Programación 2

La Previa: 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)

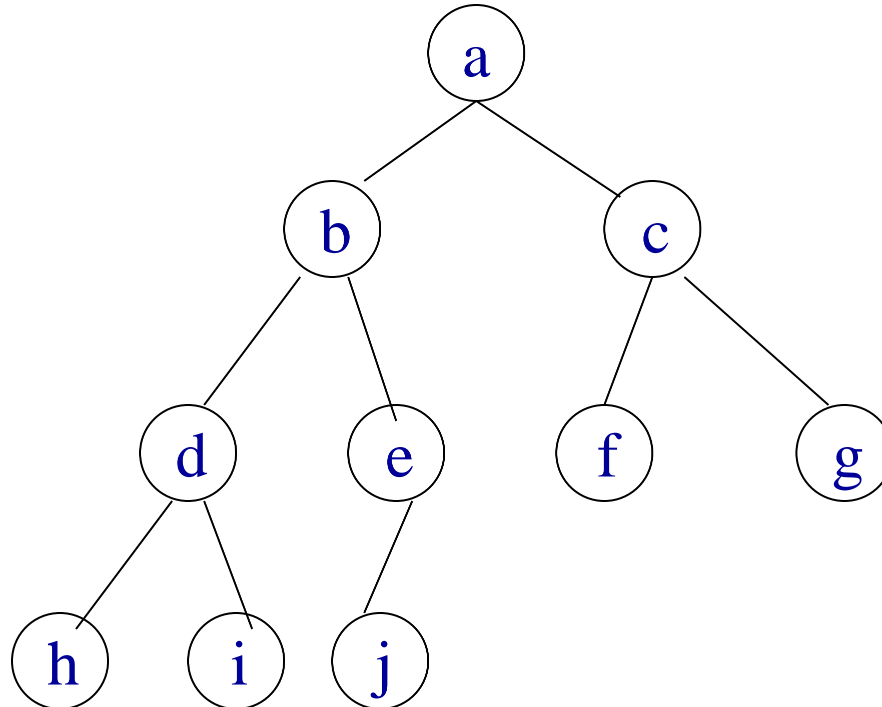
Propiedad de la estructura:

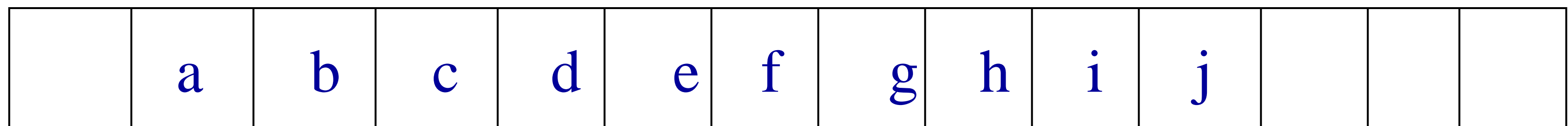
Un heap es un árbol binario completamente lleno, con la posible excepción del nivel más bajo, el cual se llena de izquierda a derecha. Un árbol así se llama un ***árbol binario completo***.

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)





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

tope

Binary Heaps

<u>Propiedad de la estructura</u> (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--

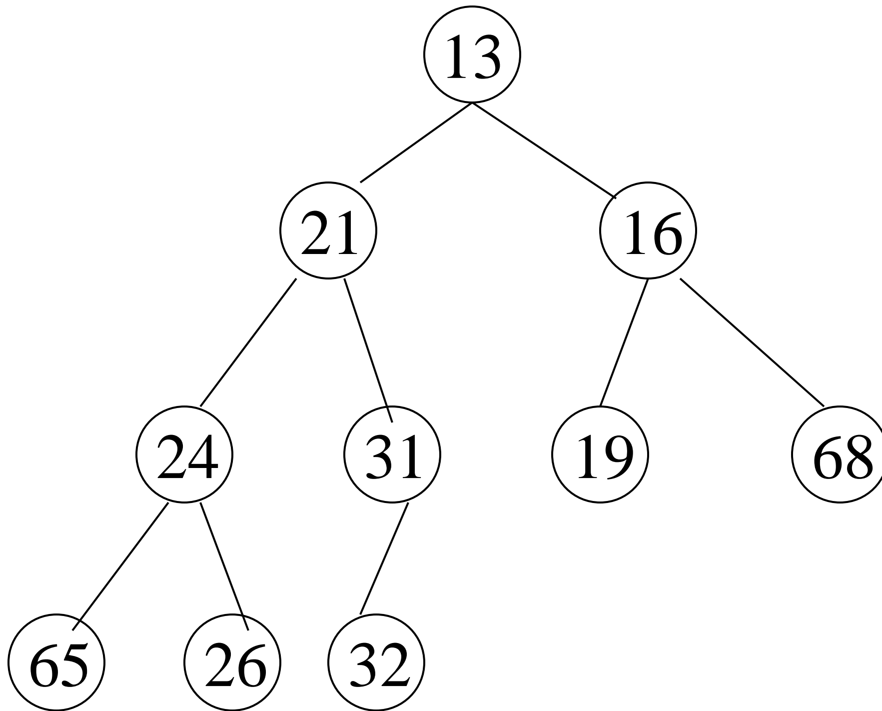
(menor o igual --si nos basamos en Multisets para prioridades--)

