Programación II

Tema 6. Colas de prioridad y Heaps

Iván Cantador y Rosa Mª Carro
Escuela Politécnica Superior
Universidad Autónoma de Madrid

Contenidos

- El TAD Cola de prioridad
- El Heap
- Implementación en C de Heap
- Algoritmo de ordenación HeapSort

Contenidos

- **El TAD Cola de prioridad**
- El Heap
- Implementación en C de Heap
- Algoritmo de ordenación HeapSort

El TAD Cola de prioridad. Definición

- En pilas, colas y listas el orden de sus elementos está determinado por la secuencia de inserciones y extracciones
  - En una **pila** el último elemento insertado es el primero en ser extraído (LIFO)
  - En una **cola** el primer elemento insertado es el primero en ser extraído (FIFO)
- En **colas de prioridad** el orden de sus elementos está determinado por un **valor de prioridad numérico asociado a cada elemento**
  - El elemento de mayor prioridad es el primero en ser extraído, independientemente de cuando fue insertado (siempre que en la cola no haya otros de igual prioridad)
  - Una mayor prioridad puede venir indicada tanto por un valor numérico más alto como por uno más bajo (dependerá de la aplicación)

El TAD Cola de prioridad. Primitivas

- **Primitivas** - las mismas que las del TAD Cola

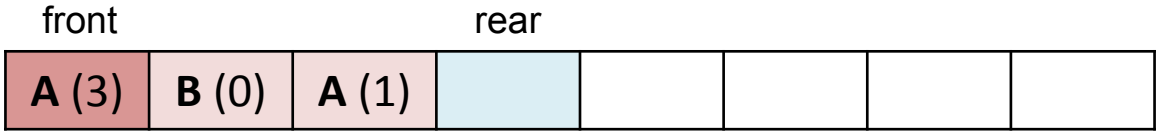
```
ColaPrioridad colaPrioridad_crear()
colaPrioridad_liberar(ColaPrioridad q)

boolean colaPrioridad_vacia(ColaPrioridad q)
boolean colaPrioridad_llena(ColaPrioridad q)

status colaPrioridad_insertar(ColaPrioridad q, Elemento e)
Elemento colaPrioridad_extraer(ColaPrioridad q)
```

- **Particularidad**: las primitivas de inserción y/o extracción tienen en cuenta la prioridad de los elementos
  - Se asume que el TAD Elemento almacena un valor de prioridad y que proporciona primitivas para su acceso y uso, p.e.:

```
elemento_setPrioridad(Elemento e, entero prioridad)
entero elemento_getPrioridad(Elemento e)
```

El TAD Cola de prioridad. EdD con array

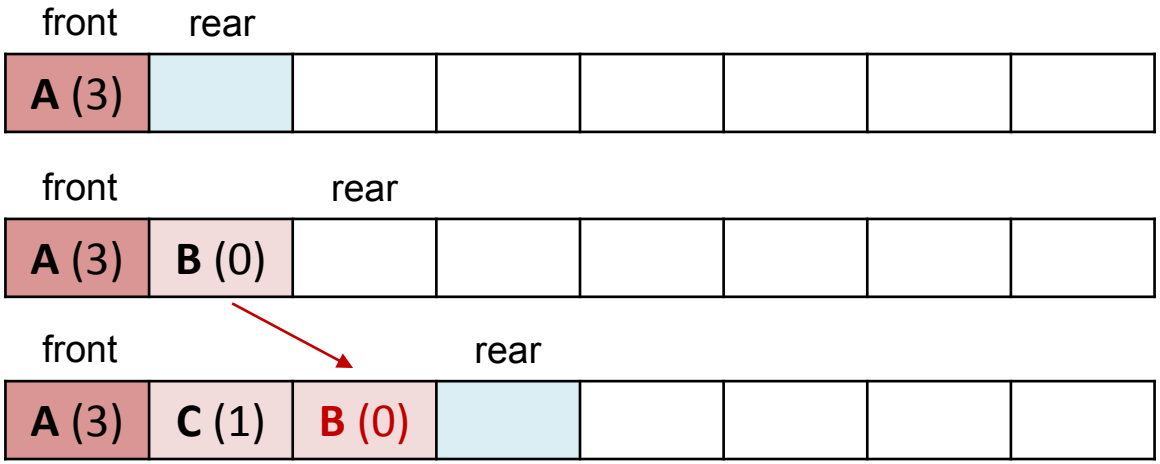
- **Solución 1:** usar la EdD del TAD Cola basada en array, almacenando los elementos no ordenados
  - **Inserción:** inserta un elemento en el rear; no tiene en cuenta las prioridades de los elementos → eficiente
    - Ejemplo: inserción de A (3), B (0), C (1) – prioridades entre paréntesis



  - **Extracción:** como los elementos no están ordenados, busca desde el front al rear aquel elemento con mayor/menor prioridad → ineficiente
    - Implementación 1: mueve el resto de elementos para no dejar posiciones vacías (incluyendo el front y el rear si procede)
    - Implementación 2: marca con un valor especial una posición vacía; cuando la cola está llena, limpia del array las posiciones vacías, recolocando elementos (incluidos el front y el rear si procede)

El TAD Cola de prioridad. EdD con array

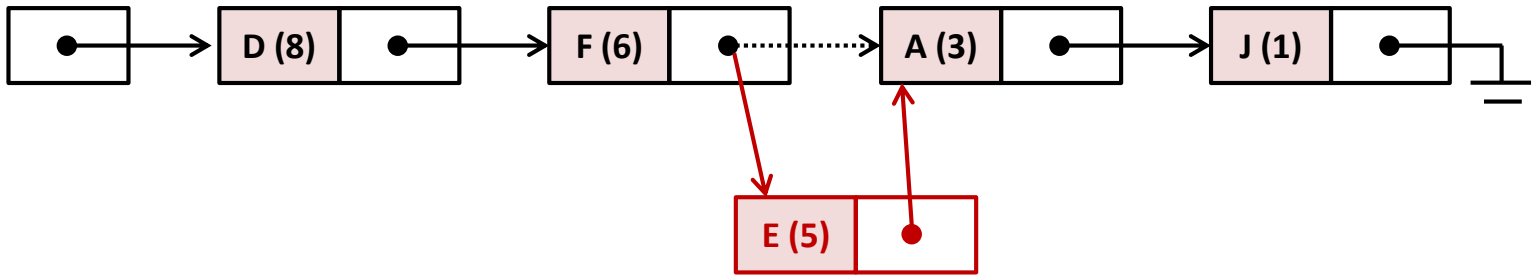
- **Solución 2:** usar la EdD del TAD Cola basada en array, manteniendo los elementos ordenados
  - **Inserción:** mueve los elementos pertinentes a la hora de insertar uno nuevo → ineficiente
    - Ejemplo: inserción de A (3), B (0), C (1) – prioridades entre paréntesis



  - **Extracción:** devuelve el elemento del front → eficiente

El TAD Cola de prioridad. EdD con lista enlazada

- **Solución 3:** usar la EdD del TAD Lista Enlazada, manteniendo los elementos ordenados
  - **Inserción:** recorre los nodos de la lista, comprobando los valores de prioridad de sus elementos, hasta encontrar la posición donde insertar el nuevo → ineficiente (en promedio necesarias N/2 comparaciones, siendo N el número de nodos de la lista)
    - Ejemplo: inserción de E (5) – prioridades entre paréntesis

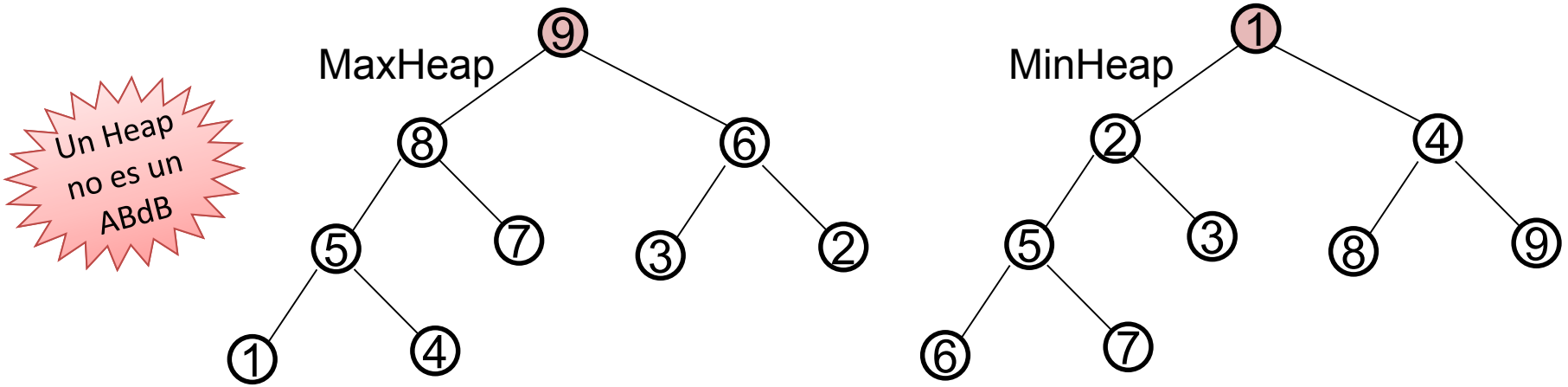


  - **Extracción:** devuelve el elemento del comienzo/fin → eficiente

El TAD Cola de prioridad. EdD con heap

- **Solución definitiva:** usar la EdD de **Heap**, un árbol binario H (estrictamente casi completo) que cumple una condición de orden recursiva:
  - *Orden descendente (MaxHeap)*: la raíz de H tiene un valor de prioridad mayor que la de cualquiera de sus hijos
  - *Orden ascendente (MinHeap)*: la raíz de H árbol tiene un valor de prioridad menor que la de cualquiera de sus hijos

