

# **Programación 2**

## **8 - TADs Colecciones Colas de Prioridad**

# Colas de Prioridad

El término “**cola de prioridad**” se relaciona con:

- La palabra “**cola**” sugiere la espera de cierto servicio por ciertas personas u objetos.
- La palabra “**prioridad**” sugiere que este servicio no se proporciona por medio de una disciplina “primero en llegar, primero en ser atendido” que es la base del TAD Cola, sino que cada elemento tiene una prioridad basada en “la urgencia de su necesidad” (de ser atendido)

# Colas de Prioridad

## Ejemplos:

- la sala de espera de un hospital, donde los pacientes con problemas potencialmente mortales serán atendidos primero.
- Aún cuando los trabajos enviados a una impresora se suelen colocar en una cola, esto puede no ser siempre lo mejor. Por ejemplo, si los trabajos tienen prioridades en ser procesados.
- En el área de sistemas operativos y políticas de *scheduling* de procesos en sistemas de tiempo compartido.

# Colas de Prioridad

Hay dos TADs especialmente utilizados, basados en el modelo de conjuntos: Los **diccionarios** y las **colas de prioridad**. El primero fue estudiado en el curso previo. Ahora veremos el segundo.

Una **cola de prioridad** es un Tad Set (o alternativamente un MultiSet) con las operaciones:

(constructores) Vacio, Insertar,

(predicado) EsVacio,

(selectores) Borrar\_Min y Min.

# Colas de Prioridad

NOTA: el modelo mínimo de colas de prioridad en el Weiss abarca dos operaciones: Insertar y Borrar\_Min. Esta última operación devuelve y elimina el mínimo elemento.

NOTA: alternatively podríamos pensar que en vez de Min y Borrar\_Min tenemos Max y Borrar\_Max, o todas estas operaciones (cola de prioridad min-max).

# Colas de Prioridad: Implementaciones

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

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

Y ¿si las lista se mantuviese ordenada?

¿Cuál de las dos opciones le parece más adecuada y por qué?

# 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)**.

En realidad *la inserción tardará un tiempo medio constante*, y nuestra implementación permitirá construir un heap de  $n$  elementos en un tiempo lineal, si no intervienen eliminaciones.

# TAD's acotados y no acotados

Como vimos anteriormente, los TAD's pueden ser acotados o no en la cantidad de elementos (por ejemplo las listas, pilas, colas, los diccionarios).

En general, las implementaciones estáticas refieren a una versión del TAD que es acotada (incluso en la especificación del TAD), mientras que las implementaciones dinámicas admiten una versión del TAD no acotada (que permite expresar la noción de estructuras potencialmente infinitas).

Lo cierto es que TAD's acotados y no acotados son conceptualmente diferentes, pero pueden unificarse en una misma especificación.



# TAD's acotados y no acotados

Los TAD's acotados incorporan una operación que permite testar si la estructura está llena. Asimismo, se agrega una precondition a la operación de inserción del TAD (“que la estructura no esté llena”).

A los fines prácticos podemos considerar que no sólo la versión acotada posee una operación `IsFull` `--Esta_llena--` (en el caso de la no acotada retorna siempre *false*).

La precondition de las operaciones de inserción se asumen sólo si la implementación que se usa es “acotada”.

# Colas de Prioridad Acotadas

La implementación de colas de prioridad con binary heaps hace entonces referencia a una especificación de colas de prioridad acotadas (con una operación adicional **Esta\_llena()** y con una precondition sobre la operación de inserción)

## Bibliografía de Colas de Prioridad:

- \* **Capítulo 6 del libro de Weiss.**
- \* Lectura adicional: secciones 4.10 y 4.11 del libro de Aho, Hopcroft and Ullman.

# 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)

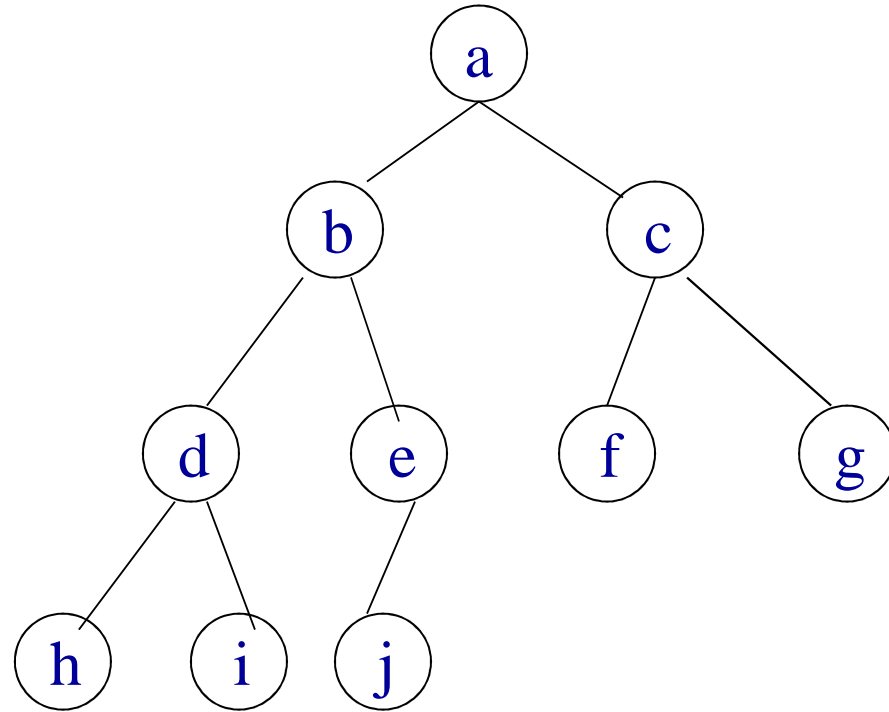
## Propiedad de la estructura:

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, sin recurrir a apuntadores.

