

# **Programación 2**

## **La Previa: Colecciones 2**

### **Multiconjuntos, Tablas y**

### **Colas de Prioridad**

# Multisets

# Multisets

## Lists

- Hay orden posicional de elementos
- Los elementos pueden repetirse

## Sets

- No hay orden posicional de elementos
- Los elementos no se repiten

## MultiSets

- No hay orden posicional de elementos
- Los elementos pueden repetirse

Ejemplo de Multiset: stock de productos

Relación entre Multisets y Permutaciones

## Un TAD Multiset

- **Vacio**  $m$ : construye el multiset  $m$  vacío;
- **Insertar  $x$   $m$ : agrega  $x$  a  $m$ ;**
- **EsVacio**  $m$ : retorna true si y sólo si el multiset  $m$  está vacío;
- **Ocurrencias**  $x$   $m$ : retorna la cantidad de veces que está  $x$  en  $m$ ;
- **Borrar**  $x$   $m$ : elimina una ocurrencia de  $x$  en  $m$ , si  $x$  está en  $m$ ;
- **Destruir**  $m$ : destruye el multiset  $m$ , liberando su memoria.

# Implementaciones

Multiset  $m = \{(e1, \#e1), \dots, (ei, \#ei), \dots, (en, \#en)\}$

Adaptar y analizar para *multisets* las siguientes implementaciones vistas para conjuntos:

- Variantes de Listas

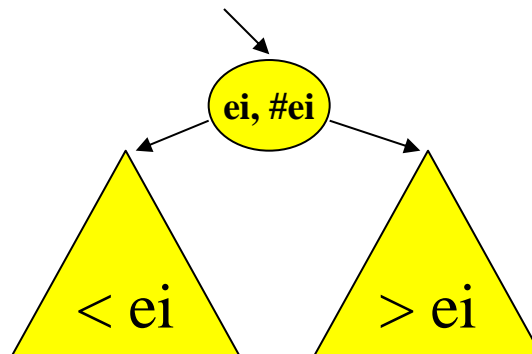
→  $e1 \rightarrow e2 \rightarrow e1 \rightarrow \dots$

→  $e1, \#e1 \rightarrow e2, \#e2 \rightarrow \dots$

- Arreglos de Booleanos (ahora...)



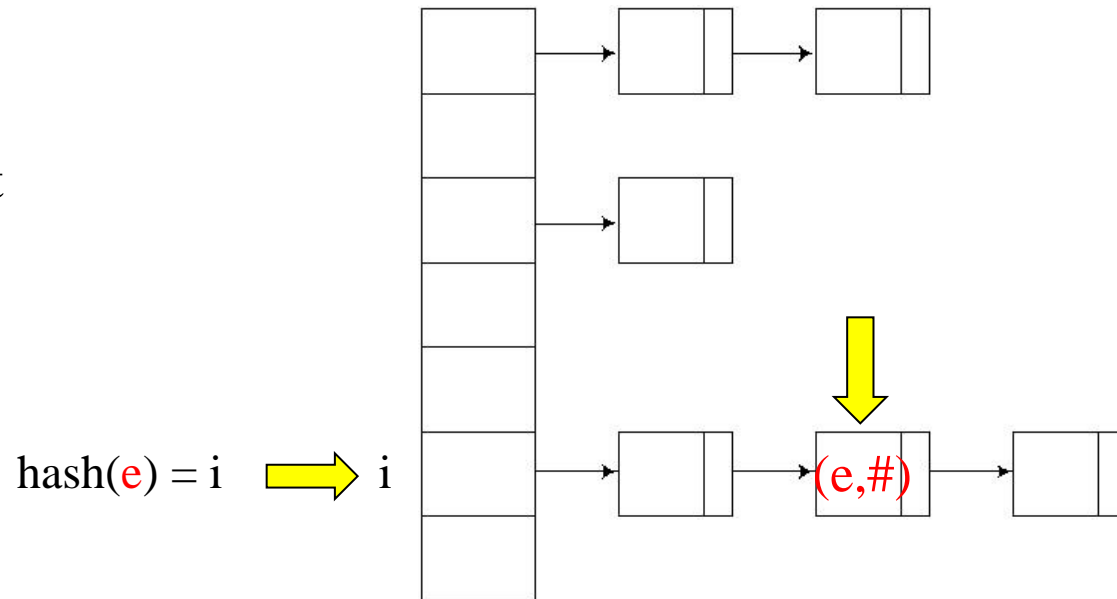
- ABBs



# Implementaciones (cont)

- Hashing abierto de elementos de tipo T (con multiplicidades)

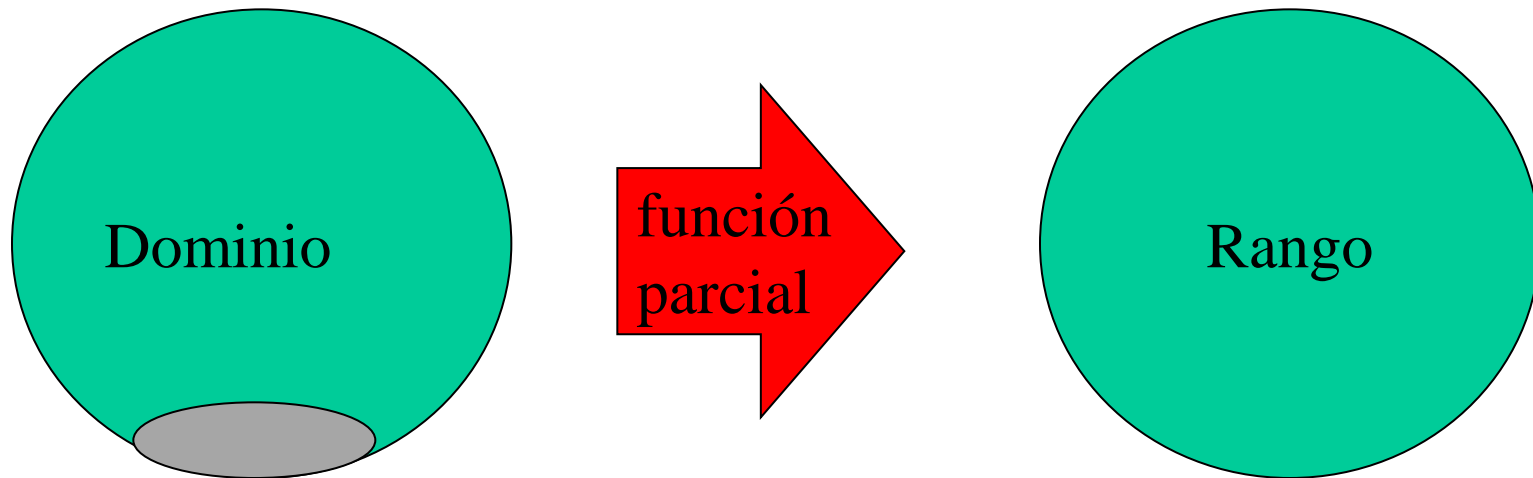
hash: T  $\rightarrow$  int



# Tablas - Funciones Parciales (*Mappings*)

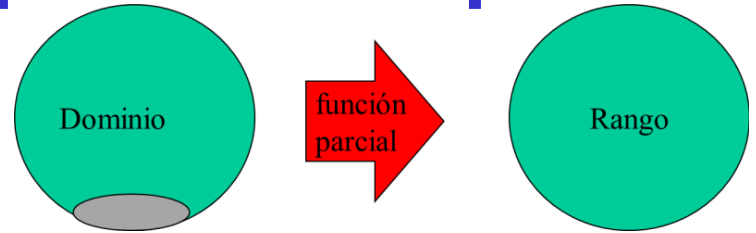
# El TAD Tabla (Función parcial, *Mapping*)

Una **tabla** es una **función parcial** de elementos de un tipo, llamado el tipo dominio, a elementos de otro (posiblemente el mismo) tipo, llamado el tipo recorrido o rango o codominio.





# TAD Tabla/Mapping. Operaciones para:



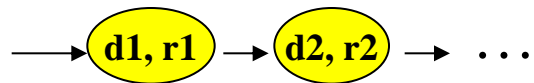
- construir una **tabla vacía**;
- **insertar** una correspondencia  $(d,r)$  en una tabla  $t$ . Si  $d$  está definida en  $t$  (tiene imagen), actualiza su correspondencia con  $r$ ;
- saber si una tabla **está vacía**;
- saber si un valor  $d$  **tiene imagen** en una tabla  $t$ ;
- **obtener la imagen** de un valor  $d$  (que existe) en una tabla  $t$ ;
- **eliminar** una correspondencia de una tabla, dado un valor del dominio.
- destruir una tabla.