  → **Extracción**: devuelve el elemento de la raíz de H

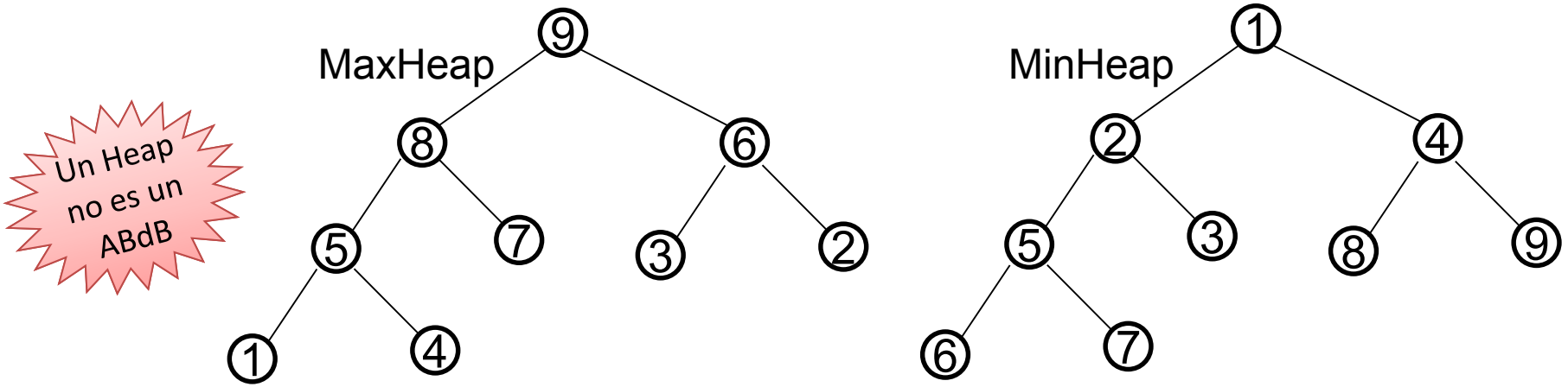


Contenidos

- El TAD Cola de prioridad
- **El Heap**
- Implementación en C de Heap
- Algoritmo de ordenación HeapSort

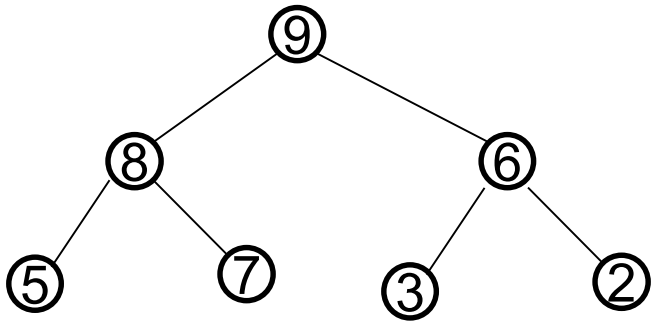
El TAD Heap. Definición

- Un **heap** (montón, montículo) es un árbol binario tal que:
  - Es estrictamente casi completo: todos los niveles excepto tal vez el último están completos, y el último, si no lo estuviese, tiene sus nodos *de izquierda a derecha*
  - Todo nodo **n** cumple la condición de orden:
    - Orden descendiente - MaxHeap: info(**n**) > info(**n'**), ∀**n'** descendiente de **n**
    - Orden ascendente - MinHeap: info(**n**) < info(**n'**), ∀**n'** descendiente de **n**

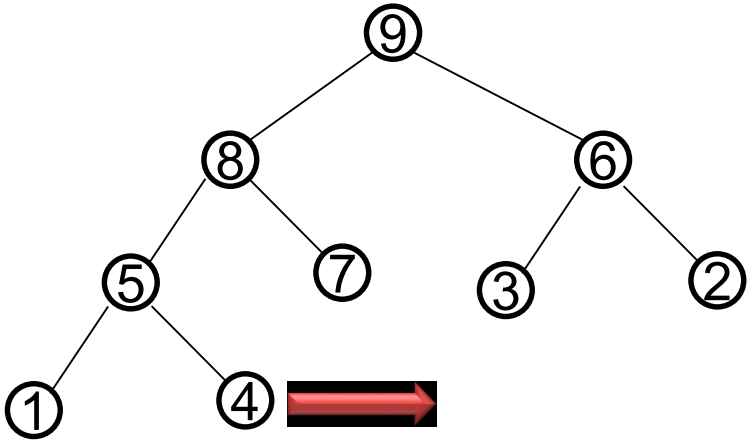


El TAD Heap. Definición

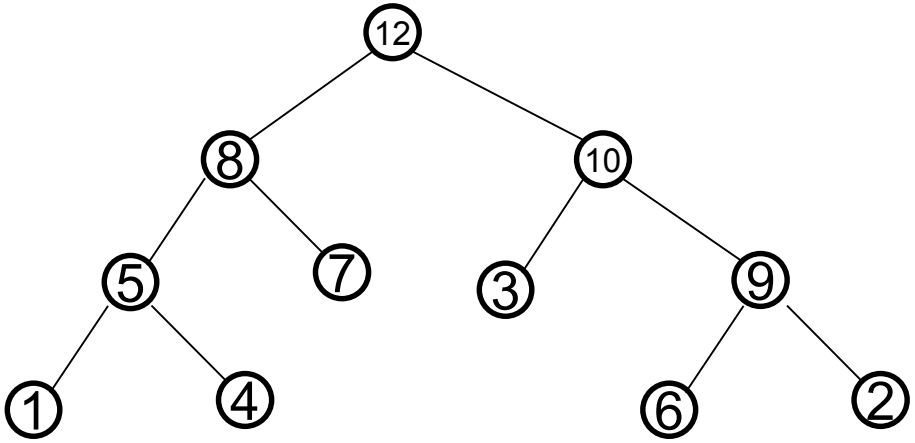
- **Completitud de árboles**



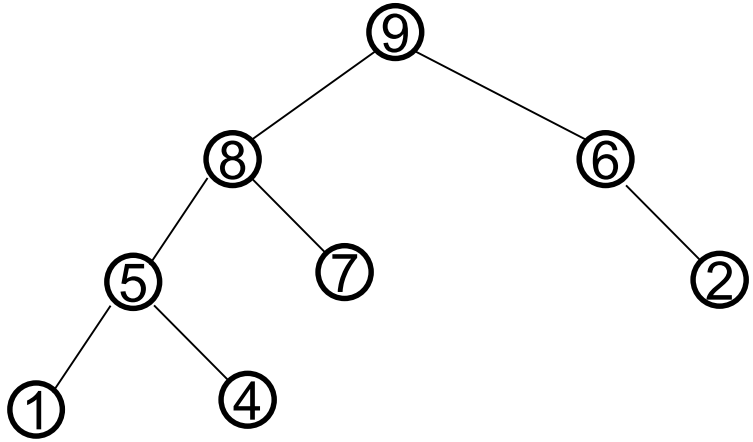
árbol completo



árbol (estrictamente) casi completo



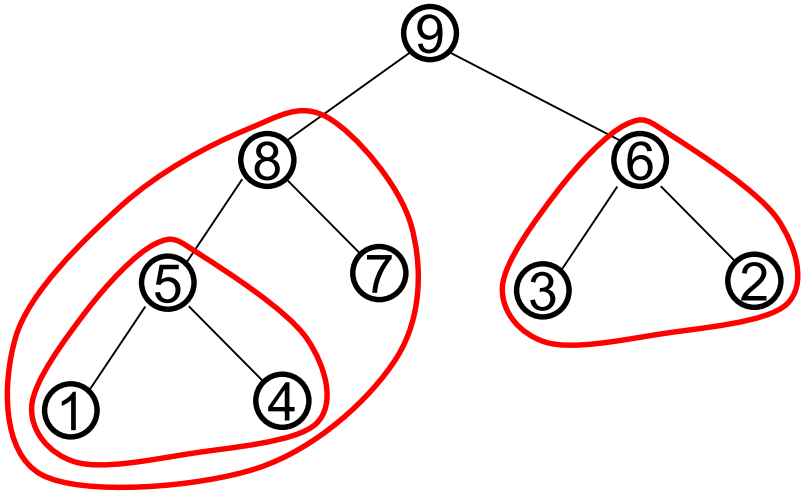
árbol casi completo
(se puede considerar no completo)



árbol no completo

El TAD Heap. Definición

- **Heap** – propiedades
  - Todo sub-árbol de un heap es a su vez un heap



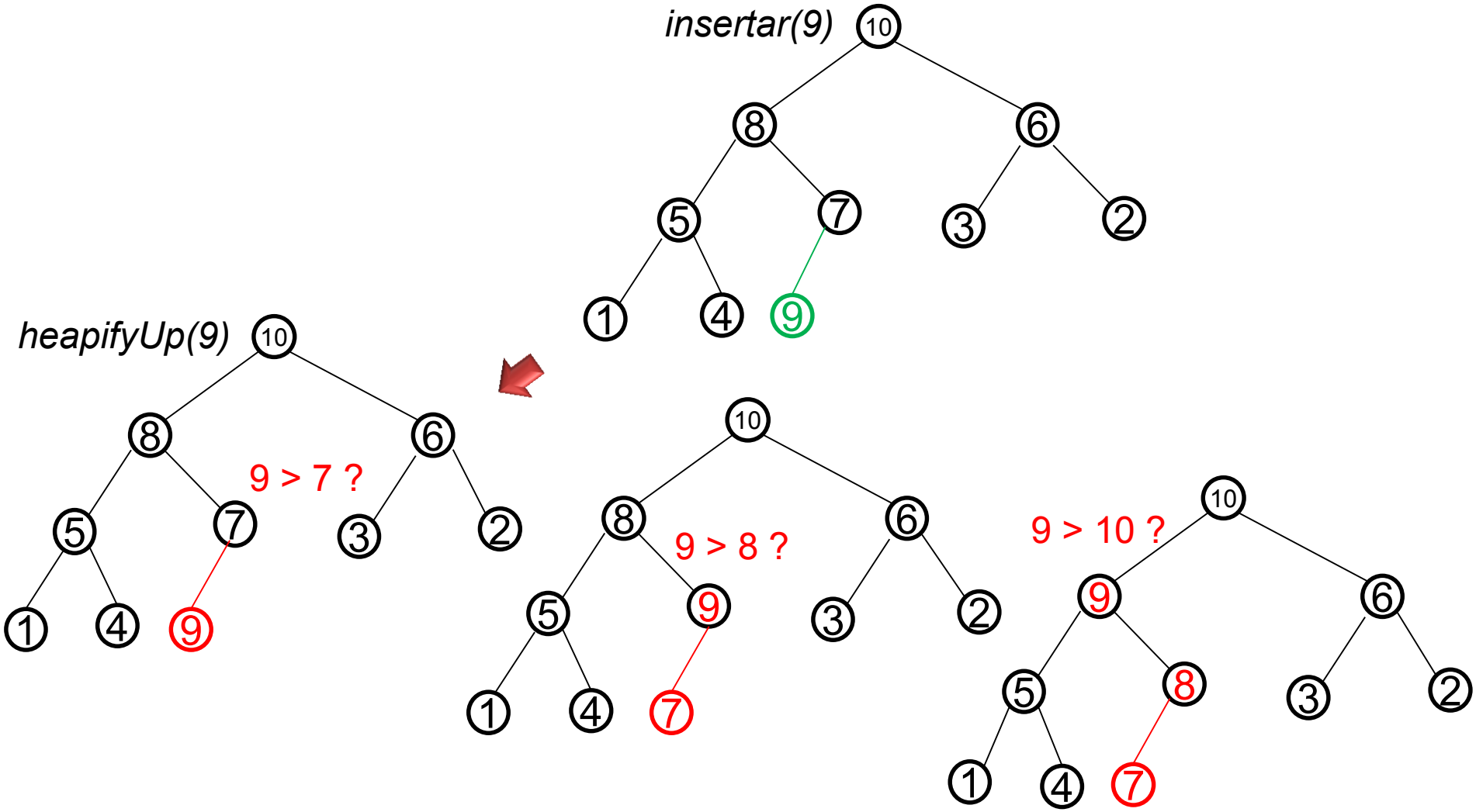
  - En un heap cualquier recorrido desde la raíz hasta una hoja proporciona un vector ordenado de elementos
    - 9 → 4: [9, 8, 5, 4]
    - 9 → 3: [9, 6, 3]