# Los montículos o Heaps (Binary Heaps)



	a	b	c	d	e	f	g	h	i	j			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

# Los montículos o Heaps (Binary Heaps)

¿ Cómo es la definición de la estructura en la parte de valores para la implementación de colas de prioridad con la representación anterior ?

# 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--)

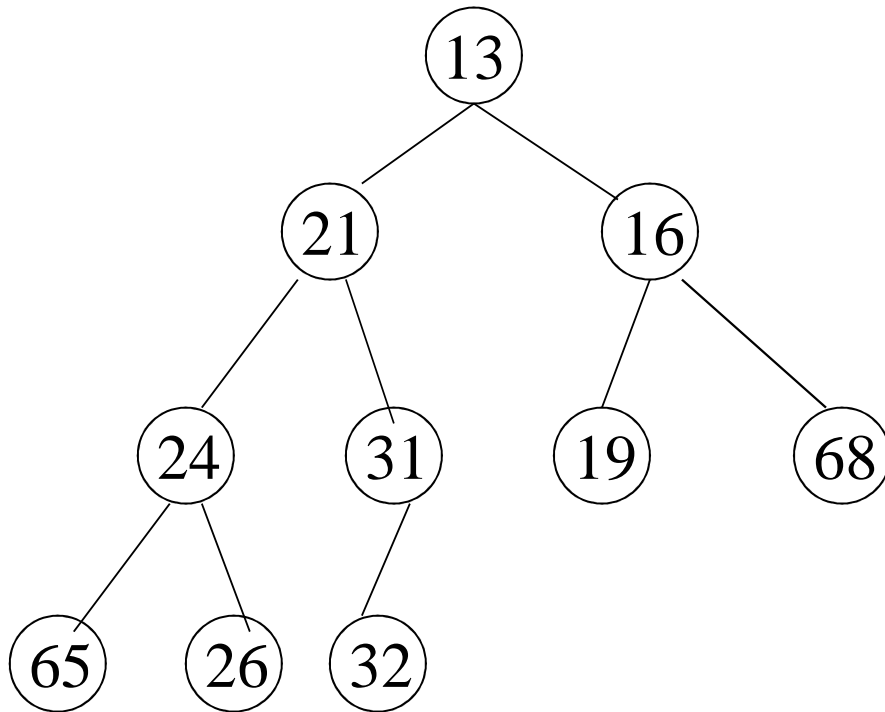
que la clave en  $X$ , con la excepción obvia de la raíz  
(donde esta el mínimo elemento).

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

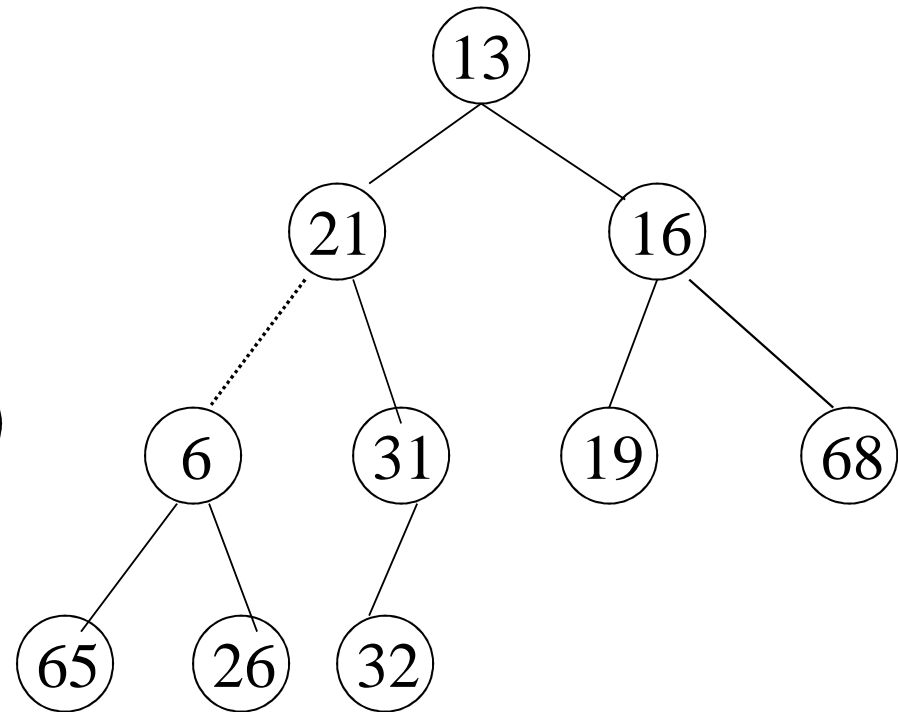
Ejemplos:



# Los montículos o Heaps (Binary Heaps)




**SI**



**NO**

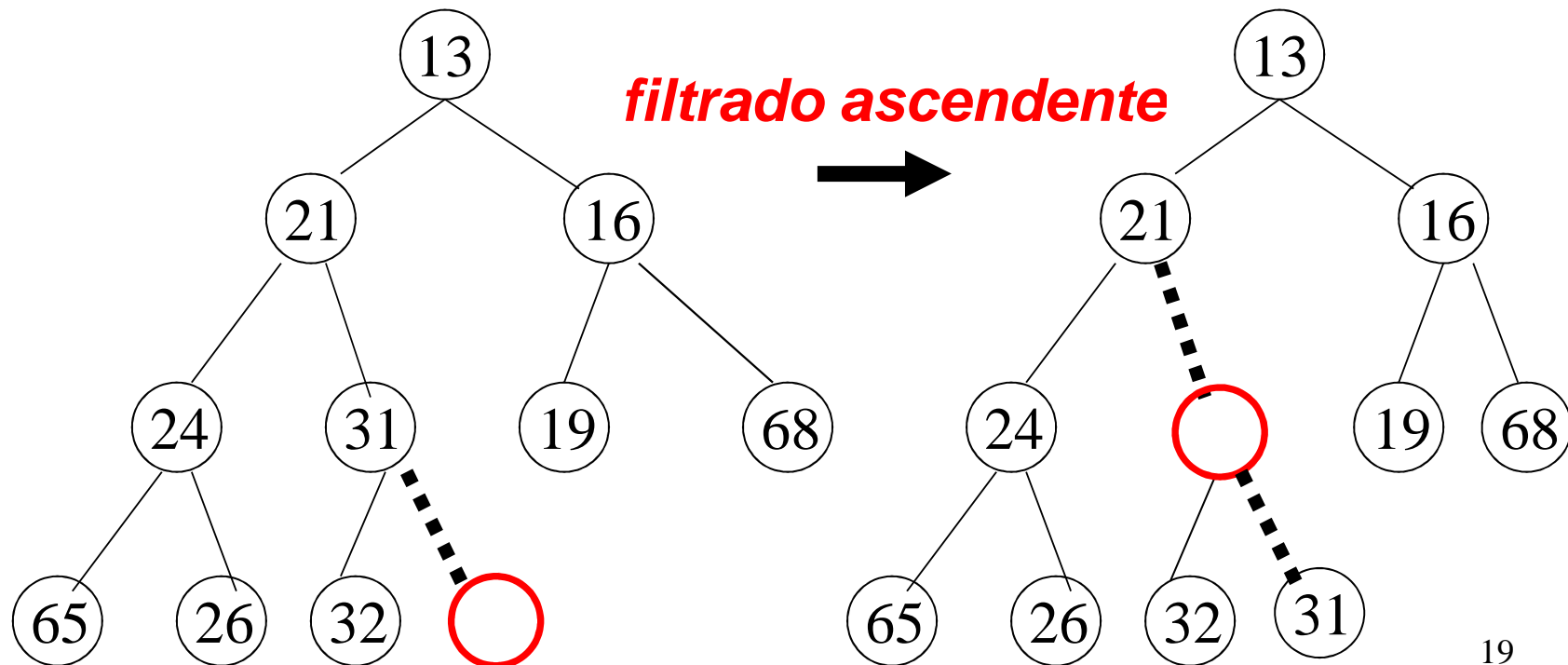
# Operaciones básicas para Heaps

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)**
- 
- Desarrollarlas....**

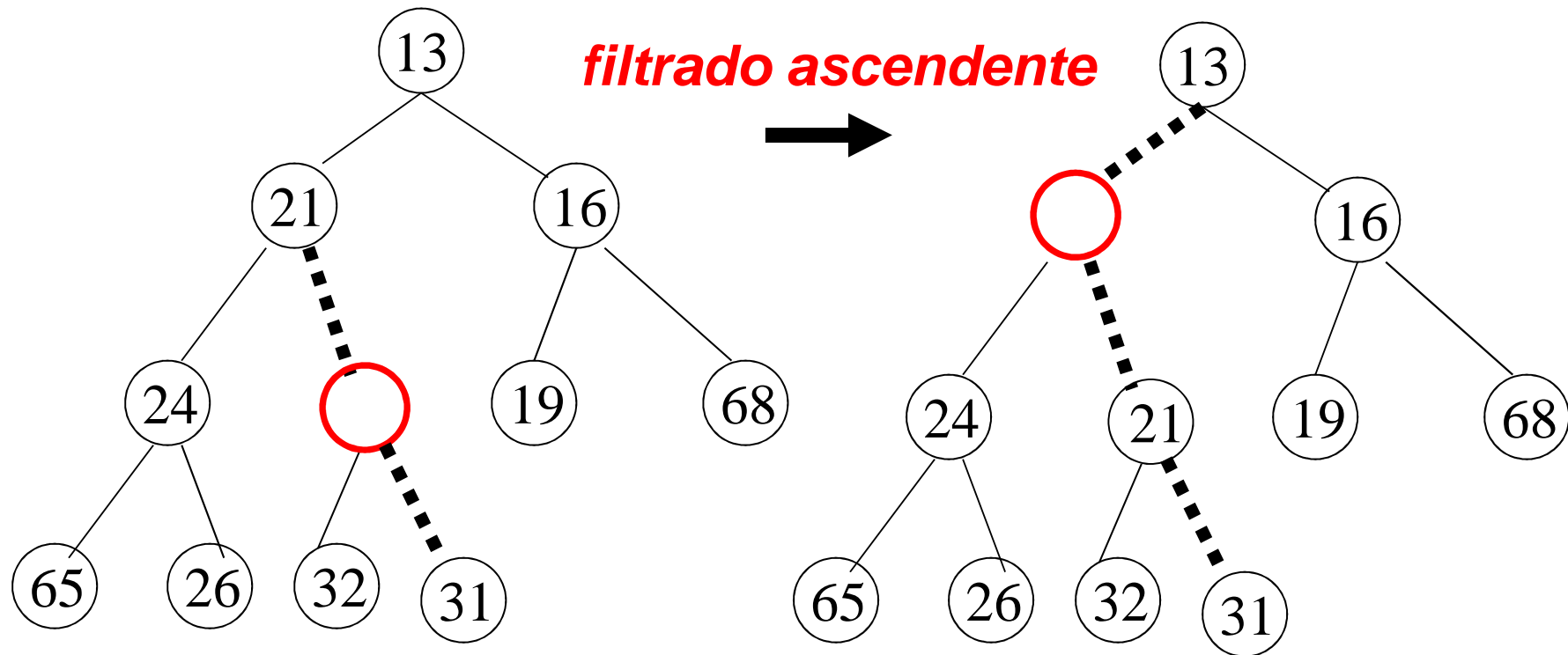
# Operaciones básicas para Heaps

- **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):



