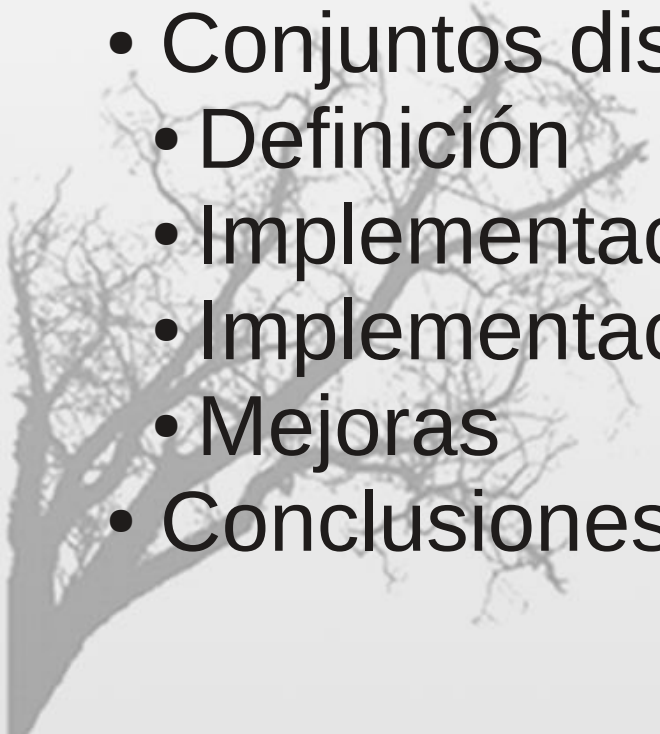


Lección 12:

Heaps y Conjuntos Disjuntos

- Motivación
- Heaps
 - Definición
 - Implementación
 - Eficiencia
- Conjuntos disjuntos
 - Definición
 - Implementación con vectores
 - Implementación con árboles
 - Mejoras
- Conclusiones



Motivación



Los maestros interinos cogen plaza según su puntuación, el que más puntuación tenga antes elige plaza.

- Cada vez que se necesita un docente, a principios de curso o para cubrir bajas, se elige el de mayor puntuación.
- Cuando el maestro toma plaza, deja de competir hasta que acaba su contrato o acaba el curso.
- Al comienzo del siguiente curso, todos vuelven a computar sus puntos y vuelven a optar a plazas.

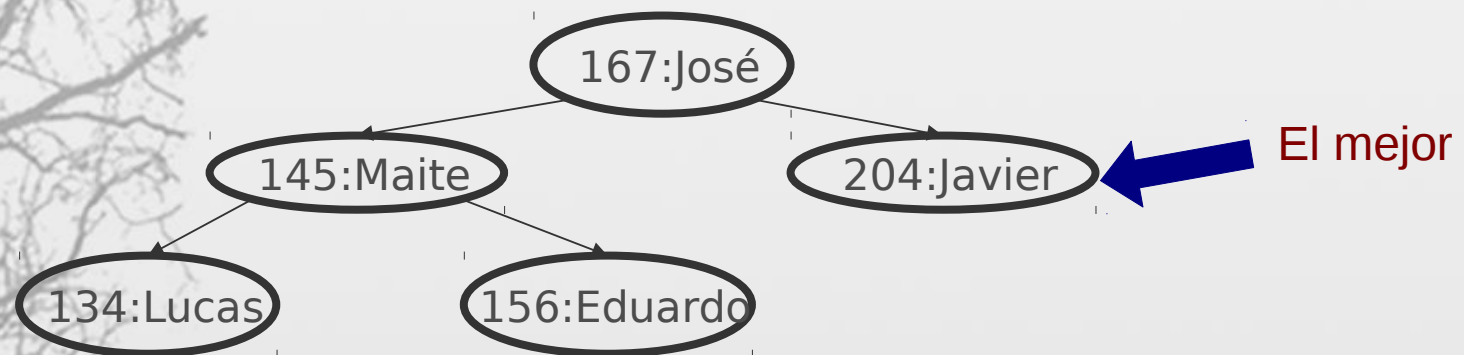
...	Maite Pérez 145	Lucas Suarez 134	Eduardo Ruíz 156	José García 167	Javier Osorio 204	...
-----	--------------------	---------------------	---------------------	--------------------	----------------------	-----

Motivación



Se me ocurre utilizar:

- Listas o vectores desordenados:
 - Insertar es $O(1)$ pero encontrar el mejor es $O(n)$
- Vectores ordenados:
 - Encontrar el mejor es $O(1)$ pero insertar es $O(n)$
- Árboles AVL
 - Inserciones y obtener el mejor es $O(\log n)$

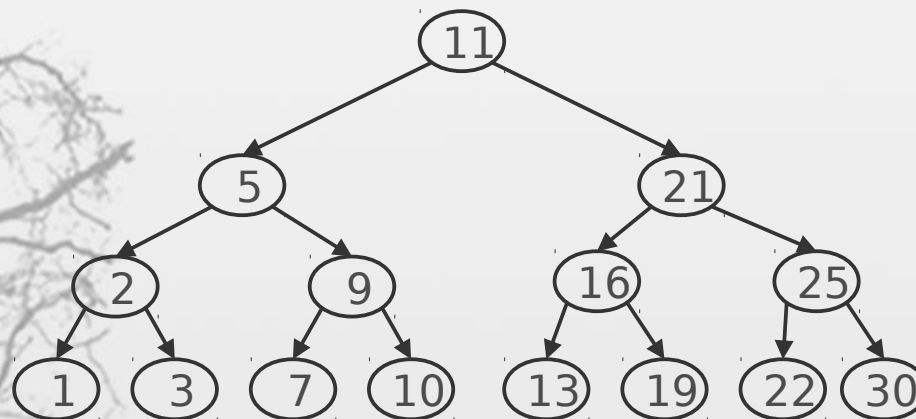


Definiciones previas



Un heap se implementa mediante un árbol binario. Veamos antes algunas definiciones.

- Un **árbol binario perfecto** es un árbol binario con todos los nodos hoja a la misma profundidad, es decir, todos los nodos internos tienen dos hijos.