**Tabla  $t \subseteq \text{Dominio} \times \text{Rango} = \{(d_1, r_1), \dots, (d_i, r_i), \dots, (d_n, r_n)\}$**

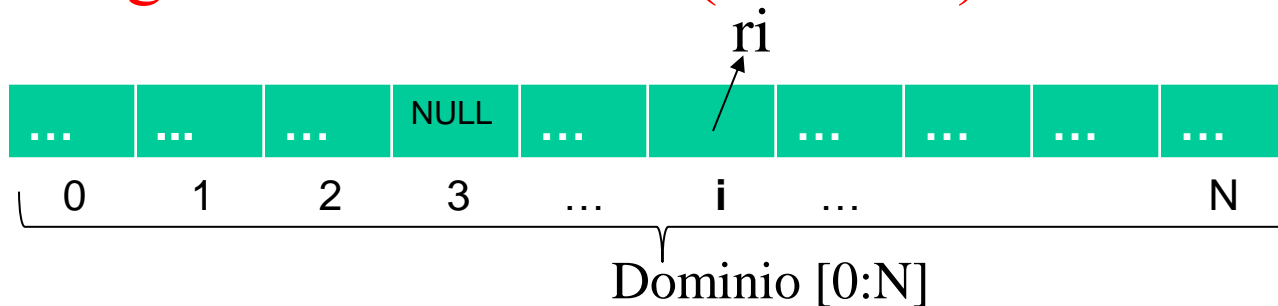
# Implementaciones

- Adaptar y analizar para *mappings* las siguientes implementaciones vistas para conjuntos:

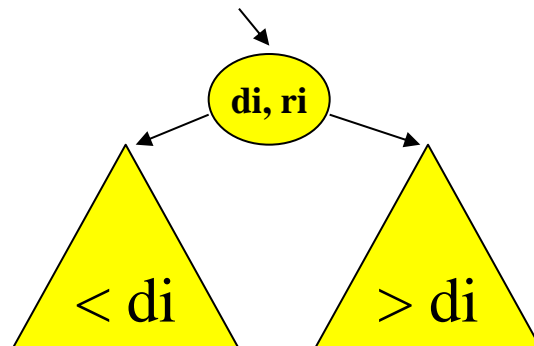
- Listas



- Arreglos de Booleanos (ahora...)



- ABBs



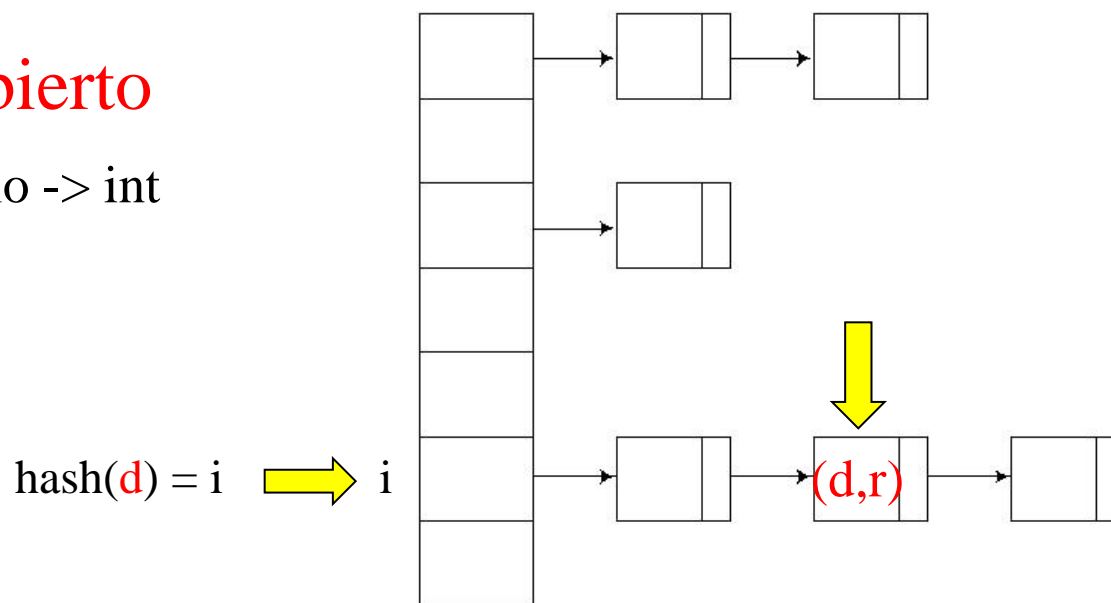
## Implementaciones (cont)

- Adaptar y analizar para *mappings* las siguientes implementaciones vistas para conjuntos:
  - Arreglos con tope de pares  $(d,r)$  ordenado por dominio



- Hashing abierto

hash: Dominio  $\rightarrow$  int



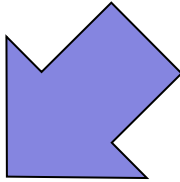
# Insertar con Hashing

```
unsigned int hash (D d){return ...;}
```

```
\\ hash podría aplicarse a d y un entero M, realizando %M internamente
```

```
struct nodoHash{  
    D dom;  
    R ran;  
    nodoHash* sig;  
}
```

```
struct RepresentacionTabla{  
    nodoHash** tabla;  
    int cantidad;  
    int cota;  
}
```



insertar una correspondencia (d,r)  
en una tabla t. Si d está definida en  
t (tiene imagen), actualiza su  
correspondencia con r.

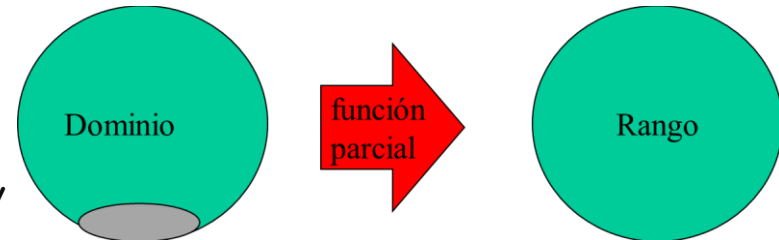
```
void insertarTabla (D d, R r, Tabla &t) {  
    int posicion = hash(d)%(t->cota); \\ hash podría aplicarse a d y t->cota...  
    nodoHash* lista = t->tabla[posicion];  
    while (lista!=NULL && lista->dom!=d)  
        lista = lista->sig;  
    if (lista==NULL){  
        nodoHash* nuevo = new nodoHash;  
        nuevo->dom = d;  
        nuevo->ran = r;  
        nuevo->sig = t->tabla[posicion];  
        t->tabla[posicion] = nuevo;  
        t->cantidad++;  
    }  
    else lista->ran = r;  
}
```

## Especificación del TAD Tabla no acotada de D en R ( $D \rightarrow R$ )

```
#ifndef _TABLA_H
#define _TABLA_H
struct RepresentacionTabla;
typedef RepresentacionTabla * Tabla;

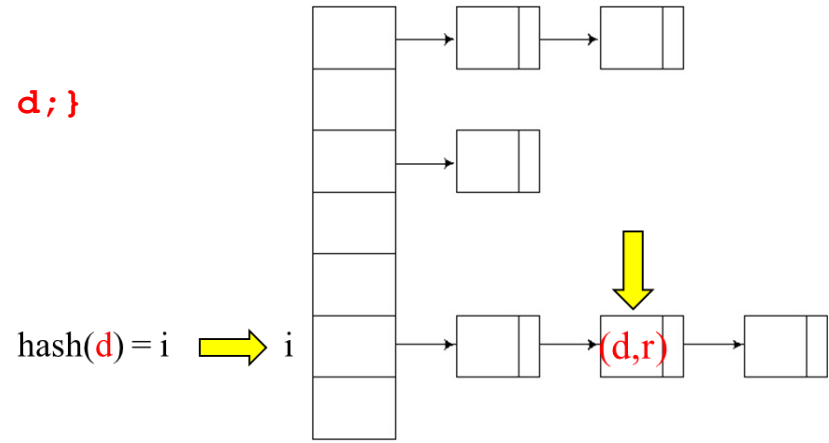
Tabla crearTabla (int cantidadEsperada);
// Devuelve la Tabla vacía no acotada, donde se estiman cantidadElementos.
void insertarTabla (D d, R r, Tabla &t);
/* Agrega la correspondencia (d,r) en t, si d no tenía imagen en t. En
   caso contrario actualiza la imagen de d con r. */
bool estaDefinidaTabla (D d, Tabla t);
// Devuelve true si y sólo si d tiene imagen en t.
bool esVaciaTabla (Tabla t);
// Devuelve true si y sólo si t es vacía.
R recuperarTabla (D d, Tabla t);
/* Retorna la imagen de d en t.
   Precondición: estaDefinidaTabla(d,t). */
void eliminarTabla (D d, Tabla &t);
/* Elimina de t la correspondencia que involucra a d, si d está definida
   en t. En caso contrario la operación no tiene efecto. */
int cantidadEnTabla (Tabla &t);
// retorna la cantidad de correspondencias (d,r) en t.
Tabla copiarTabla (Tabla t);
// retorna una copia de t sin compartir memoria.
void destruirTabla (Tabla &t);
// Libera toda la memoria ocupada por t.

#endif /* _Tabla_H */
```