El TAD Heap. Inserción de un elemento

- **Heap** – inserción de un elemento
  1. Se inserta el elemento en un **"último" nodo** para mantener la condición de *árbol estrictamente casi completo*
  2. *heapifyUp*: a partir del elemento insertado, se "recoloca" el heap para mantener la condición de MaxHeap (o MinHeap)
     - de forma iterativa el elemento se compara con su nodo padre, y si no se cumple la condición de MaxHeap, se hace un swap entre el elemento y el padre

El TAD Heap. Inserción de un elemento

- **Heap** – inserción de un elemento

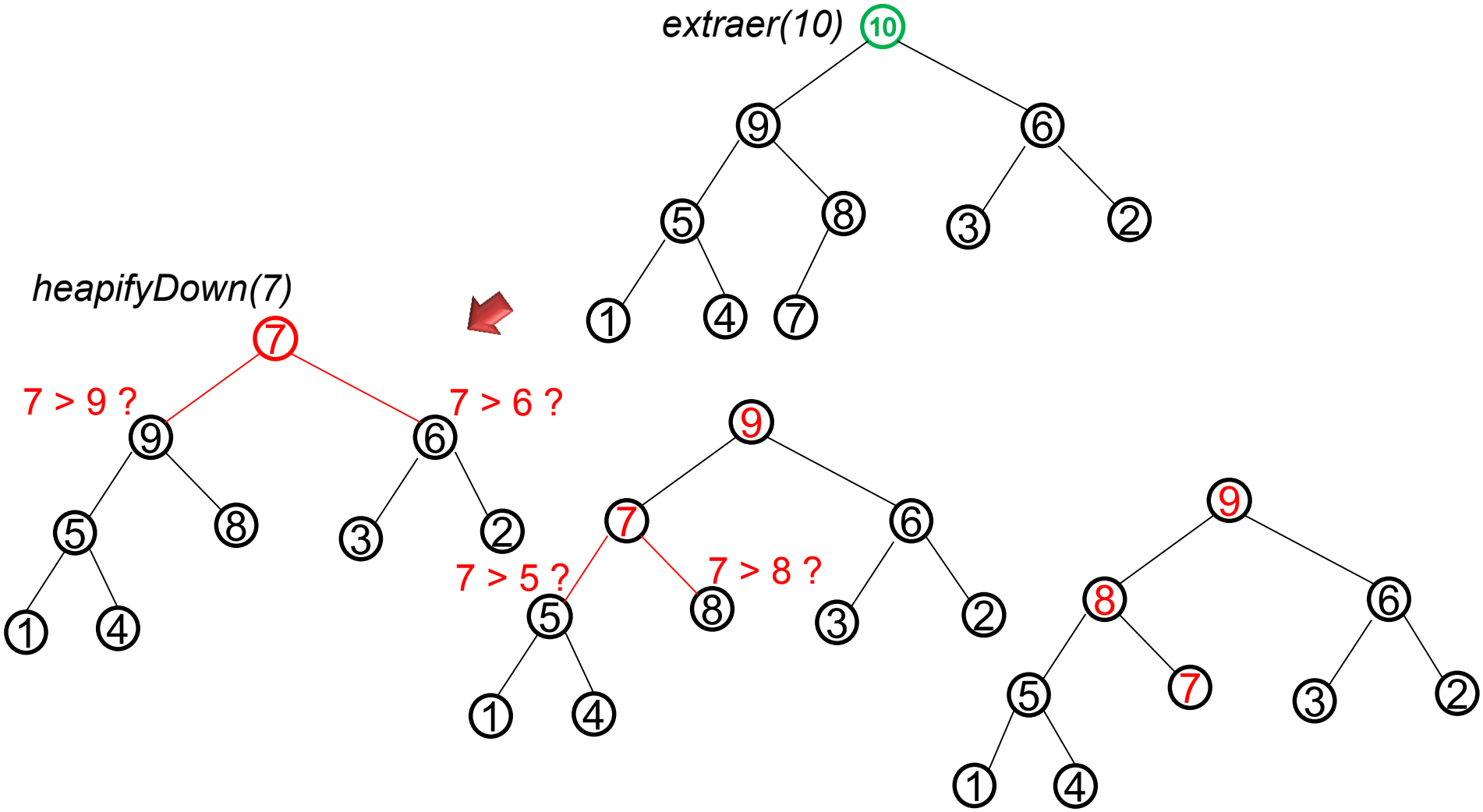


El TAD Heap. Extracción del elemento raíz

- **Heap** – extracción del elemento raíz
  1. Se guarda el elemento del **nodo raíz**
  2. Se extrae el elemento de más a la derecha del último nivel y se copia en el nodo raíz
  3. *heapifyDown*: a partir del elemento raíz, se "recoloca" el heap para mantener la condición de MaxHeap (o MinHeap)
     - de forma iterativa el elemento se compara con sus hijos, y si no se cumple la condición de MaxHeap (o MinHeap), se hace un swap entre el elemento y el hijo con elemento de mayor (o menor) valor

El TAD Heap. Extracción del elemento raíz

- **Heap** – extracción del elemento raíz

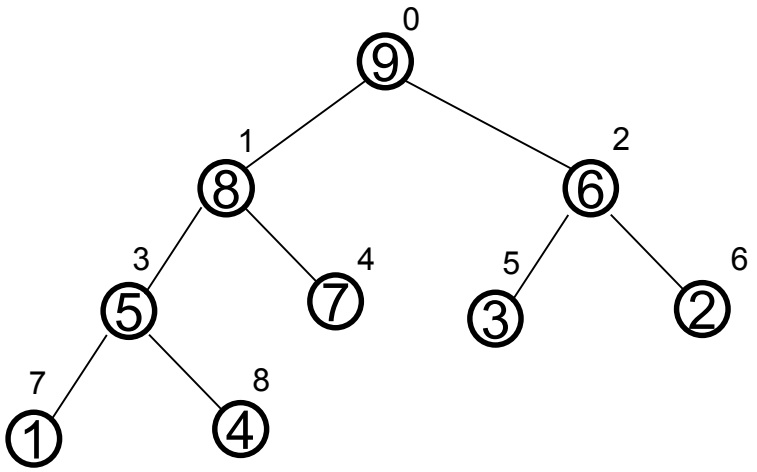


Contenidos

- El TAD Cola de prioridad
- El Heap
- **Implementación en C de Heap**
- Algoritmo de ordenación HeapSort

Implementación en C de Heap. EdD

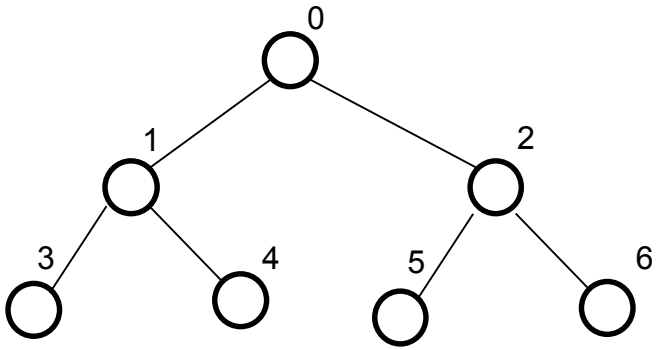
- **Heap** – EdD
  - heap = array en el que los nodos del heap se identifican con los índices del array
    - raiz = T[0]
    - izq(n) = 2n+1
    - der(n) = 2n+2
    - pad(n) = ⌊(n-1)/2⌋

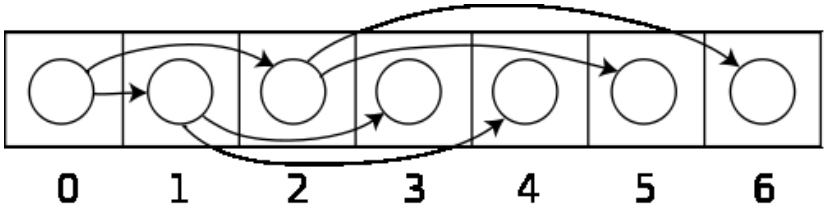


9	8	6	5	7	3	2	1	4	
0	1	2	3	4	5	6	7	8	

Implementación en C de Heap. EdD

- **Heap** – EdD
  - izq(p) = (2 * (p) + 1) ≡ **2p + 1**
  - der(p) = (2 * (p) + 2) ≡ **2p + 2**
  - pad(p) = ((int) (((p) - 1) / 2)) ≡ **floor((p – 1) / 2)**

