Problema

- Ver Problema 11503
- Discutir una posible estructura de datos útil para este problema

Union-find Disjoint Set

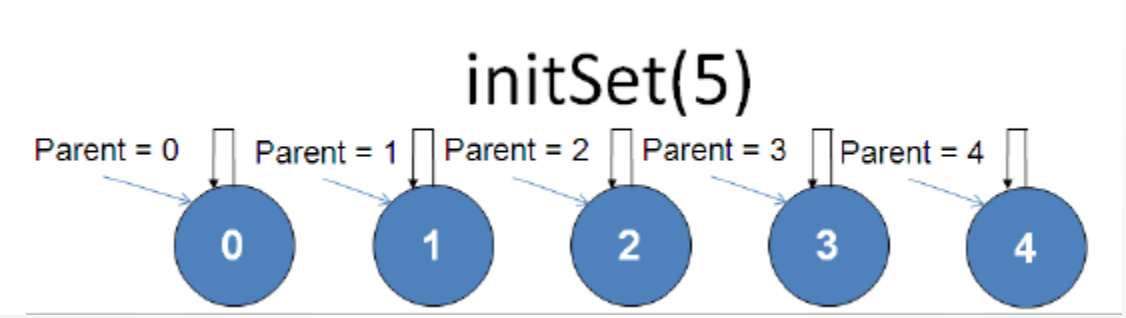
- Es una estructura de datos para modelar datos que inicialmente son disjuntos con la habilidad de hacer las siguientes operaciones de forma eficiente:
  - Comprobar si dos elementos pertenecen al mismo conjunto
  - Unir dos conjuntos
  - Obtener el tamaño de un conjunto

- Estas operaciones no se pueden conseguir con un set de la STL porque con set solo se maneja un conjunto.

- Se puede implementar con un vector < set < … > >, pero no sería eficiente: la unión seria lineal y ocuparía O(n^2) en espacio.

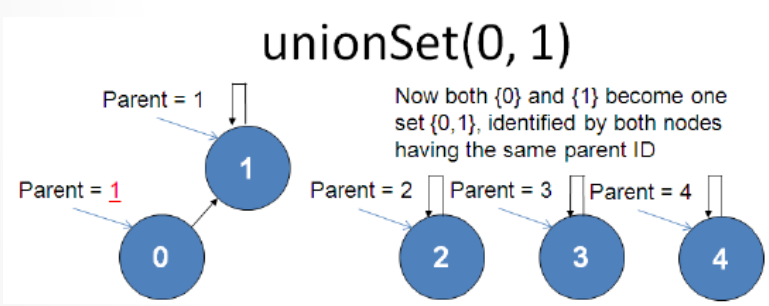
Union-find Disjoint Set

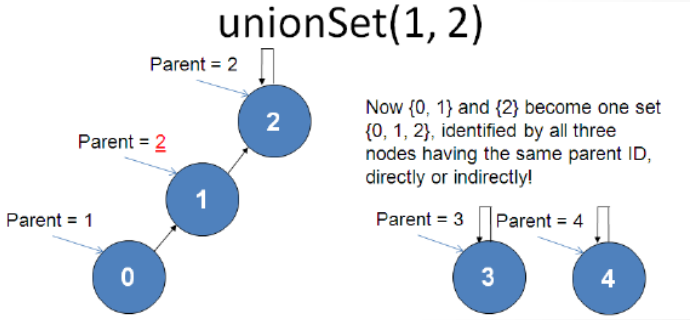
- La idea clave en el Union-find Disjoint Set es mantener para cada conjunto un elemento representante.
- Se forman árboles con los nodos de un grupo, y la raíz es el representante. El padre de cada uno se almacena en vector < int > pset.
- Al inicializar, como cada elemento forma su propio conjunto, pset[i] = i. Las raíces o representantes, se apuntan a sí mismas.

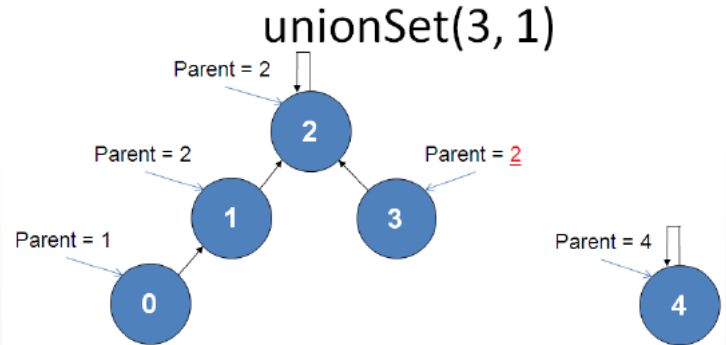


Union-find Disjoint Set

- En la versión más básica del UFDS, para unir los conjuntos del elemento i y el elemento j, se llama a unionSet(i, j) y para hacer esta operación, el representante de i, pasa a apuntar al representante de j.