Altura h

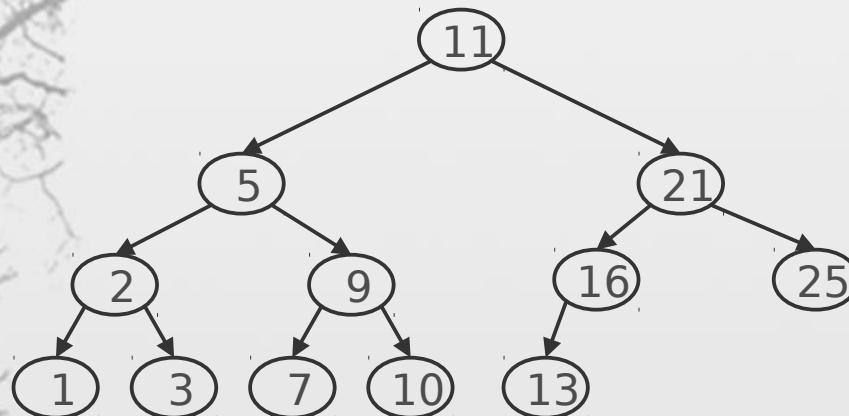
- $2^{h+1} - 1$ nodos
- $2^h - 1$ no-hojas
- 2^h hojas

Definiciones previas



Pero el número de datos a almacenar no siempre es $2^{h+1}-1$

- Un **árbol binario completo** es un árbol que cumple la propiedad de ser perfecto en todos sus niveles excepto quizás en el último nivel, en el cual los nodos deben estar agrupados a la izquierda
- Un árbol perfecto es un árbol completo



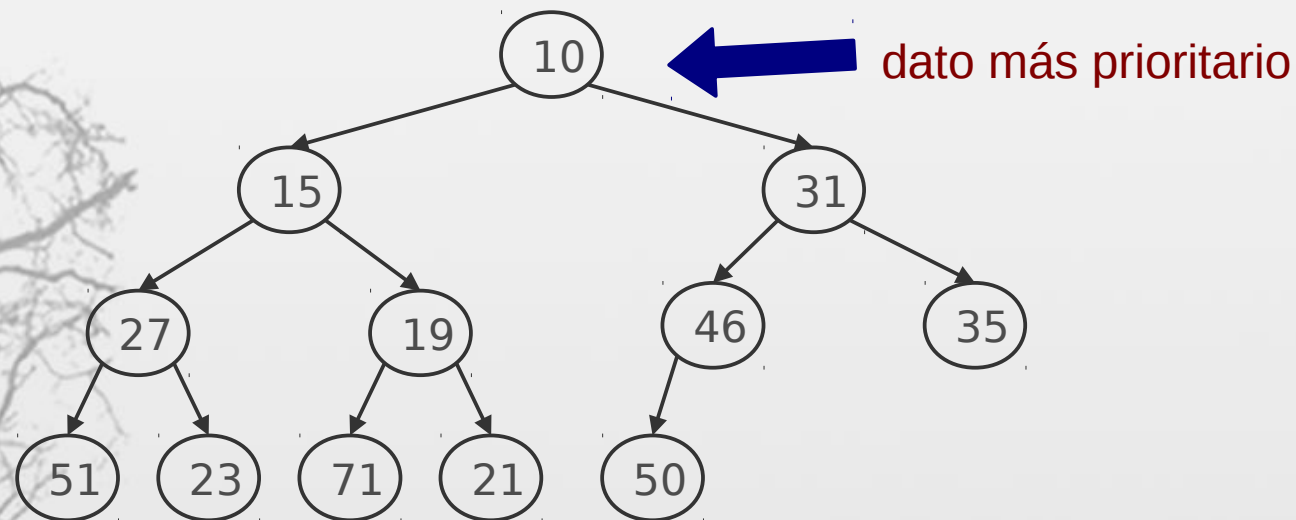


Definición de heap

Un **heap** es un árbol completo con la propiedad:

- Todo nodo (no raíz) tiene un padre con menor o igual prioridad

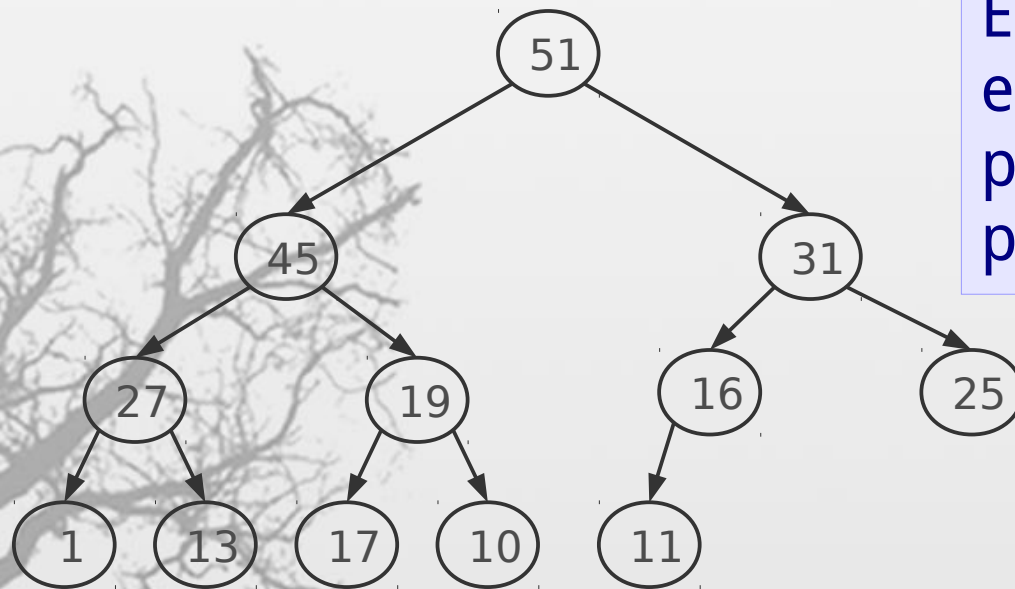
Esta propiedad permite que el nodo raíz tenga siempre el dato más prioritario.



Heaps y colas de prioridad

Pero a veces el dato más prioritario puede ser el de más valor, entonces la definición es la opuesta:

- Cada nodo padre tiene un igual o mayor prioridad



Esta definición sería la válida en el caso de los maestros porque escogemos al de más puntos

Implementación del heap

Un árbol completo es compacto y tiene una implementación óptima utilizando un vector.

