

Estructuras de Datos y Algoritmos 1

Estructuras Múltiples y Colas de Prioridad

Carlos Luna

cluna@fing.edu.uy

Estructuras Múltiples

Con frecuencia, un problema aparentemente simple de representación de un conjunto o correspondencia, conlleva un difícil problema de elección de estructuras de datos.

La elección de una estructura de datos para el conjunto simplifica ciertas operaciones, pero hace que otras lleven demasiado tiempo, y, al parecer no existe una estructura de datos que haga más sencillas todas las operaciones.

En tales casos, la solución suele ser el uso simultáneo de dos o más estructuras diferentes para el mismo conjunto o correspondencia.

Ejemplos (1)

PROBLEMA: Ranking de la FIFA

Se desea mantener una escala de equipos de futbol en la que cada equipo esté situado en un único puesto. Los equipos nuevos se agregan en la base de la escala, es decir, en el puesto con numeración más alta. Un equipo puede retar a otro que esté en el puesto inmediato superior (el i al $i-1$, $i > 1$), y si le gana, cambia de puesto con él.

Pensar en una representación para esta situación !!

Ejemplos (1)

Se puede representar la situación anterior mediante un TAD cuyo modelo fundamental sea una **correspondencia de nombres de equipos** (cadenas de char) **con puestos** (enteros 1, 2, ...).

Las 3 operaciones a realizar son:

- **AGREGA(nombre)**: agrega el equipo nombrado al puesto de numeración más alta.
- **RETA(nombre)**: es una función que devuelve el nombre del equipo del puesto $i-1$ si el equipo nombrado está en el puesto i , $i > 1$.
- **CAMBIA(i)**: intercambia los nombres de los equipos que estén en los puesto i e $i-1$, $i > 1$.

Ejemplos (1)

ALTERNATIVA 1:

Un arreglo de ESCALA, donde ESCALA[i] sea el nombre del equipo en el puesto i.

- AGREGA: Cómo sería?, Qué tiempo llevaría?
- CAMBIA: Cómo sería?, Qué tiempo llevaría?
- RETA(nom): Cómo sería?, Qué tiempo llevaría?

ALTERNATIVA 2: Qué otra representación podría considerarse?

Ejemplos (1)

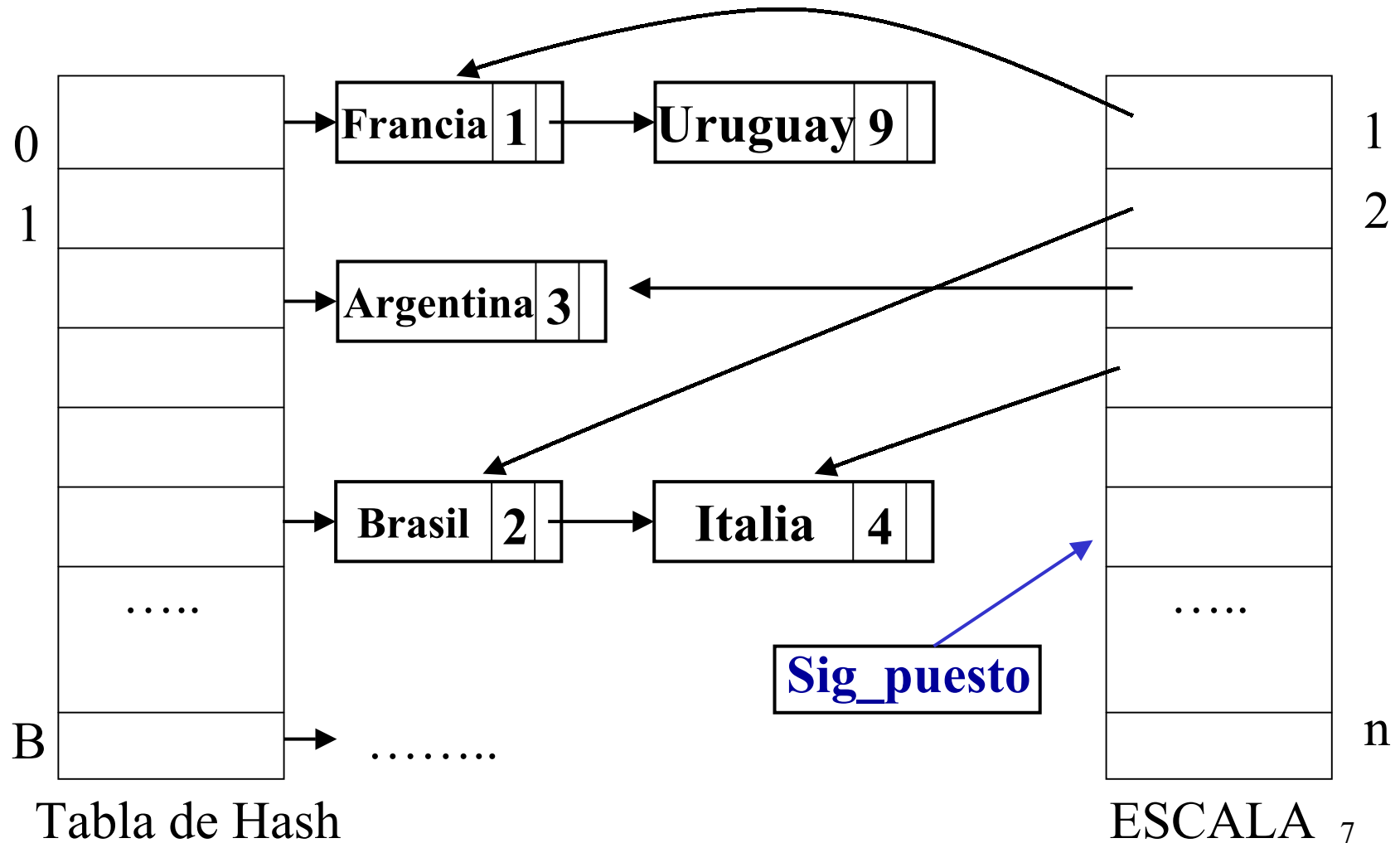
ALTERNATIVA 2: Qué otra representación podría considerarse?

