.1. Min Heaps

Heap → Árbol binario quasicompleto

Min Heap → Heap pero se cumple en cada nodo T': (El padre es menor que los hijos).

$info(T') < min\{info(left(T')), info(right(T'))\}$

- Primitivas: `extract` y `insert`

Implementación

Se guardan los datos en un array de n elementos:

- Los hijos del nodo en el índice `m` son: `2*i + 1` y `2*2 + 2`
- El padre de un nodo en el índice `m` es: `(i-1)//2`
- La función `heapify` se encarga de colocar los elementos en su sitio y que siempre se cumpla la condición.

```
def heapify(h: List, i: int):
  while 2*i+1 < len(h):
    n_i = i
    if h[n_i] > h[2*i+1]:
      n_i = 2*i+1
    if 2*i+2 < len(h) and h[n_i] > h[2*i+2]:
      n_i = 2*i+2
    if n_i > i:
      h[i], h[n_i] = h[n_i], h[i]
      i = n_i
    else:
      return
```

Eliminar la raíz:

1. Eliminamos la raíz
2. Ponemos el último elemento como raíz
3. Realizar `heapify` al nodo raíz.

Insertar un nuevo elemento

1. Incluir el nuevo nodo al final
2. Mientras sea más pequeño que su padre, los cambiamos

```
def insert_min_heap(h, k):
  h += [k]
  j = len(h) - 1

  while j >= 1 and h[(j-1) // 2] > h[j]:
    h[(j-1) // 2], h[j] = h[j], h[(j-1) // 2]
    j = (j-1) // 2
```

2.2. Colas de prioridad

Colas de prioridad → Colas, pero en las que cada elemento tiene una prioridad asignada que determina su puesto en la cola.

Si suponemos que cuanto más pequeño un número, mayor prioridad.

- Algunas primitivas: `insert`, `remove`

2.3. Colas de prioridad vs. Min Heaps

Los min heaps están creados utilizando una prioridad, como las colas. Al final:

→ `insert` es simplemente la inserción en el min heap.

→ `remove` es simplemente eliminar la raíz del min heap.

3. Tipo de dato Conjunto Disjunto (CD)

Primer enfoque

- Todos los elementos se numeran de 1 a n.
- Cada subconjunto tomará su nombre de uno de sus elementos, su representante (por ejemplo, el valor más pequeño)
- Mantenemos en un vector el nombre del subconjunto disjunto de cada elemento.

vodafone yu

FANTASÍA DE FIBRA

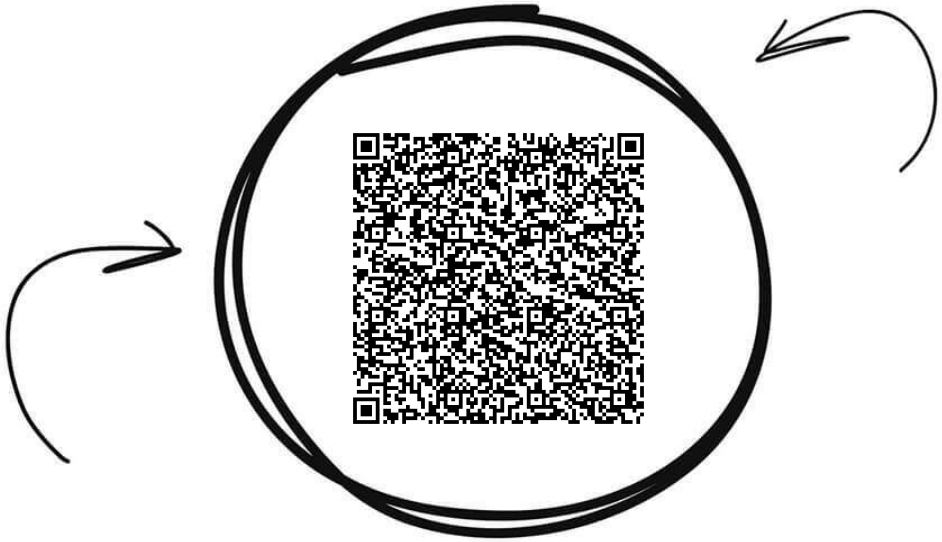
SIN PERMANENCIA

yu Fibra

300Mbps

27€ /mes

Algoritmia y Estructuras de...