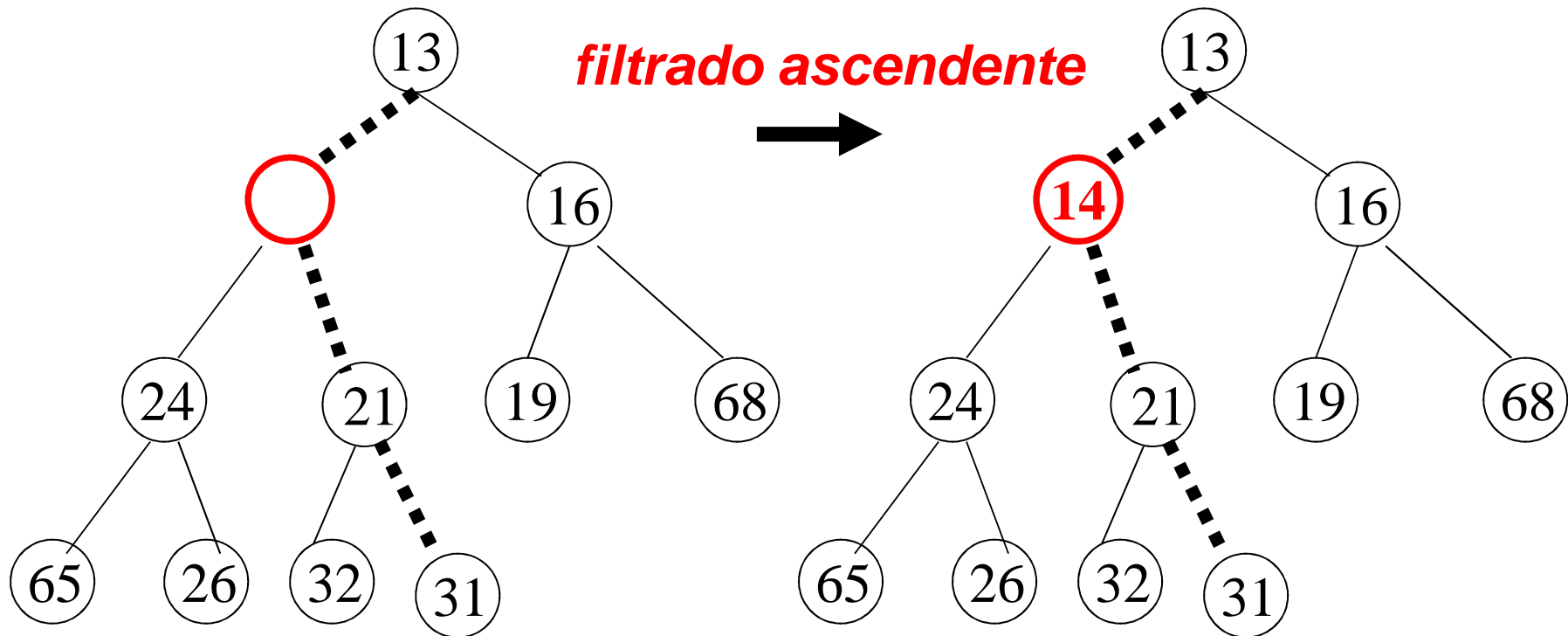
# Operaciones básicas para Heaps

## Inserción del 14



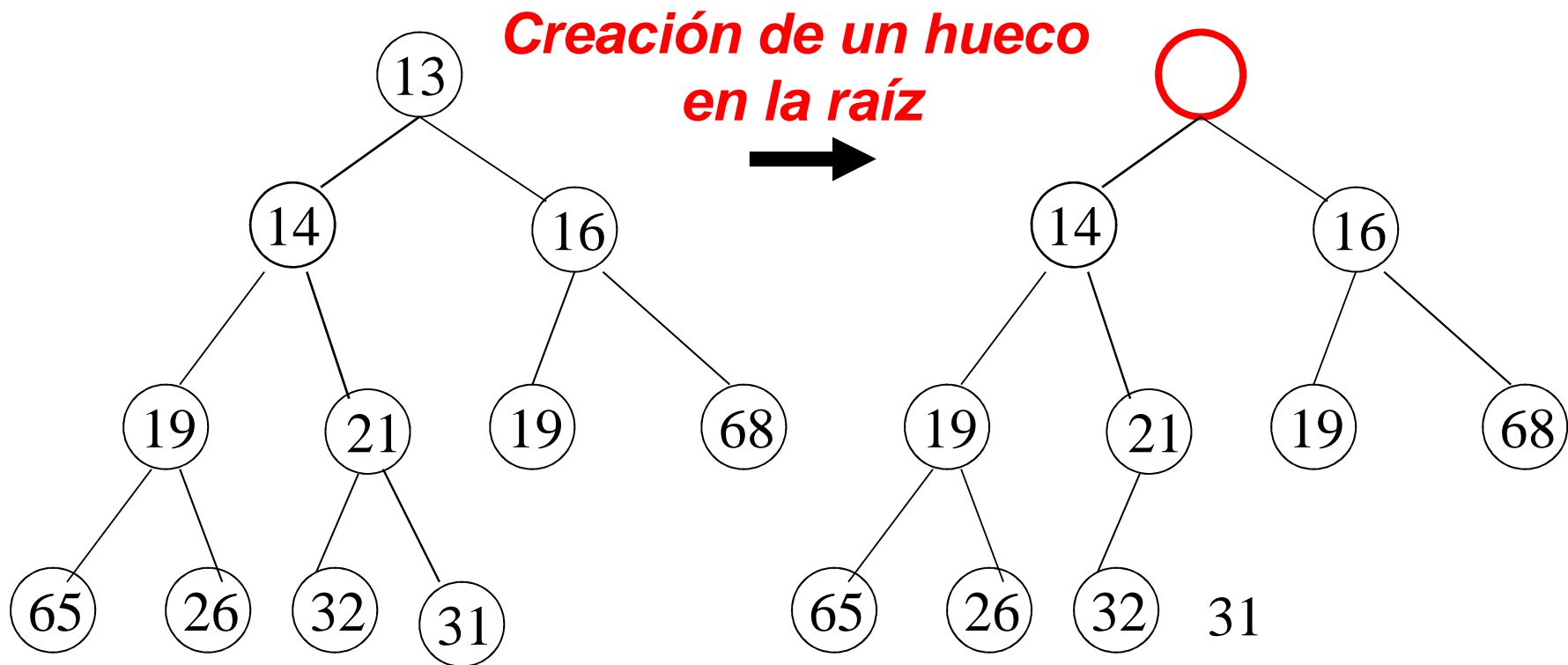
# Operaciones básicas para Heaps

## Inserción del 14



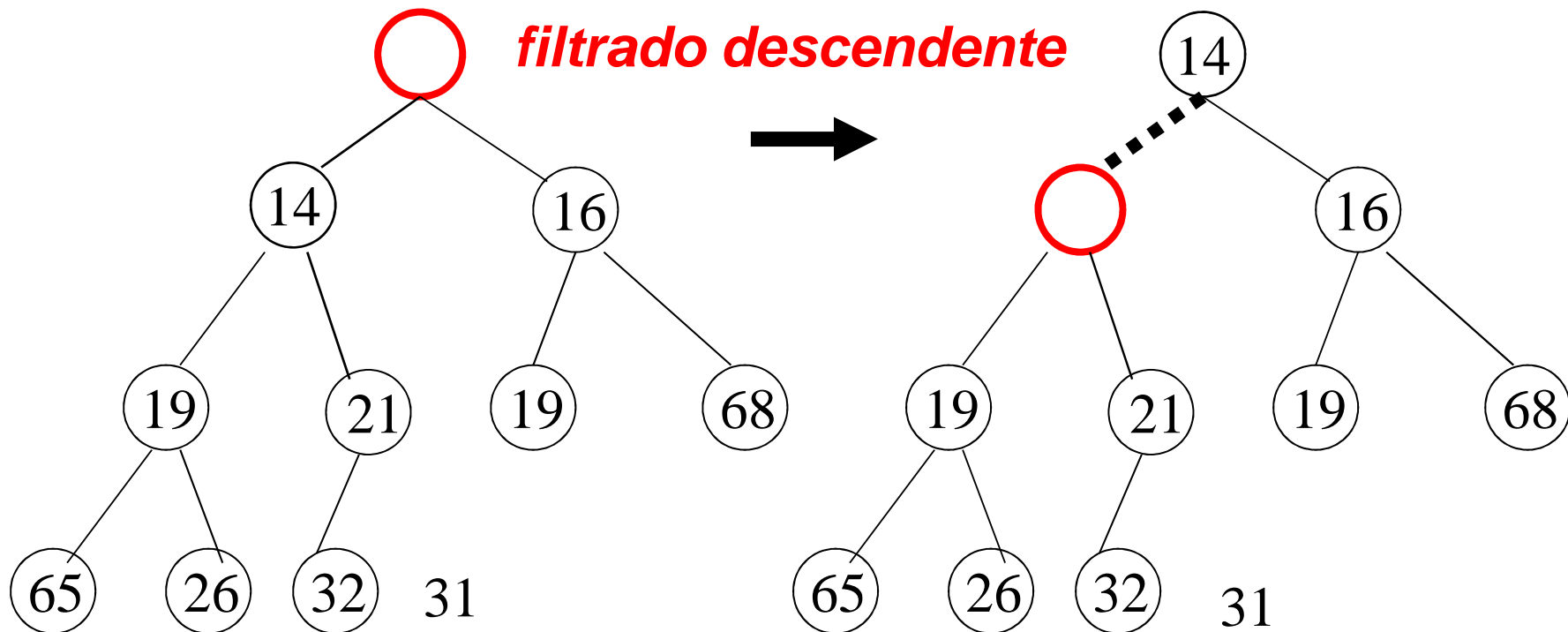
# Operaciones básicas para Heaps

Eliminar el mínimo, `Borrar_Min`:  $O(\log n)$



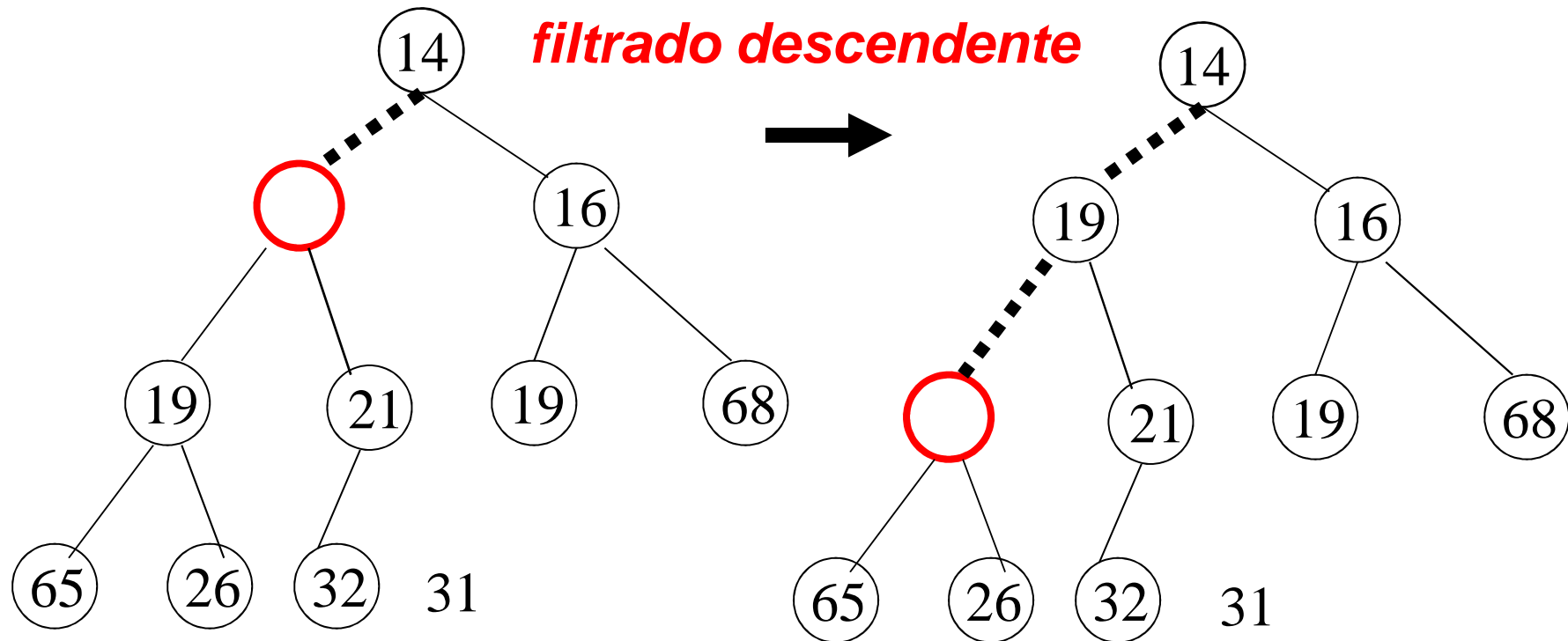
# Operaciones básicas para Heaps

## Eliminar el mínimo



# Operaciones básicas para Heaps

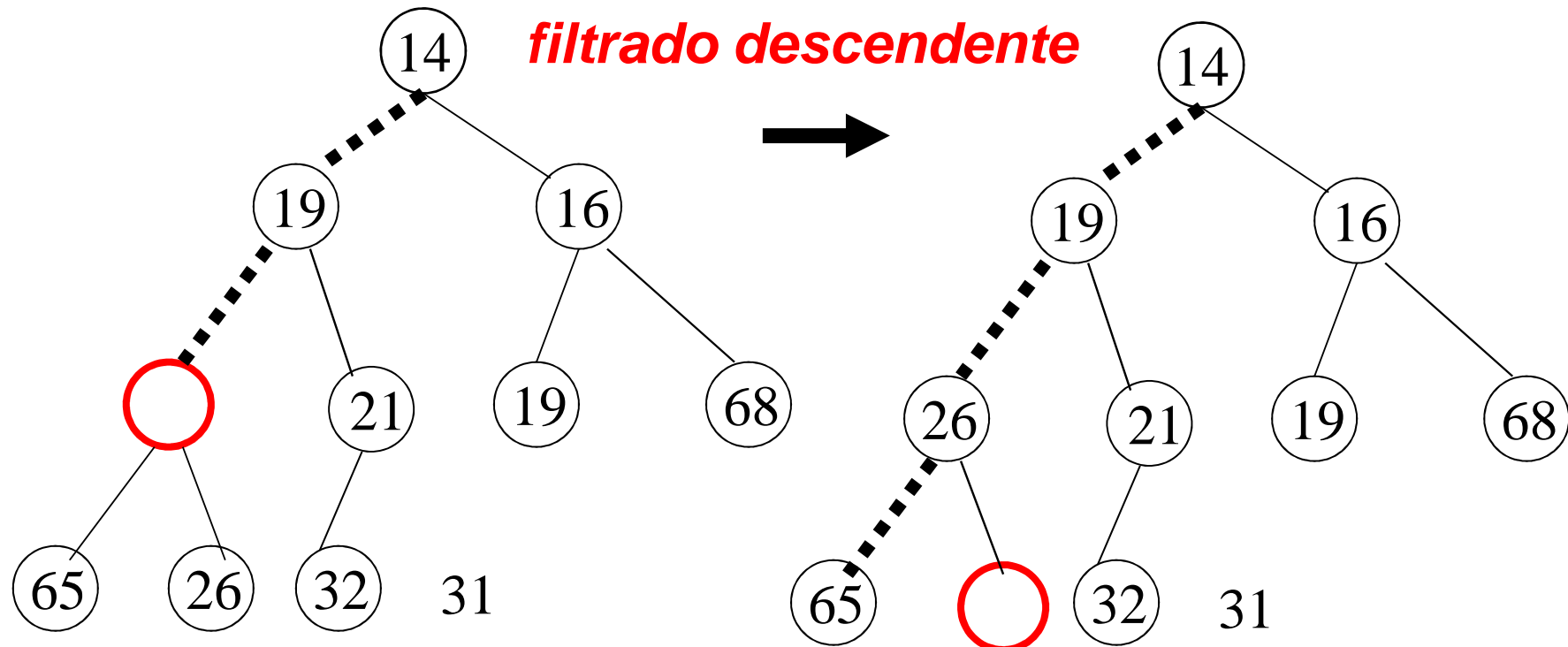
## Eliminar el mínimo





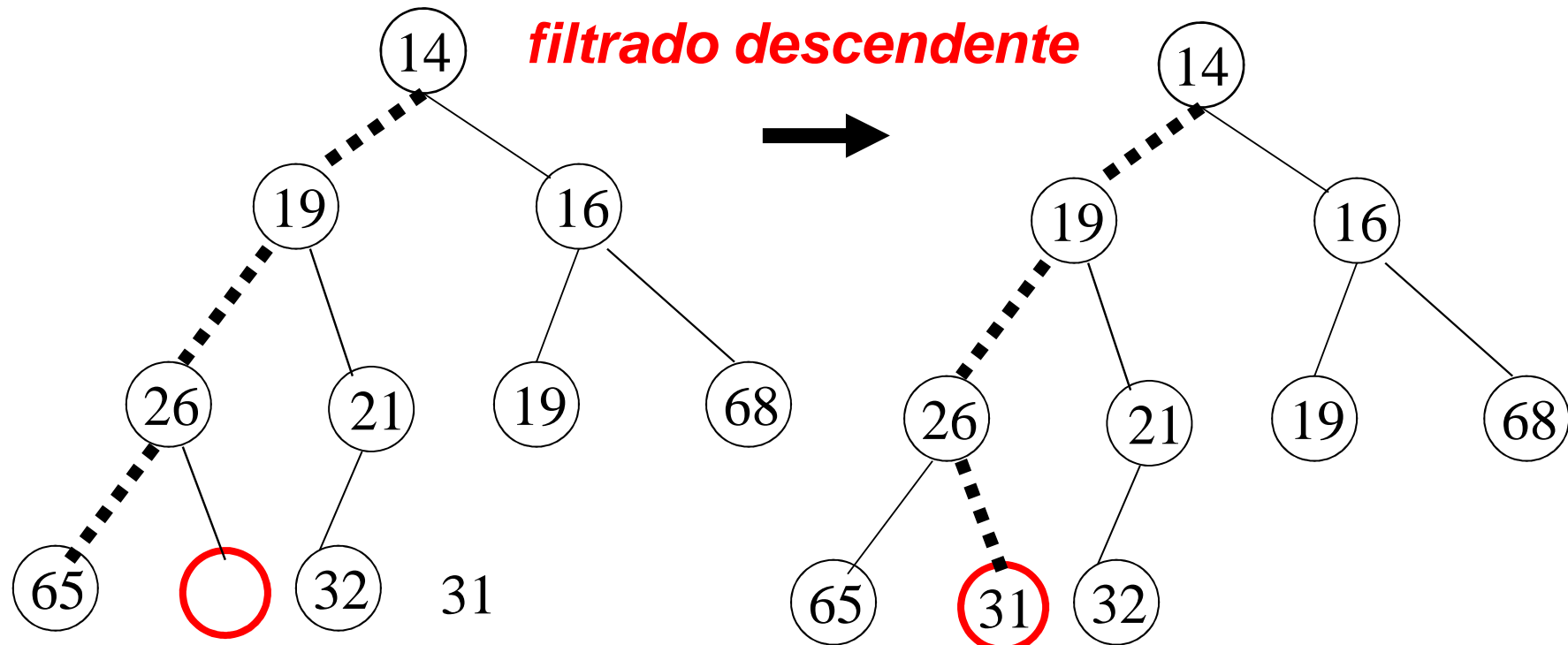
# Operaciones básicas para Heaps

## Eliminar el mínimo



# Operaciones básicas para Heaps

## Eliminar el mínimo



# Colas de Prioridad: Aplicaciones

- **Aplicaciones en el área de los sistemas operativos**