# Implementación de un Tabla no acotada de D en R (D→R) con hashing abierto

```
#include ...
#include "Tabla.h"
int hash (D d){ return ... }
\\unsigned int hash (unsigned int d){return d;}
struct nodoHash{
    D dom;
    R ran;
    nodoHash* sig;
}
struct RepresentacionTabla{
    nodoHash** tabla;
    int cantidad;
    int cota;
}
Tabla crearTabla (int cantidadEsperada) {
    Tabla t = new RepresentacionTabla();
    t->tabla = new (nodoHash*) [cantidadEsperada];
    for (int i=0; i<cantidadEsperada; i++) t->tabla[i]=NULL;
    t->cantidad = 0;
    t->cota = cantidadEsperada;
    return t;
}
```

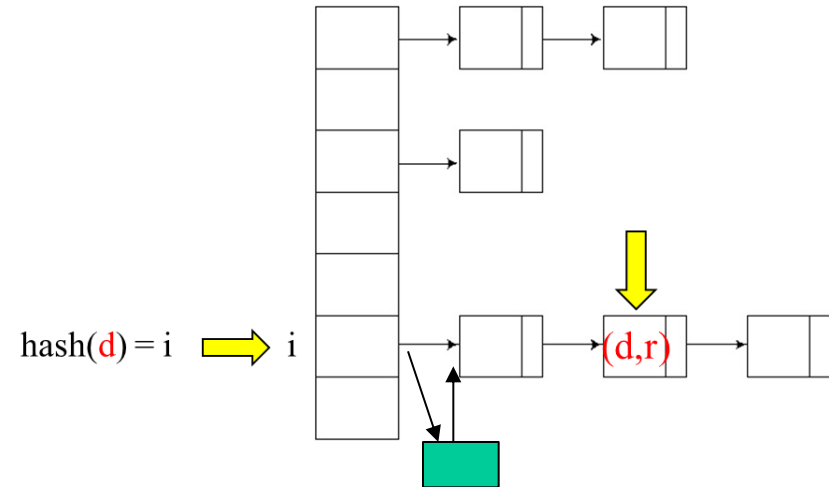


# Implementación de Tabla acotado de D en R (D→R) con hashing abierto

```
void insertarTabla (D d, R r, Tabla &t) {  
    int posicion = hash(d) % (t->cota);  
    nodoHash* lista = t->tabla[posicion];  
    while (lista!=NULL && lista->dom!=d)  
        lista = lista->sig;  
    if (lista==NULL){  
        nodoHash* nuevo = new nodoHash;  
        nuevo->dom = d;  
        nuevo->ran = r;  
        nuevo->sig = t->tabla[posicion];  
        t->tabla[posicion] = nuevo;  
        t->cantidad++;  
    }  
    else lista->ran = r;  
}
```

```
bool estaDefinidaTabla (D d, Tabla t) {  
    int posicion = hash(d) % (t->cota);  
    nodoHash* lista = t->tabla[posicion];  
    while (lista!=NULL && lista->dom!=d)  
        lista = lista->sig;  
    return lista!=NULL;  
}
```

```
bool esVacíaTabla (Tabla t) { return t->cantidad==0; }  
...
```



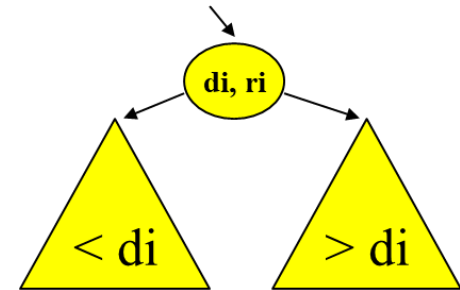
# Implementación de un Tabla no acotada de D en R ( $D \rightarrow R$ ) con un ABB

```
#include ...
#include "Tabla.h"

struct nodoABB{
    D dom;
    R ran;
    nodoABB* izq;
    nodoABB* der;
};

struct RepresentacionTabla{
    nodoABB* abb;
    int cantidad;
};

Tabla crearTabla (int cantidadEsperada) {
    Tabla t = new RepresentacionTabla();
    t->abb = NULL;
    t->cantidad = 0;
    return t;
}
```



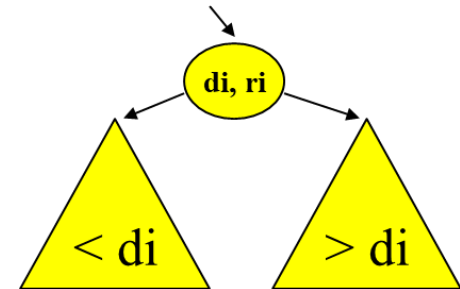


# Implementación de Tabla acotado de D en R ( $D \rightarrow R$ ) con un ABB

/\* Inserta en el ABB la pareja (d,r); si d está en a actualiza el r y devuelve false. En caso contrario retorna true. \*/

```
bool insertarABB (D d, R r, nodoABB* & a){
    if (a==NULL){
        a = new nodoABB;
        a->dom = d; a->ran = r;
        a->izq = a->der = NULL;
        return true;
    }
    else if (d == a->dom){
        a->ran = r;
        return false;
    }
    else if (d < a->dom) return insertarABB(d, r, a->izq);
    else return insertarABB(d, r, a->der);
}
```

```
void insertarTabla (D d, R r, Tabla &t) {
    if (insertarABB(d, r, t->abb))
        t->cantidad++;
}
```



## Ejemplo - Especificación

Considere la especificación del TAD *Tabla* no acotada de *unsigned int* (dominio) en *float* (codominio):

```
struct RepTabla;
```

```
typedef RepTabla * Tabla;
```

```
typedef unsigned int nat;
```

```
// POS: Devuelve la Tabla vacía, sin correspondencias.
```

```
Tabla crear();
```

```
/* POS: Agrega la correspondencia (d,c) en t, si d no tenía imagen en t. En caso contrario actualiza la imagen de d con c. */
```

```
void insertar (nat d, float c, Tabla & t);
```

```
// POS: Devuelve true si y sólo si d tiene imagen en t.
```

```
bool definida (nat d, Tabla t);
```

```
// POS: Devuelve la cantidad de correspondencias en t. En particular, 0 si t es la tabla vacía.
```

```
int cantidad (Tabla t);
```

## Ejemplo - Especificación

*// PRE: definida(d,t). POS: Retorna la imagen de d en t.*

*float recuperar (nat d, Tabla t);*

*/\* POS: Elimina de t la correspondencia que involucra a d, si d está definida en t. En otro caso la operación no tiene efecto. \*/*

*void eliminar (nat d, Tabla & t);*

*// PRE: cantidad(t) != 0. POS: Retorna el mínimo valor del dominio que tiene imagen en t.*

*nat minDomino (Tabla t);*

*// PRE: cantidad(t) != 0. POS: Retorna el máximo valor del dominio que tiene imagen en t.*

*nat maxDominio (Tabla t);*

*//POS: Imprime las correspondencias (d,c) de t, ordenadas de mayor a menor por los valores del dominio (d).*

*void imprimir(Tabla t);*

## Ejemplo - Uso

Una empresa almacena los precios de sus productos en tablas (de tipo *Tabla*), donde el dominio de tipo *unsigned int* (*nat*) corresponde a los identificadores de los productos (no acotados) y el codominio de tipo *float* corresponde a los precios. La empresa quiere evitar inconsistencias de precios de productos de varias tablas y para esto se propone implementar una función iterativa *preciosUnicos* que, dadas dos tablas *t1* y *t2* (de tipo *Tabla*) no vacías genere una nueva tabla (de tipo *Tabla*) que contenga las correspondencias entre productos y precios que no generan conflictos entre *t1* y *t2*. Esto es, una correspondencia (producto, precio) estará en la tabla resultado si y solo si:

- su producto está en una sola tabla (ó en *t1* ó en *t2*), ó
- si el producto está en ambas tablas (*t1* y *t2*), el precio tiene que ser el mismo.

Implemente *preciosUnicos* sin acceder a la representación del TAD *Tabla* y sin modificar las tablas parámetro.

```
// PRE: t1 y t2 no vacías
```

```
Tabla preciosUnicos(Tabla t1, Tabla t2)
```

## Ejemplo - Uso

```
Tabla preciosUnicos(Tabla t1, Tabla t2){  
    Tabla res = crear();  
    nat inf = min (minDominio(t1), minDominio(t2)); // min: mínimo en nat  
    nat sup = max (maxDominio(t1), maxDominio(t2)); // max: máximo en nat  
    bool def_t1, def_t2;  
    float precio;  
    for (nat i = inf; i <= sup; i++) {  
        def_t1 = definida(i, t1); def_t2 = definida(i, t2);  
        if (def_t1 && def_t2){  
            precio = recuperar(i, t1);  
            if (precio == recuperar(i, t2)) insertar(i, precio, res);  
        }  
        else if (def_t1) insertar(i, recuperar(i, t1), res);  
        else if (def_t2) insertar(i, recuperar(i, t2), res);  
    }  
    return res;  
}
```

# Colas de Prioridad

# Colas de Prioridad

Una cola de prioridad es un Tad Set o un MultiSet con las operaciones básicas:

**(constructores) Vacio, Insertar**

**(predicado) EsVacio**

**(selectores) Borrar\_Min y Min (Borrar\_Max y Max)**

Cada elemento podría ser su prioridad o tener asociado un valor de prioridad (número natural).