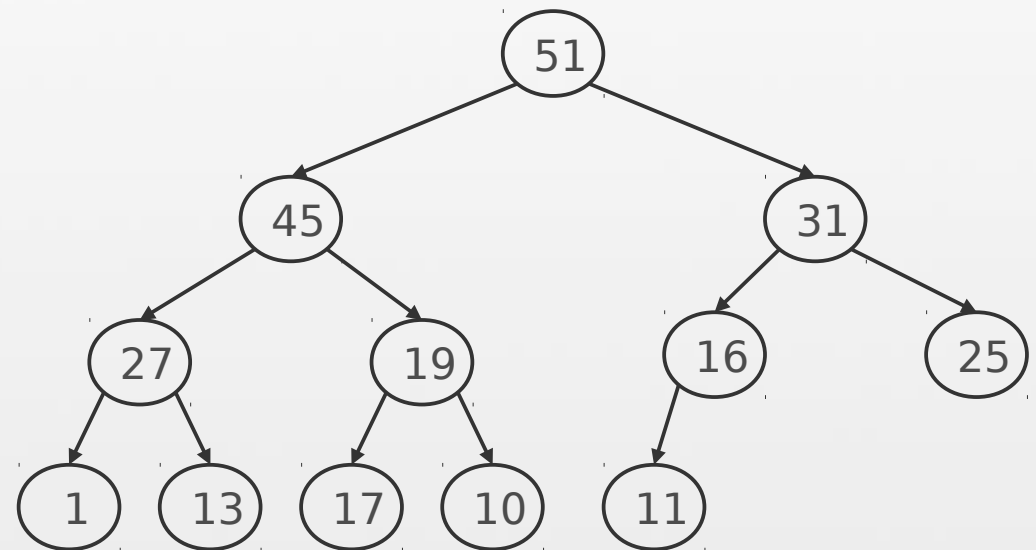
Nodo i :

hijo izquierdo: $2*i+1$

hijo derecho: $2*i+2$

padre: $\lfloor (i - 1)/2 \rfloor$

soy el 45,
dónde están mis hijos?
izdo $\square 2*1+1 = 3$
decho $\square 2*1+2 = 4$
padre $\square 1-1/2 = 0$



0	1	2	3	4	5	6	7	8	9	10	11
51	45	31	27	19	16	25	1	13	17	10	11

Operaciones del heap



Operaciones básicas (sin contar creación/destrucción)

- Inserción y sacar el dato más prioritario
- Actualizar no es una operación típica
- Criterio seguido: más prioritario el de más valor

```
template <class T>
class Heap {
    T *arr; // el contenedor
    int tamal, tamaf; // tamaños logico y físico
    void mueveAbajo(int nodo); // recoloca el nodo
    inline void swap (int a, int b);
public:
    Heap(int MaxTama=500);
    Heap(const Heap<T> &hp);
    ~Heap(void);
    void inserta(const T &item);
    T elimina();
    inline int tamanio();
};
```



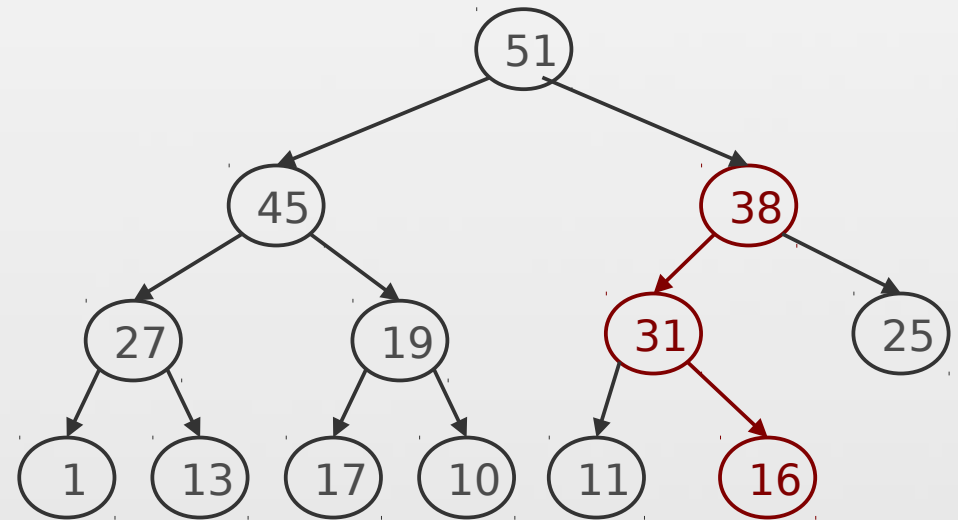
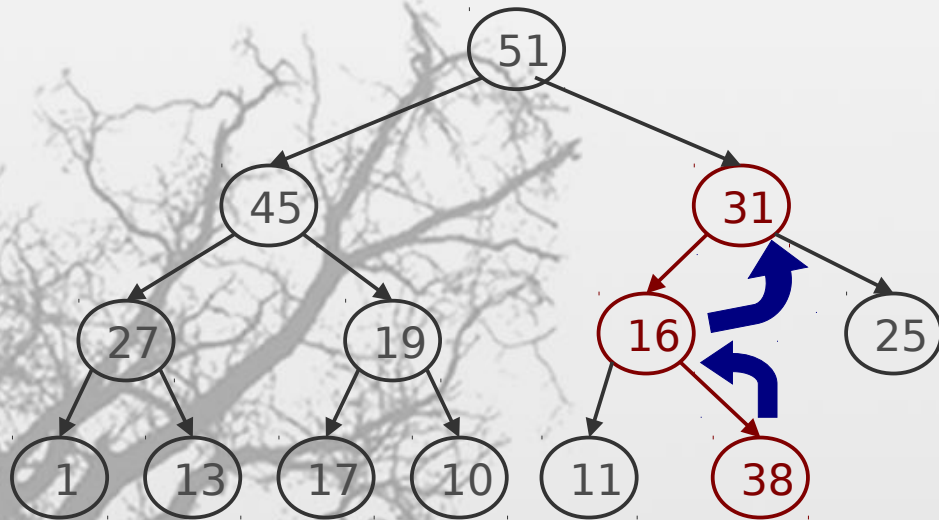
Insertar dato

La inserción sigue los siguientes pasos:

- Insertar el dato en la última posición:

```
arr[tamal]=item
```

- Intercambiar este dato con su padre mientras que no se cumpla la condición (padre \geq hijo)