- **Ordenación externa**
- **Algoritmos *greedy* en los cuales interesa encontrar el mínimo repetidamente.**
- **Simulación de eventos discretos**
- **...**

# Colas de Prioridad Extendidas

- **Ejemplos**
- **Definición**
- **Operaciones**
- **Aplicaciones**

Lectura a cargo de los estudiantes...

# Colas de Prioridad Extendidas (Heap)

Algunas operaciones adicionales sobre heaps que tienen  $O(\log n)$  --si se admiten Multisets para las prioridades-- y presuponen conocidas las posiciones de los elementos :

- **decrementar\_clave (x, d, Heap)**

reduce el valor de la clave (clave) en la posición x por una cantidad positiva d. Como esto puede violar el orden del heap, debe arreglarse con un *filtrado ascendente*.

Esta operaciones podría ser útil para administradores de sistemas: pueden hacer que sus programas se ejecuten con la mayor prioridad.

# Colas de Prioridad Extendidas (Heap)

- **incrementar\_clave (x, d, Heap)**

aumenta el valor de la clave en la posición x en una cantidad positiva d. Esto se obtiene con un *filtrado descendente*.

Muchos planificadores bajan automáticamente la prioridad de un proceso que consume un tiempo excesivo de CPU.

# Colas de Prioridad Extendidas (Heap)

- **eliminar (x, Heap)**

retira el nodo en la posición x del Heap. Esto puede hacerse ejecutando primero `decrementar_clave(x,  $\infty$ , Heap)` y después `Borrar_Min(Heap)`.

Cuando un proceso termina por orden del usuario (en vez de terminar normalmente) se debe eliminar de la cola de prioridad.

- **construir\_heap**: leerla del libro (y en el práctico). Tiene  $O(n)$  para crear un heap con n claves.
- .....

# Colas de Prioridad Extendidas (Heap)

¿Cómo encontrar el máximo elemento de un heap?

¿Cuánto cuesta en tiempo de ejecución?

¿Qué se conoce acerca del orden de los elementos y de la posición del máximo?

Las colas de prioridad permiten encontrar el elemento mínimo en tiempo constante y borrar el mínimo e insertar un elemento en  $O(\log(n))$ .