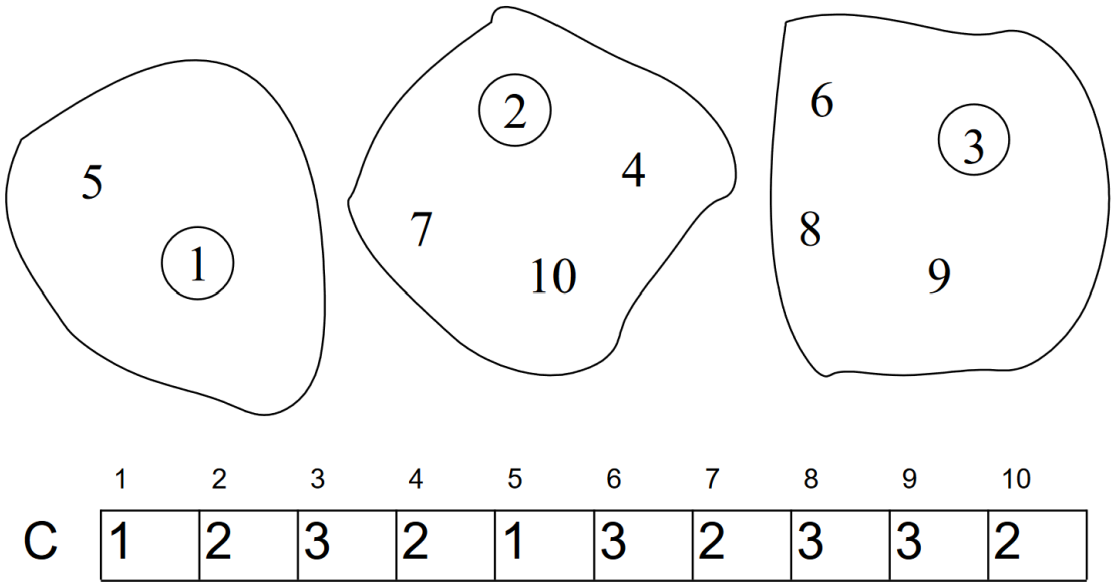


Banco de apuntes de la

WUOLAH

Comparte estos flyers en tu clase y consigue más dinero y recompensas

1. Imprime esta hoja
2. Recorta por la mitad
3. Coloca en un lugar visible para que tus compis puedan escanar y acceder a apuntes
4. Llévate dinero por cada descarga de los documentos descargados a través de tu QR



La representación inicial es una colección de n conjuntos C_i.

- Cada conjunto tiene un elemento diferente, $C_i \cap C_j$.

Hay 2 operaciones válidas:

- **Búsqueda**: devuelve el nombre del conjunto de un elemento dado.
- **Unión**: Combina dos subconjuntos que contienen a y b en un subconjunto nuevo, destruyéndose los originales.

Pseudocódigo:

```
tipo
  Elemento = entero;
  Conj = entero;
  ConjDisj = vector [1..N] de entero

función Buscar1 (C, x) : Conj
  devolver C[x]
fin función
```

Vemos que la búsqueda es una simple consulta → $O(1)$. Lo único que importa es que `búsqueda(x) = búsqueda(y)` ⇔ `x` e `y` están en el mismo conjunto.