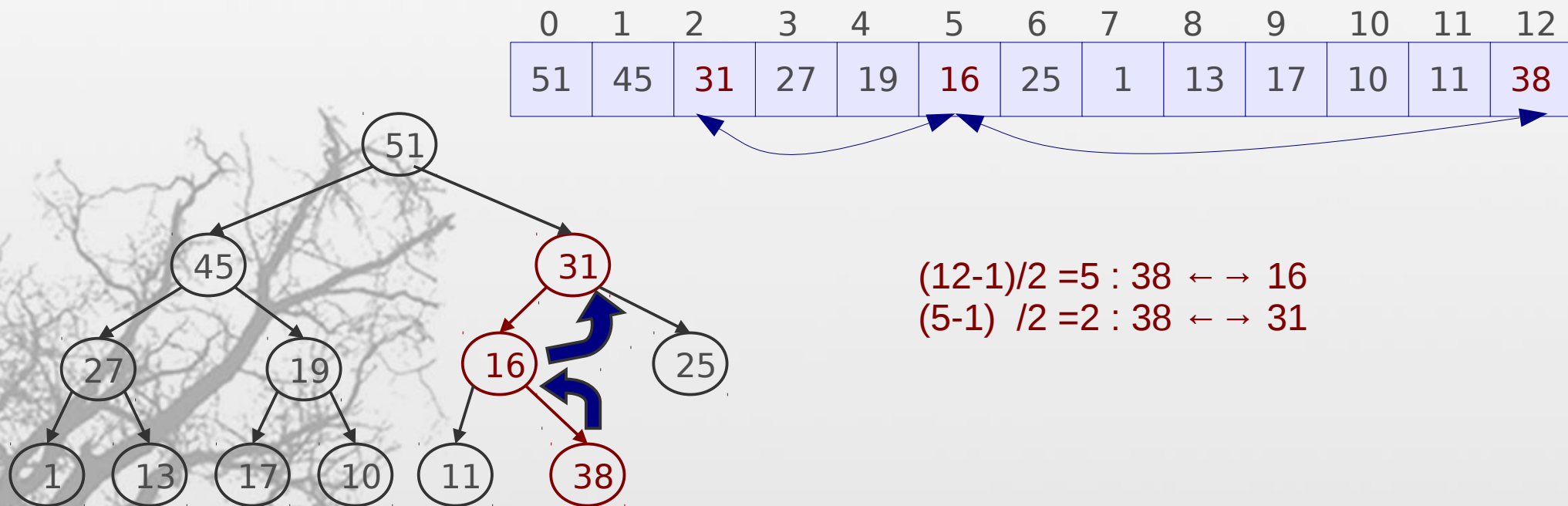
insertamos 38



Insertar dato

La implementación sobre vectores:

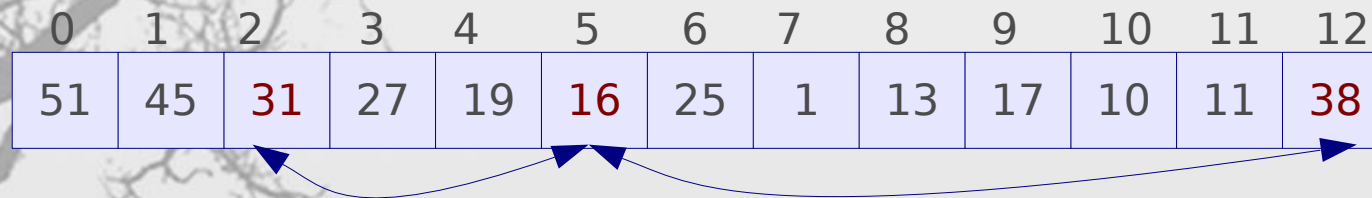
- Consideramos un tamaño estático
- Es eficiente al conocerse las posiciones padre/hijo



código de inserción



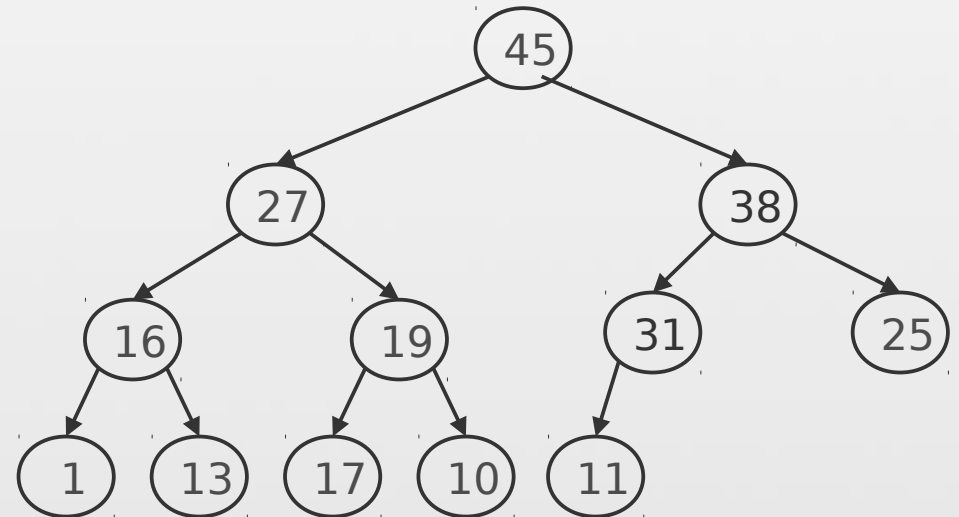
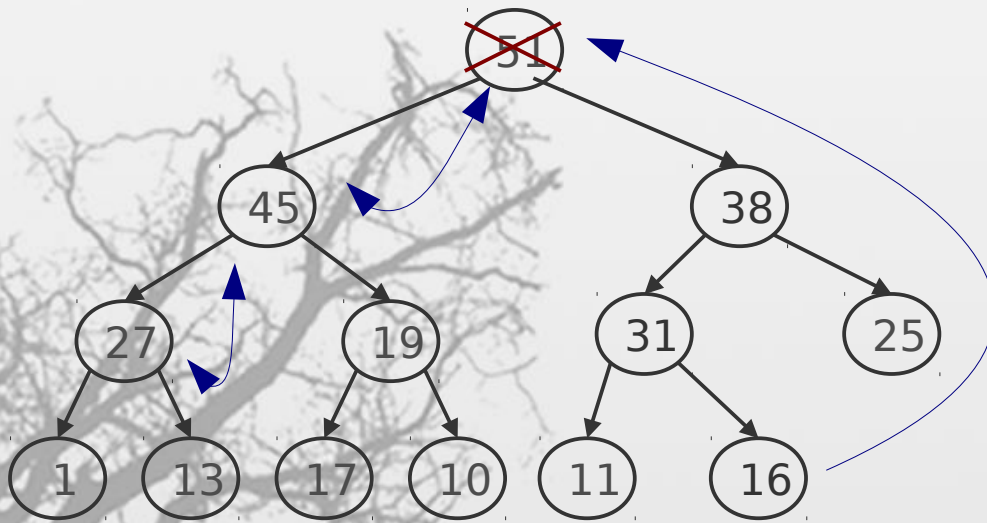
```
template <class T>
void Heap<T>::inserta(const T &item)
{
    if (tamal >= tamaf)
        throw ErrorMaximoTam();
    arr[tamal] = item;
    int npos = tamal;
    int padre = (npos-1)/2;
    while (padre >= 0 && arr[npos] > arr[padre]){
        swap (npos, padre);
        npos = padre;
        padre = (npos-1)/2;
    }
    tamal++;
}
```