=> OPEN HASING (en el supuesto que es posible mantener en número de *buckets* proporcional al número de equipos)

- AGREGA: Qué tiempo llevaría?
- CAMBIA: Qué tiempo llevaría?
- RETA(nom): Qué tiempo llevaría?

Ejemplos (1)

Y si combinamos las dos estructuras, ¿ cuáles son los tiempos de ejecución ?



Ejemplos (2)

Analicen el ejemplo de asociaciones muchos a muchos y el uso de una estructura de listas múltiples del libro de Aho, Hopcroftand y Ullman (cap 4, secc 4.12).

estudiantes - cursos

- un estudiante inscripto a más de un curso.
- un curso tiene más de un estudiante.

Pensar en una representación simple y alguna más eficiente (sobre todo en cuanto a espacio de almacenamiento)

Ejemplos (3)

Se desea una estructura de datos para mantener información de atletas que han participado en la competencia de los 100 metros llanos en los juegos olímpicos en de los últimos 20 años. Los datos que interesan sobre cada competidor son la posición (1 a 8) y el año en que compitió. Cada competidor está identificado por un código (entero). Si un atleta participó más de una vez, aparece con la mejor posición que obtuvo.

Pensar en una representación del problema, para que los siguientes requerimientos se realicen eficientemente:

Ejemplos (3)

- ¿ Cuáles competidores salieron alguna vez en la primera posición ?.
- ¿ Saber el año en que un competidor obtuvo el máximo puesto ?.
- Imprimir los competidores ordenados por código.
- Imprimir todos los competidores que están en el puesto k ($1 \leq k \leq 8$).
- Saber cuál es el puesto más alto obtenido por un competidor.
- Las naturales de inserción, supresión y búsqueda de competidores en una posición.

Ejemplos (3)

Considere la siguiente estructura:

```
typedef struct nodo {  
    int codigo, año;  
    struct nodo *izq, *der;  
    struct nodo *sig;    // Siguiendo en lista por puestos  
} celdaCompe;  
  
typedef celdaCompe *abbCompe;  
  
typedef struct {  
    abbCompe competidores;  
    abbCompe puestos[8];  
} resultados100m;
```

Ejemplos (3)

La anterior es una estructura dual que permite ingresar por código de jugador o por puesto. El campo competidores es un árbol binario de búsqueda por código del competidor. El campo puestos es un arreglo, que en el lugar i -ésimo contiene un puntero a la lista de competidores que obtuvieron el puesto i . Esta lista esta hilvanada sobre el mismo árbol.

- Ejercicio: escribir un procedimiento con el encabezado:
void BorrarCompetidor (resultados100m *Res, int cod)
Este procedimiento borra de Res al competidor con código cod.

Ejemplos (3)

- Indique el orden de ejecución para el caso medio del procedimiento **BorrarCompetidor**.

Modifique la estructura de forma tal que ese tiempo pase a ser $O(\log n)$.

- Ejercicio: escriba el procedimiento BorrarCompetidor para la modificación propuesta en el punto anterior.

Ejemplos (4)

Los empleados de cierta compañía se representan en la base de datos de la compañía por su nombre (que se supone único), el número de empleado y el número de seguridad social.

Sugerir una estructura de datos que permita, dada una representación de un empleado, encontrar las otras dos representaciones del mismo individuo. ¿Qué rápida, en promedio, puede lograrse que sea cada una de esas operaciones?.

Más Ejemplos...

En el práctico...

Conclusiones:

¿para qué sirven las multiestructuras?

¿qué relación tienen éstas con TADs?

Bibliografía

Libros

- Estructuras de Datos y Análisis de Algoritmos en Pascal

Mark Allen Weiss; Benjamin/Cummings Inc., 1993.

- **Estructuras de Datos y Algoritmos**

A. Aho, J. E. Hopcroft & J. D. Ullman; Addison-Wesley, 1983.

(Cap 4, secc. 4.12)

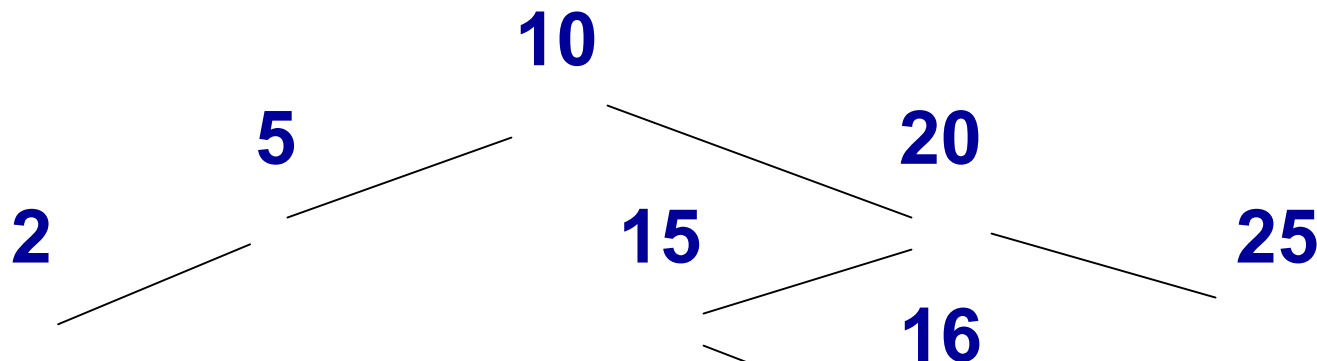
Hacer el práctico: es la mejor teoría...

Un poco de práctica...

Ejercicios:

- impresión por niveles en un AB
- búsqueda del k-ésimo menor elemento en un ABB (o AVL).

Ayuda: considerar que cada nodo n contiene un campo ***lsize*** que indica la cantidad de nodos en el sub-árbol izquierdo de n más uno.



Buscar el 2 menor, el 4, el 5 y el 10 menor.

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).