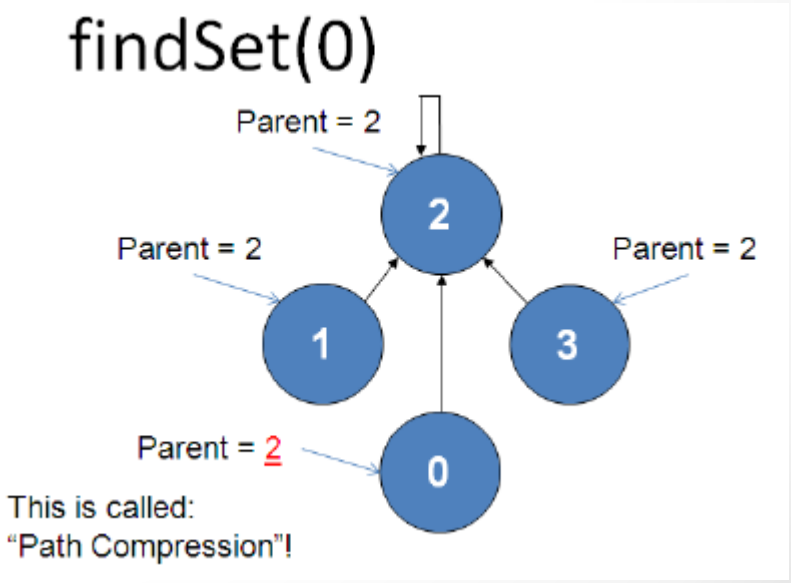






Union-find Disjoint Set

- Para calcular el representante del elemento i, se llama a findSet(i). Esta función comprueba si el padre del elemento es él mismo, y en tal caso ese es el valor que hay que devolver, si no, se puede obtener llamando recursivamente a findSet(pset[i])
- Al recorrer los caminos **se comprimen**.
- Para comprobar si dos elementos son del mismo conjunto, se comprueba si su representante es el mismo.



Union-find Disjoint set

- La implementación de un Union-find Disjoint Set con estas características es muy corta.
- 4 funciones de una o dos líneas:
  - initSet
  - findSet
  - isSameSet
  - unionSet

- No hacemos el análisis de complejidad, pues es una estructura de los cursos anteriores, pero recordamos que todas las operaciones son O(log n) amortizado, salvo el initSet que es O(n). Recordatorio del sigificado de amorizado: una operación específica podría llevar más de O(n), pero eso provoca que otras llevarán menos coste y en media al realizar n operaciones, tenemos coste n * log n.

Union-Find Disjoint Set

- La complejidad anterior, se puede mejorar almacenando para cada conjunto un valor Rank, de forma que Rank + 1 es una cota superior de la altura del árbol y que se calcula de la siguiente forma:
  - Al inicializar, todos los nodos tienen Rank = 0
  - Al unir dos conjuntos, si tienen Rank diferente, se apunta al de menor rango, al de mayor rango, por lo que el Rank del representante no cambia.
  - Al unir dos conjuntos, si tienen igual rango, se suma 1 al rango.

Algorithm	Average	Worst case
Space	$O(n)$[1]	$O(n)$[1]
Search	$O(\alpha(n))$[1]	$O(\alpha(n))$[1]
Merge	$O(\alpha(n))$[1]	$O(\alpha(n))$[1]

- La inversa de la función de Ackermann se puede considerar prácticamente como una constante a niveles prácticos, pues incluso para conjuntos que ocupen toda la memoria de la computadora, el valor es menor que 5.

$\alpha(9876!) = 5.$

Union-Find Disjoint Set

- Ver Código en UnionFind.cpp

bitset

- Es una plantilla de la STL que permite emular el comportamiento de un array de booleanos, pero que está optimizado para el ahorro de memoria.
- Cada valor ocupa un único bit
- Habitualmente hace mejor uso de la Caché, por ocupar menos memoria, con lo que también se pueden obtener beneficios en tiempo.

bitset

```cpp
// constructing bitsets
#include <iostream>       // std::cout
#include <string>         // std::string
#include <bitset>         // std::bitset

int main ()
{
  std::bitset<16> foo;
  std::bitset<16> bar (0xfa2);
  std::bitset<16> baz (std::string("0101111001"));

  std::cout << "foo: " << foo << '\n';
  std::cout << "bar: " << bar << '\n';
  std::cout << "baz: " << baz << '\n';

  return 0;
}
```

```
foo: 0000000000000000
bar: 0000111110100010
baz: 0000000101111001
```

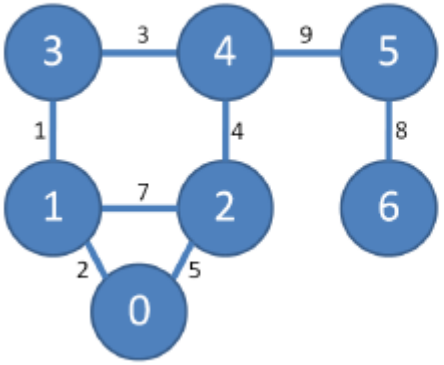
- Tiene métodos set y reset, para poner los bits a 1 ó 0
- Con los corchetes se puede acceder a los valores de los bits.

Máscaras de bits

```
#define isOn(S, j) (S & (1ll << j))

#define setBit(S, j) (S |= (1ll << j))

#define clearBit(S, j) (S &= ~(1ll << j))

#define toggleBit(S, j) (S ^= (1ll << j))

#define lowBit(S) (S & (-S))

#define setAll(S, n) (S = (1ll << n) - 1ll)

#define modulo(S, N) ((S) & (N - 1))   // returns S % N, where N is a power of 2

#define isPowerOfTwo(S) (!(S & (S - 1)))

#define nearestPowerOfTwo(S) ((int)pow(2.0, (int)((log((double)S) / log(2.0)) + 0.5)))

#define turnOffLastBit(S) ((S) & (S - 1))

#define turnOnLastZero(S) ((S) | (S + 1))

#define turnOffLastConsecutiveBits(S)  ((S) & (S + 1))

#define turnOnLastConsecutiveZeroes(S)  ((S) | (S - 1))
```

Grafos

- Matriz de adyacencia
  - Útil si en el algoritmo necesitamos consultar de forma continua el coste de ir de un nodo a otro nodo.
  - No se puede usar en grafos dispersos con muchos nodos.