# Colas de Prioridad: Implementaciones

Prácticamente todas las realizaciones estudiadas para conjuntos (diccionarios) o multiconjuntos son también apropiadas para colas de prioridad. **Es decir ¿?**

Usando un lista no ordenada, ¿ cuáles son los tiempos de Insertar y Min (o Borrar\_Min) ?

Y ¿ si la lista se mantuviese ordenada ?



# Colas de Prioridad: Implementaciones

Usando un ABB, ¿ cuáles son los tiempos de Insertar y Min (o Borrar\_Min) ?, ¿ en el peor caso o en el caso promedio ?. Y un AVL ?.

Existe una alternativa para implementar colas de prioridad que lleva  $O(\log n)$  --en el peor caso-- para las operaciones referidas (Min es  $O(1)$ ) y no necesita punteros: **Los montículos o Heaps (Binary Heaps).**

# Los montículos o Heaps (Binary Heaps)

Los **Heaps** tienen 2 propiedades (al igual que los AVL):

- una propiedad de la estructura
- una propiedad de orden del heap

Las operaciones van a tener que preservar estas propiedades.

NOTA: la prioridad (el orden) puede ser sobre un campo de la información y no sobre todo el dato.

# Los montículos o Heaps (Binary Heaps)

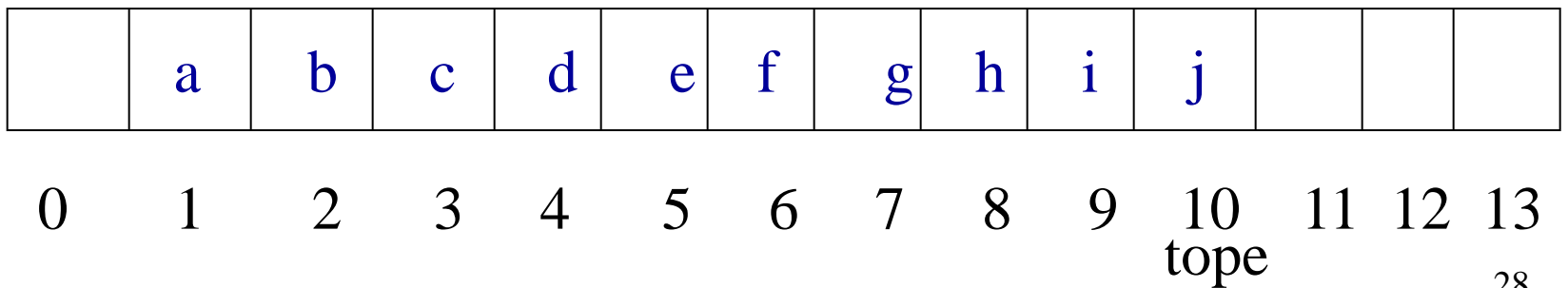
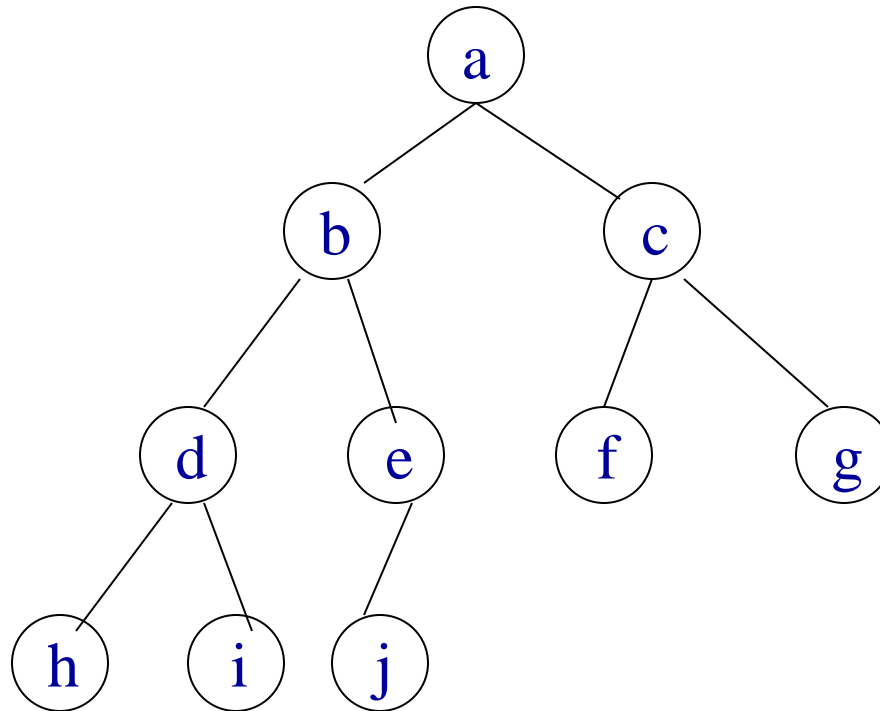
## Propiedad de la estructura:

Un heap es un árbol binario completamente lleno, con la posible excepción del nivel más bajo, el cual se llena de izquierda a derecha. Un árbol así se llama un ***árbol binario completo***.

La altura  $h$  de un *árbol binario completo* tiene entre  $2^h$  y  $2^{h+1}-1$  nodos. Esto es, la altura es  $\lfloor \log_2 n \rfloor$ , es decir  $O(\log_2 n)$ .

Debido a que un *árbol binario completo* es tan regular, se puede almacenar en un arreglo con tope, sin recurrir a apuntadores.

# Los montículos o Heaps (Binary Heaps)



# Binary Heaps

## Propiedad de la estructura (cont):

Para cualquier elemento en la posición  $i$  del arreglo, el hijo izquierdo está en la posición  $2*i$ , el hijo derecho en la posición siguiente:  $2*i+1$  y el padre está en la posición  $\lfloor i / 2 \rfloor$ .

Como vemos, no sólo no se necesitan punteros, sino que las operaciones necesarias para recorrer el árbol son muy sencillas y rápidas.

El único problema es que requerimos previamente un cálculo de tamaño máximo del heap, pero por lo general esto no es problemático. El tamaño del arreglo en el ejemplo es 13 --no 14 (el 0 es distinguido)--

# Los montículos o Heaps (Binary Heaps)

## Propiedad de orden del heap:

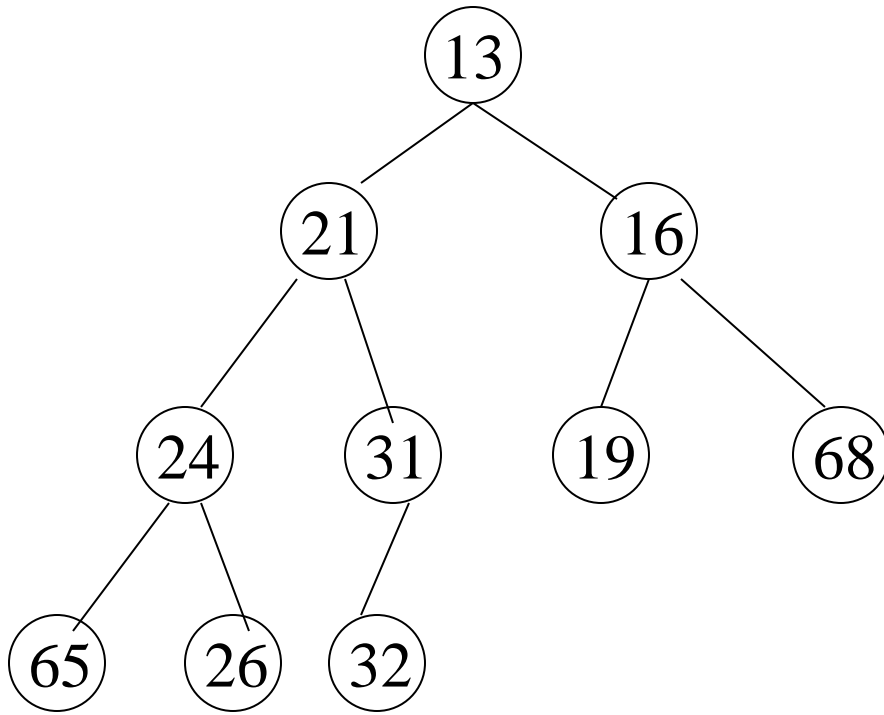
Para todo nodo  $X$ , la clave en el padre de  $X$  es:  
menor --si nos basamos en Sets para prioridades--  
(menor o igual --si nos basamos en Multisets para prioridades--)

que la clave en  $X$ , con la excepción obvia de la raíz  
(donde esta el mínimo elemento).