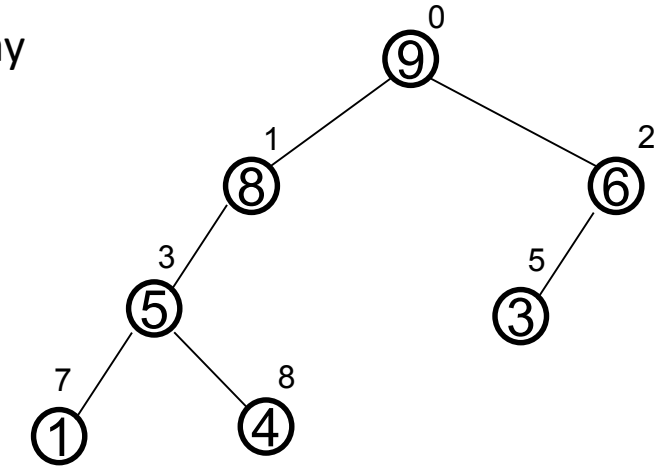




Implementación en C de Heap. EdD

- En árboles no completos
  - Representación ineficiente: huecos en el array

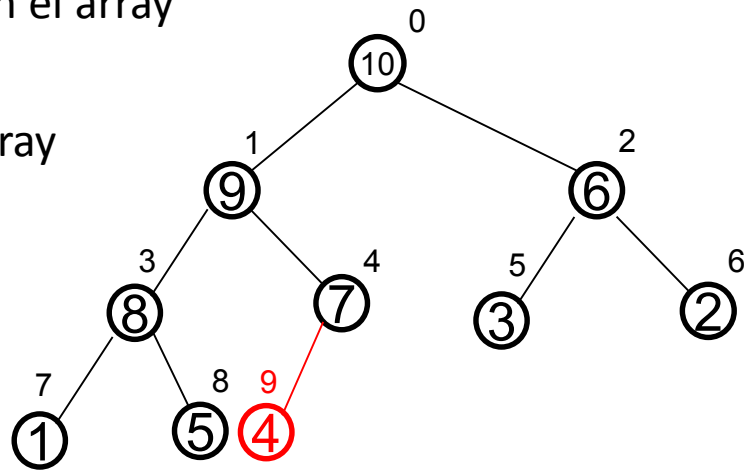
9	8	6	5		3		1	4	
0	1	2	3	4	5	6	7	8	



- En árboles completos o estrictamente casi completos
  - Los elementos se colocan de forma secuencial en el array
  - Añadir un elemento al árbol

    = asignar un valor en el primer índice libre del array

10	9	6	8	7	3	2	1	5	4
0	1	2	3	4	5	6	7	8	9



Implementación en C de Heap. EdD

- **Uso de un array como EdD para Heap**
  - **Ventajas**
    - Simplificación de la representación y manejo de los datos: no es necesaria la gestión dinámica de instancias de la estructura Nodo
    - No desperdicio de memoria, porque el árbol es estrictamente casi completo
    - Uso eficaz de la (reserva de) memoria cuando se conoce el número de elementos máximo que se va a manejar
  - **Inconvenientes**
    - Desperdicio de memoria si el árbol es no completo (pocos nodos en el último nivel)
    - Posible necesidad de reservas de memoria adicionales para el array si no se conoce el número de elementos máximo que se va a manejar

Implementación en C de Heap. EdD

- **Heap** – EdD

```
// heap.h
#ifndef HEAP_H
#define HEAP_H

#include "elemento.h"
#include "tipos.h"

typedef struct _Heap Heap;

Heap *heap_crear();
void heap_liberar(Heap *ph);
boolean heap_vacio(const Heap *ph);
boolean heap_lleno(const Heap *ph);
status heap_insertar(Heap *ph, Elemento* pe);
Elemento *heap_extraer(Heap *ph);

#endif
```

Implementación en C de Heap. EdD

- **Heap** – EdD

```
// heap.c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "heap.h"
#include "elemento.h"
#include "tipos.h"

#define HEAP_MAX 100

#define izq(n) (2*(n)+1)
#define der(n) (2*(n)+2)
#define pad(n) (floor(((n)-1)/2))

struct _Heap {
   Elemento *nodos[HEAP_MAX];   // Array estático: dinámico sería mejor
   int numNodos;
};

Heap *heap_crear() { ... }
void heap_liberar(Heap *ph) { ... }
boolean heap_vacio(const Heap *ph) { ... }
boolean heap_lleno(const Heap *ph) { ... }
status heap_insertar(Heap *ph, Elemento* pe) { ... }
Elemento *heap_extraer(Heap *ph) { ... }
```

Implementación en C de Heap. Primitivas

- **Heap** – creación y liberación de un heap

```
Heap *heap_crear() {
    Heap *ph = NULL;

    ph = (Heap *) malloc(sizeof(Heap));
    if (!ph) return NULL;

    ph->numNodos= 0;

    return ph;
}

void heap_liberar(Heap *ph) {
    int i;

    if (ph) {
        for (i=0; i<ph->numNodos; i++) {
            elemento_liberar(ph->nodos[i]);
        }
        free(ph);
    }
}
```

```
struct _Heap {
    Elemento *nodos[HEAP_MAX];
    int numNodos;
};
```

Implementación en C de Heap. Primitivas

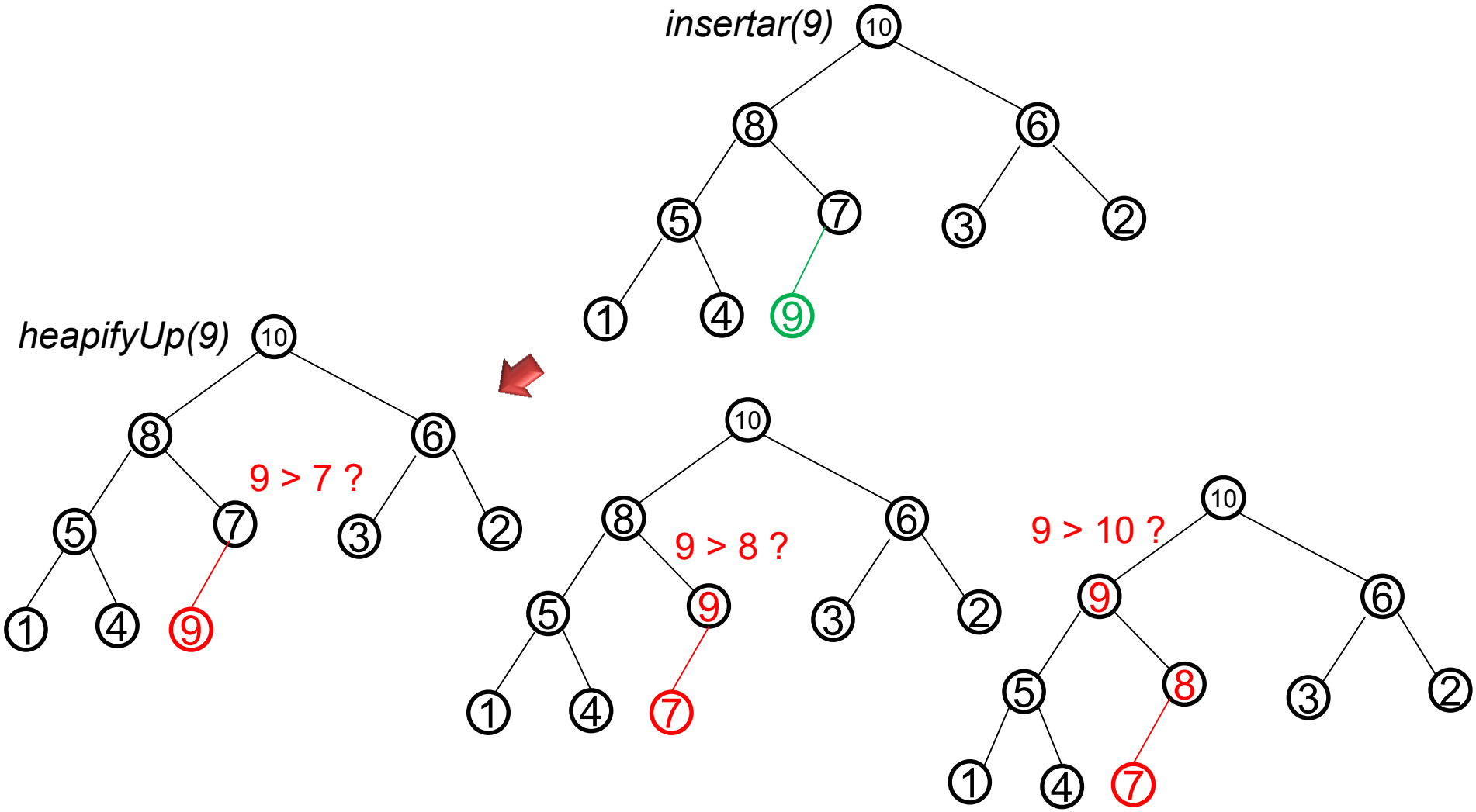
- **Heap** – heap vacío y lleno

```
boolean heap_vacio(const Heap *ph) {
    if (!ph) return TRUE; // Caso de error

    return ph->numNodos == 0 ? TRUE : FALSE;
    // equivale a:
    // if (ph->numNodos == 0) return TRUE; else return FALSE;
}

boolean heap_lleno(const Heap *ph) {
    if (!ph) return TRUE; // Caso de error

    return ph->numNodos == HEAP_MAX ? TRUE : FALSE;
    // equivale a:
    // if (ph->numNodos == HEAP_MAX) return TRUE; else return FALSE;

}
```

```
struct _Heap {
    Elemento *nodos[HEAP_MAX];
    int numNodos;
};
```

Implementación en C de Heap. Primitivas

- **Heap** – inserción de un elemento



Implementación en C de Heap. Primitivas

- Implementar la función **heap_insertar**

Implementación en C de Heap. Primitivas

- Implementar la función **heap_insertar**

```
status heap_insertar(Heap *ph, Elemento *pe);
```

Se tiene:

```
struct _Heap {
   Elemento *nodos[HEAP_MAX];
   int numNodos;
};
```