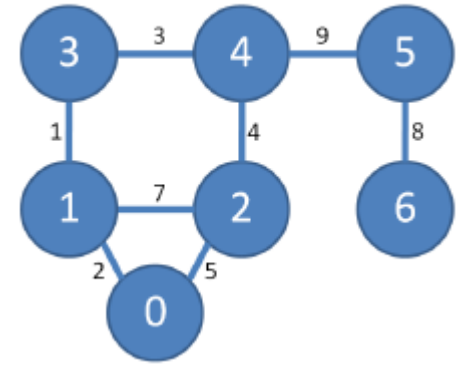


A

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

Grafos

- Lista de adyacencia
  - Útil cuando necesitamos enumerar los vecinos de un nodo

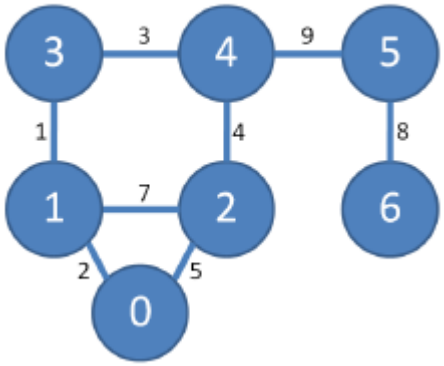


B

0	→	(1,2)	→	(2,5)		
1	→	(0,2)	→	(2,7)	→	(3,1)
2	→	(0,5)	→	(1,7)	→	(4,4)
3	→	(1,1)	→	(4,3)		
4	→	(2,4)	→	(3,3)	→	(5,9)
5	→	(4,9)	→	(6,8)		
6	→	(5,8)				

Grafos

- Lista de aristas
  - Generalmente se mantienen en algún orden determinado
  - Útil en algunos algoritmos que requieren acceder a toda la lista de aristas, como para implementar el algoritmo de Kruskal



C

w	v1	v2
1	1	3
2	0	1
3	3	4
4	2	4
5	0	2
7	1	2
8	5	6
9	4	5

- Nota: es muy habitual que se requieran mantener de forma simultánea varias de las representaciones para resolver un problema de forma eficiente.

Colas de Prioridad Avanzadas

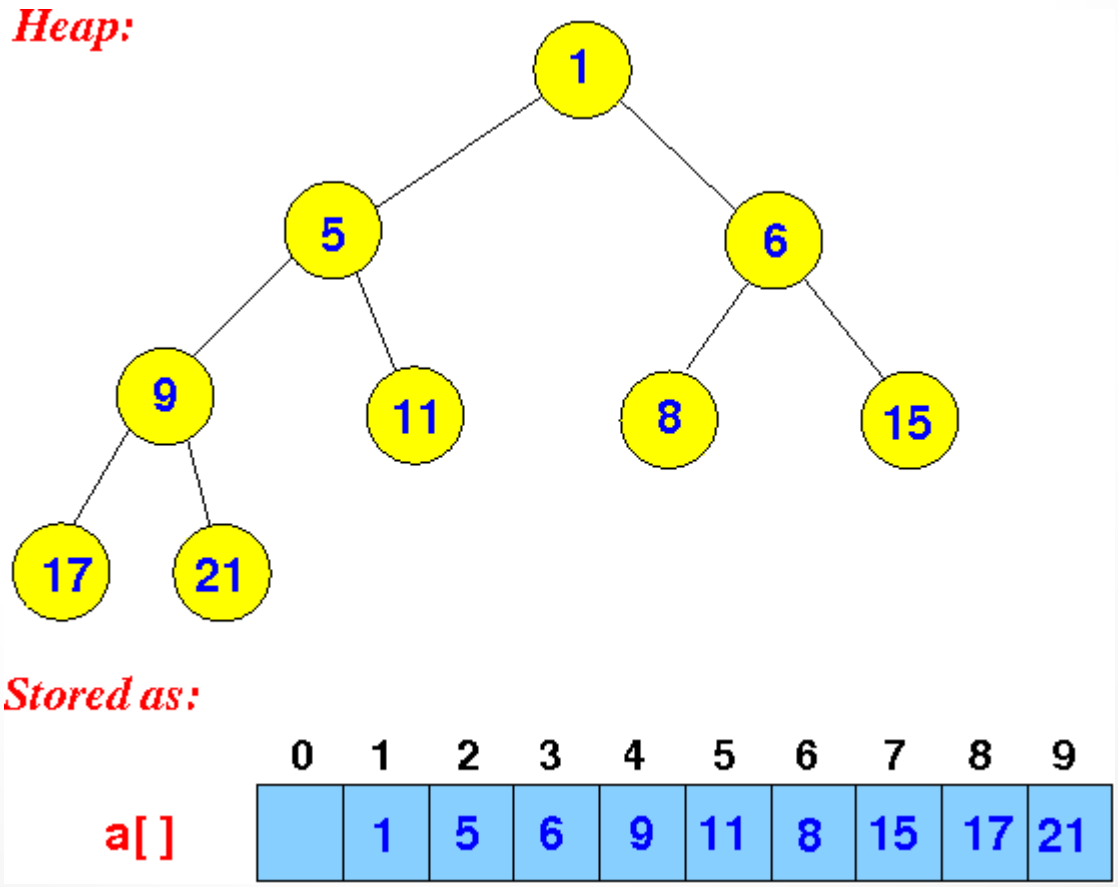
- Supongamos que queremos desarrollar un heap en la que los nodos tienen un identificador único (además de su prioridad). Queremos tener la posibilidad de borrar y/o actualizar (sumar o restar un valor a la prioridad) de un nodo (a partir de su id) de forma eficiente (logaritmo).
- ¿Cómo lo hacemos?
- A esta estructura se le llama "Priority queue with updatable priorities".