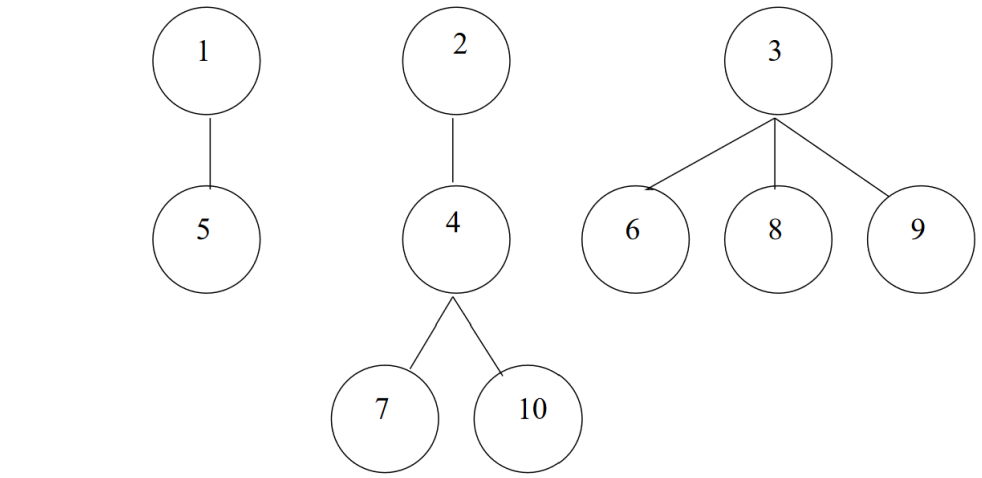
```
procedimiento Unir1 (C, a, b)
  i = min (C[a], C[b]);
  j = max (C[a], C[b]);
  para k = 1 hasta N hacer
    si C[k] = j entonces C[k] = i
  fin para
fin procedimiento
```

El tiempo de ejecución de `union` es $O(n)$ → Una secuencia de $n-1$ uniones toma $O(n^2)$.

Segundo enfoque

Utilizamos un **árbol** para caracterizar cada subconjunto.

- La raíz nombra al subconjunto
- Representación es fácil → solo importa apuntar al padre.
- Cada entrada `p[i]` en el vector contiene el padre del elemento `i`.



	1	2	3	4	5	6	7	8	9	10
C	1	2	3	2	1	3	4	3	3	4

Pseudocódigo:

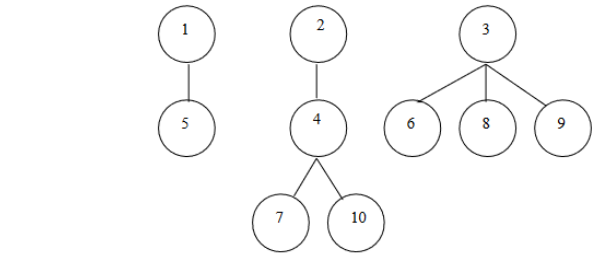
```
función Buscar2 (C, x) : Conj
  r := x;
  mientras C[r] <> r hacer
    r := C[r]
  fin mientras;
  devolver r
fin función
```

La **búsqueda** sobre el elemento `x` se efectúa devolviendo la raíz del árbol que contiene `x`. → Es proporcional a la profundidad del nodo con `x` → en el peor caso es $O(n)$

```
# supone que raíz1 y raíz2 son raíces
procedimiento Unir2 (C, raíz1, raíz2)
  si ra´ız1 < ra´ız2 entonces C[ra´ız2] := ra´ız1
  sino C[ra´ız1] := ra´ız2
fin procedimiento
```

La unión se efectúa combinando ambos árboles → apuntamos la raíz de un árbol a la del otro. → Tiempo de ejecución $O(1)$.

Uniones por altura: Realizar las uniones haciendo del árbol menos profundo un subárbol del árbol más profundo. La altura se incrementa solo si se unen árboles de igual altura.



	1	2	3	4	5	6	7	8	9	10
C	1	2	3	2	1	3	4	3	3	4

	1	2	3	4	5	6	7	8	9	10
A	1	2	1							

```
procedimiento Unir3 (C, A, raíz1, raíz2)
  si A[ra´ız1] = A[ra´ız2] entonces
    A[ra´ız1] = A[ra´ız1] + 1;
    C[ra´ız2] = ra´ız1
  sino si A[ra´ız1] > A[ra´ız2] entonces C[ra´ız2] = ra´ız1
  sino C[ra´ız1] = ra´ız2
fin procedimiento
```

- La profundidad de cualquier nodo nunca es mayor que $\log_2 n$
  - Todo nodo está inicialmente en la profundidad 0.
  - Cuando su profundidad se incremente como resultado de una unión, se coloca en un´arbol al menos el doble de grande → su profundidad se puede incrementar $\log_2 n$ veces