Obtener dato

- Sacar el dato que está en la raíz
- Poner el último nodo como raíz
- Intercambiar el nodo con su hijo mayor mientras no se cumpla la condición padre-hijo del heap



Obtener dato



```
template <class T>
T Heap<T>::obtener()
{
    assert(tamal > 0);
    T aux = arr[0];
    arr[0] = arr[--tamal]; // ultimo elemento
    mueveAbajo(0);
    return aux;
}
```

```
template <class T>
void Heap<T>::mueveAbajo(int raiz)
{
    int hijo = 2*raiz+1;
    if (hijo<tamal){
        int hijodecha = hijo + 1;
        if (hijodecha < tamal && arr[hijodecha]>arr[hijo])
            hijo = hijodecha;
        if (arr[raiz]<arr[hijo])
            swap(raiz,hijo);
        mueveAbajo(hijo);
    }
}
```



sube el mayor de los dos hijos

Eficiencia de los heaps



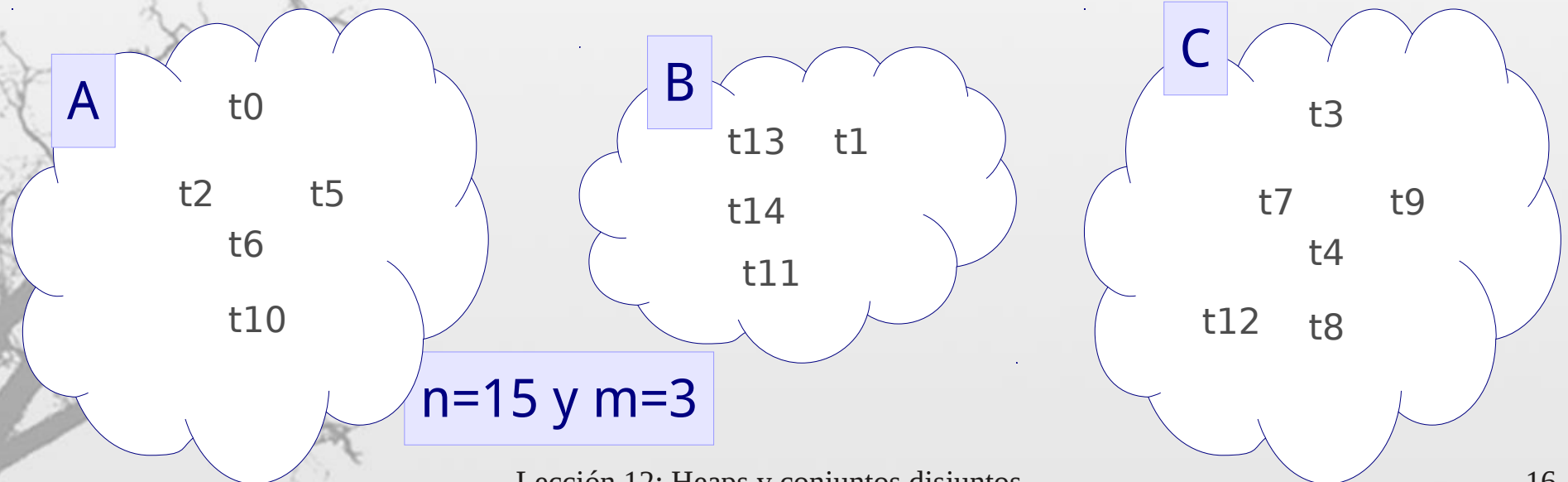
- La operación de inserción necesita un tiempo:
 - $O(1)$ en el mejor caso
 - $O(\log n)$ en el peor caso, la altura del árbol
- La operación de obtener el más prioritario
 - Es siempre $O(\log n)$ en el mejor y peor de los casos
- El tiempo logarítmico siempre está garantizado porque el árbol es completo

Conjuntos disjuntos



Un gran proyecto software se ha dividido en n tareas, todas ellas independientes, que serán asignadas a m grupos de trabajo distintos de modo que cada uno de ellos realice tareas de modo exclusivo.

Posteriormente algunos grupos afines de unirán para unificar resultados parciales.



Conjuntos disjuntos



Hay varias estructuras de datos que pueden manejar este tipo de problemas:

- Un vector o una lista simplemente enlazada: pero controlar datos no repetidos cuesta $O(n)$

El problema se resuelve en matemáticas con los denominados **conjuntos disjuntos**, concepto también existente como estructura de datos.

- Permiten resolver problemas cómo identificar la tarea de un miembro del equipo
- Saber si dos personas pertenecen al mismo equipo
- Unir dos equipos existentes



Conjuntos disjuntos (def.)

Un **conjunto disjunto** de tamaño **n** divide dichos elementos en **m** subconjuntos de modo que:

- Cada elemento pertenece a un solo subconjunto
- Ningún elemento pertenece a más de un subconjunto
- La unión de todos los subconjuntos da el total