Colas de Prioridad Avanzadas

- En primer lugar, dado un identificador, debemos tener la posibilidad de saber cuál es su posición en el Heap. Esto lo podemos hacer utilizando un map, o una tabla Hash.
- También podemos fijarnos que no necesitamos la operación actualizar. Si tenemos la operación borrar e insertar de forma eficiente, actualizar consiste en primero borrar y luego insertar.
- Nos centramos entonces en el borrado de una posición cualquiera del Heap.

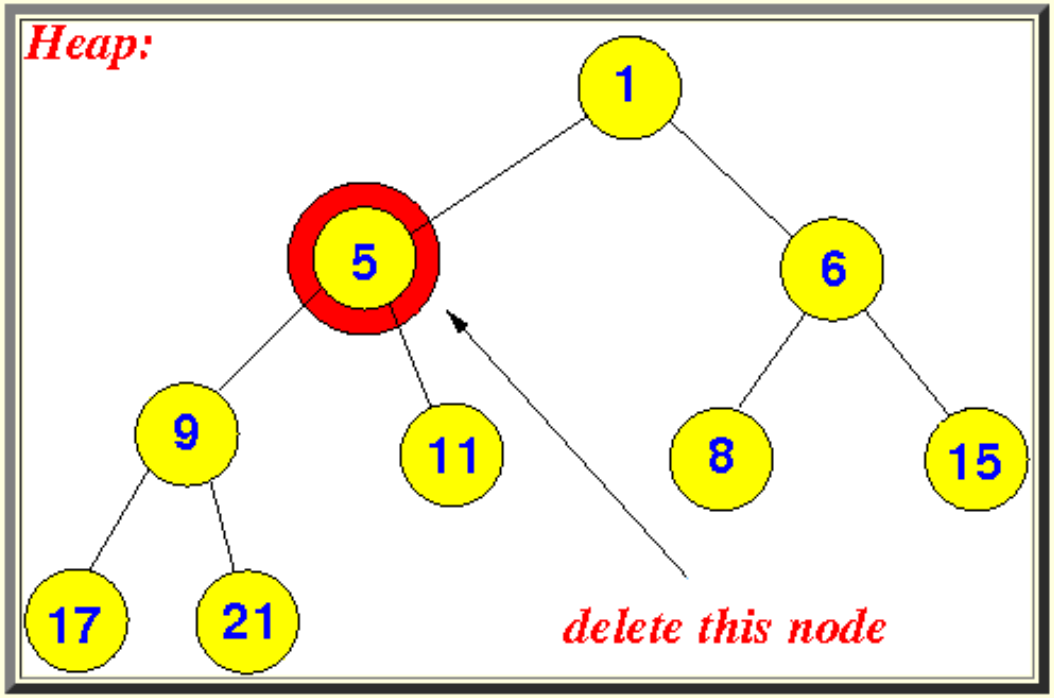
Colas de Prioridad Avanzadas

Ojo: consideramos Min-Heap

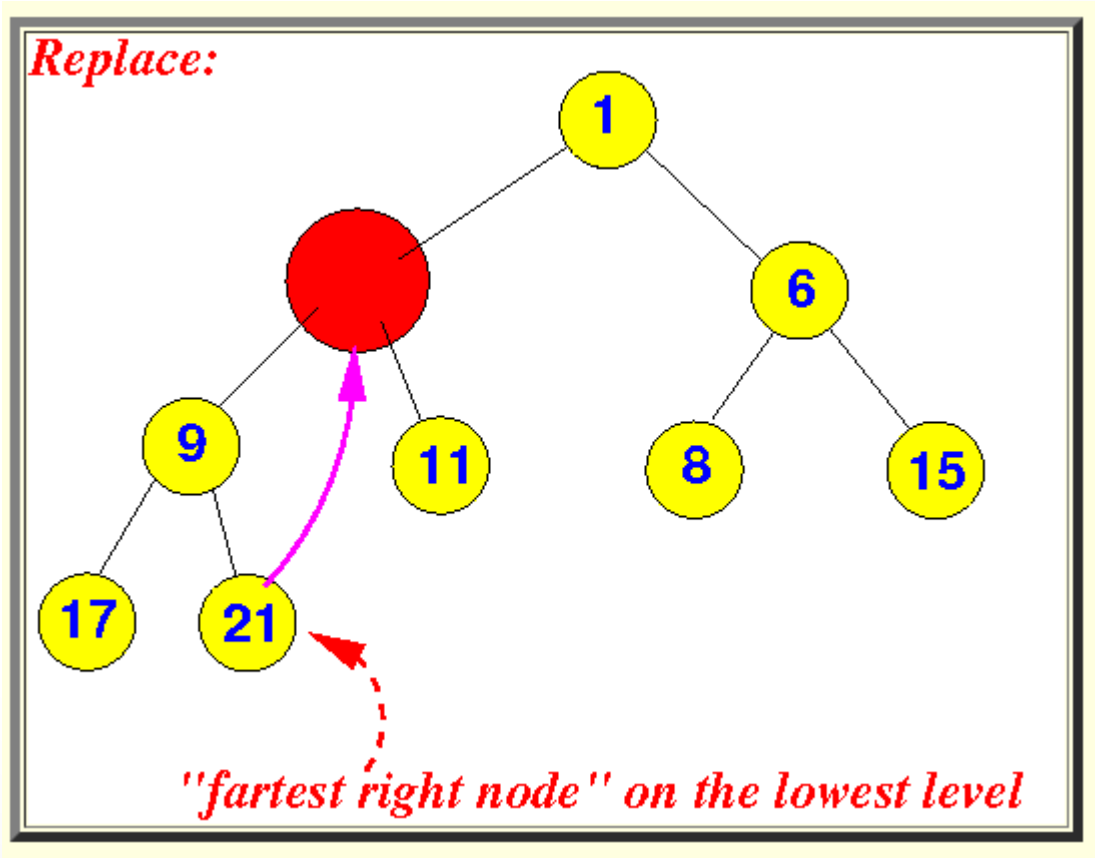


Colas de Prioridad Avanzadas

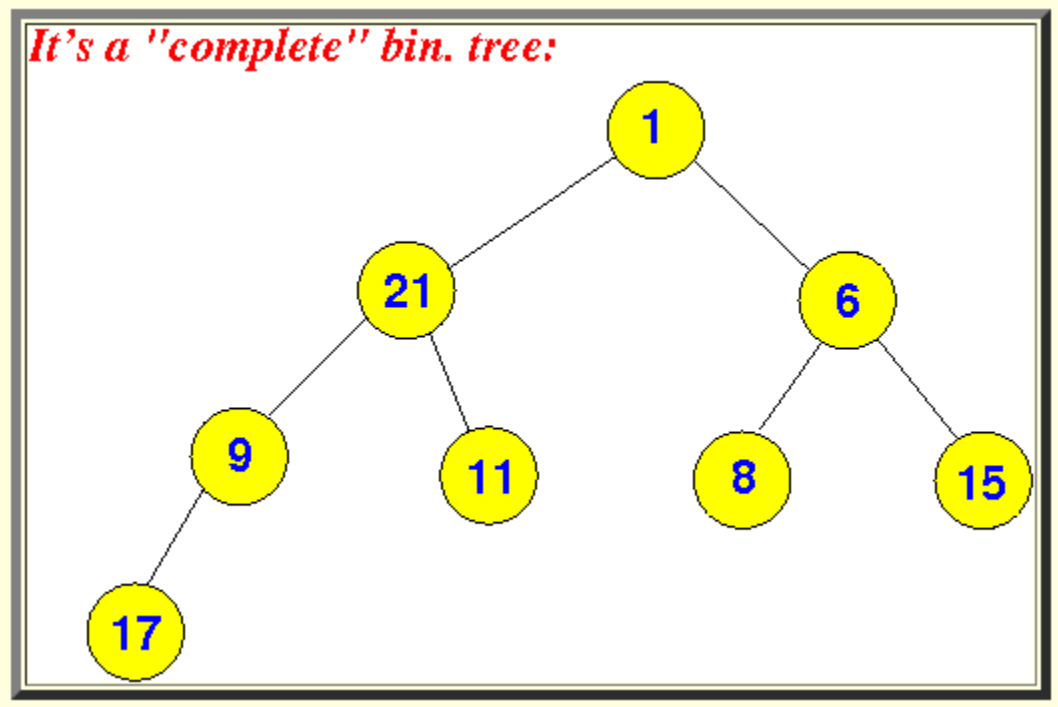
- *Delete* the **node** containing the **value 5** from the following **heap**:



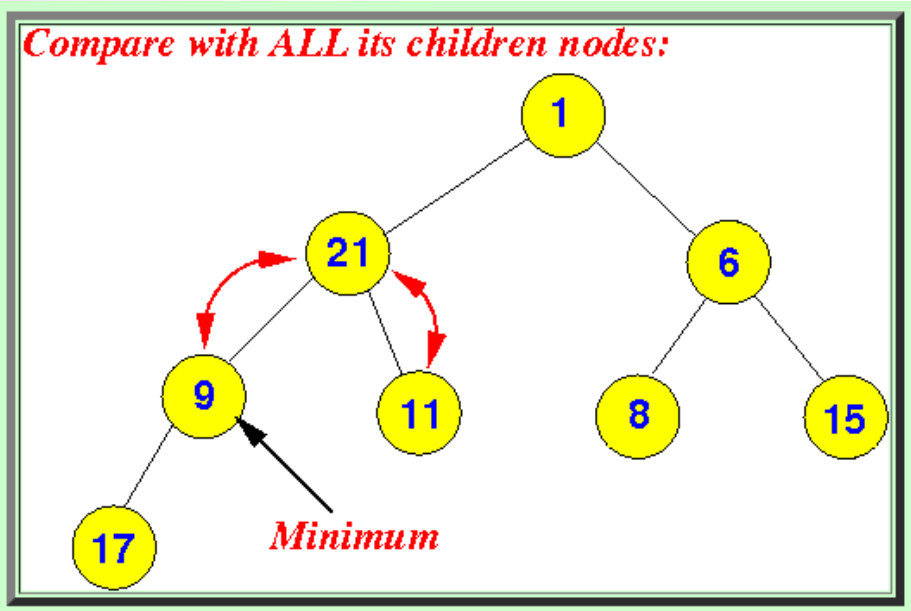
Colas de Prioridad Avanzadas

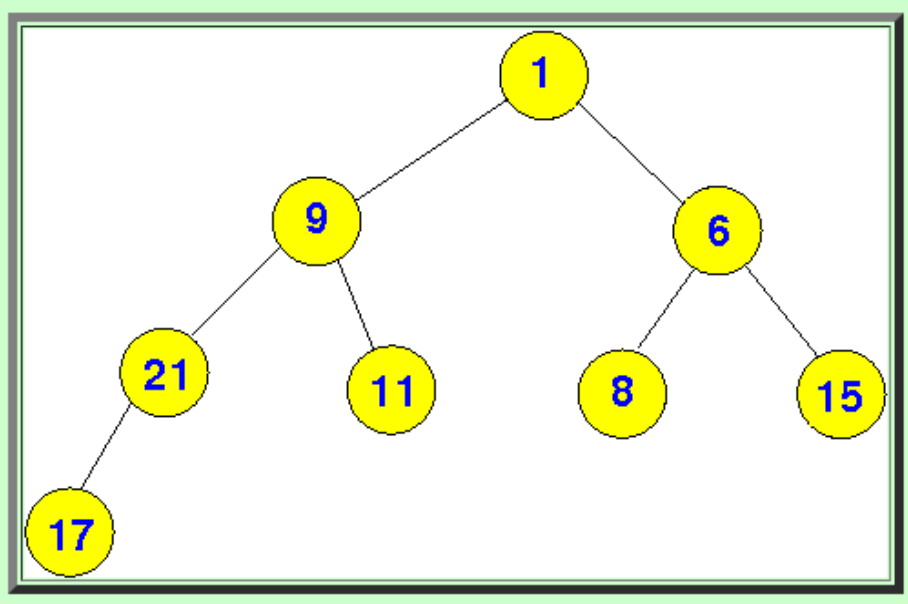


Colas de Prioridad Avanzadas

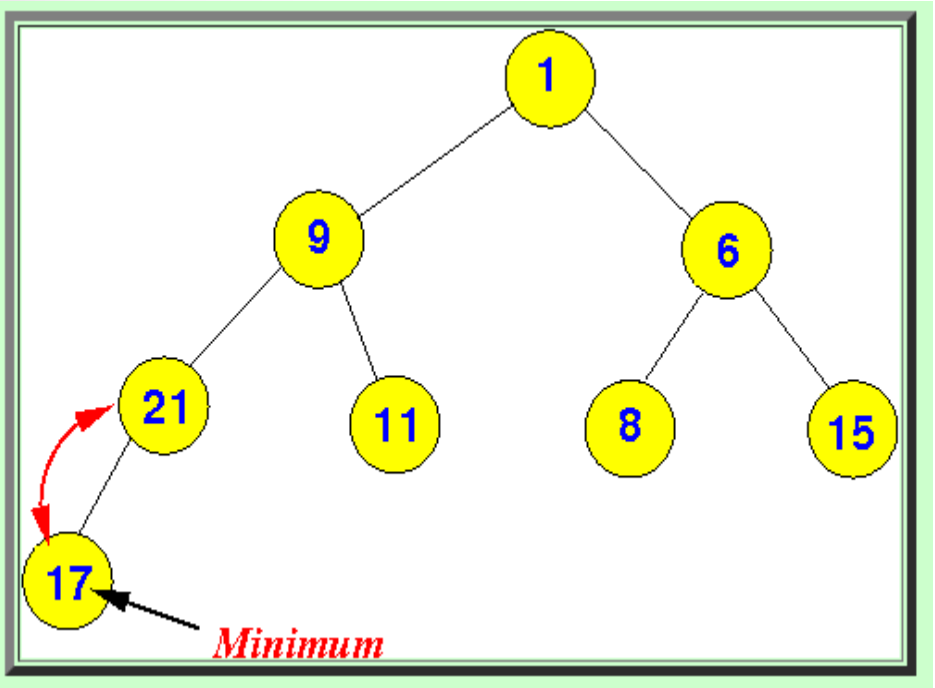


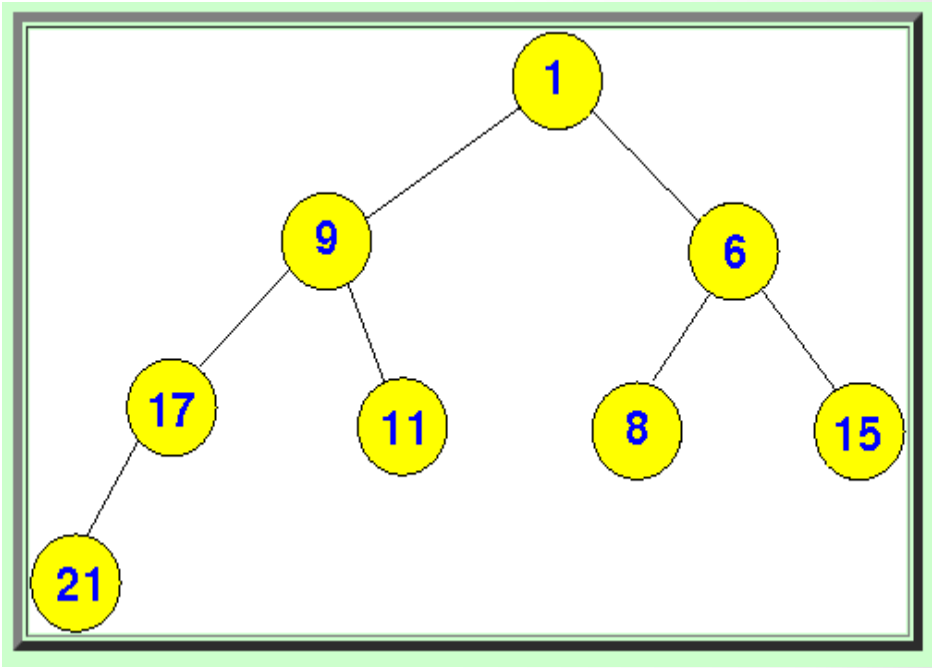
Colas de Prioridad Avanzadas





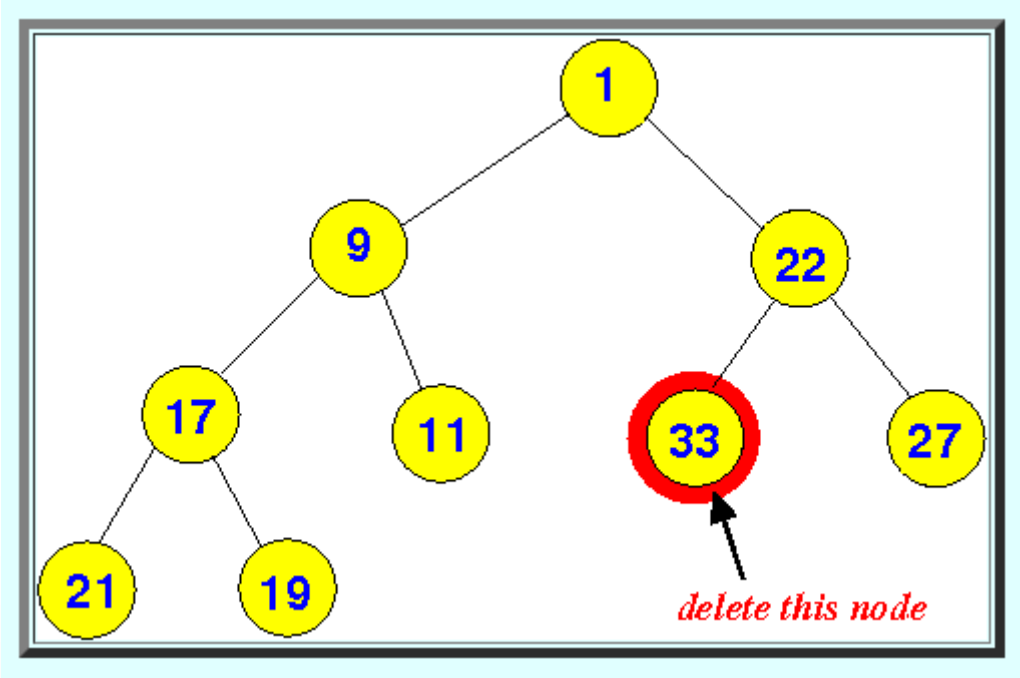
Colas de Prioridad Avanzadas



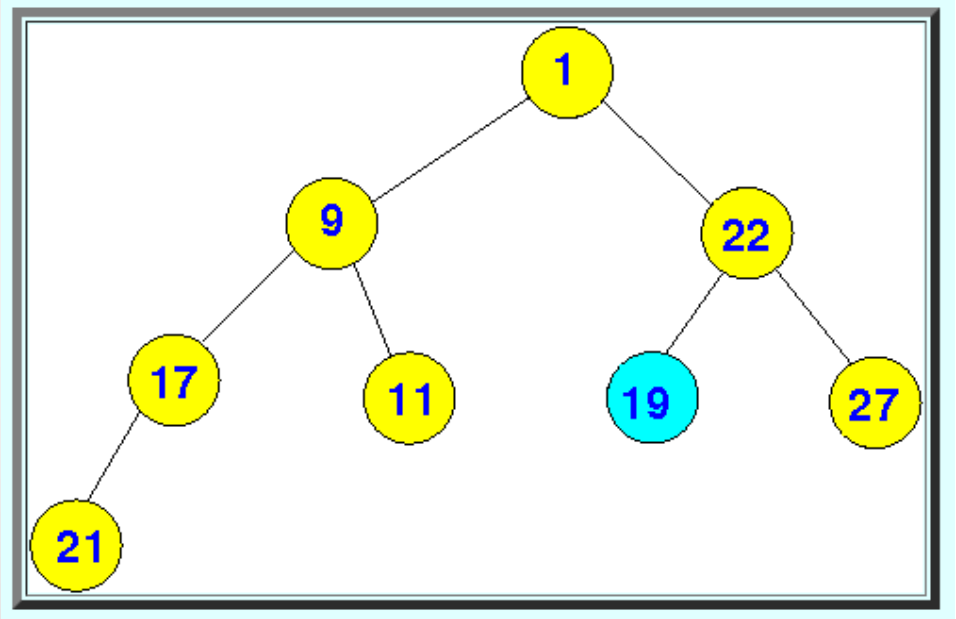


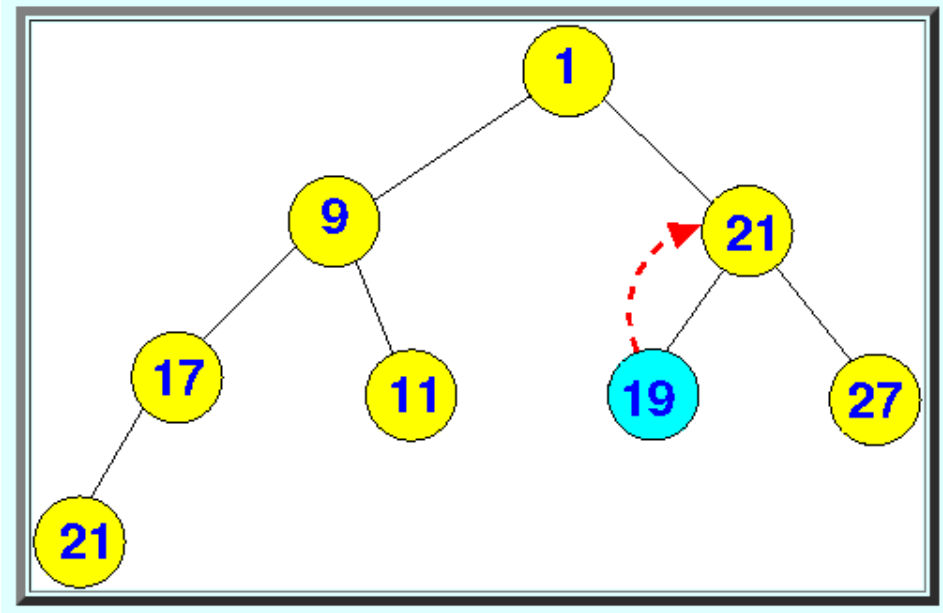
Colas de Prioridad Avanzadas