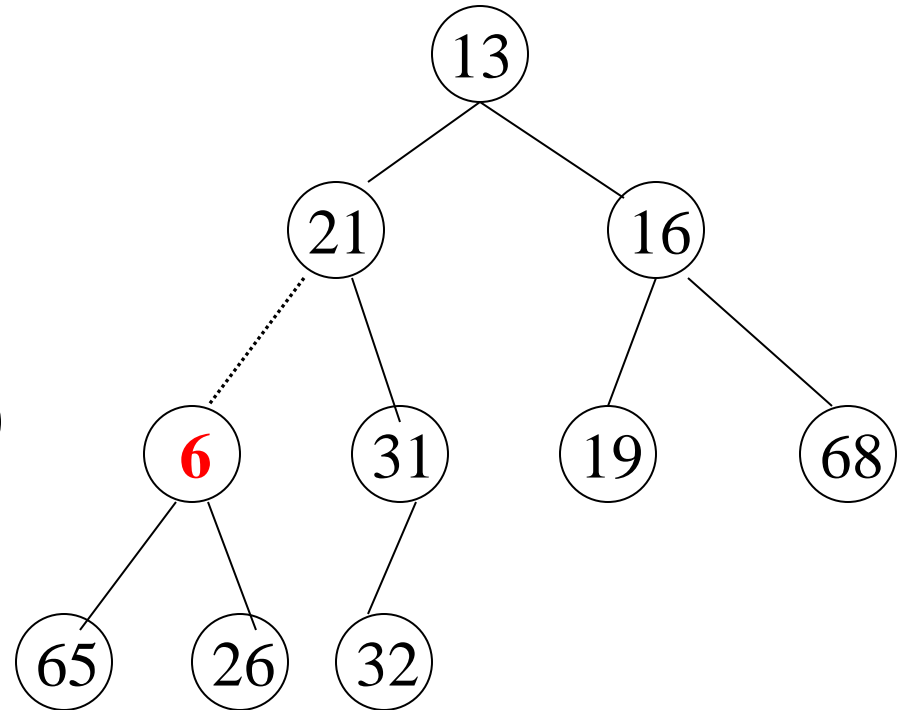
Esta propiedad permite realizar eficientemente las  
propiedades de una cola de prioridad que refieren  
al mínimo.

Ejemplos:

# Los montículos o Heaps (Binary Heaps)




**SI**



**NO**

# Operaciones básicas para Heaps

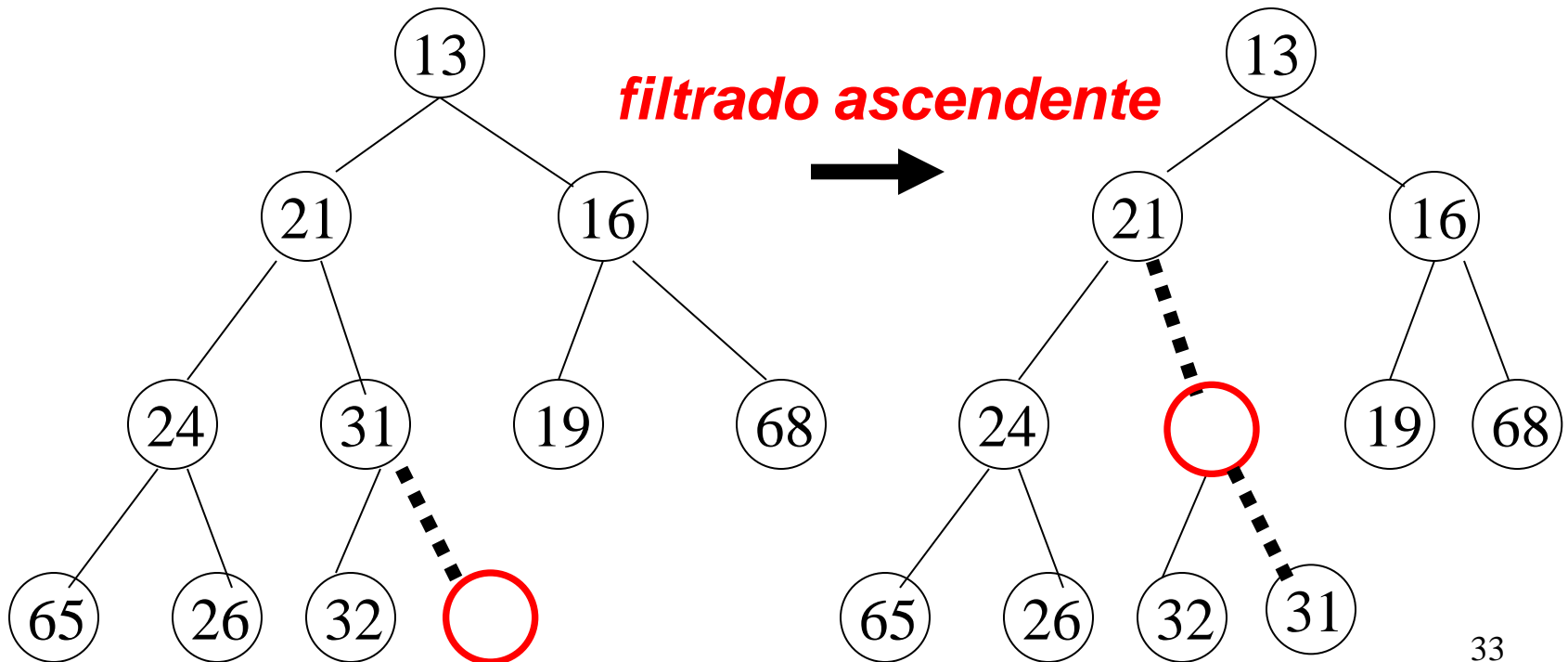
Todas las operaciones son fáciles (conceptual y prácticamente de implementar). Todo el trabajo consiste en asegurarse que se mantenga la propiedad de orden del Heap.

- **Min**
  - **Vacio**
  - **EsVacio**
  - **Insertar**
  - **Borrar\_Min**
  - **Otras operaciones sobre Heaps...(cap. 6 del Weiss)**
- 



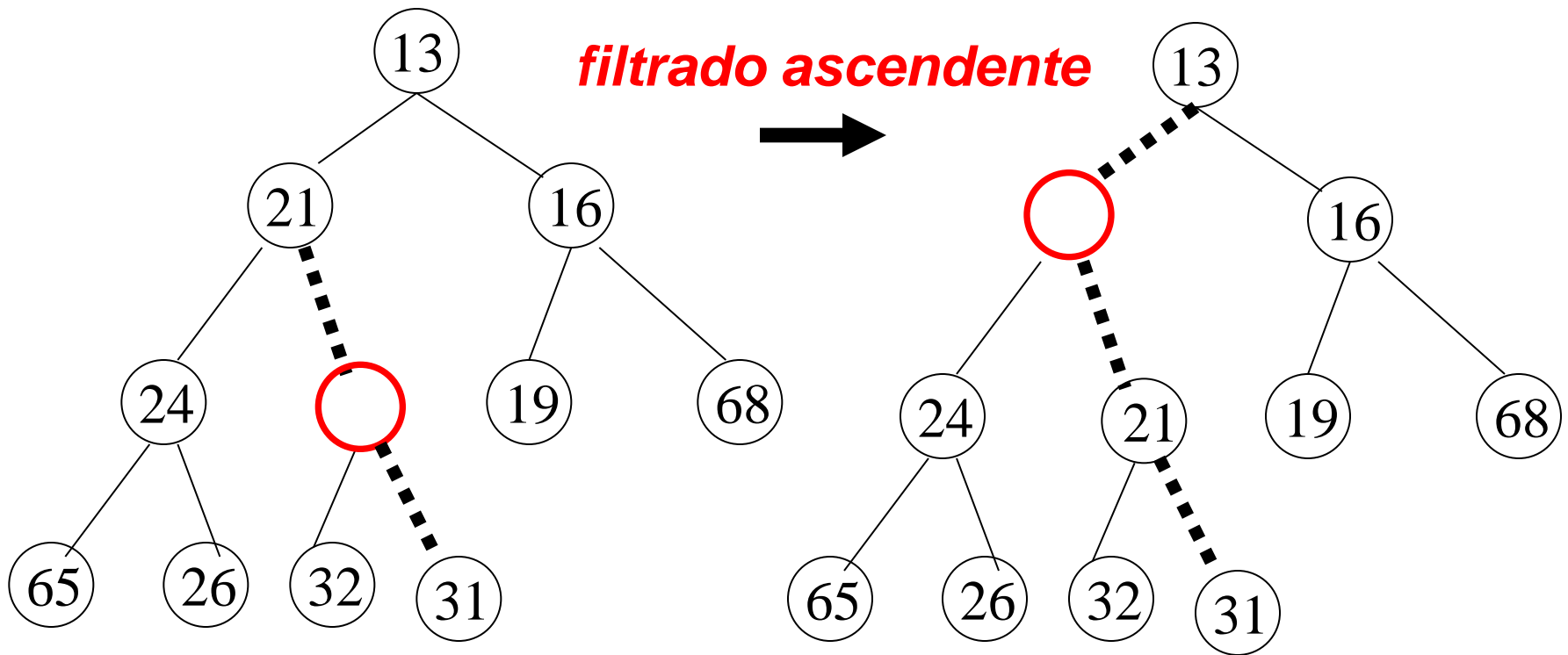
# Operaciones básicas para Heaps

- **Min**: simple,  $O(1)$ .
- **Vacio**, **EsVacio**: simples,  $O(1)$ .
- **Insertar**:  $O(\log n)$  peor caso y  $O(1)$  en el caso promedio. Insertar el 14 en (vale incluso si hay prioridades repetidas):