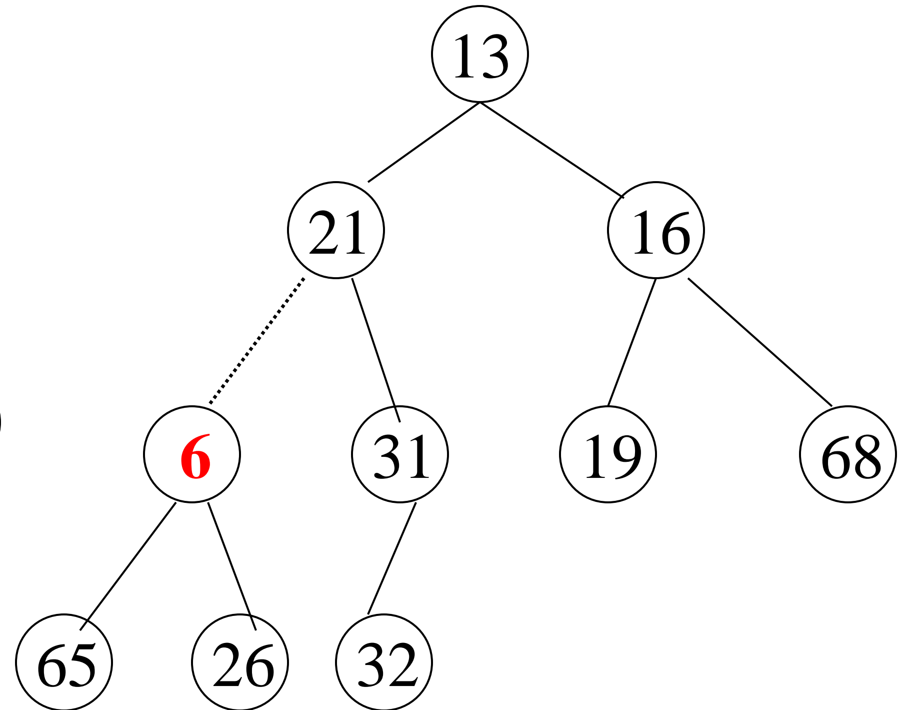
que la clave en X, con la excepción obvia de la raíz (donde esta el mínimo elemento).

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

Ejemplos:

Los montículos o Heaps (Binary Heaps)