10	9	6	8	7	3	2	1	5	4
0	1	2	3	4	5	6	7	8	9

```
boolean heap_vacio(const Heap *ph);
boolean heap_lleno(const Heap *ph);

Elemento *elemento_copiar (Elemento *pe);
int elemento_getPrioridad(Elemento *pe);
```

Hay que implementar:

```
void heapifyUp(Heap *ph, int k);
```

Implementación en C de Heap. Primitivas

- **Heap** – inserción de un elemento

```
status heap_insertar(Heap *ph, Elemento *pe) {
   if (!ph || !pe || heap_lleno(ph) == TRUE) return ERROR;

   // Insertamos el dato en la ultima posicion
   ph->nodos[ph->numNodos] = elemento_copiar(pe);
   if (!ph->nodos[ph->numNodos]) return ERROR;

   // Recolocamos el heap: heapify up del ultimo nodo
   heapifyUp(ph, ph->numNodos);

   // Incrementamos el numero de nodos del heap
   ph->numNodos++;

   return OK;
}
```

```
struct _Heap {
   Elemento *nodos[HEAP_MAX];
   int numNodos;
};
```

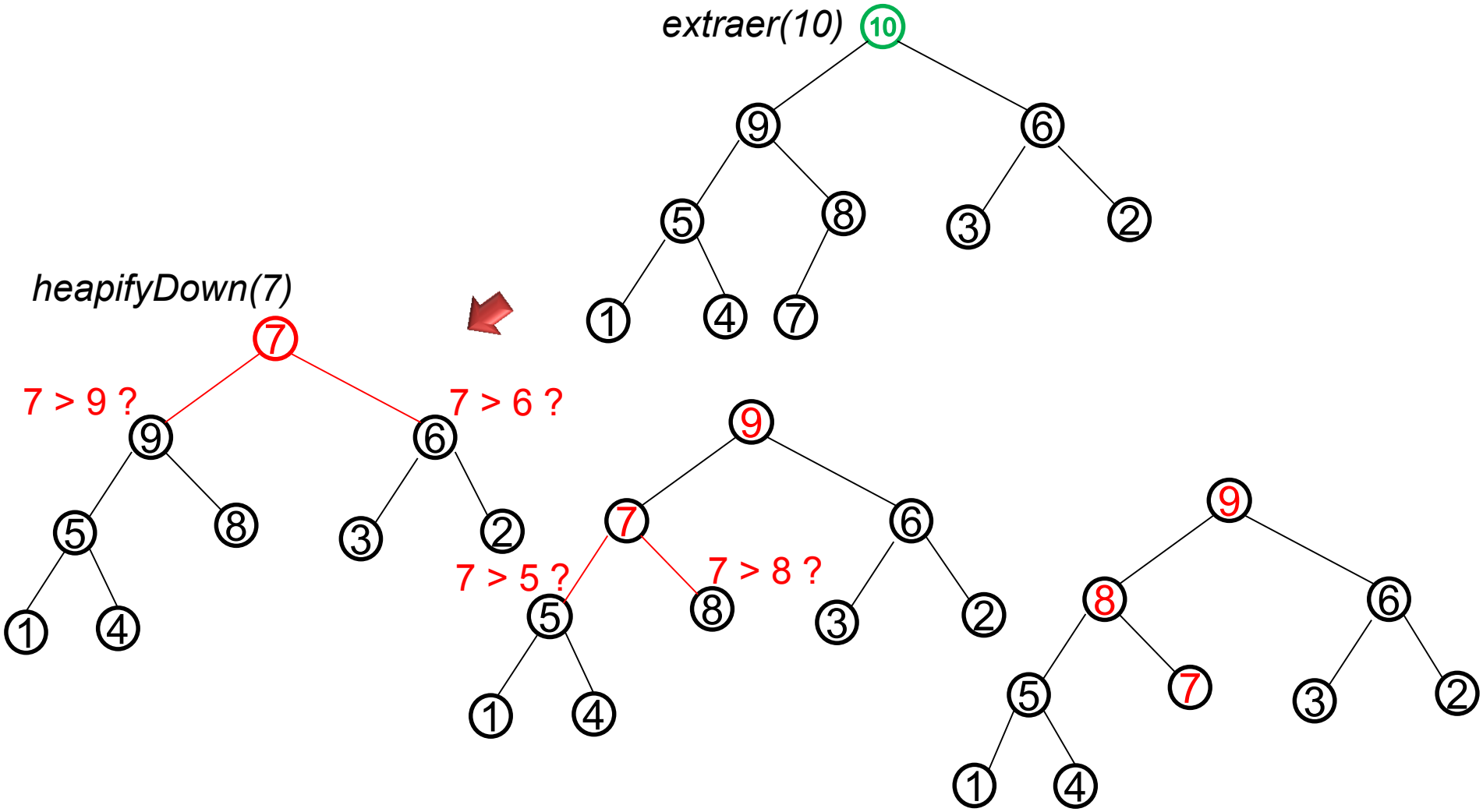
Implementación en C de Heap. Primitivas

- **Heap** – heapify up

```
void heapifyUp(Heap *ph, int k) {  // Funcion privada en heap.c
    Elemento *aux = NULL;
    int p;

    if (!ph || k < 0 || k > ph->numNodos) return;

    p = pad(k);

    // Condición de parada (caso base)
    if (p < 0) return;

    // Llamada recursiva
    if (elemento_prioridad(ph->nodos[k]) > elemento_prioridad(ph->nodos[p])) {
        aux = ph->nodos[p];
        ph->nodos[p] = ph->nodos[k];
        ph->nodos[k] = aux;

        heapifyUp(ph, p); // p = índice desde donde comparar hacia arriba
    }
}
```

Implementación en C de Heap. Primitivas

- **Heap** – Extracción del elemento raíz



Implementación en C de Heap. Primitivas

- Implementar la función **heap_extraer**

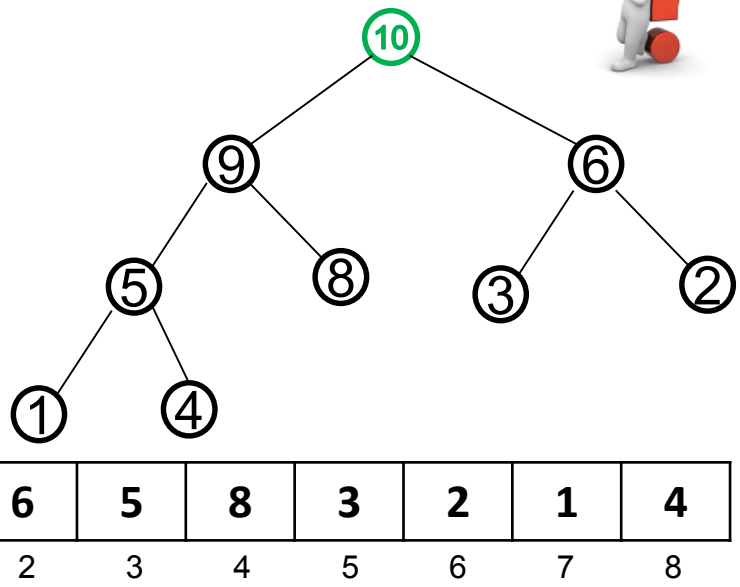
Implementación en C de Heap. Primitivas

- Implementar la función **heap_extraer**

```
Elemento *heap_extraer(Heap *ph);
```

Se tiene:

```
struct _Heap {
   Elemento *nodos[HEAP_MAX];
   int numNodos;
};
```




```
boolean heap_vacio(const Heap *ph);
boolean heap_lleno(const Heap *ph);

Elemento *elemento_copiar (Elemento *pe);
int elemento_getPrioridad(Elemento *pe);
```

Hay que implementar:

```
void heapifyDown(Heap *ph, int k);
```

Implementación en C de Heap. Primitivas

- **Heap** – Extracción del elemento raíz

```
Elemento *heap_extraer(Heap *ph) {
    Elemento *pe = NULL;

    if (!ph || heap_vacio(ph) == TRUE) return NULL;

    // Guardamos el elemento raiz
    pe = ph->nodos[0];

    // Movemos el ultimo elemento a la raiz
    ph->nodos[0] = ph->nodos[ph->numNodos-1];

    ph->nodos[ph->numNodos-1] = NULL;

    // Decrementamos el numero de nodos
    ph->numNodos--;

    // Recolocamos el heap: heapify down de la raiz
    heapifyDown(ph, 0);

    return pe;
}
```

```
struct _Heap {
    Elemento *nodos[HEAP_MAX];
    int numNodos;
};
```

Implementación en C de Heap. Primitivas

- **Heap** – heapify down

```
void heapifyDown(Heap *ph, int k) {  // Funcion privada en heap.c
    int izq, der, max;
    Elemento *aux = NULL;

    if (!ph || k < 0 || k >= ph->numNodos) return;

    izq = izq(k);
    der = der(k);

    // Obtenemos el maximo de k, izq(k) y der(k)
    if (izq < ph->numNodos &&
        elemento_getPrioridad(ph->nodos[k]) < elemento_getPrioridad(ph->nodos[izq])) {
        max = izq;
    } else {
        max = k;
    }
    if (der < ph->numNodos &&
        elemento_getPrioridad(ph->nodos[max])< elemento_getPrioridad(ph->nodos[der])) {
        max= der;
    }
    // Swap entre k y max, y llamada recursive a heapify
    if (max != k) {  // Condicion de parada
        aux = ph->nodos[max];
        ph->nodos[max] = ph->nodos[k];
        ph->nodos[k] = aux;
        return heapifyDown(ph, max);
    }
}
```

El TAD Cola de prioridad. Heap

- **Cola de prioridad = Heap**
  - Una **cola de prioridad** almacena una serie de elementos ordenados (descendientemente) por su prioridad
    - La extracción en una cola de prioridad devuelve el elemento de mayor prioridad
  - Un **heap** puede almacenar una serie de elementos con prioridades asignadas, si estas prioridades se utilizan para satisfacer la condición de MaxHeap
    - La extracción en un heap devuelve el elemento del nodo raíz, i.e. el de mayor prioridad