Veamos una especificación de este TAD:

Colas de Prioridad

```
// módulo "Diccionario.h"
template <class Etype>
class ColaPrioridad {
public:
    // Constructoras
    virtual void Vacio() = 0;
    // construye la cola de prioridad vacía

    virtual void Insertar(const Etype & x) = 0;
    /* inserta el elemento x en la cola de
    prioridad */
```

Colas de Prioridad

// Predicados

```
virtual bool EsVacio() = 0;
```

```
/* retorna true si y sólo si la cola de  
prioridad es vacía */
```

// Selectoras

```
virtual Etype Min() = 0;
```

```
// retorna el elemento de menor prioridad  
de la cola de prioridad. Pre-cond: la cola  
no está vacía.
```

Colas de Prioridad

```
virtual void Borrar_Min()= 0;  
/* Borra el elemento de menor proriad de  
la cola de prioridad */  
  
/* esta última operacion podría tener o no  
precondición. Alternativamente, Borrar_Min  
podría retornar el elemento eliminado (en  
este caso si sería necesaria una  
precondición, salvo que se retorne un  
puntero al elemento */  
  
};
```


Colas de Prioridad: Implementaciones

Con excepción de las tablas de hashing, todas las realizaciones estudiadas para conjuntos (diccionarios) son también apropiadas para colas de prioridad. **Es decir ¿?**

¿Por qué una tabla de hash no es adecuada (más adecuada que una lista enlazada)?

Usando un lista no ordenada, ¿ 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 en el curso de Estruct. y Algoritmos 1, 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 nivel de C++, la especificación acotada de un TAD es conceptualmente una subclase de la no acotada.

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 *falso*).

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)

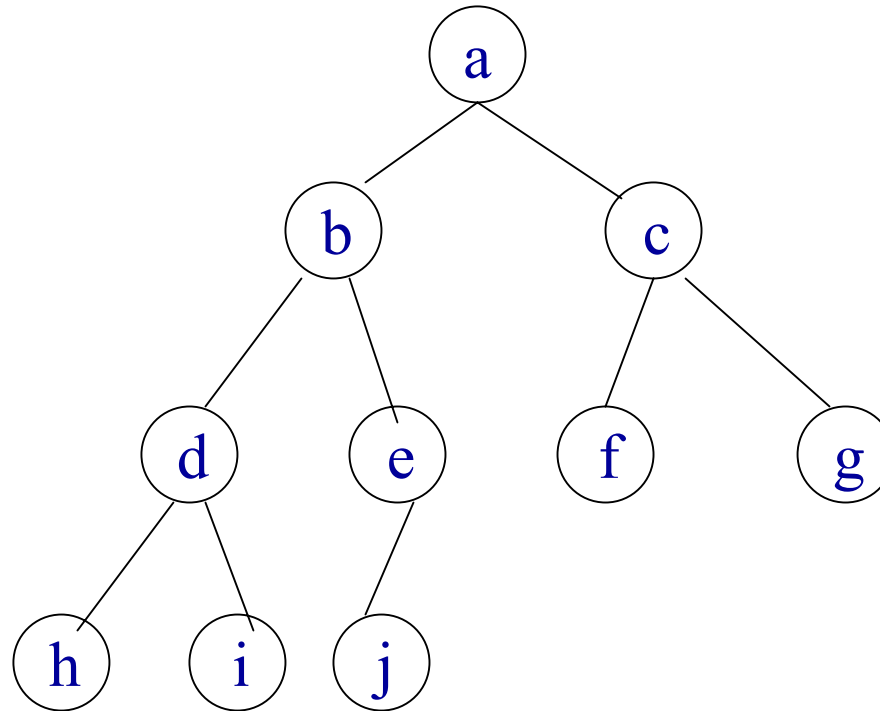
Propiedad de la estructura:

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)

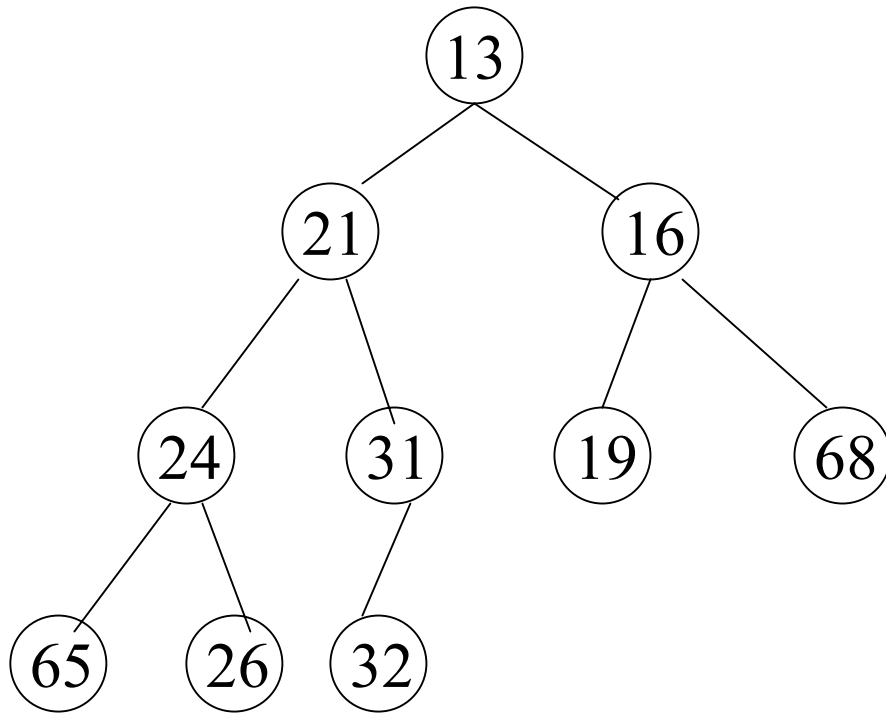
Propiedad de orden del heap:

Para todo nodo X , la clave en el padre de X es menor (o igual) que la clave en X , con la excepción obvia de la raíz (donde esta el mínimo elemento).