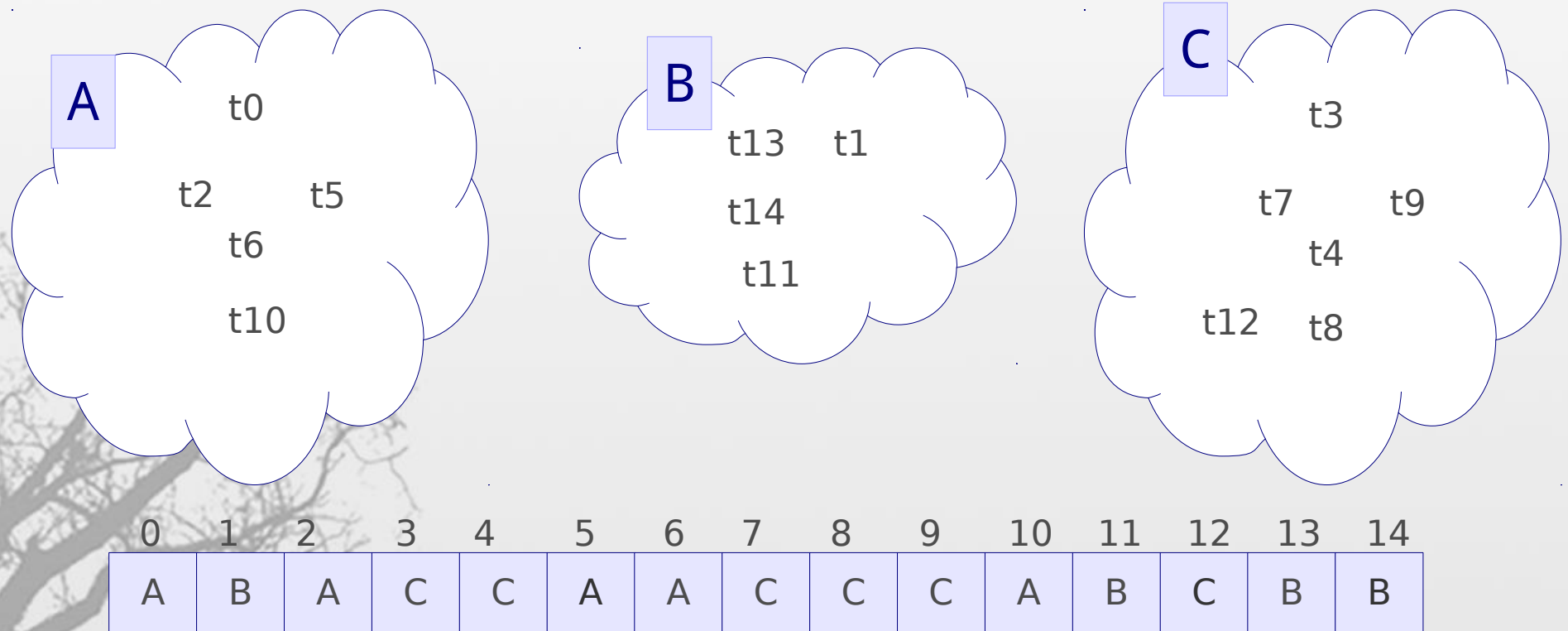
Las operaciones típicas de un conjunto disjunto son:

- **Buscar** el conjunto al que pertenece un elemento
- **Mezclar** dos conjuntos en uno solo

Implementación con vectores



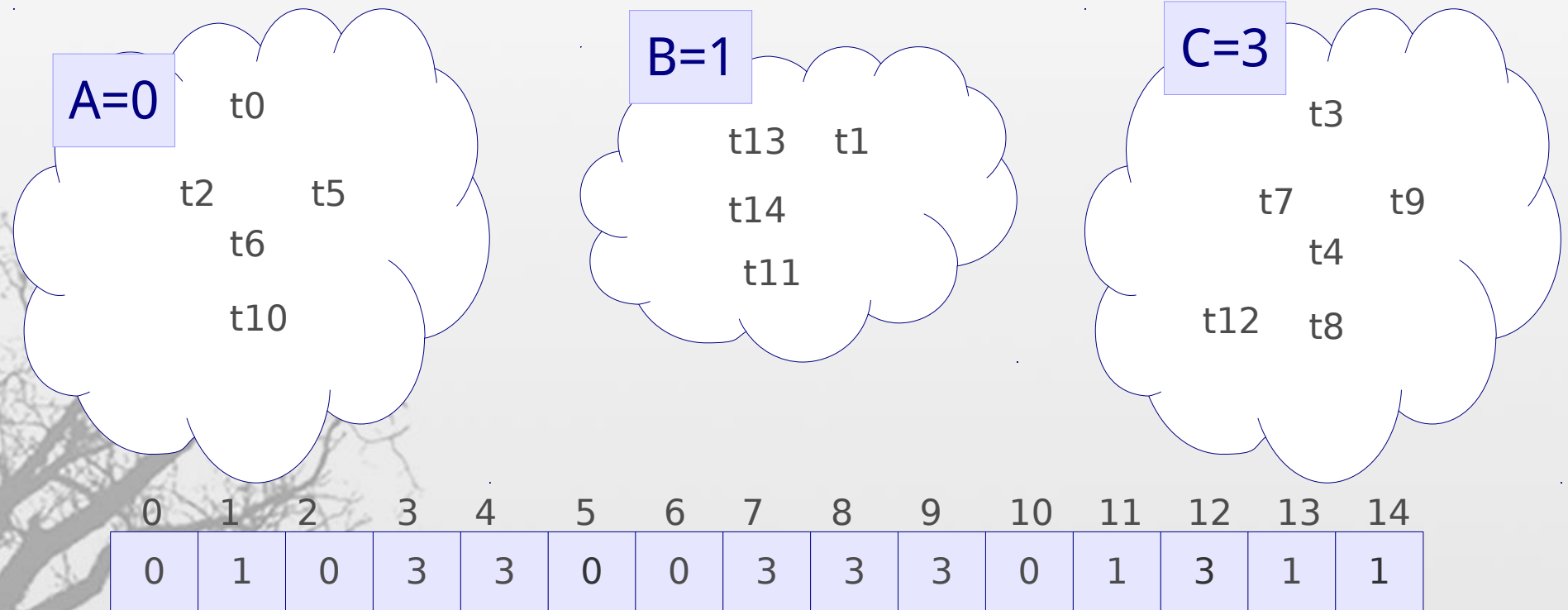
La representación en vectores considera los índices como datos y el contenido el conjunto al que pertenece el dato



Implementación con vectores



Como nombre del conjunto se considera el valor del menor dato del conjunto. Si los datos no son de tipo numérico se les asigna una numeración correlativa.



Operaciones usando vectores



Descripción de operaciones en pseudocódigo

- Crear un conjunto unitario: al inicio cada elemento está en un conjunto distinto
- Buscar ($O(1)$): el índice es el elemento
- Union ($O(n)$): se cambian todas las posiciones iguales a y por x (se asume $x < y$)

```
procedure CreaConjunto(x)
    rep[x] := x;
end

function Buscar(x)
    return rep[x];
end
```

```
procedure Union(x, y)
    for i:=1 to n do
        if rep[i] = y then
            rep[i] := x;
        endif
    endfor
end
```