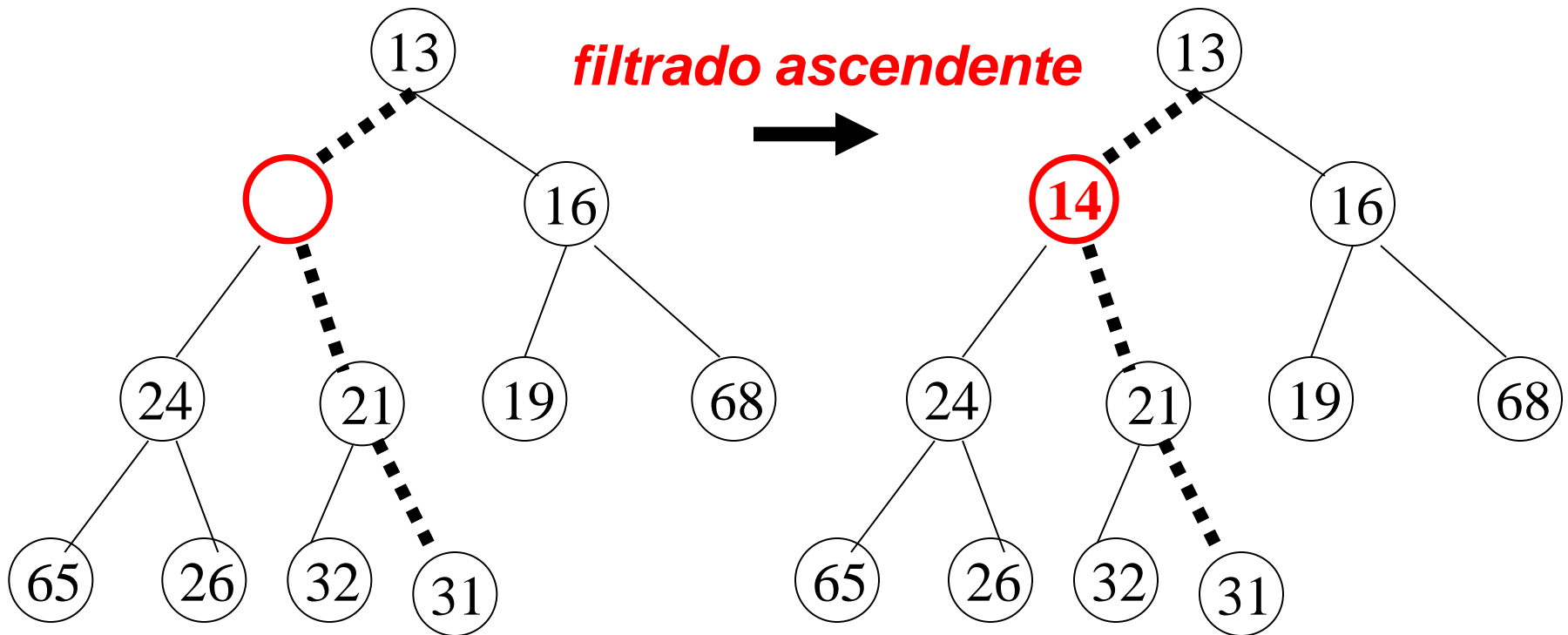
# Operaciones básicas para Heaps

## Insertión del 14



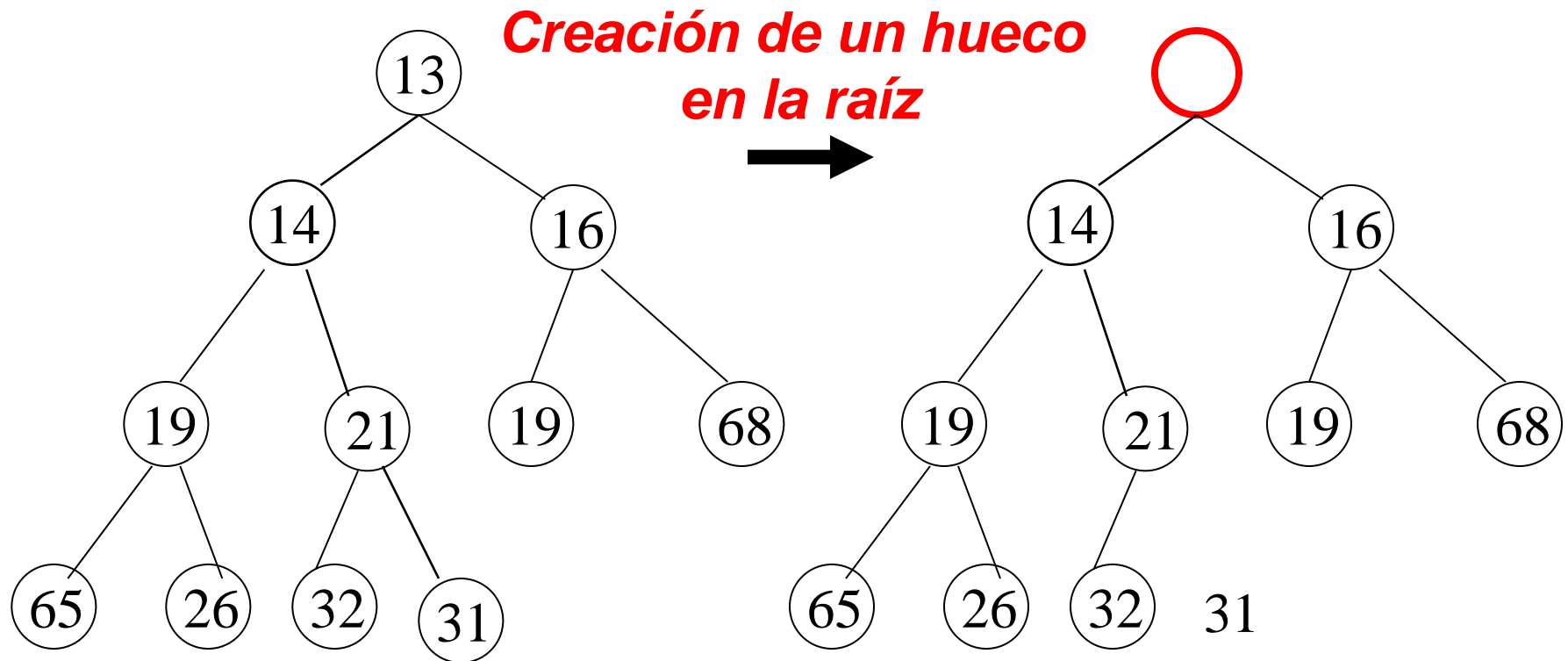
# Operaciones básicas para Heaps

## Inserción del 14



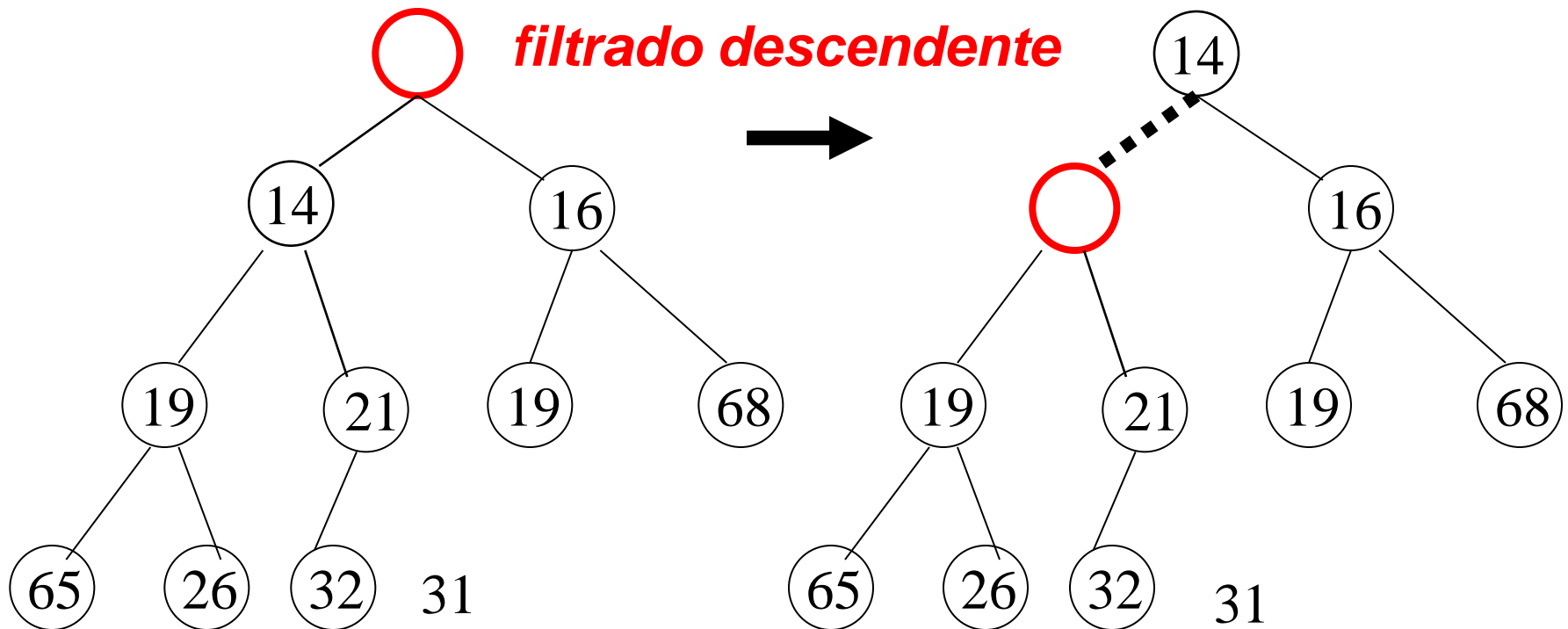
# Operaciones básicas para Heaps

Eliminar el mínimo, Borrar\_Min:  $O(\log n)$



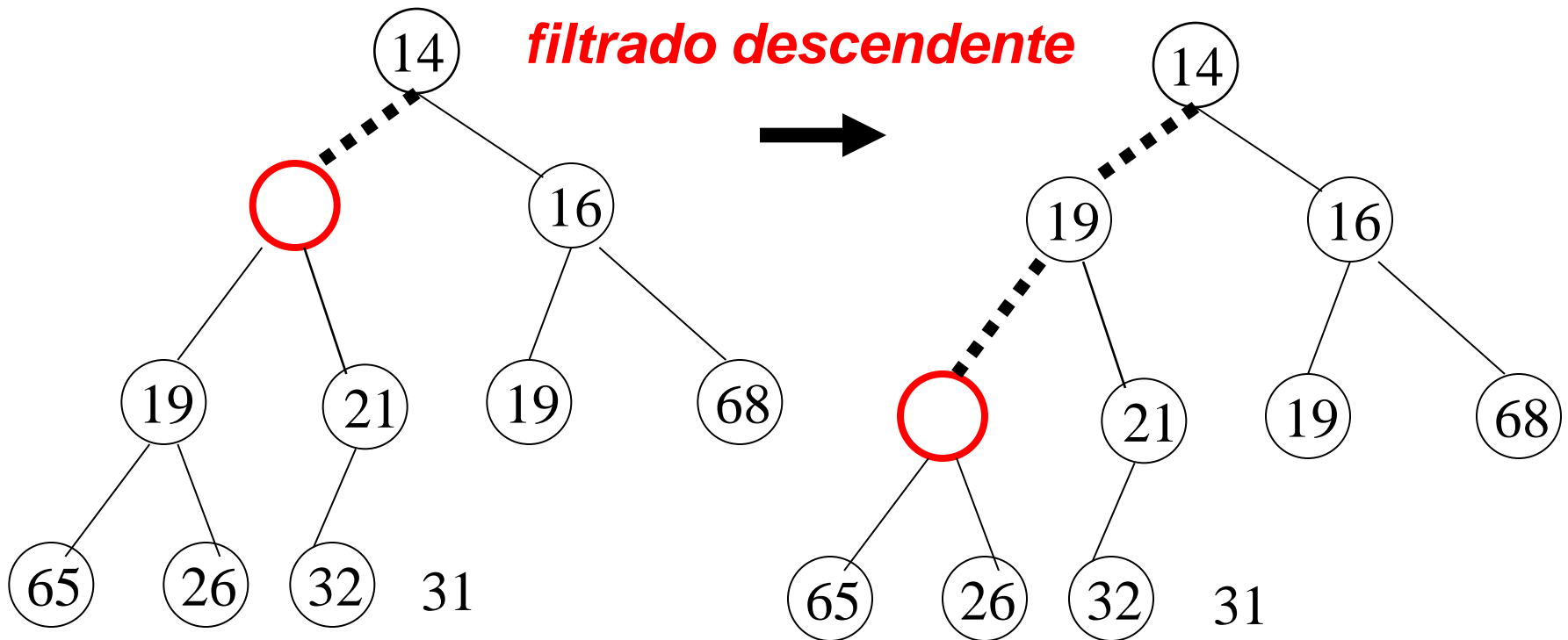
# Operaciones básicas para Heaps

## Eliminar el mínimo



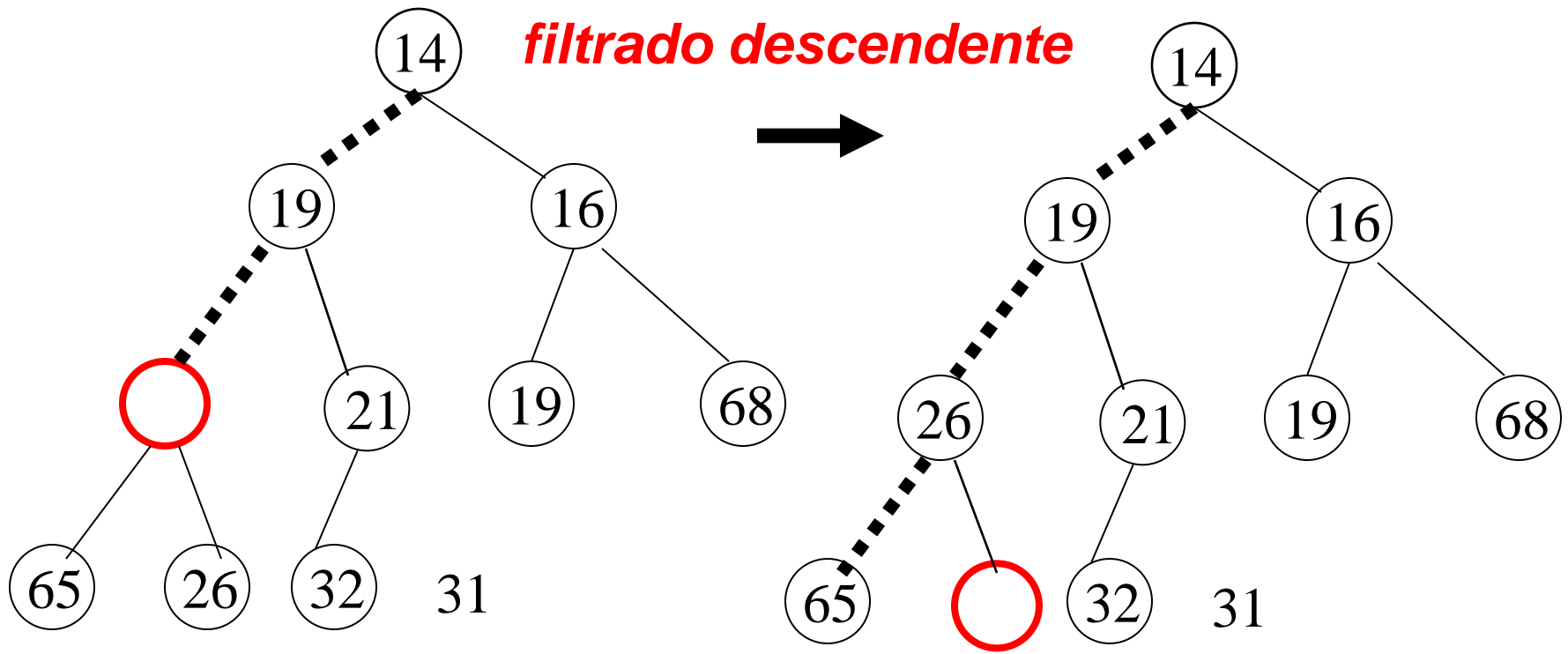
# Operaciones básicas para Heaps

## Eliminar el mínimo



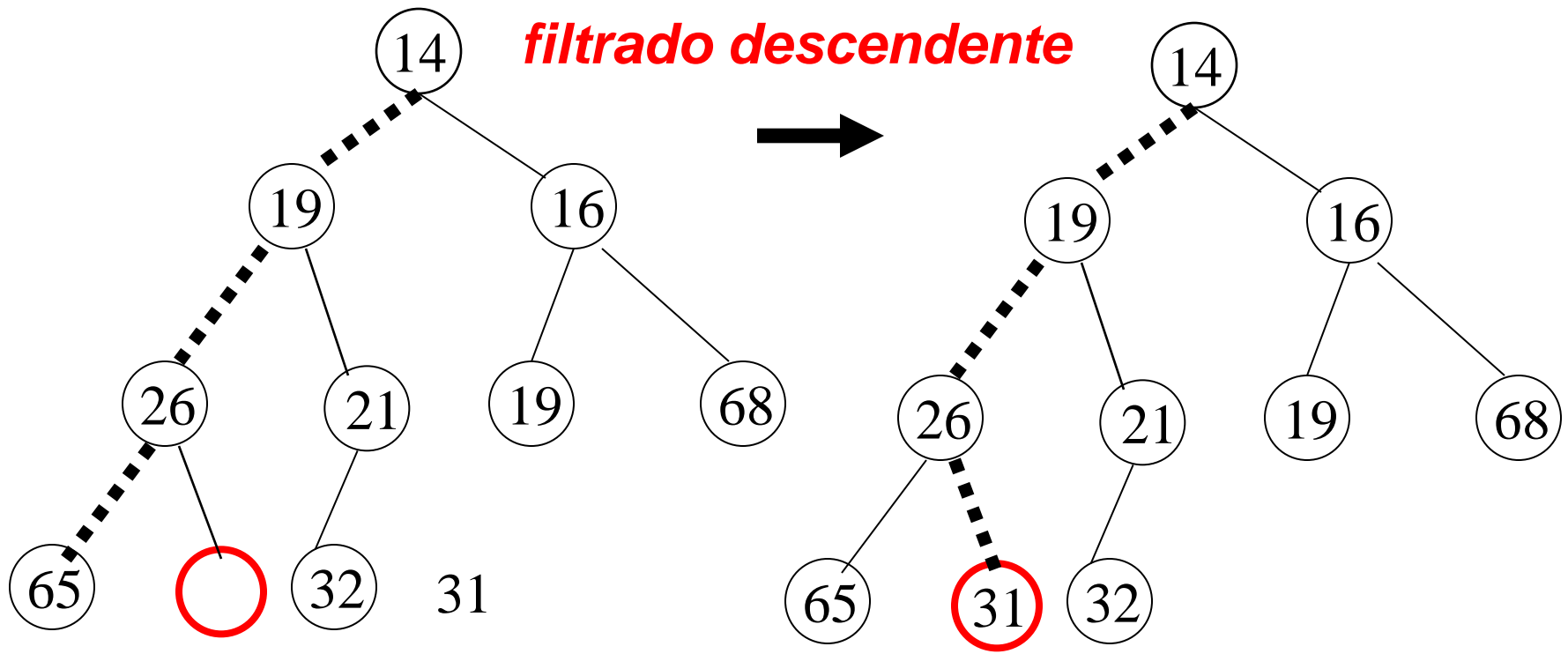
# Operaciones básicas para Heaps

## Eliminar el mínimo



# Operaciones básicas para Heaps

## Eliminar el mínimo





# Ejercicio

Considere un TAD **Cola de Prioridad (CP)** no acotada de elementos de un tipo genérico  $T$  donde las prioridades están dadas por números naturales. Se admiten prioridades y elementos repetidos.

```
struct representacionCP;  
typedef representacionCP * CP;  
  
CP crear ();  
// Devuelve la cola de prioridad vacía  
  
void agregar (T x, unsigned int p,  
CP & cp);  
// Agrega x con prioridad p a cp  
  
bool esVacia (CP cp);  
// Retorna true si y solo si cp es vacía
```

```
T prioritario (CP cp);  
/* Retorna el elemento con mayor prioridad (valor más grade) ingresado en cp. Ante igual prioridad retorna el primero en ingresar */  
// Precondición: !esVacia(cp)  
  
void eliminar (CP & cp);  
/* Remueve el elemento con mayor prioridad (valor más grade) ingresado en cp. Ante igual prioridad elimina el primero en ingresar */  
// Precondición: !esVacia(cp)
```

# Ejercicio

## Parte a)

Defina una **representación** para el TAD *CP* (*representacionCP*) de tal manera que las operaciones *crear*, *agregar*, *esVacia* y *prioritario* tengan  $O(1)$  de tiempo de ejecución en el peor caso. **Explique** para cada una de estas cuatro operaciones como se cumplen las restricciones establecidas para la representación propuesta. Escriba luego el **código de *crear* y *agregar***, y asuma implementadas las restantes operaciones (no escriba sus códigos).