Comprensión de caminos → se ejecuta durante búsqueda

- Durante la búsqueda de un dato x, todo nodo en el camino de x a la raíz cambia su padre por la raíz.
- Es independiente del modo en que se efectúen las uniones.

```
función Buscar3 (C, x) : Conj
  r = x;
  mientras C[r] <> r hacer
    r = C[r]
  fin mientras;
  i = x;
  mientras i <> r hacer
    j = C[i]; C[i]= r; i = j
  fin mientras;
  devolver r
fin función
```

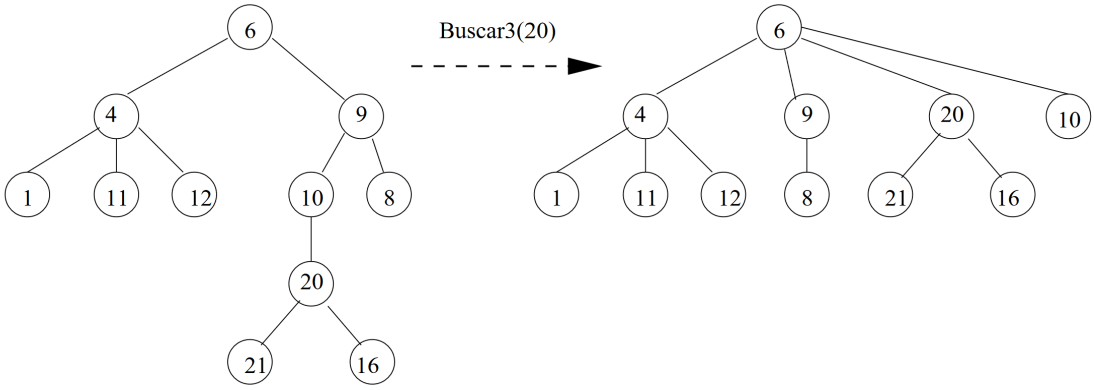
La compresión de caminos no es totalmente compatible con la unión por alturas.

- Al buscar, la altura de un árbol puede cambiar.

Las alturas almacenadas para cada árbol se convierten en alturas estimadas (rangos).

- Pero la unión por rangos es tan eficiente, en teoría, como la unión por alturas.

Ejemplo de comprensión de caminos:



Fuente: http://quegrande.org/apuntes/EI/2/Alg/teoria/08-09/tema_7_-_conjuntos_disjuntos.pdf

Explicación presentación

- Un Conjunto Disjunto (DS) sobre un **conjunto universal** U es una familia **dinámica** S de subconjuntos disjuntos de U, i.e., una **partición** de U.
- Cada uno de los **subconjuntos** estará representado por un cierto **elemento** x. Denominaremos S_x al subconjunto del DS representado por el elemento x
- El *Conjunto Disjunto* tiene las siguientes primitivas:
  - `DS_init (U)` recibe el conjunto universal U y devuelve la partición inicial S como la familia de subconjuntos disjuntos $\{\{u\} : u \in U\}$.
  - `DS_find (u, S)` recibe un elemento $u \in U$ y devuelve el representante del subconjunto S_x de S que contiene a u.
  - `DS_union (u, v)` recibe dos elementos $u, v, \in U$. Si u y v pertenecen a subconjuntos disjuntos diferntes, $u \in S_x$ y $v \in S_y$, calcula la unión $S_x \cup S_y$. Devuelve el representante del subconjunto $S_x \cup S_y$. Normalmente el representante de $S_x \cup S_y$ será x o y.

Recuerda:

- Conjunto: Colección de elementos no repetidos
- Conjunto universal: Es un conjunto formado por todos los objetos (elementos) de estudio en un contexto dado.
- ***Subconjuntos disjuntos***: Dos subconjuntos S_1 y S_2 son disjuntos si no tienen ningún elemento (objeto) en común $S_1 \cap S_2 = \varnothing$.

Características DS

- El Conjunto Disjunto nunca está vacío.
- Los subconjuntos de un Conjunto Disjunto nunca se dividen. Solo pueden cambiar a subconjuntos más grandes mediante la unión de subconjuntos disjuntos.
- Después de invocar a `DS_init ()` tendremos una partición con $|U|$ subconjuntos (i.e tantos subconjuntos como elementos tenga el conjunto universal). Por tanto, el máximo número de uniones que se puede realizar es $|U|-1$.