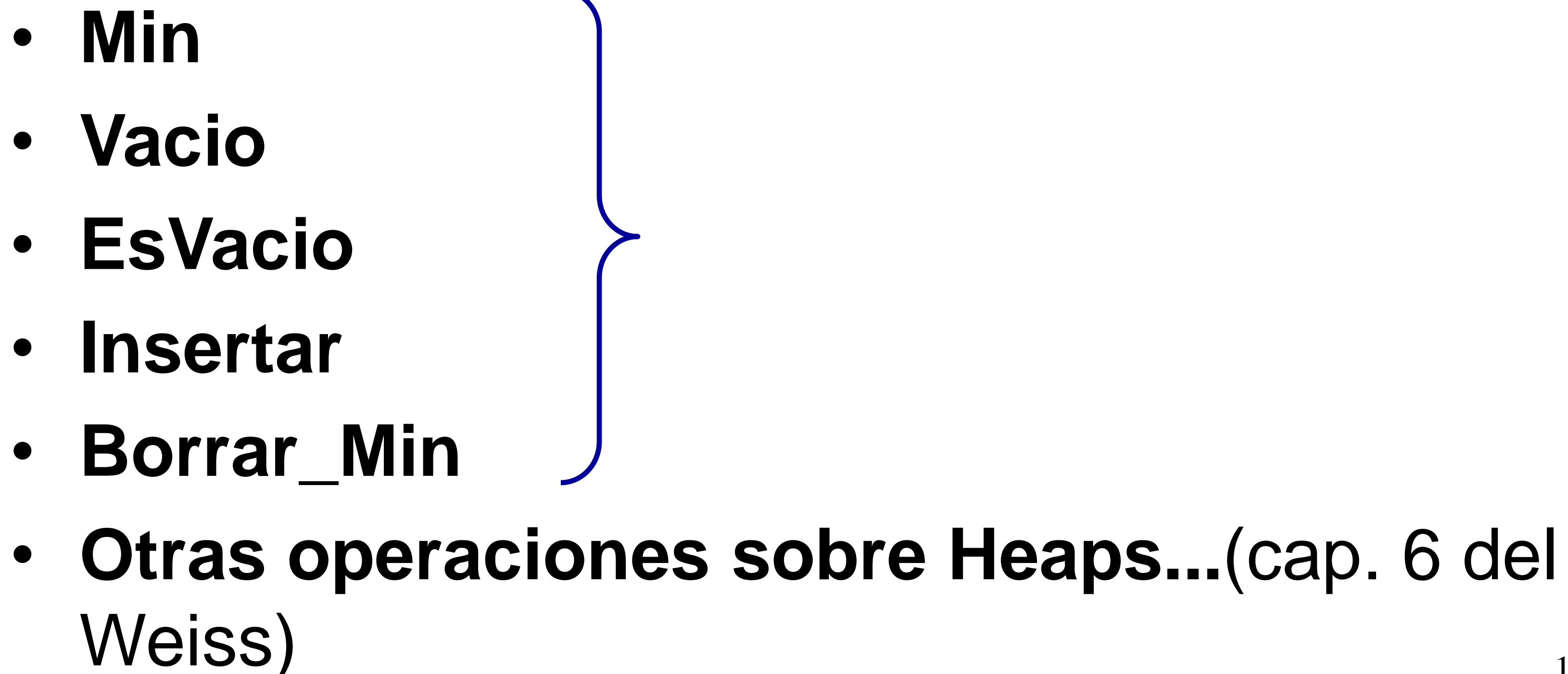




SI **NO**

Operaciones básicas para Heaps

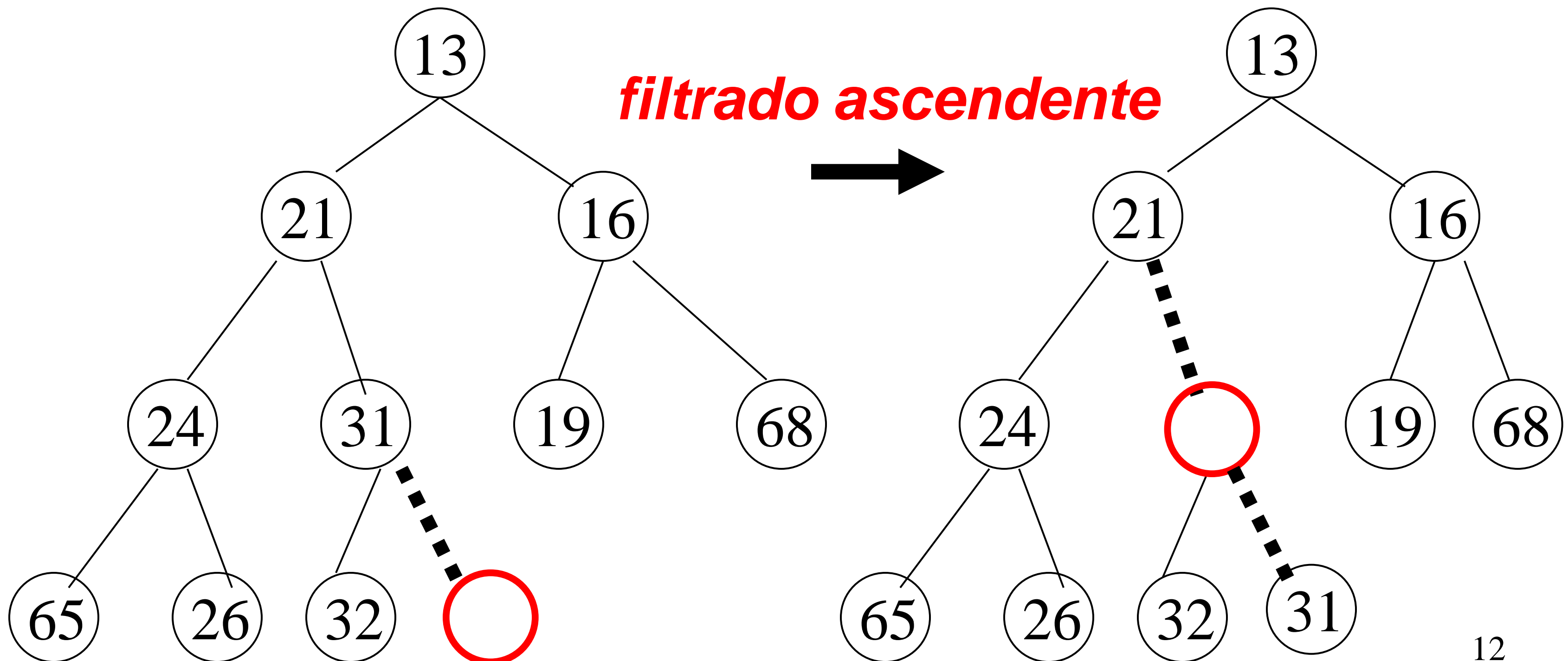
Todas las operaciones son fáciles (conceptual y prácticamente de implementar). Todo el trabajo consiste en asegurarse que se mantenga la propiedad de orden del Heap.