Operaciones usando vectores



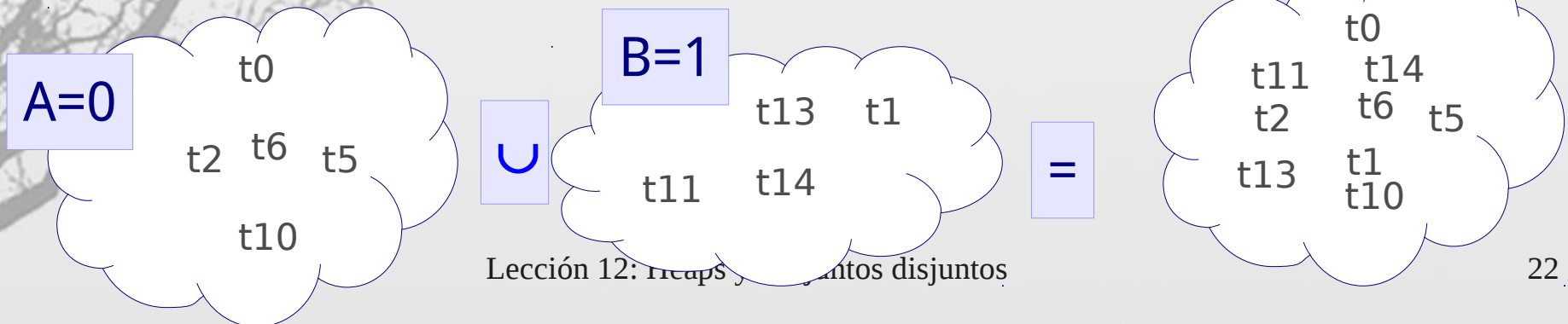
Supongamos que el trabajo de los grupos 0 y 1 va a fusionarse tras obtener los resultados parciales:

Inicialmente:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	1	0	3	3	0	0	3	3	3	0	1	3	1	1

Tras unir conjuntos 0 y 1

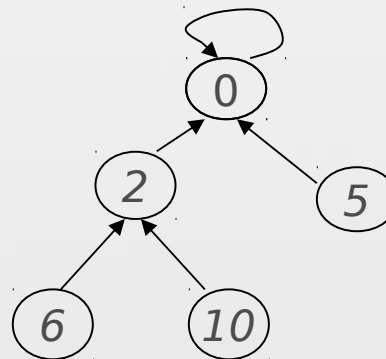
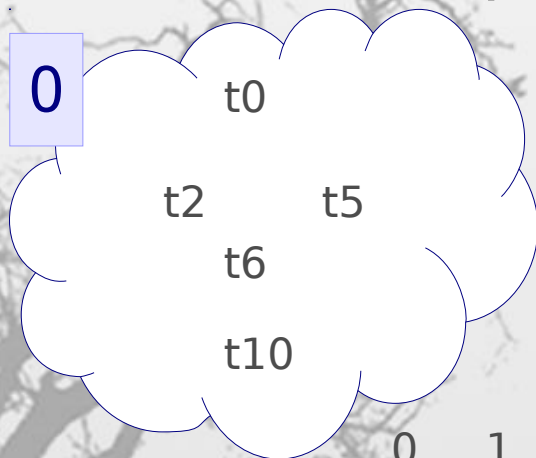
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	0	0	3	3	0	0	3	3	3	0	0	3	0	0



Implementación con árboles

La eedd es un bosque de árboles, cada árbol representa un conjunto (el árbol puede no ser binario)

- La raíz nombra al conjunto
- El índice i representa al elemento i y $p[i]$ contiene el índice del padre
- La raíz cumple: $p[i]=i$

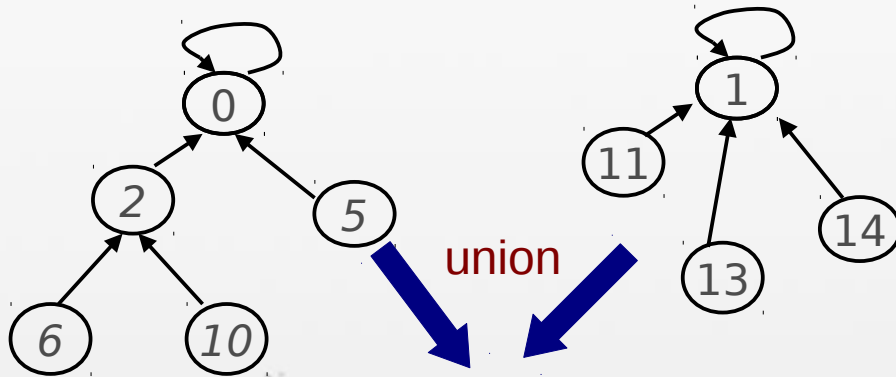


El conjunto 0 queda representado en forma de árbol, el resto como estaba

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	1	0	3	3	0	2	3	3	3	2	1	3	1	1

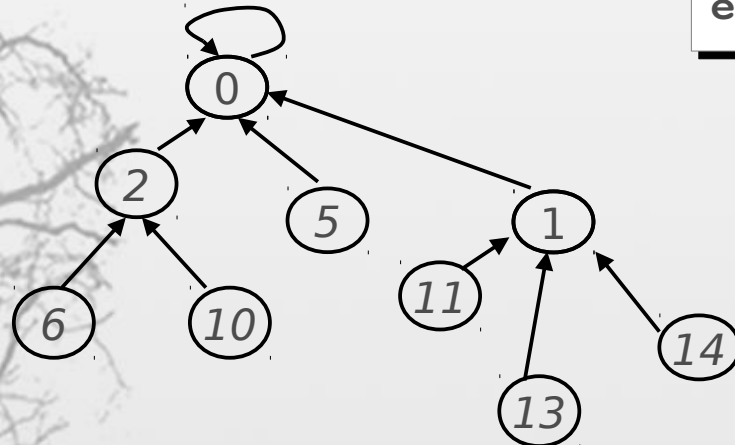
Operaciones usando árboles

- Union: $O(1)$: sólo es necesario enlazar las raíces



```
procedure Unir(x,y)
  if x < y then
    rep[y] = x;
  else
    rep[x] = y;
  endif
end
```

la única
operación




0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	0	0	3	3	0	2	3	3	3	2	1	3	1	1

Operaciones usando árboles

- Crear un conjunto unitario, todos con la misma raíz
- Buscar ($O(n)$): recorro el árbol hasta llegar a la raíz

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	1	0	3	3	0	2	3	3	3	2	1	3	1	1



Buscar (10):

1 rep[10]=2 \neq raíz
2 rep[2]=0 \neq raíz
3 rep[0]=0 \square resultado


```
function Buscar(x)
    r := x;
    while rep[r] <> r do
        r := rep[r]
    endwhile

    return r
end
```

Operaciones usando árboles

- Crear un conjunto unitario, todos con la misma raíz
- Buscar ($O(n)$): recorro el árbol hasta llegar a la raíz