- El TaD Conjunto Disjunto no es tan común como los diccionarios, pilas o colas pero son muy valioso porque sus implementaciones son muy eficientes: cuando pueden utilizarse se consiguen grandes mejoras. https://courses.cs.washington.edu/courses/cse373/14sp/lecture10.pdf

---

Ejemplo:

Una de las muchas aplicaciones del TaD Conjuntos Disjuntos es hallar las componentes conexas de un **grafo no dirigido**. Dos nodos pertenecerán a una misma componente conexa cuando exista un camino entre ellos.

- Por ejemplo, el grafo no-dirigido G donde $G[V]$ representa al conjunto de nodos y $G[E]$ al de aristas

$G[V] = \{0, 1, 2, 3, 4, 5, 6\}$
$G[E] = \{(0, 1), (4, 5), (0, 2), (3, 0), (1, 3)\}$

tiene tres componentes conexas formadas por los nodos: $\{(0, 1, 2, 3), (4, 5), (6)\}$.

- Podemos preguntarnos:

> ¿Cómo obtener las componentes conexas del grafo? ¿Cómo saber si existe un camino entre dos nodos u y v cualesquiera del grafo?

- *Respuesta:*

> El conjunto de nodos del grafo es un conjunto universal. Las componentes conexas son los subconjuntos disjuntos de un Conjunto Disjunto. Por tanto podemos utilizar las funciones del TaD Conjunto de Disjunto para obtener las componentes conexas de un grafo. Los algoritmos connected_components() y same_component() obtienen las componentes conexas y determinan si dos nodos están conectados.

```
def connected_components (G: Grafo) -> DS:
    ''' Devuelve un Subconjunto Disjunto con las componentes conexas del grafo G'''

    # Se genera un subconjunto disjunto por cada vértice
    dsj = DS_init (G[V])
    # Se procesan todas las aristas del grafo
    for u, v in G[E]:
        DS_union (u, v, dsj)
    return dsj
```

Evolución del algoritmo paso a paso

Primitive DS	edge processed					disjoin subsets S_x	
S = init (G[V])	· · ·	{**0**}	{**1**}	{**2**}	{**3**}	{**4**)	{**5**}
union (0, 1, S)	(0,1)	{**0**, 1}		{**2**}	{**3**}	{**4**)	{**5**}
union (4, 5, S)	(4,5)	{**0**, 1}		{**2**}	{**3**}	{**4**, 5}	
union (0, 2, S)	(0,2)	{**0**, 1, 2}			{**3**}	{**4**, 5}	
union (0, 3, S)	(0,3)	{**0**, 1, 2, 3}				{**4**, 5}	
union (1, 3, S)	(1,3)	{**0**, 1, 2, 3}				{**4**, 5}	

- Cuando se inicializa el conjunto disjunto se crean tantos subconjuntos como nodos tenga el grafo.
- Cada vez que se itera la lista de arista se invoca a la función `unón()`
- Notad como *union(1,3)* no modifica los subconjuntos ya que cuando se ejecuta los nodos 1 y 3 ya pertenecían al mismo subconjunto.
- Existe un camino entre dos nodos cualesquiera si ambos pertenecen a la misma componente conexa y, por tanto, al mismo subconjunto disjunto. El procedimiento $\mathrm{same\_component}()$ recibe un Conjunto Disjunto y determina si tienen el mismo

representante.

```
def same_component (u:int, v:int, dsj:DS):
    if find(u, dsj) == find(v, dsj):
        return True
    return False
```

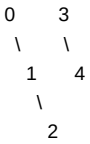
Estructuras de datos para la DS

- Hay varias opciones para la *estructura de datos* con la que implementar DS: Por ejemplo
  - Con **Listas**:
    - Para cada subconjunto S_x se utiliza una lista para cada uno de los subconjuntos y un array de punteros (enlaces) a ellas. Todos los nodos de las listas tienen un puntero al primer nodo de la lista que será el representante del subconjunto. En Cormen et al. puedes encontrar una descripción de esta estructura.
- Nosotros optaremos por una estructura más simple basada en **árboles**.