- ***Sometimes***, you have to **filter** the **replacement node** ***up*** the Binary tree !!!!!

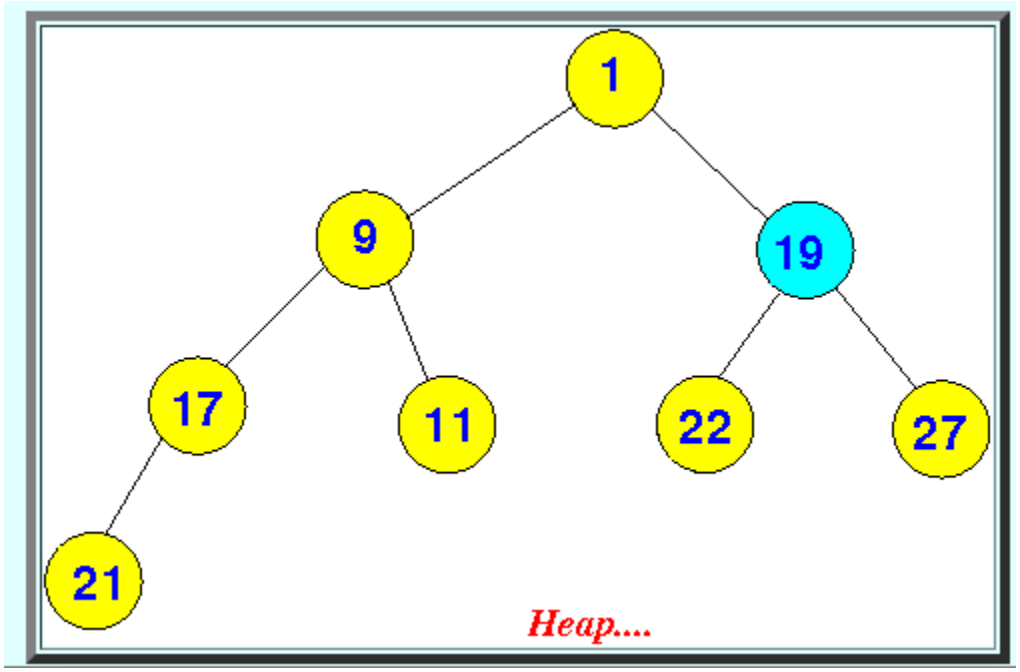


Colas de Prioridad Avanzadas





Colas de Prioridad Avanzadas



Colas de Prioridad Avanzadas

```
1, Delete a node from the array
   (this creates a "hole" and the tree is no longer "complete")

2. Replace the deletion node
   with the "fartest right node" on the lowest level
   of the Binary Tree
   (This step makes the tree into a "complete binary tree")

3. Heapify (fix the heap):

     if ( value in replacement node < its parent node )
        Filter the replacement node UP the binary tree
     else
        Filter the replacement node DOWN the binary tree
```

- Cada vez que cambiamos un nodo de posición, debemos actualizar la estructura auxiliar, que mapeo identificadores a posiciones en el heap.

Treap

- Árbol balanceado, con posibilidad de Split y Merge eficiente.