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

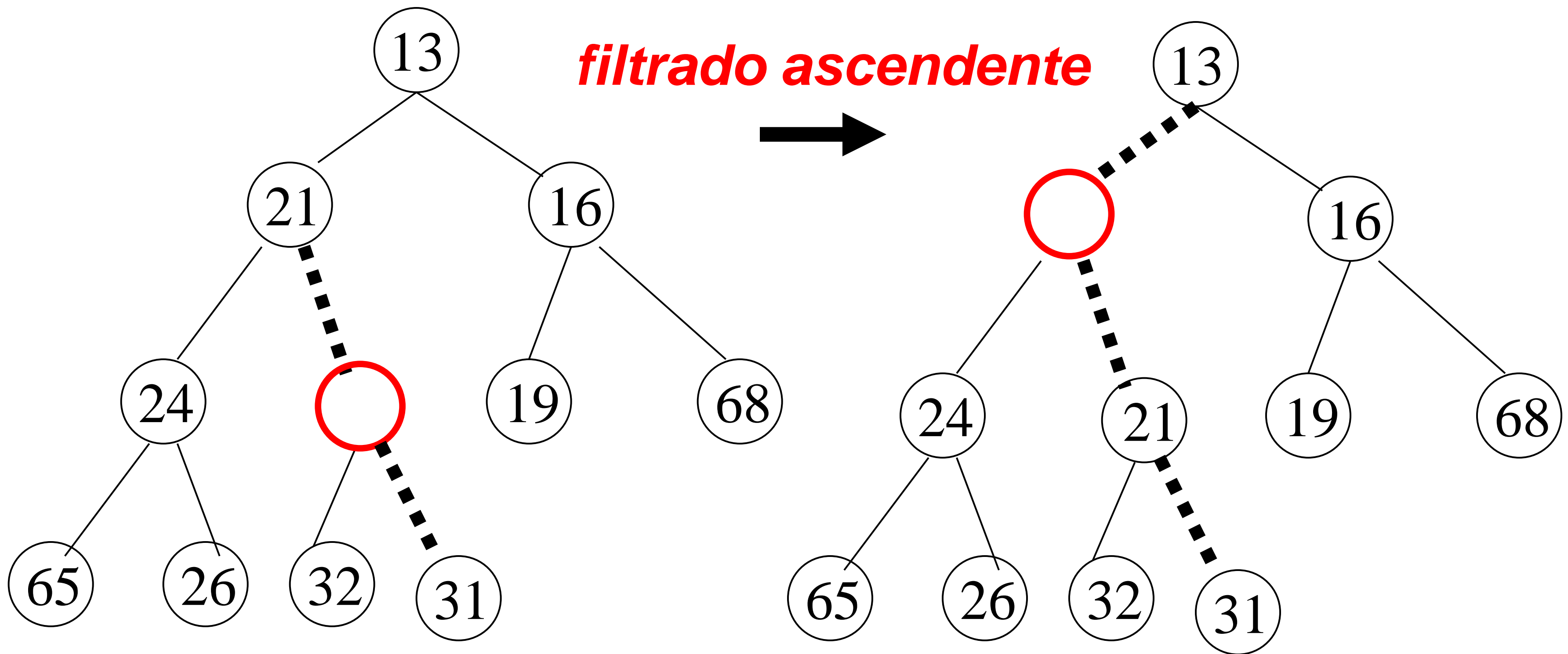
Operaciones básicas para Heaps

- **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):