Explicar como modificar la estructura de cola de prioridad y las operaciones para proveer la implementación de una cola de prioridad con dos extremos que tenga las siguientes características:



# Colas de Prioridad Extendidas (Heap)

- la estructura se puede construir en tiempo  $O(n)$ .
- un elemento se puede insertar en tiempo  $O(\log(n))$ . El máximo y el mínimo se pueden borrar en tiempo  $O(\log(n))$ .
- el máximo o el mínimo se pueden encontrar en tiempo constante.

# ANEXO

## Especificación e implementación de una Cola de Prioridad

# Especificación

```
#ifndef _COLAPRIORIDAD_H
#define _COLAPRIORIDAD_H

#include "tipoT.h"

struct RepresentacionColaPrioridad;
typedef RepresentacionColaPrioridad * ColaPrioridad;

void crearColaPrioridad (ColaPrioridad &cp, int tamano);
/* Devuelve en 'cp' una cola de prioridad vacía,
   que podrá contener hasta 'tamano' elementos.
   Precondición: tamano > 0. */

void encolarColaPrioridad (ColaPrioridad &cp, T t, int prio);
/* Agrega a 'cp' el elemento 't' con prioridad 'prio'.
   Precondición: ! esLlenaColaPrioridad (cp). */
```

# Especificación

```
bool esVacíaColaPrioridad (ColaPrioridad cp);  
/* Devuelve 'true' si 'cp' es vacía, 'false' en otro caso. */  
  
bool esLlenaColaPrioridad (ColaPrioridad cp);  
/* Devuelve 'true' si 'cp' está llena, 'false' en otro caso. */  
  
T minimoColaPrioridad (ColaPrioridad cp);  
/* Devuelve el elemento de 'cp' que tiene asignada menor  
   prioridad (si más de uno cumple esa condición devuelve  
   cualquiera de ellos).  
   Precondición: ! esVacíaColaPrioridad (cp). */  
  
void removerMinimoColaPrioridad (ColaPrioridad &cp);  
/* Remueve de 'cp' el elemento de menor prioridad  
   (si más de uno cumple esa condición remueve cualquiera de  
   ellos).  
   Precondición: ! esVacíaColaPrioridad (cp). */
```

# Especificación

```
void destruirColaPrioridad (ColaPrioridad &cp);  
/* Libera toda la memoria ocupada por 'cp'. */
```

```
#endif /* _COLAPRIORIDAD_H */
```

Donde, tipoT.h:

```
#ifndef _TIPO_T  
#define _TIPO_T
```

```
typedef unsigned int T;  
bool esMayor (T t1, T t2);  
bool esMenor (T t1, T t2);  
bool esIgual (T t1, T t2);
```

```
#endif /* _TIPO_T */
```

# Implementación

```
#include "ColaPrioridad.h"
#include <assert.h>
#include <stdio.h>

struct Elem
{
    int prio;
    T info;
};

struct RepresentacionColaPrioridad
{
    Elem * array;
    int tope;
    int capacidad;
};
```

# Implementación

```
// auxiliar
int posicionMinimo (ColaPrioridad cp)
{
    int i_min = 0;
    int min = cp->array [0].prio;
    for (int i=1; i<cp->tope; i++)
    {
        if (cp->array [i].prio < min)
        {
            i_min = i;
            min = cp->array [i].prio;
        }
    }
    return i_min;
}
```

# Implementación

```
// auxiliar
int posicionMaximo (ColaPrioridad cp)
{
    int i_max = 0;
    int max = cp->array [0].prio;
    for (int i=1; i<cp->tope; i++)
    {
        if (cp->array [i].prio > max)
        {
            i_max = i;
            max = cp->array [i].prio;
        }
    }
    return i_max;
} // posicionMaximo
```



# Implementación

```
// auxiliar
void intercambiar (Elem &a, Elem &b)
{
    Elem aux = a;
    a = b;
    b = aux;
} // intercambiar

void crearColaPrioridad (ColaPrioridad &cp, int tamanio)
{
    assert (tamanio > 0);
    cp = new RepresentacionColaPrioridad;
    cp->array = new Elem [tamanio];
    cp->tope = 0;
    cp->capacidad = tamanio;
} // crearColaPrioridad

bool esVaciaColaPrioridad (ColaPrioridad cp)
{
    return cp->tope == 0;
} // esVaciaColaPrioridad
```

# Implementación

```
bool esLlenaColaPrioridad (ColaPrioridad cp)
{
    return cp->tope == cp->capacidad;
} // esLlenaColaPrioridad

void encolarColaPrioridad (ColaPrioridad &cp, T t, int prio)
{
    cp->array[cp->tope].prio = prio;
    cp->array[cp->tope].info = t;
    cp->tope ++;
} // encolarColaPrioridad

T minimoColaPrioridad (ColaPrioridad cp)
{
    return cp->array [posicionMinimo (cp)].info;
} // minimoColaPrioridad

void removerMinimoColaPrioridad (ColaPrioridad &cp)
{
    int i_min = posicionMinimo (cp);
    cp->tope--;
    intercambiar (cp->array [i_min], cp->array [cp->tope]);
} // removerMinimoColaPrioridad
```

# Implementación

```
void destruirColaPrioridad (ColaPrioridad &cp)
{
    delete [] cp->array;
    delete cp;
} // destruirColaPrioridad
```

Donde, tipoT.cpp:

```
#include "tipoT.h"
#include <stddef.h>
#include <assert.h>
```

```
bool esMayor (T t1, T t2) {
    return (t1 > t2);
}
```

```
bool esMenor (T t1, T t2) {
    return (t1 < t2);
}
```

```
bool esIgual (T t1, T t2) {
    return (t1 == t2);
}
```