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	1	0	3	3	0	2	3	3	3	2	1	3	1	1



Buscar (10):

1 rep[10]=2 \neq raíz
2 rep[2]=0 \neq raíz
3 rep[0]=0 \square resultado

```
function Buscar(x)
    r := x;
    while rep[r] <> r do
        r := rep[r]
    endwhile

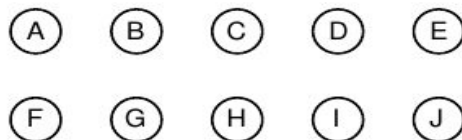
    return r
end
```

Ejemplo de construcción

A	B	C	D	E	F	G	H	I	J

0 1 2 3 4 5 6 7 8 9

(a)

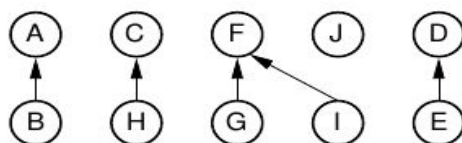


En un principio, cada conjunto tiene un elemento

	0			3		5	2	5	
A	B	C	D	E	F	G	H	I	J

0 1 2 3 4 5 6 7 8 9

(b)

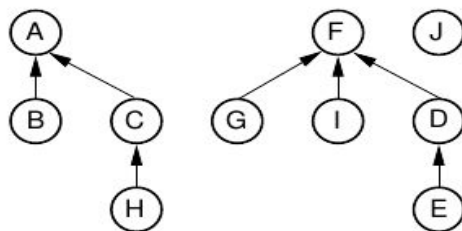


$A \cup B$, $C \cup H$, $(F \cup G) \cup I$, $D \cup E$

	0	0	5	3		5	2	5	
A	B	C	D	E	F	G	H	I	J

0 1 2 3 4 5 6 7 8 9

(c)

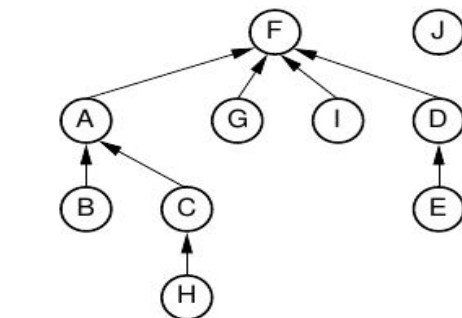


$A \cup C$, $F \cup D$

5	0	0	5	3		5	2	5	
A	B	C	D	E	F	G	H	I	J

0 1 2 3 4 5 6 7 8 9

(d)



$A \cup F$

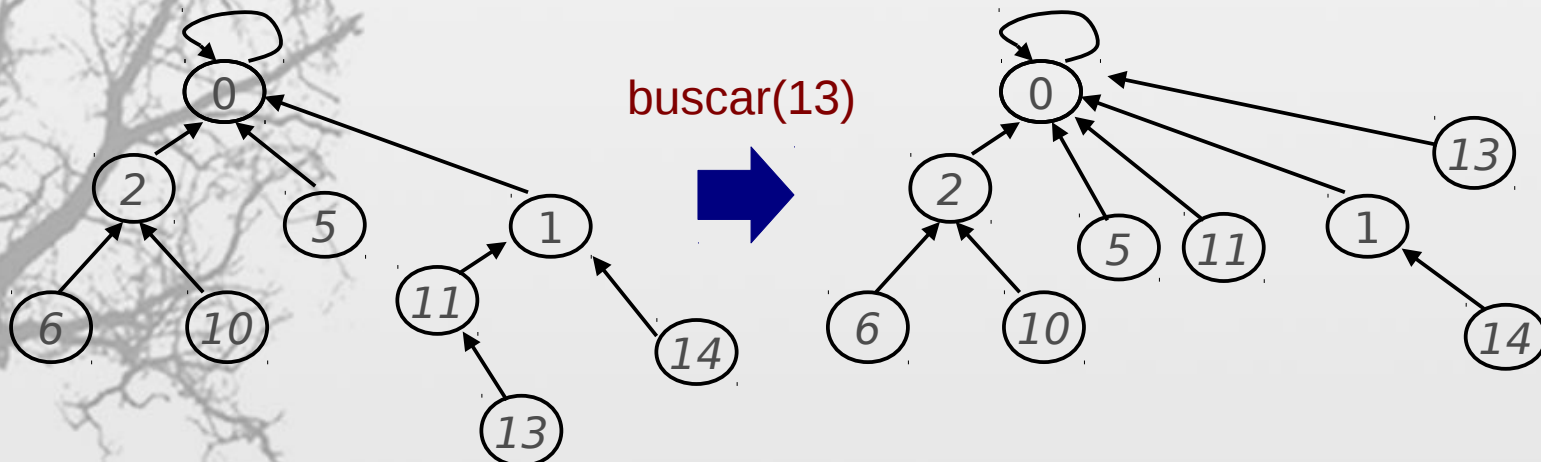
Mejora en la unión



- El procedimiento anterior es aleatorio y puede crear árboles muy descompensados
- Es mejor considerar la altura de los árboles y hacer que el árbol menos profundo sea subárbol del más profundo (como en el ejemplo anterior)
- Si son iguales, entonces elegir uno cualquiera y obtener un nuevo árbol con un nivel más
- **Con este criterio el nodo raíz no tiene por qué ser el de menor valor!**
- Se pueden añadir otra serie de heurísticas que mejoren la estructura de los árboles tras la unión

Mejora en la búsqueda

- La búsqueda será menos eficiente cuanto mayor sea la altura de los árboles
- **Compresión de caminos:** Si tras una búsqueda obtengo el conjunto al que pertenece un elemento y por ende el de los que me encuentro por el camino ¿Por qué no conectarlos directamente con la raíz



Mejora en la búsqueda

- ¡Cuántas más búsquedas se realicen, más eficiente será la EEDD!
- Con la mejora en la unión y la compresión de caminos la búsqueda necesita tiempo $O(n/\log^*n)$ (logaritmo estrella, casi constante)

```
function Buscar(x)
  r := x;
  while rep[r] <> r do
    r := rep[r]
  endwhile

  s := x;
  while rep[s] <> r do
    rep[s] := r
  endwhile

  return r
end
```

Conclusiones



- Aunque varias estructuras de datos pueden solucionar un problema, existen soluciones específicas para problemas como los relativos a los conjuntos disjuntos o a las colas con prioridad
- En estos casos la estructura de datos se puede concebir como un árbol, aunque se use un vector como contenedor
- En ambos casos la solución mediante árboles es la más eficiente