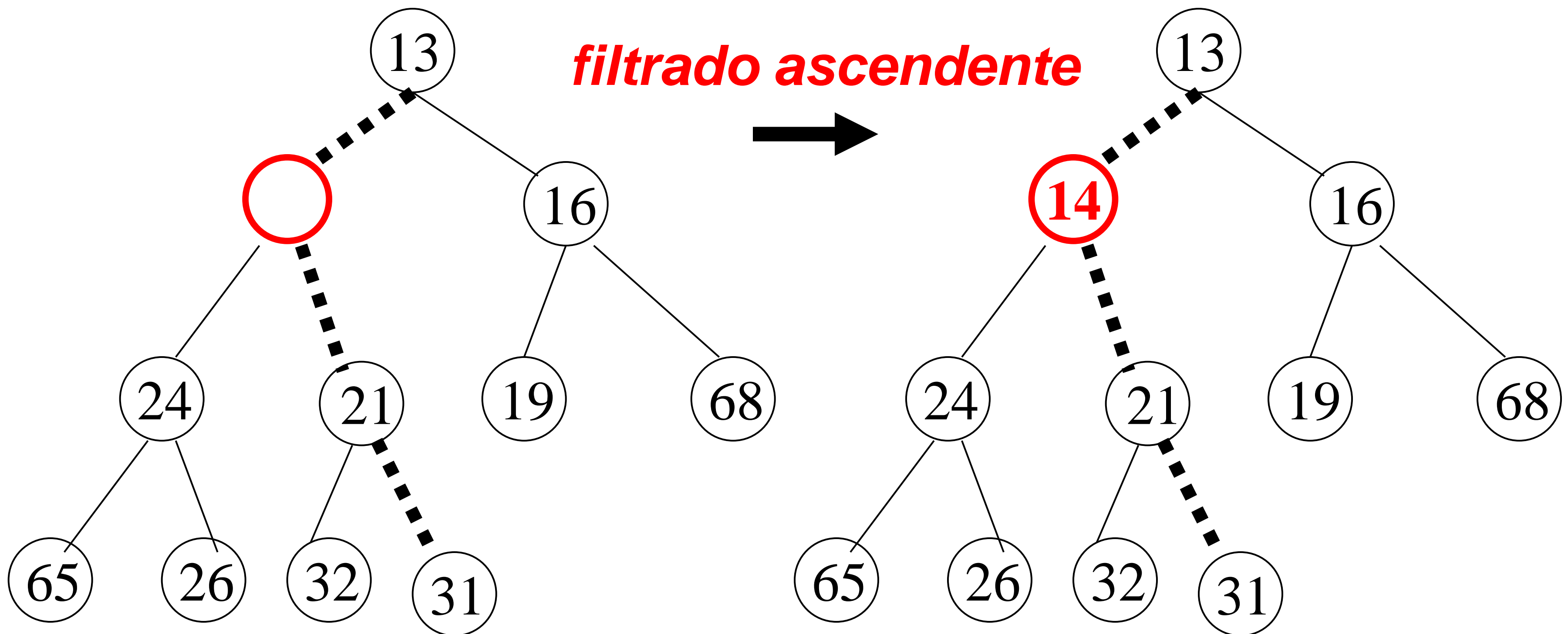
Operaciones básicas para Heaps

Inserción del 14



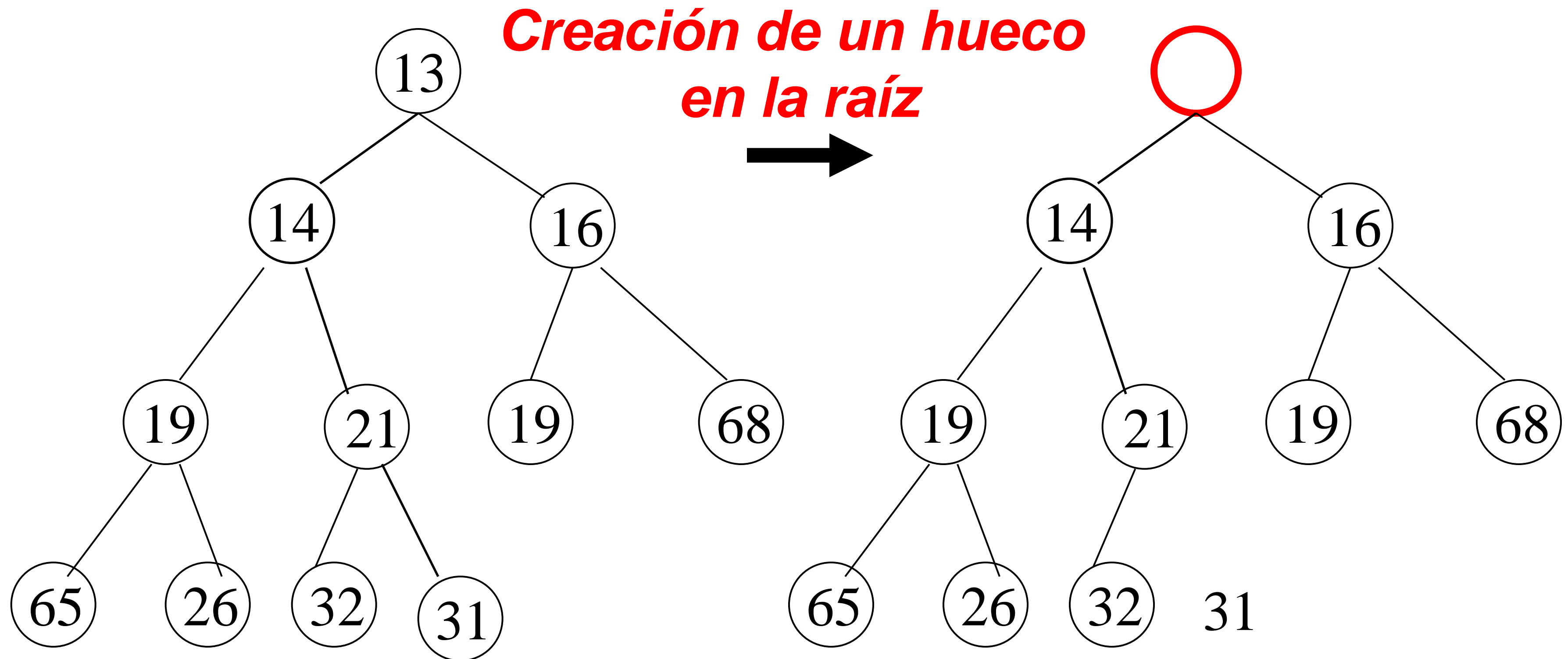
Operaciones básicas para Heaps