El TAD Cola de prioridad. Implementación en C

- **Cola de prioridad** – implementación en C

```
typedef struct _Heap ColaPrioridad;

status colaPrioridad_crear(ColaPrioridad *pq) {
    return heap_crear();
}

void colaPrioridad_liberar(ColaPrioridad *pq) {
    heap_liberar(pq);
}

boolean colaPrioridad_vacia(ColaPrioridad *pq) {
    return heap_vacio(pq);
}

boolean colaPrioridad_llena(ColaPrioridad *pq) {
    return heap_lleno(pq);
}

boolean colaPrioridad_insertar(ColaPrioridad *pq, Elemento *pe) {
    return heap_insertar(pq, pe);
}

Elemento *colaPrioridad_extraer(ColaPrioridad *pq) {
    return heap_extraer(pq);
}
```

Contenidos

- El TAD Cola de prioridad
- El Heap
- Implementación en C de Heap
- **Algoritmo de ordenación HeapSort**

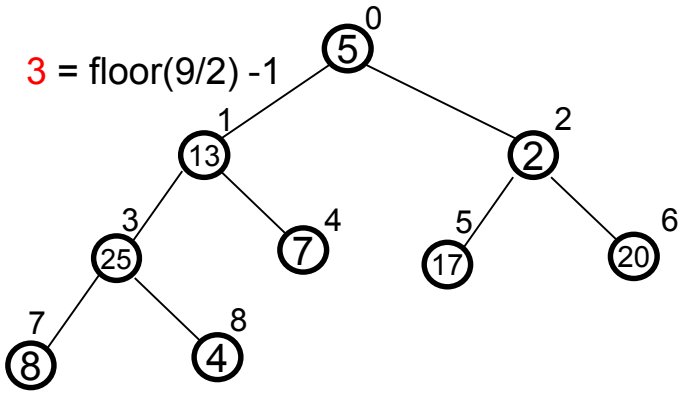
Algoritmo de ordenación HeapSort

- **HeapSort**
  - Algoritmo que ordena una lista de elementos haciendo uso de un heap
  - Consta de 2 fases
- **Fase 1: construcción del heap *(versión in-place)***
  - Crea un heap a partir de la lista de elementos desordenada sin necesidad de llamar a heap_insertar para cada uno de ellos
  - Consiste en una sucesión de llamadas a heapifyDown sobre los nodos (excepto las hojas) del heap, desde “el último padre” (índice: floor(N/2) -1)
- **Fase 2: extracción iterativa de los elementos del heap**
  - Ordena una lista de elementos a través de un heap
  - Crea un heap a partir de la lista de entrada mediante buildHeap
  - Mientras el heap no esté vacío, extrae de él un elemento (la raíz) y lo inserta en la lista (desde el final hasta el principio)

HeapSort. Ejemplo

- **HeapSort** – fase 1: construcción de heap *(in-place)*

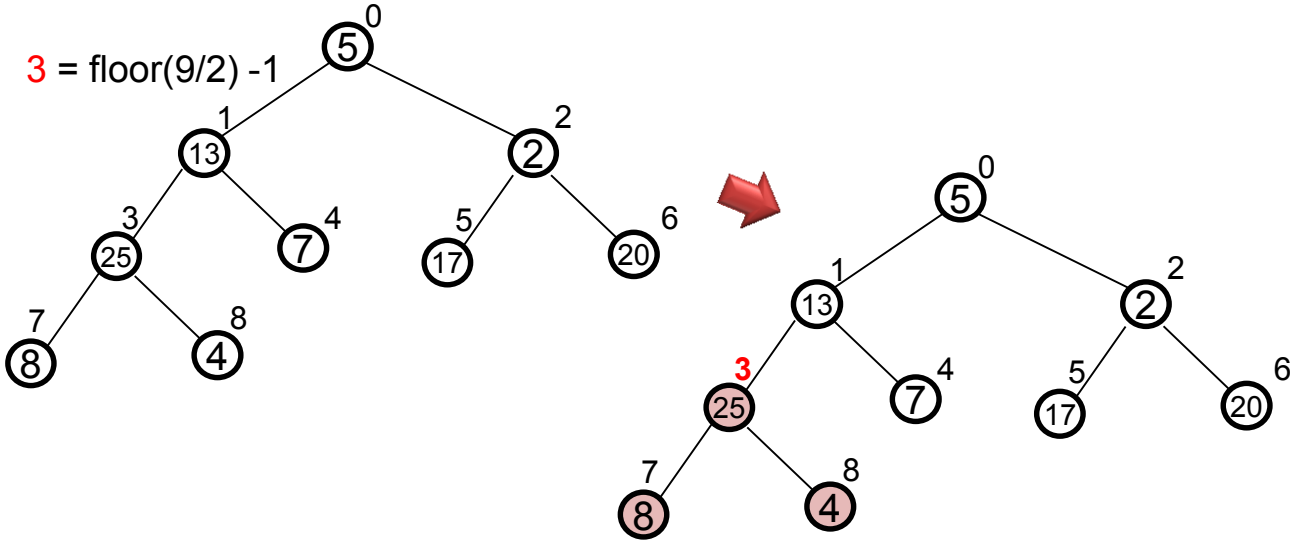
HeapSort	5	13	2	25	7	17	20	8	4
	0	1	2	3	4	5	6	7	8



HeapSort. Ejemplo

- **HeapSort** – fase 1: construcción de heap *(in-place)*

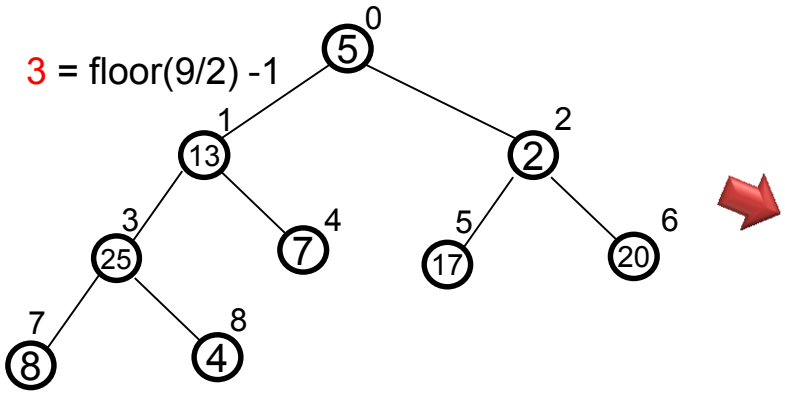
HeapSort	5	13	2	25	7	17	20	8	4
	0	1	2	3	4	5	6	7	8