Implemantación de DS con árboles

- Cada subconjunto disjunto S_x de DS se almacena en un **tipo particular de árbol** que denominaremos T_x.
- El representante x de S_x se sitúa en la raíz de su correspondiente árbol T_x.
- El resto de elementos del subconjunto S_x serán nodos del árbol T_{S_x}.
- Todo nodo del árbol tiene un enlace a su nodo padre, excepto la raíz, que podemos considerar que tiene un enlace a si mismo.

El DS del siguiente ejemplo esta formado por los tres árboles correspondientes a los tres subconjuntos $\{(0,1,2),(3,4)\}$ de DS



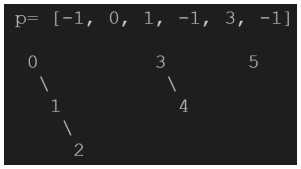
Coste de las funciones

- El coste de la función $\text{union}(x,y,S)$ cuando x e y son representantes de sus respectivos subconjuntos es $O(1)$. Por ejemplo, si tras la unión de ambos subconjuntos, queremos que el árbol T_y sea un subárbol de T_x tan solo habrá enlazar x con la raíz de T_y.
- Sin embargo, para implementar $\text{union}(u,v,S)$ sobre cualesquiera nodos u y v primero habría que hallar los representantes de sus respectivos subconjuntos, es decir invocar a la función `find()`. Por tanto necesitamos una manera eficiente de identificar a que árboles pertenecen los elementos u y v. Una vez identificados los árboles habría que recorrerlos hasta hallar sus respectivas raíces y enlazarlas. ¿Cómo hacerlo?

Implementación de un árbol con una tabla

- Se puede obtener fácilmente a que árbol pertenece un nodo dado si implementamos los árboles por una tabla $p[\,]$ de tamaño $|U|$ e identificamos cada elemento del conjunto universal con un índice de la tabla.

  El valor de p correspondiente al índice u, $p[u]$ denota el índice del padre de u. Si queremos indicar que u es un raíz asignaremos a $p[u]=-1$. Por ejemplo, el siguiente DS se implementa por la tabla p



Algoritmos para las primitivas

```
def init (u) ->  list :
    p = len(u) * [-1] # se crea una tabla con el tamaño del conjunto universal,|U|, donde todas sus componentes se inicializan a -1
    return p

def find (x:int, p:list) -> int:
    # se recorre la lista hasta que llego a la ráiz del árbol
    while p[x] > -1:
        x = p[x]
    return x

def union(u:int, v:int, p:list) -> int:
    ''' PsC: Versión 1 (no optimizada)'''
    # find the representants
    x = find (u, p)
    y = find (v, p)

    if x == y:
        return -1

    p[y] = x     #join second tree to first return x
    return x
```

Ejemplo:

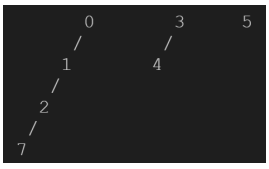
Primitive					Disjoin Sets			p
init ()	{**0**}	{**1**}	{**2**}	{**3**}	{**4**}	{**5**}	{**6**}	[-1,-1,-1,-1,-1,-1,-1]
union (0, 1)	{**0**,1}		{**2**}	{**3**}	{**4**}	{**5**}	{**6**}	[-1,0,-1,-1,-1,-1,-1]
union (4,5)	{**0**,1}		{**2**}	{**3**}	{**4**,5}		{**6**}	[-1,0,-1,-1,-1,4,-1]
union (0,2)	{**0**,1,2}			{**3**}	{**4**,5}		{**6**}	[-1,0,0,-1,-1,4,-1]
union (3,0)	{**0**,1,2,3}				{**4**,5}		{**6**}	[-1,0,0,0,-1,4,-1]
union (1,3)	{**0**,1,2,3}				{**4**,5}		{**6**}	[-1,0,0,0,-1,4,-1]

(en negrita denotamos a los representantes de cada uno de los subconjuntos disjuntos)

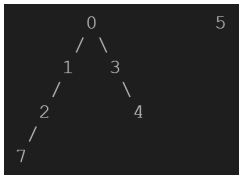
Mejorando la *unión( ): unión por alturas*

El coste de buscar un elemento $\mathrm{find}(u)$ depende de la profundidad del árbol en que se encuentre u, $O\left(\mathrm{height}(T_x)\right)$, por tanto:

- Deberíamos unir el árbol menos profundo al más profundo. Por ejemplo, si tenemos el Conjunto Disjunto con los árboles



e invocamos a la función union (1, 4) obtendríamos

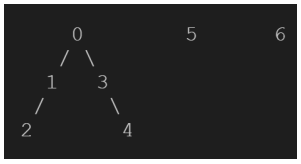


- Pero entonces *necesitamos conocer la altura de los árboles* y, por tanto, **debemos guardar la altura de todos los arboles** T_i. Una posibilidad es que cuando x sea una raíz en vez de guardar en la tabla el valor $p[x] = -1$ guardar el valor $-h$ siendo h la altura del árbol (número de nodos en el mayor camino desde la raíz hasta una hoja).

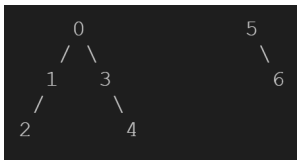
Ejemplo:

- Por ejemplo la tabla `[-3, 0, 1, -2, 3, -1, -1]` representa al DS formado por los tres árboles

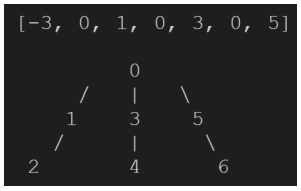
- Si ahora invocásemos a la función union (1, 4) obtendríamos [-3, 0, 1, 0, 3, -1, -1]



- En el caso de invocar a la función `union()` sobre árboles que tuviesen la misma altura



- Notad como los árboles **no** son binarios.



El pseudocódigo de la función `union` con unión por alturas sería:

```
def union (u:int, v:int, p:list):
        ''' PsC: union by height
        of the set {S_x | u \in S_x}  and {S_y | v \in S_y}  '''

        x = self.find (u)
        y = self.find (v)
```

```
        if x == y:
            return -1

        if p[y] < p[x]:      #T_y is taller
            p[x] = y
            ret = y
        elif p[y] > p[x]:    #T_x is taller
            p[y] = x
            ret = x
        else:
            #T_x, T_y have the same lenght
            p[y] = x
            p[x] -= 1
            ret =  x
        return ret
```

Mejorando find()

El coste de find ()

- El coste de $\text{find}(x, p)$ es $O(\log |S_x|) = O(\log N)$.
- Demostración (ver transparencias)

find() con compresión:

¿Podríamos mejorar el rendimiento de *find()* un poco más?

- Observad que cuando recorremos el árbol buscando el representante de u también encontramos el representante de todos los nodos v que se hallan entre u y la raíz de su árbol.
- Por tanto podemos utilizar $\text{find}(u)$ además de para hallar la raíz, actualizar el padre $p[v]$ de todos los v entre u y la raíz.
- En otras palabras, cada vez que invoquemos a la función `find()` podemos **comprimir** el camino desde u a la raíz.

Para *comprimir* el camino necesitamos ejecutar dos bucles. En el primer bucle obtenemos la raíz del árbol y en el segundo actualizamos el padre de todos los nodos entre u y la raíz

```
def find_cc(u, p):
    # find the representative
    z = u

    # get the root (representant)
    while p[z] > -1:
        z = p[z]

    # compress the path from u to the root
    while p[u] >-1:
        y = p[u]
        p[u] = z
        u = y
    return z
```

Podemos obtener una versión recursiva del código anterior. En las llamadas de la recursión recorremos el árbol hasta llegar a la ráiz (caso base) y es en los retornos de la recursión cuando vamos actualizamos el valor del padre de todos los nodos en el camino.

```
def find_compress (u, p):
        ''' find compress recursive'''
        if p[u] < 0:
            return u

        p[u] = find_compress (p[u])
        return p[u]
```

Ejemplo:

> Suponga que en el árbol [-4, 0, 0, 2, 3, 1]

```
     0
    /  \
   1    2
  /      \
 5        3
           \
            4
```

invocamos a find(4, p) con compresión. Obtendríamos el árbol:

```
[-4, 0, 0, 0, 0, 1]

        0
    /  |  \   \
   /   |   \   \
  1    2    3   4
 /
5
```