Inserción del 14



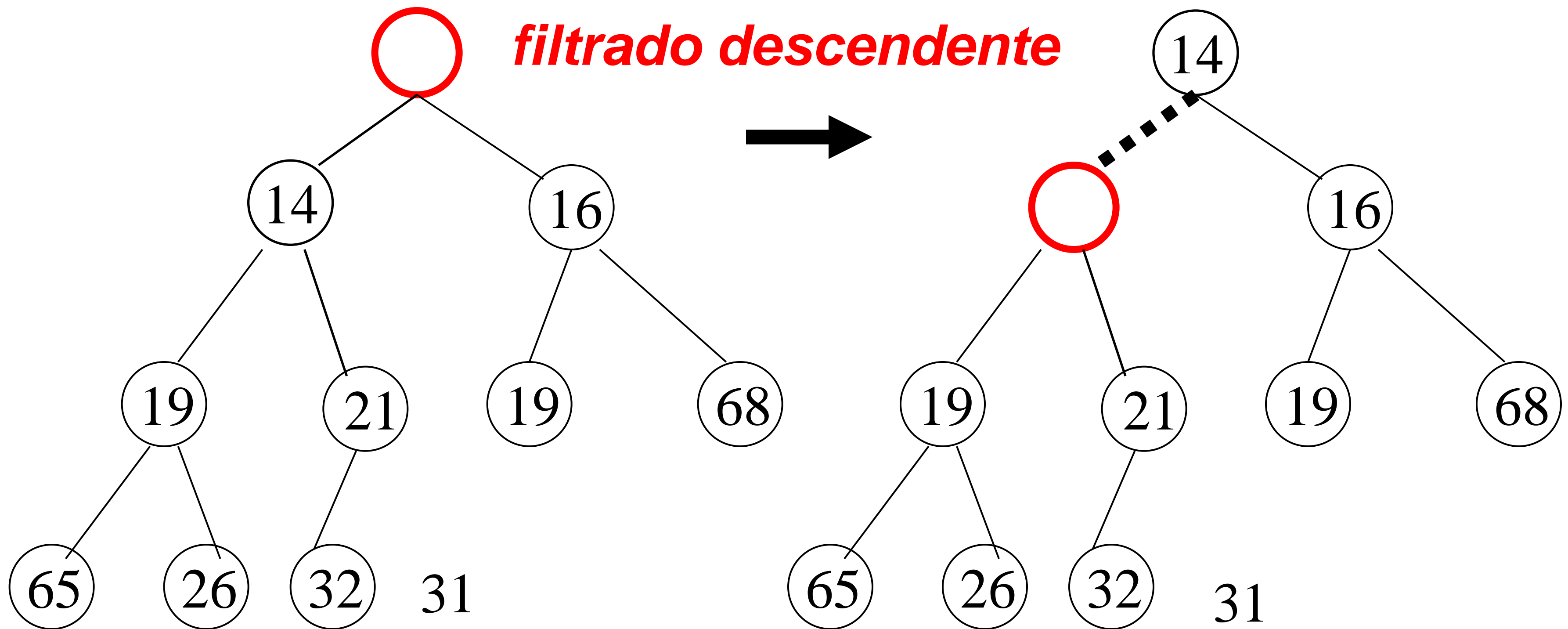
Operaciones básicas para Heaps

Eliminar el mínimo, Borrar_Min: O(log n)