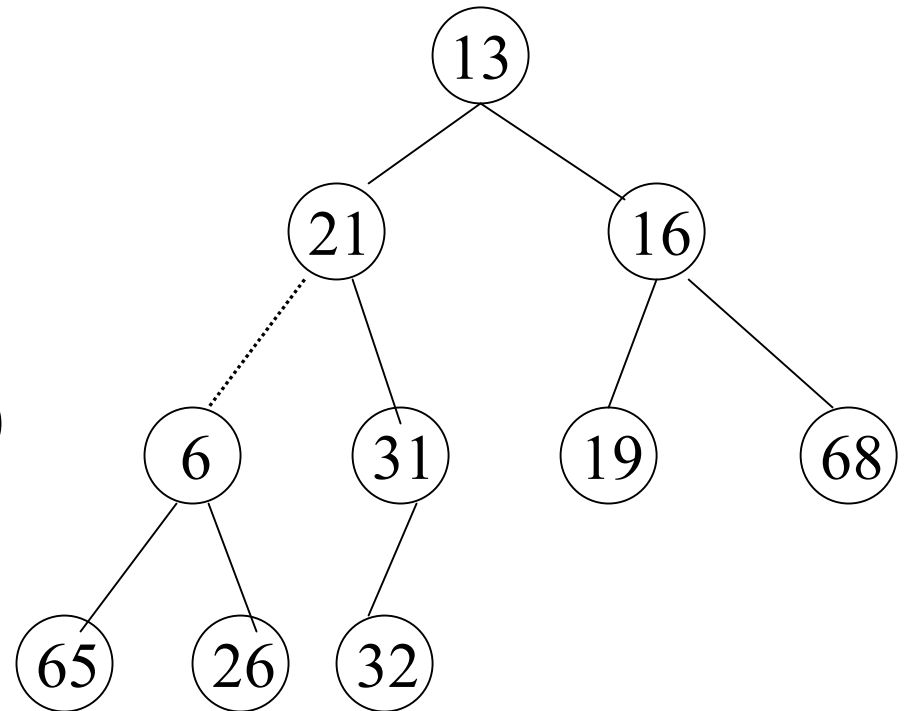
Esta propiedad permite realizar eficientemente las propiedades de una cola de prioridad que refieren al mínimo.

Ejemplos:

Los montículos o Heaps (Binary Heaps)




SI



NO

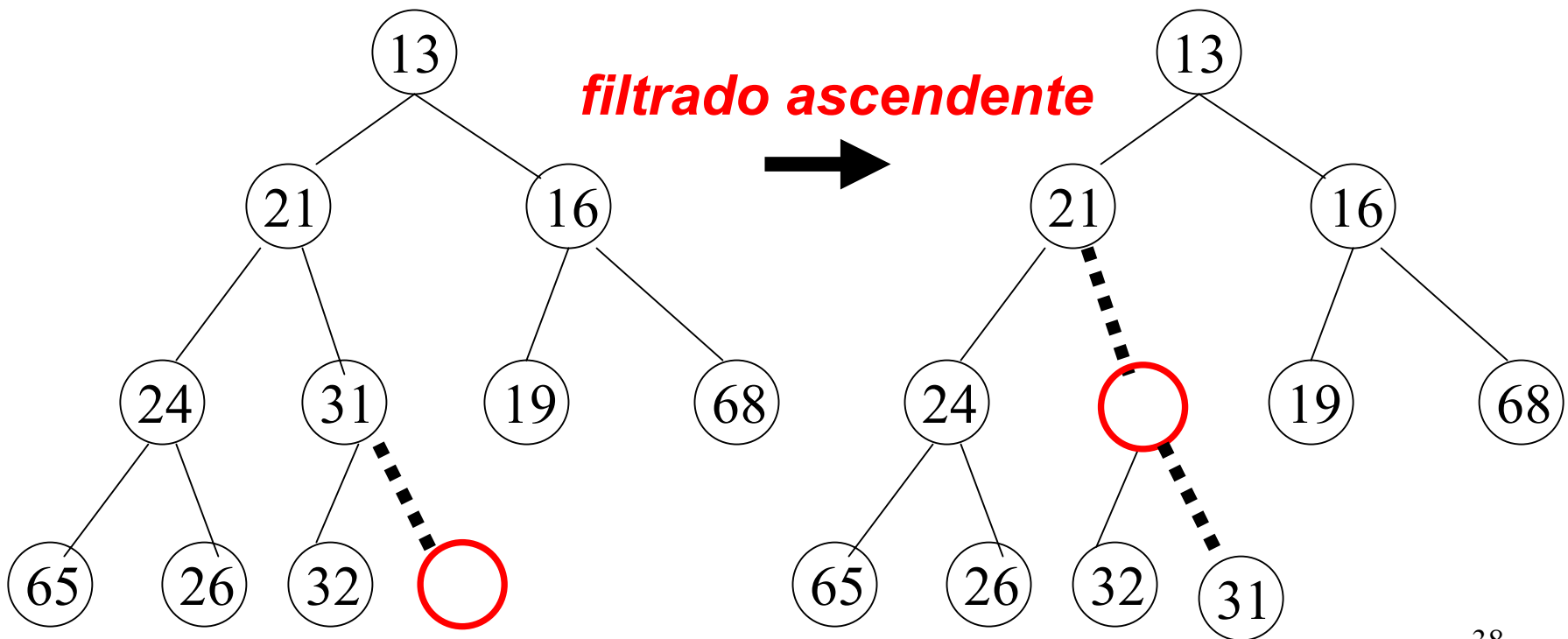
Operaciones básicas para Heaps

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...(secc 6.3.4 del Weiss)**
- 
- Desarrollarlas....**