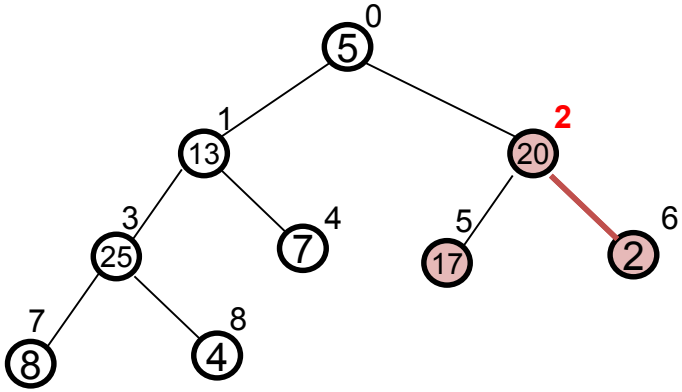


HeapSort. Ejemplo

- **HeapSort** – fase 1: construcción de heap *(in-place)*

HeapSort	5	13	2	25	7	17	20	8	4
	0	1	2	3	4	5	6	7	8

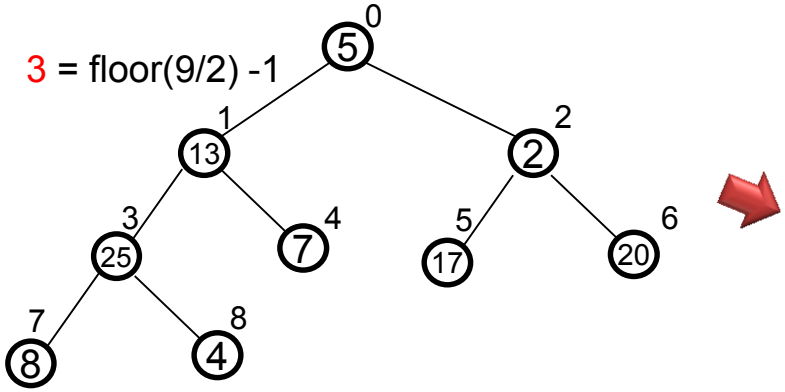


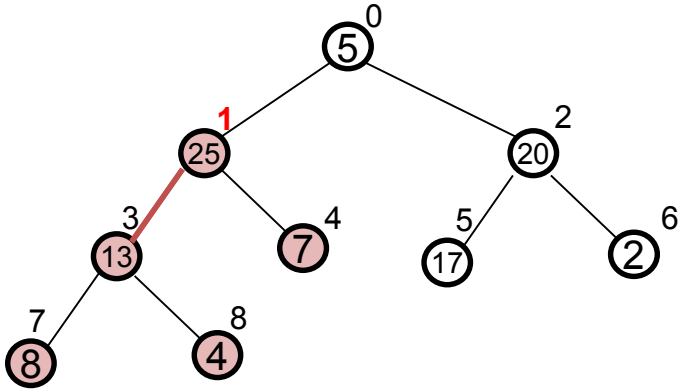


HeapSort. Ejemplo

- **HeapSort** – fase 1: construcción de heap *(in-place)*

HeapSort	5	13	2	25	7	17	20	8	4
	0	1	2	3	4	5	6	7	8

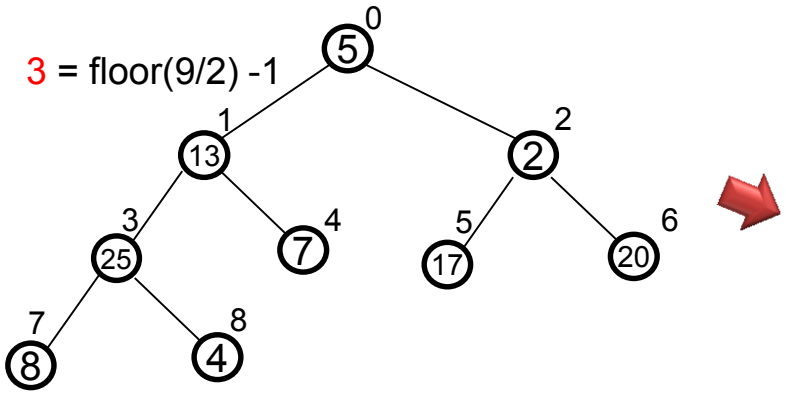


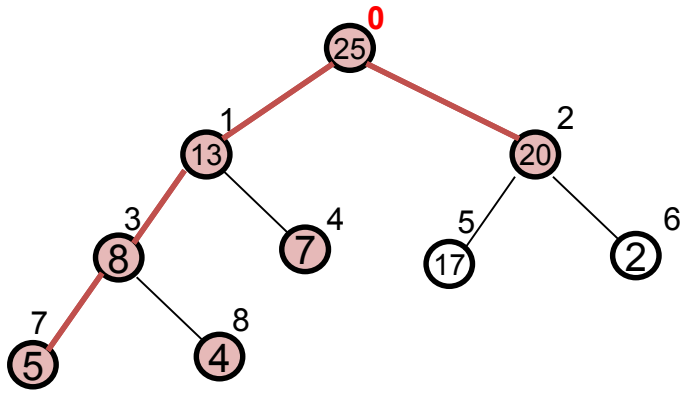


HeapSort. Ejemplo

- **HeapSort** – fase 1: construcción de heap *(in-place)*

HeapSort	5	13	2	25	7	17	20	8	4
	0	1	2	3	4	5	6	7	8





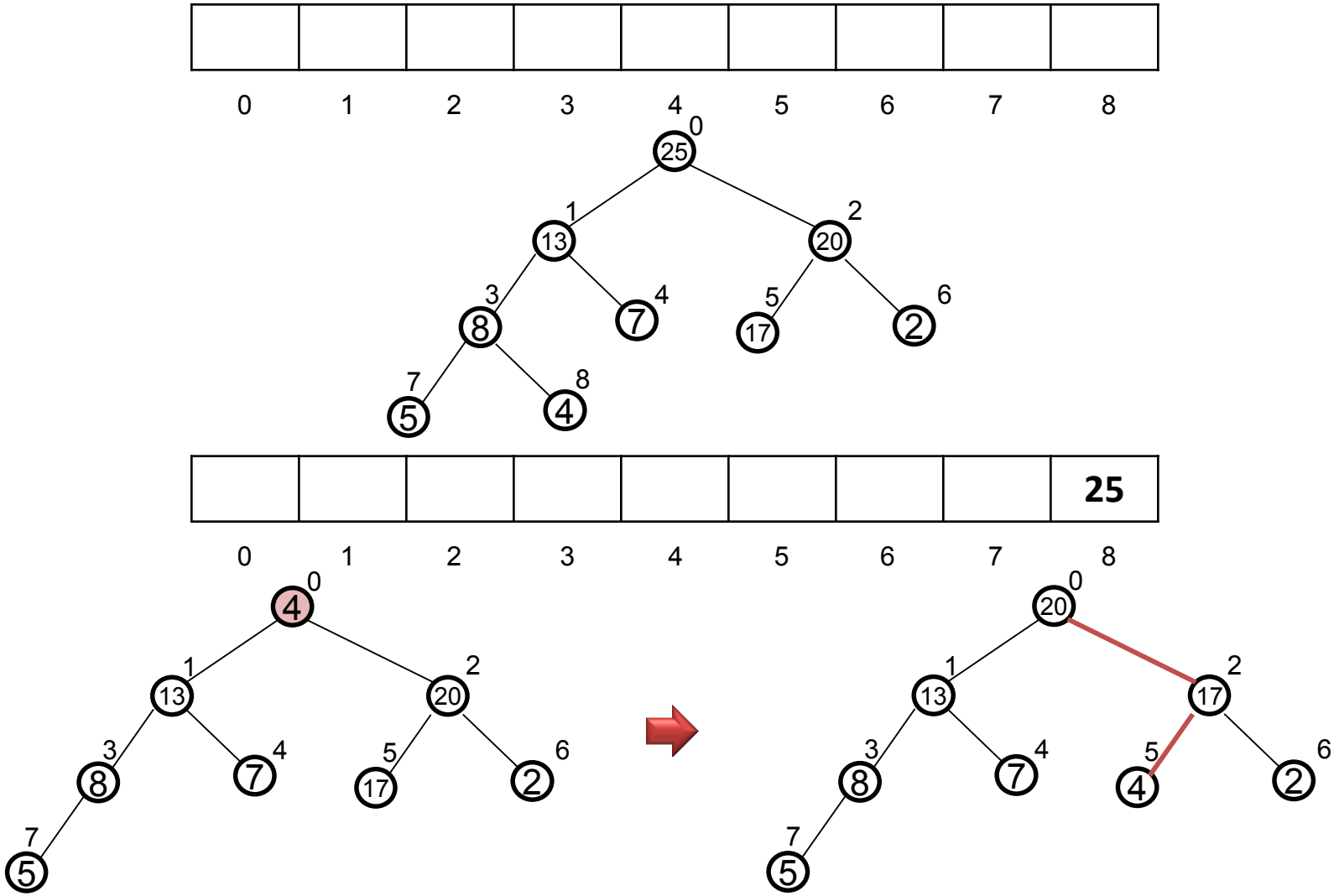
Implementación en C de Heap. Primitivas

- Versión no in-place: ¿Cuál sería la evolución de la fase de construcción del heap insertando en él elemento a elemento de la lista?