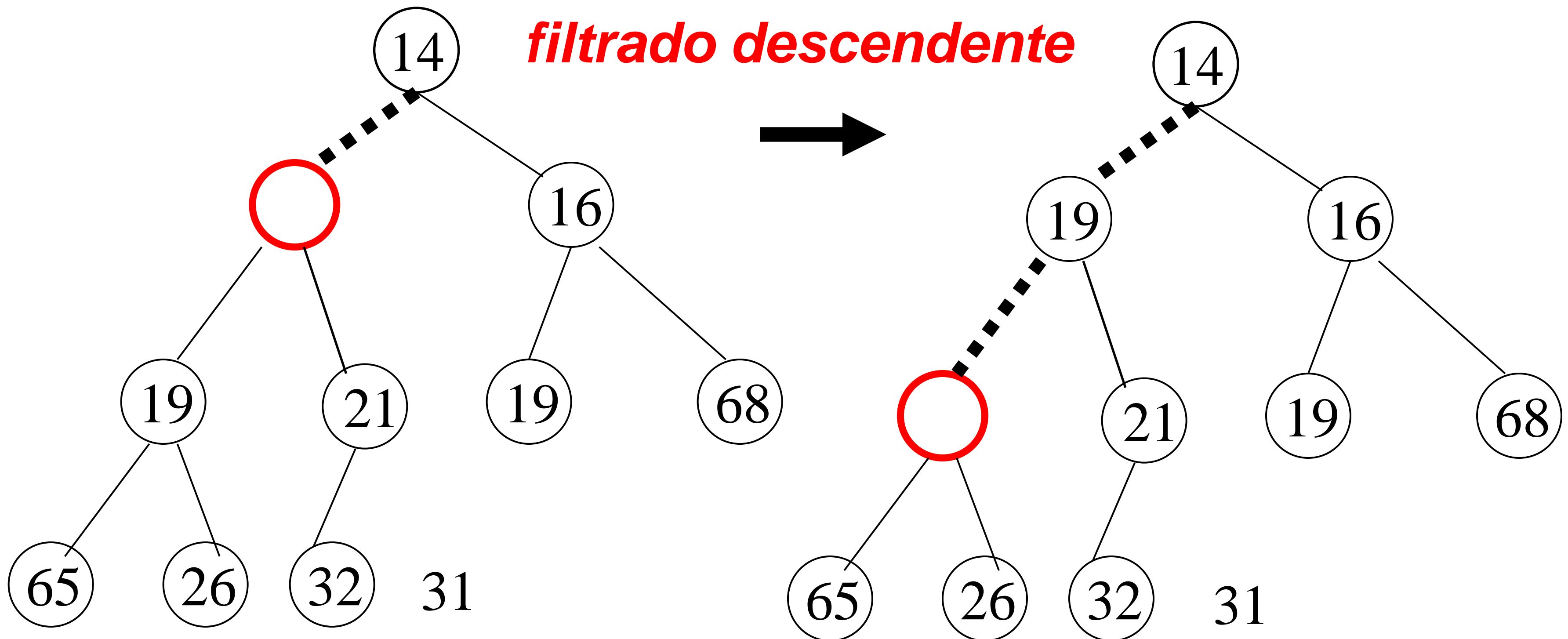
Operaciones básicas para Heaps

Eliminar el mínimo



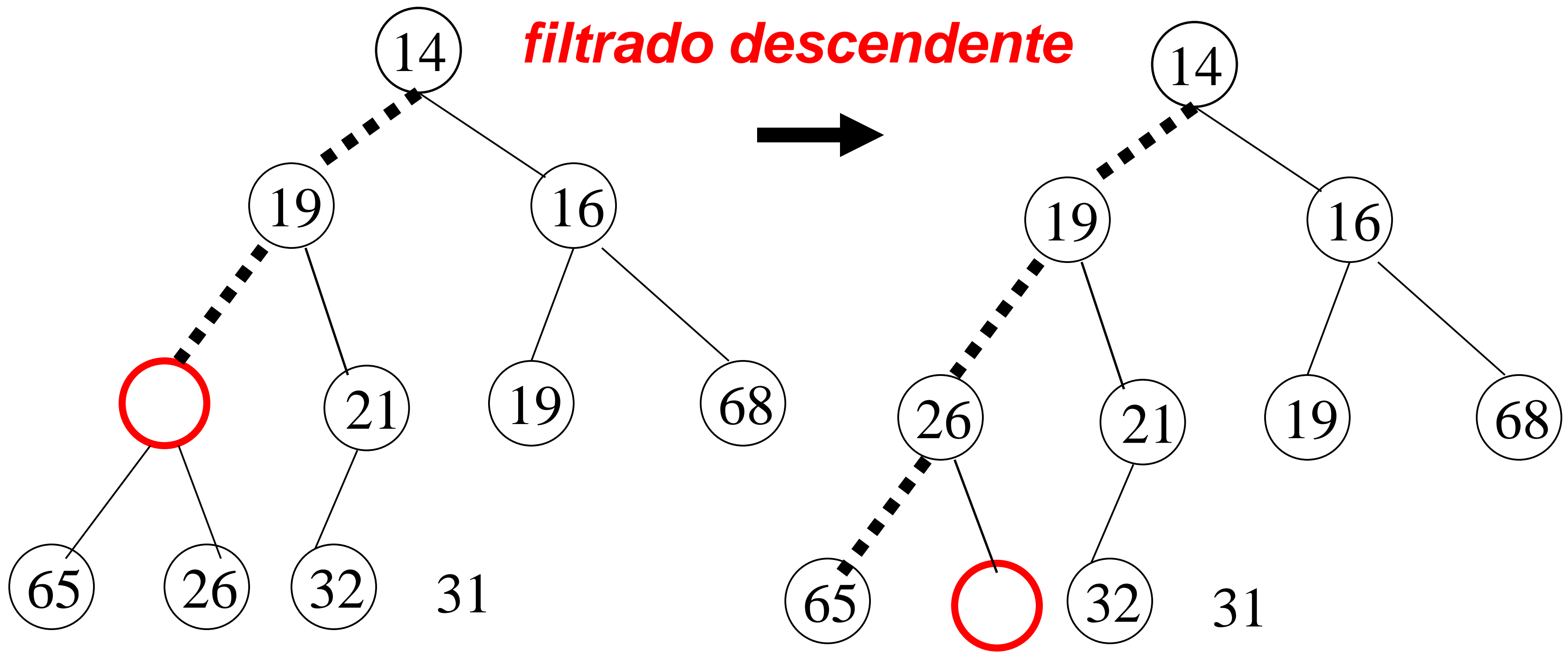
Operaciones básicas para Heaps

Eliminar el mínimo



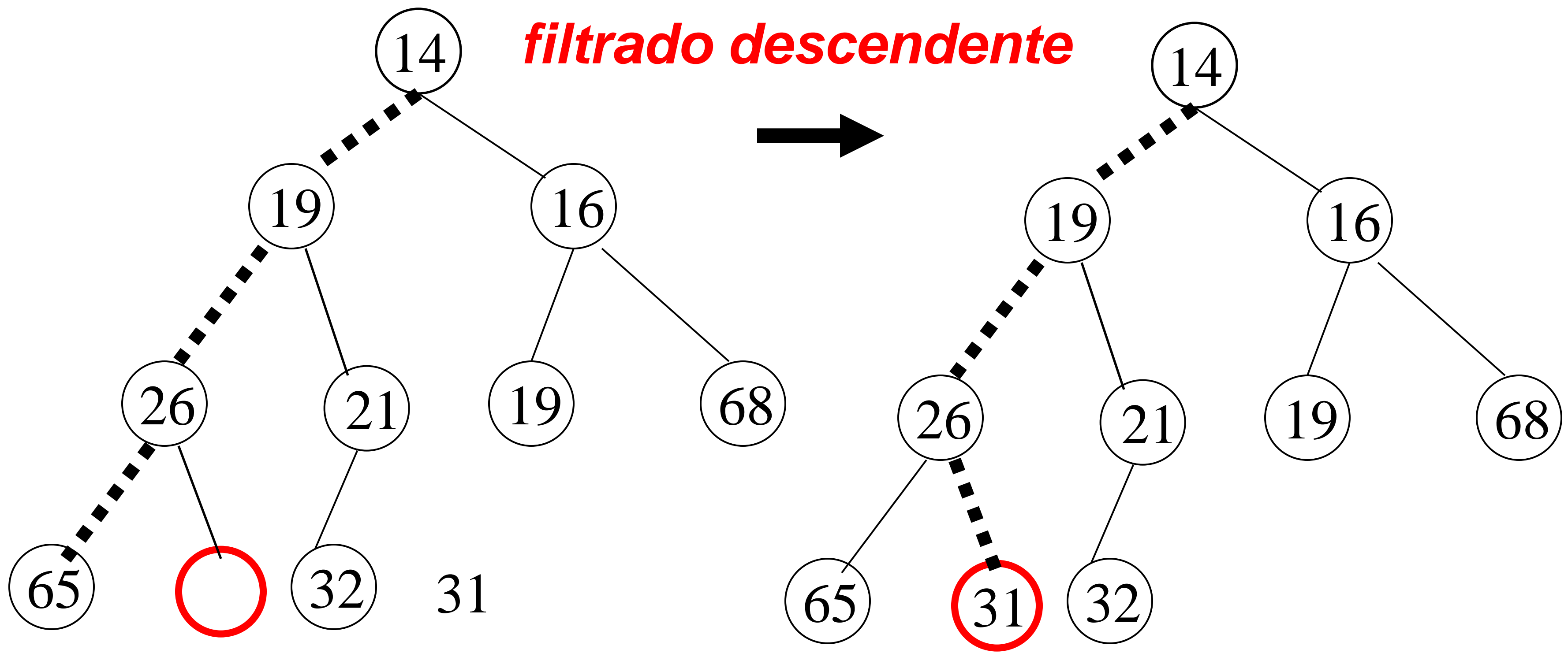
Operaciones básicas para Heaps

Eliminar el mínimo



Operaciones básicas para Heaps

Eliminar el mínimo



Colas de Prioridad 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)**

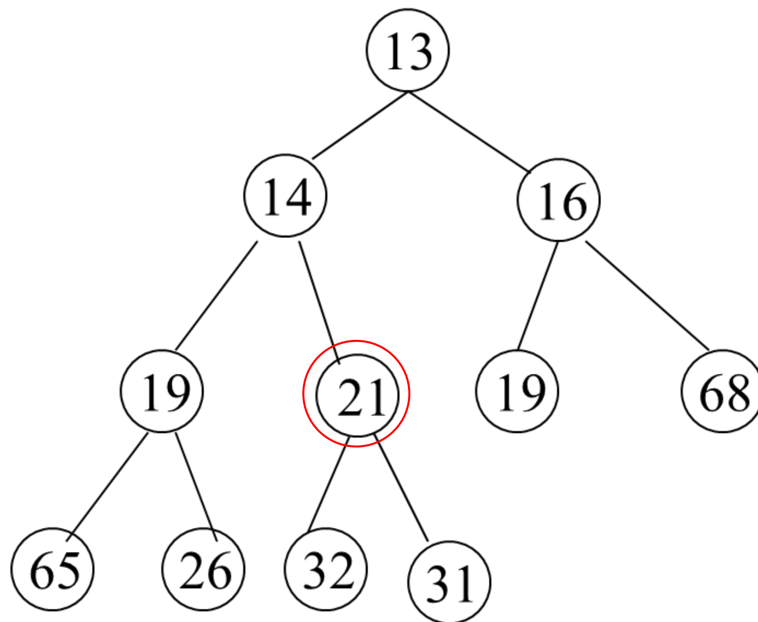
reduce el valor de la clave (clave) en la posición x por una cantidad positiva d. Como esto puede violar el orden del heap, debe arreglarse con un *filtrado ascendente*.

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)

- **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*.

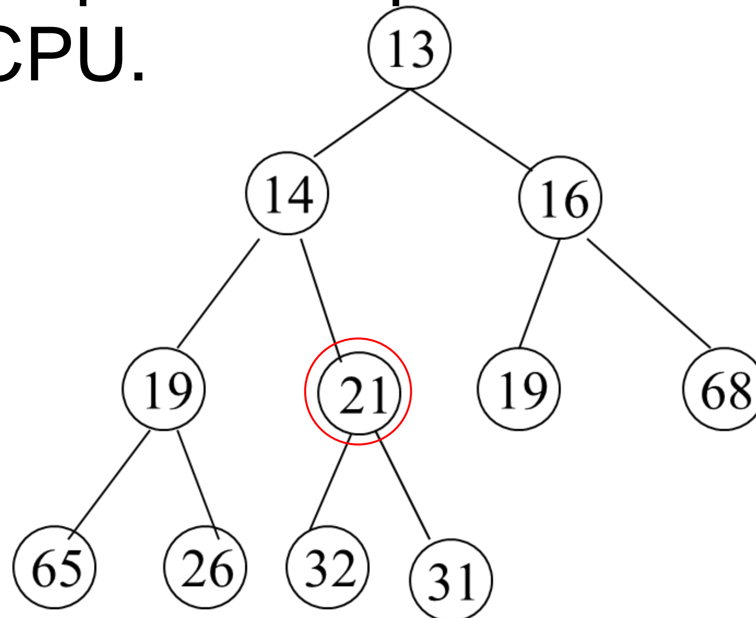


Colas de Prioridad Extendidas (Heap)