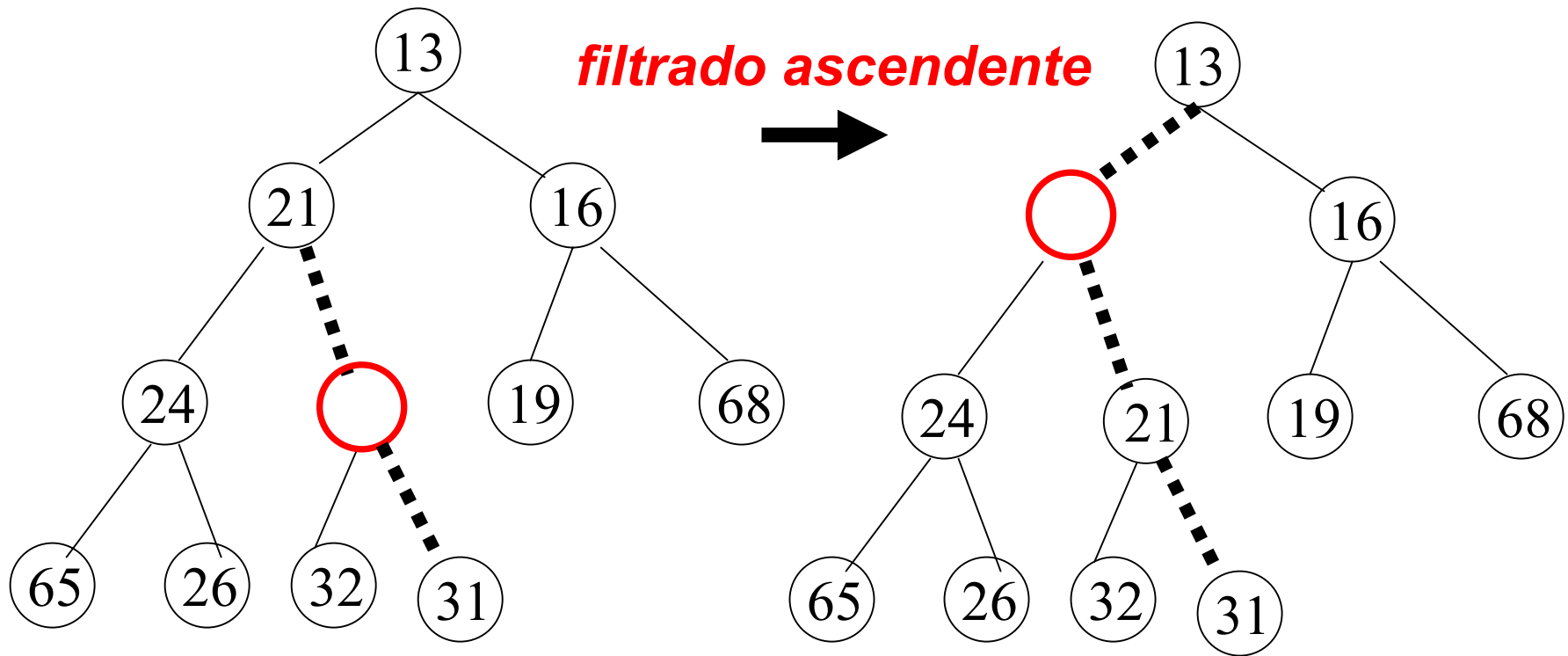
Operaciones básicas para Heaps

- **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:



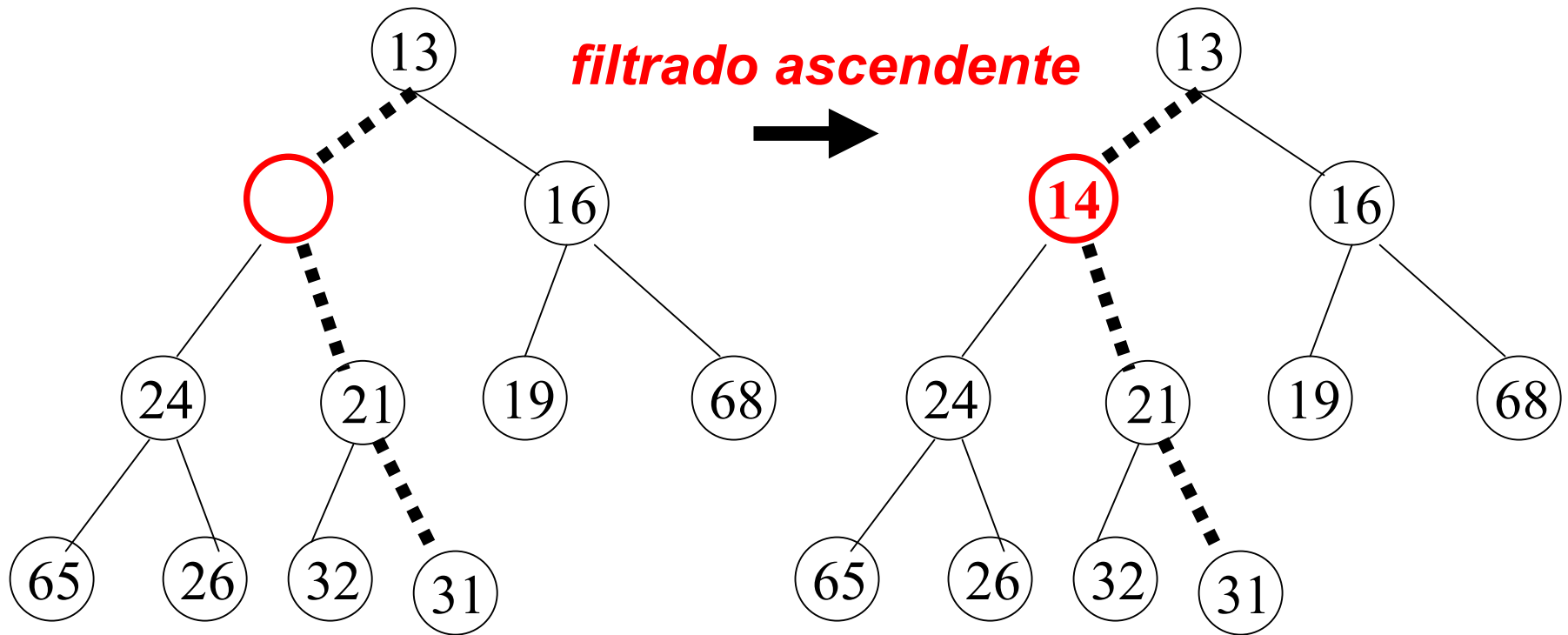
Operaciones básicas para Heaps

Inserción del 14



Operaciones básicas para Heaps

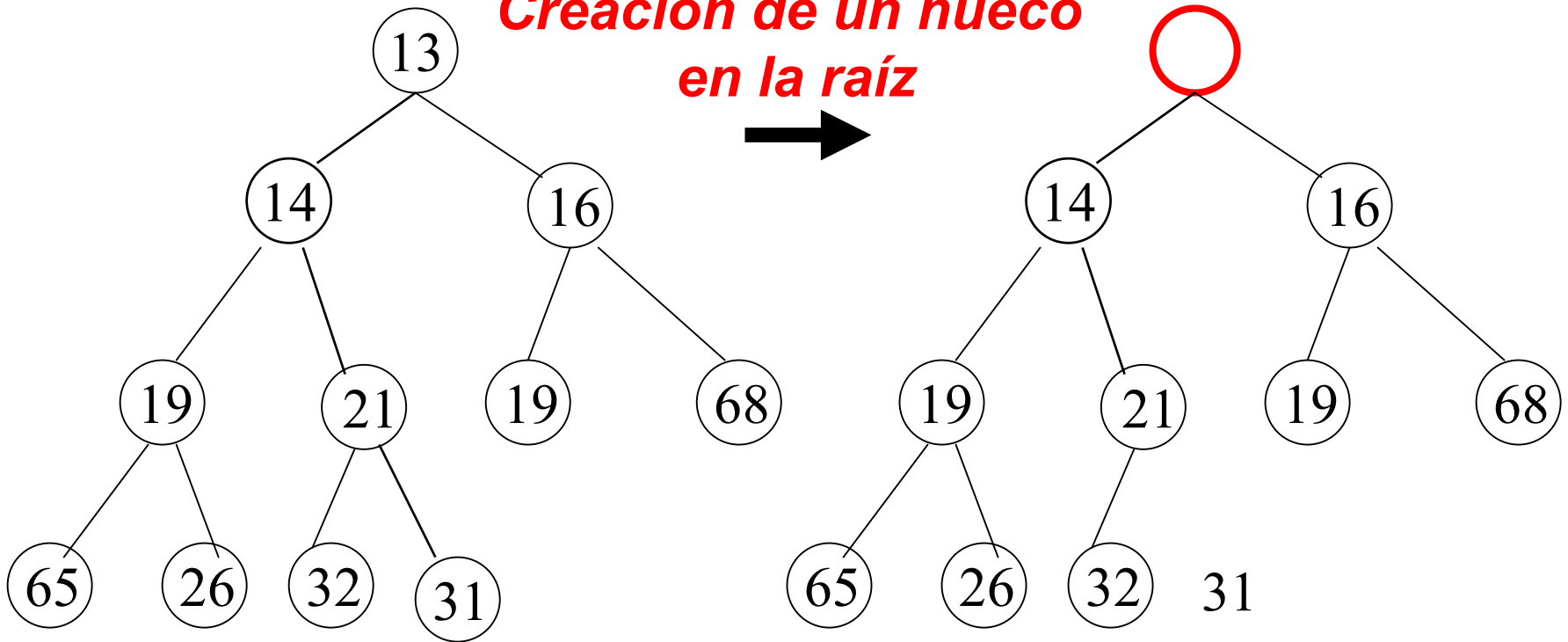
Insertión del 14



Operaciones básicas para Heaps

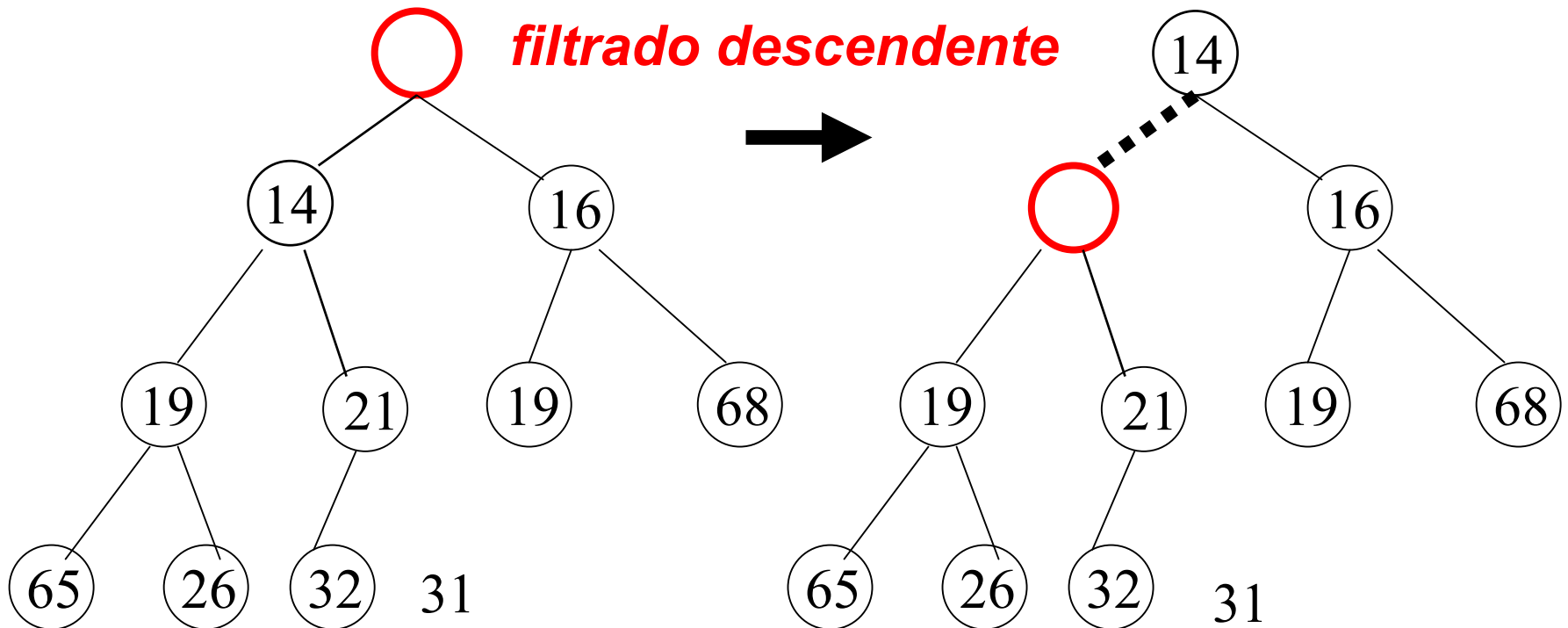
Eliminar el mínimo, Borrar_Min: $O(\log n)$