## Parte b)

Sin considerar la Parte a), defina ahora una **representación** para el TAD *CP* (*representacionCP*) de tal manera que las operaciones *crear*, *esVacia*, *prioritario* y *eliminar* tengan  $O(1)$  de tiempo de ejecución en el peor caso. **Explique** para cada una de estas cuatro operaciones como se cumplen las restricciones establecidas para la representación propuesta. No escriba el código de las operaciones del TAD.

# Ejercicio – parte a

```
struct nodoLista{  
    T dato;  
    unsigned int prioridad;  
    nodoLista * sig;  
}
```

```
CP crear(){  
    CP ret = new representacionCP;  
    ret->lista = NULL;  
    ret->prioritario = NULL;  
    return ret;  
}
```

```
struct representacionCP{  
    nodoLista * lista;  
    nodoLista * prioritario;  
}
```

```
CP agregar(T x, unsigned int p, CP & cp){  
    nodoLista * nodo = new nodoLista;  
    nodo->dato = T;  
    nodo->prioridad = p;  
    nodo->sig = cp->lista;  
    cp->lista = nodo;  
    if (cp->prioritario == NULL ||  
        p > cp->prioritario->prioridad)  
        cp->prioritario = nodo;  
}
```

## Ejercicio – parte b

Una lista simplemente encadenada (de nodos de tipo *nodoLista*, de la Parte a) ordenada por prioridad (de mayor a menor) permitiría tener al elemento prioritario al inicio.

Las inserciones serían ordenadas y en caso de prioridades repetidas, al agregar un elemento debería ir al final de los que tienen la misma prioridad.

De esta manera, obtener el elemento prioritario y eliminarlo es  $O(1)$  peor caso, al igual que crear la lista vacía y chequear si está vacía.

Esto es *crear*, *esVacia*, *prioritario* y *eliminar* tendrían  $O(1)$  de tiempo de ejecución en el peor caso.

# Ejercicio 2

Especifique, con pre y postcondiciones, un TAD *cola de prioridad* no acotado de elementos de tipo *char \** con prioridades que toman valores enteros y con las siguientes operaciones:

**1.crear**, que genera una cola de prioridad vacía.

**2.encolar**, que inserte un string en una cola de prioridad asociándole cierta prioridad dada.

**3.obtener**, que retorna el elemento de tipo *char \** con mayor prioridad (mayor valor entero) de una cola de prioridad no vacía. En el caso en que exista más de un elemento en la cola con la misma prioridad máxima, deberá retornarse el primero de ellos en ingresar a la cola.

**4.desencolar**, que elimina el elemento con mayor prioridad de una cola de prioridad no vacía. En el caso en que exista más de un elemento en la cola con la misma prioridad máxima, deberá eliminarse el primero de ellos en ingresar a la cola.

**5.cantidad**, que retorna la cantidad de elementos de una cola de prioridad.

**6.destruir**, que destruye la cola de prioridad, liberando toda su memoria

## Ejercicio 2

Implementar la operación *imprimir* que, dada una lista de elementos de tipo `char *` (cadenas), imprima las cadenas ordenadas por su largo, de mayor a menor. El orden entre cadenas de igual largo no es relevante. Implemente *imprimir* usando una *cola de prioridad* según la especificación de la parte a). Asuma que elementos de tipo `char *` (cadenas) se pueden imprimir directamente con *printf* o *cout*. Pueden utilizarse otras funciones de cadenas de caracteres como *strlen*, *strcpy* u otras.

*void imprimir (LCadenas l)* donde:

```
typedef nodoLCadenas * Lcadenas;
```

```
struct nodoLCadenas { char * cadena; LCadenas sig; };
```

# Ejercicio 2

```
void imprimir (LCadenas l){
    char * cad;
    ColaPrioridad cp = crearColaPrioridad();
    while (l!=NULL){
        encolar(cp, l->cadena, strlen(l->cadena));
        l = l->sig;
    }
    // si l fuera un TAD ?
    while (cantidad(cp)!=0){
        cout << obtener(cp);
        desencolar(cp);
    }
    destruir(cp);
}
```