- **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.

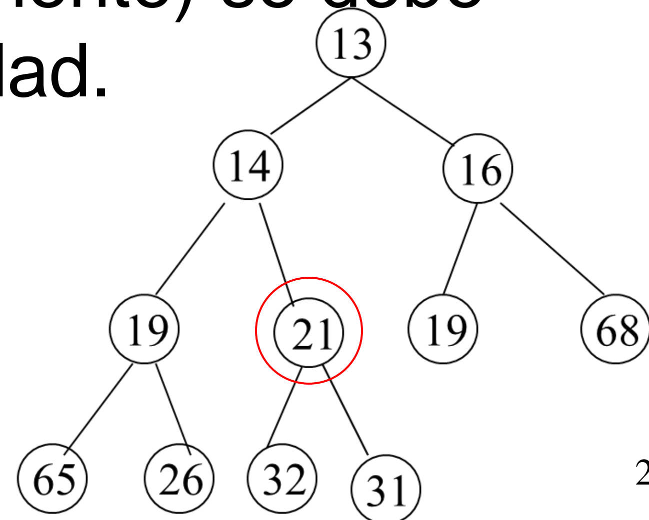


Colas de Prioridad Extendidas (Heap)

- **eliminar (x, Heap)**

  retira el nodo en la posición x del Heap. Esto puede hacerse ejecutando primero decrementar_clave(x, ∞, 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.



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**
- **…**

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:
}
```

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

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

```
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 destrute 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, lista->cadena, strlen(lista->cadena));
        l = l->sig;
   }
   while (cantidad(cp)!=0){
        cout << obtener(cp);
        desencolar(cp);
   }
   destruir(cp);
}
```