*Creación de un hueco
en la raíz*



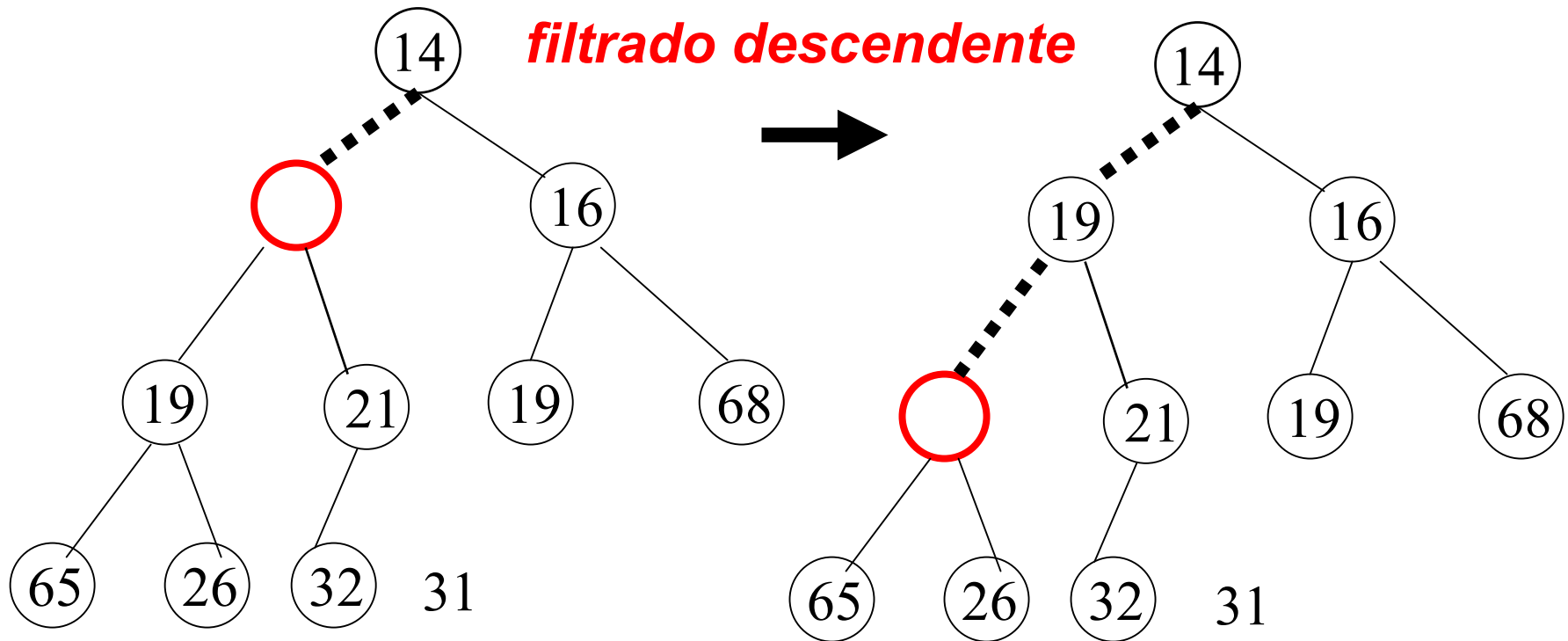
Operaciones básicas para Heaps

Eliminar el mínimo



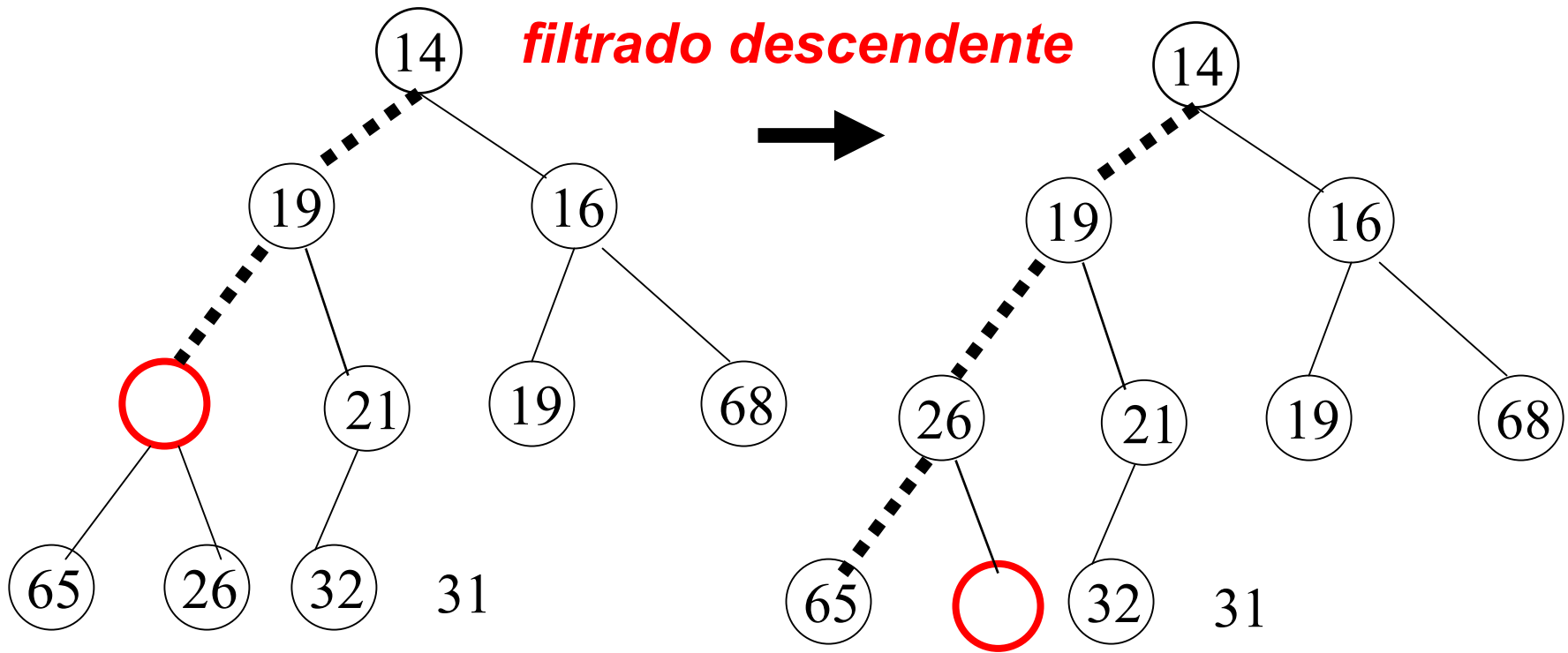
Operaciones básicas para Heaps

Eliminar el mínimo



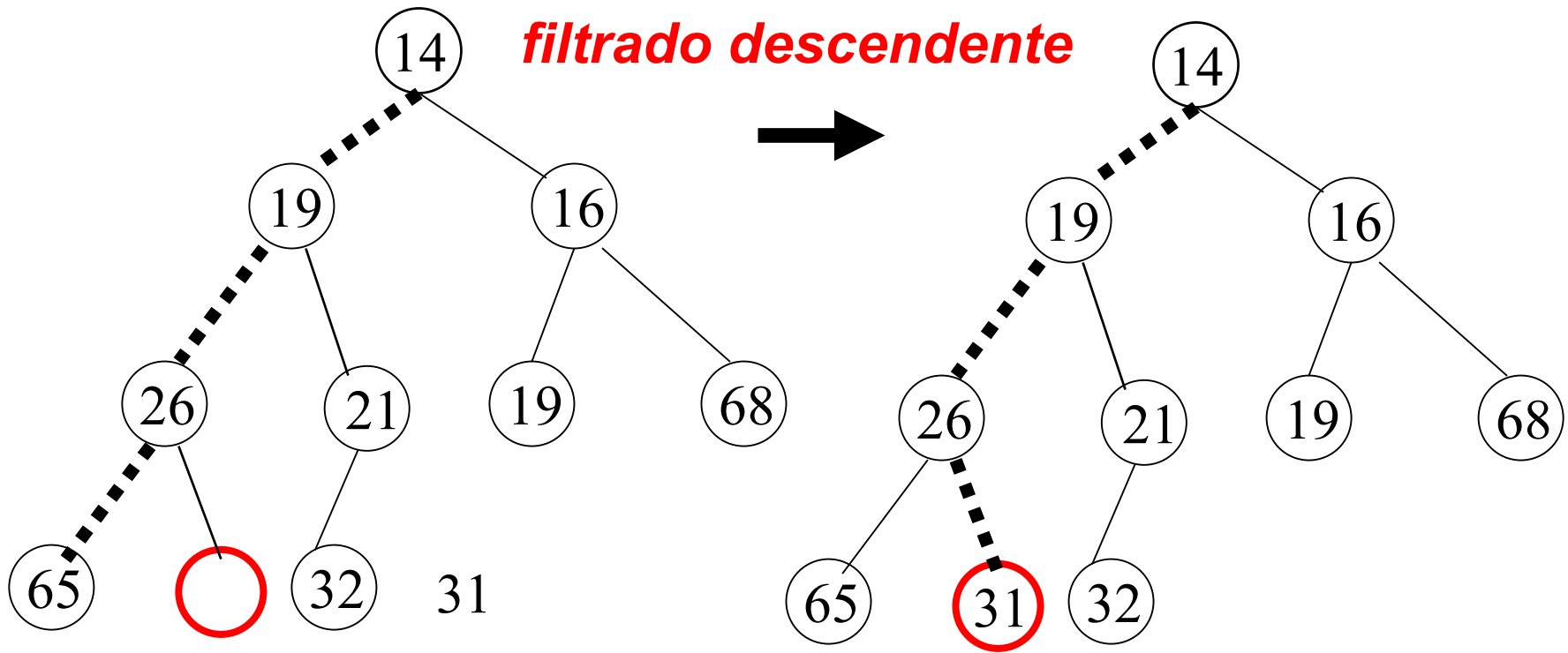
Operaciones básicas para Heaps

Eliminar el mínimo



Operaciones básicas para Heaps

Eliminar el mínimo



Ejercicios

- Implementar el TAD Cola de Prioridad acotada de elementos de un tipo genérico usando un *binary heap*.
- Implementar el TAD Cola de Prioridad no acotada de elementos de un tipo genérico usando listas encadenadas ordenadas y árboles binarios de búsqueda (o un AVL).

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**
- **Leer las aplicaciones de la sección 6.4 del Weiss**

Colas de Prioridad Extendidas

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

Lectura a cargo de los estudiantes...

Colas de Prioridad Extendidas

Algunas operaciones adicionales sobre heaps que tienen $O(\log n)$ y presuponen conocida las posiciones de los elementos :

- **decrementar_llave (x, d, Heap)**

reduce el valor de la clave (llave) 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

- **incrementar_llave (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

- **eliminar (x, Heap)**

retira el nodo en la posición x del Heap. Esto puede hacerse ejecutando primero `decrementar_llave(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.

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

Colas de Prioridad Extendidas

- ¿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

- 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.

Bibliografía

Libros

- Estructuras de Datos y Análisis de Algoritmos en Pascal (o la versión en C++)

Mark Allen Weiss; Benjamin/Cummings Inc., 1993.

Capítulo 6

- **Estructuras de Datos y Algoritmos**

A. Aho, J. E. Hopcroft & J. D. Ullman; Addison-Wesley, 1983.

(Cap 4, secc. 4.10)