5	13	2	25	7	17	20	8	4
0	1	2	3	4	5	6	7	8

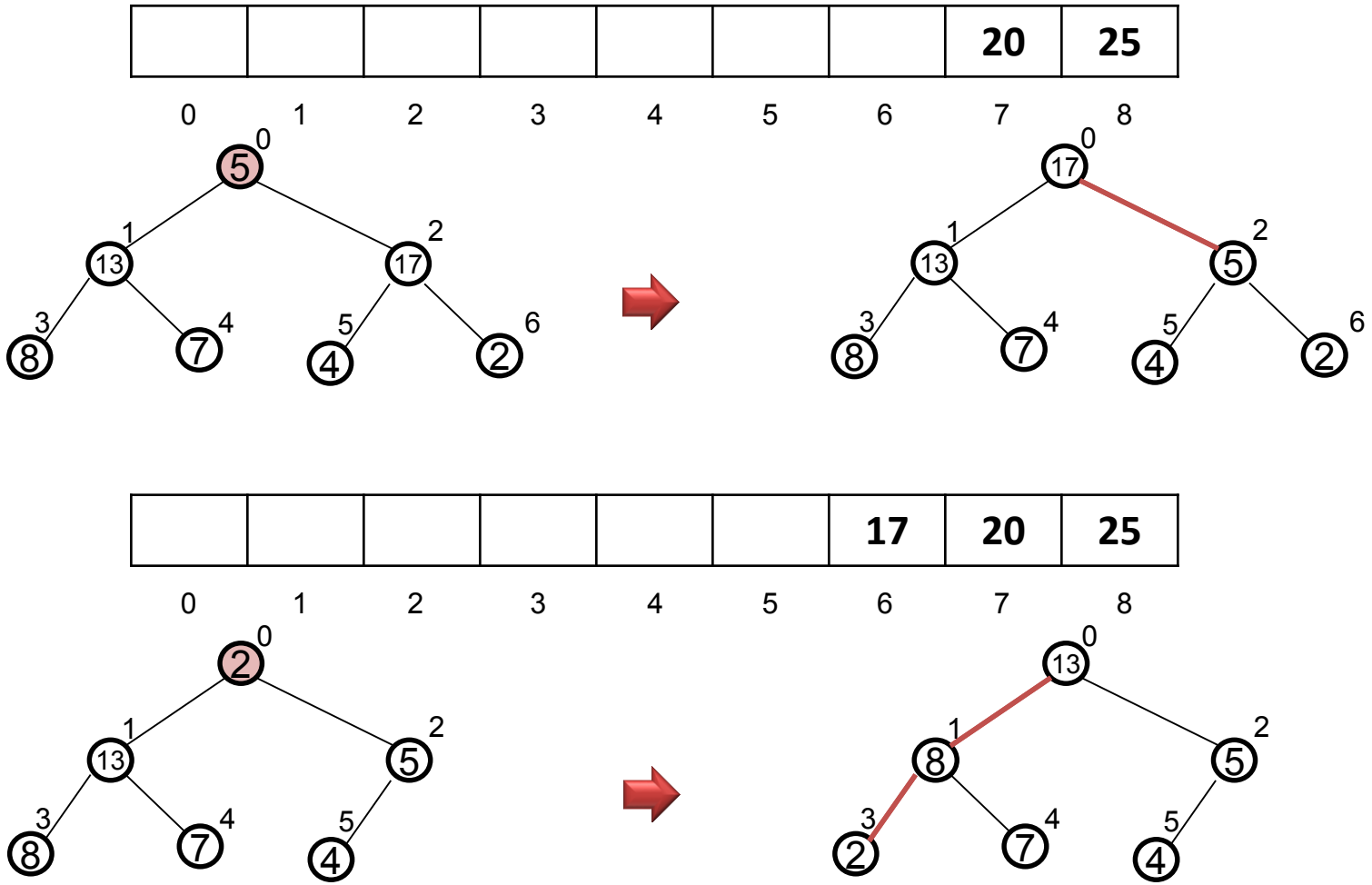
HeapSort. Ejemplo

- **HeapSort** – fase 2: extracción y ordenación de elementos



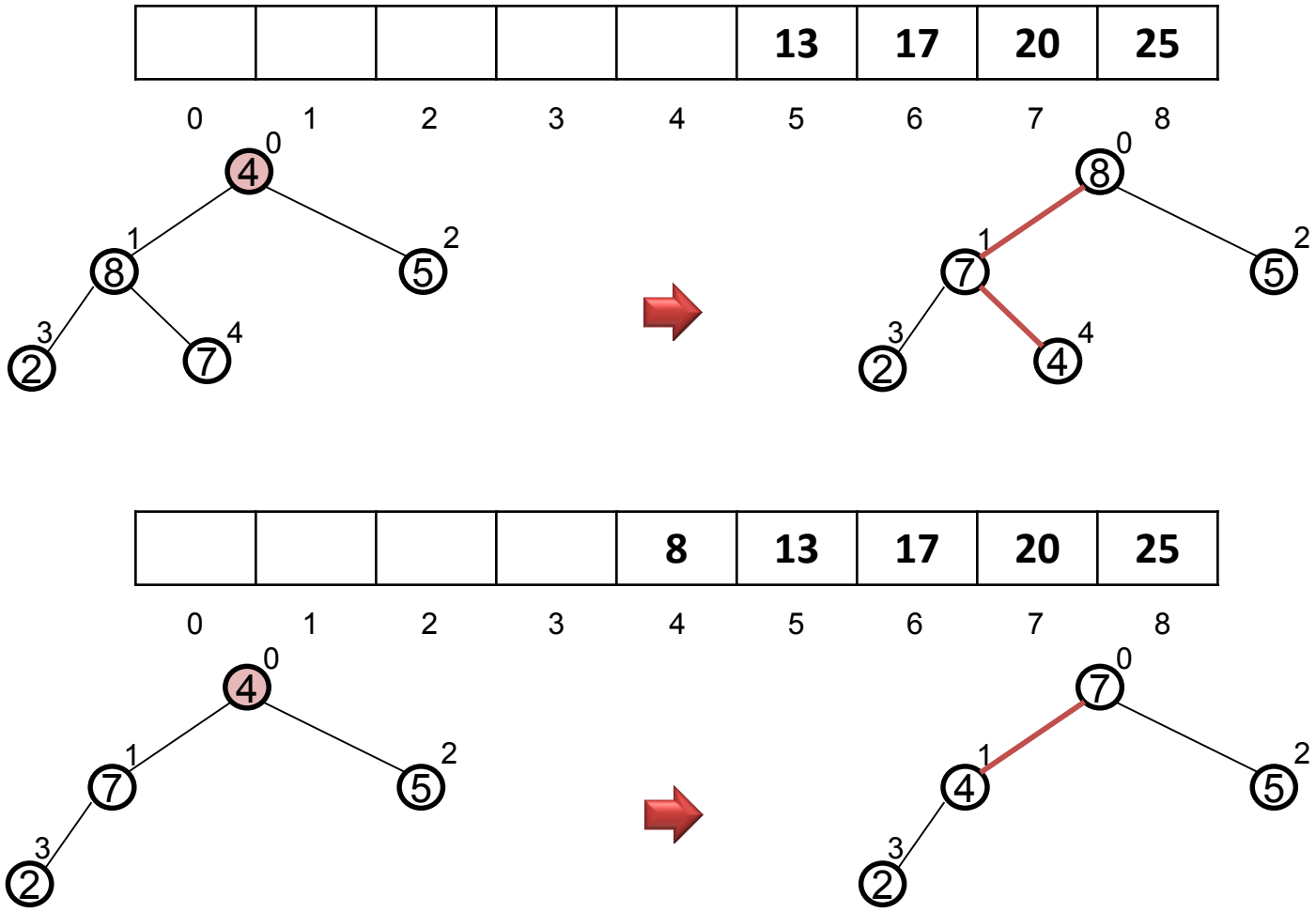
HeapSort. Ejemplo

- **HeapSort** – fase 2: extracción y ordenación de elementos



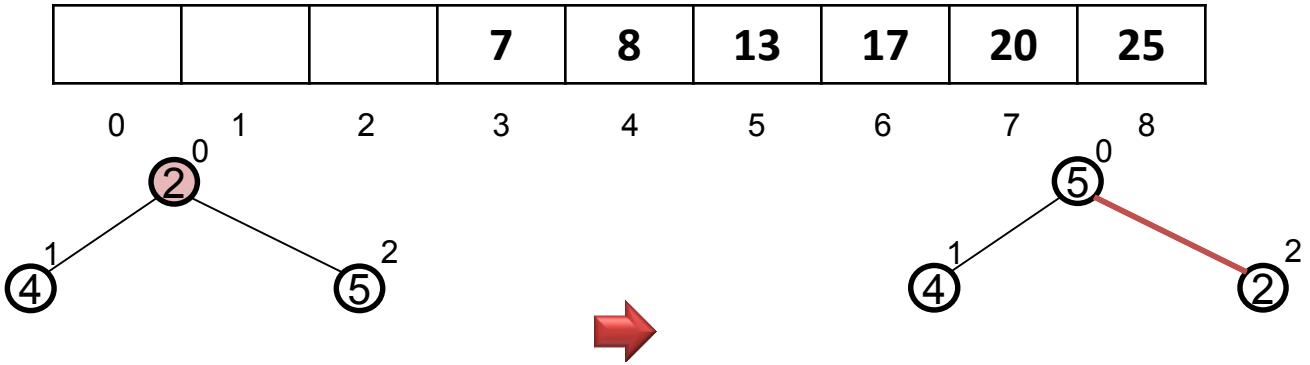
HeapSort. Ejemplo

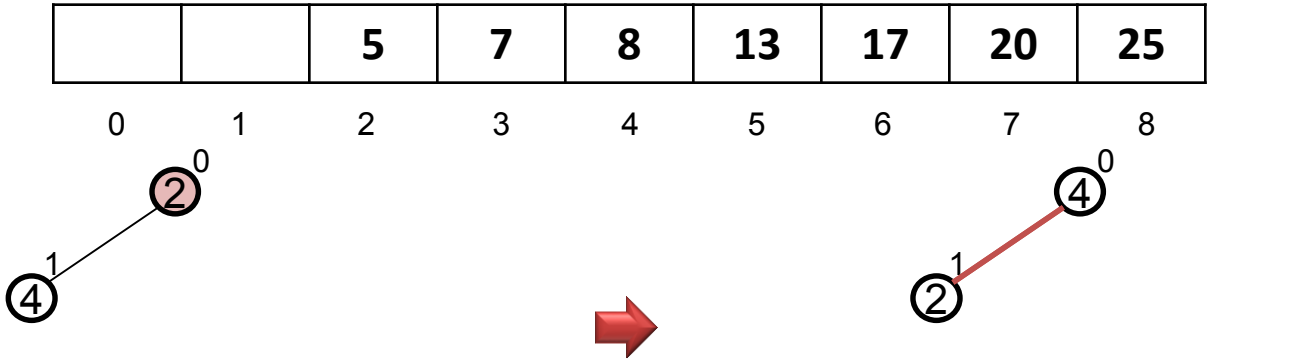
- **HeapSort** – fase 2: extracción y ordenación de elementos



HeapSort. Ejemplo

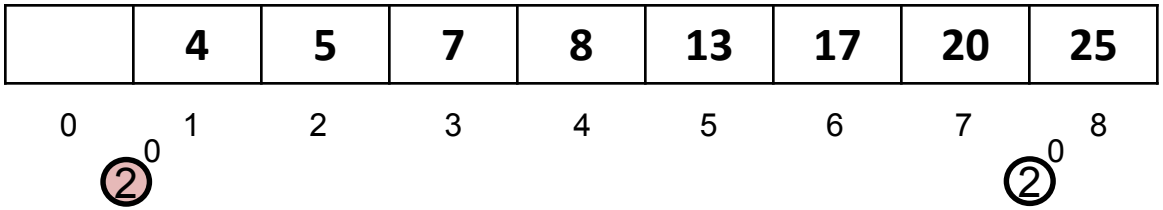
- **HeapSort** – fase 2: extracción y ordenación de elementos

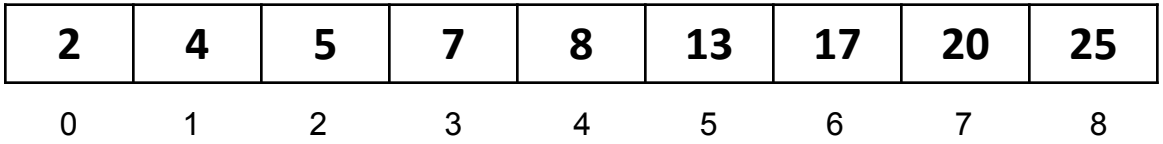




HeapSort. Ejemplo

- **HeapSort** – fase 2: extracción y ordenación de elementos





HeapSort. Construcción del heap

- **HeapSort** – implementación de buildHeap (*versión in-place*)
  - Crea un heap a partir de la lista de elementos desordenada sin necesidad de llamar a heap_insertar para cada uno de ellos
  - Consiste en una sucesión de llamadas a heapifyDown sobres los nodos (excepto las hojas) del heap

```
Heap *buildHeap(Elemento **lista, int numElementos) {
    Heap *ph = NULL;
    int i;

    if (!lista || numElementos < 0 || numElementos > HEAP_MAX)
        return NULL;

    ph = heap_crear();                          // Ojo: falta comprobar errores
    for (i=0; i<numElementos; i++ )
        ph->nodos[i] = elemento_copiar(lista[i]);
    ph->numNodos = numElementos;

    for (i=ph->floor(ph->numNodos/2)-1; i >= 0; i--)
        heapifyDown(ph, i);

    return ph;
}
```

HeapSort. Ordenación a partir del heap

- **HeapSort**
  - Ordena una lista de elementos a través de un heap
  - Crea un heap a partir de la lista de entrada mediante buildHeap
  - Mientras el heap no esté vacío, extrae de él un elemento (la raíz) y lo inserta en la lista (desde el final hasta el principio)

```
Elemento **heapSort(Elemento **lista, int numElementos) {
    Elemento **listaOrdenada = NULL;
    Heap *ph = NULL;
    int i;

    if (!lista || numElementos <= 0 || numElementos > HEAP_MAX) return NULL;

    listaOrdenada = (Elemento **) malloc(numElementos*sizeof(Elemento *));
    if (!listaOrdenada) return NULL;

    ph = buildHeap(lista, numElementos);          // Ojo: falta comprobar errores
    for (i=numElementos-1; i >= 0; i--) {
        listaOrdenada[i] = heap_extraer(ph);
    }
    return listaOrdenada;
}
```

HeapSort. Complejidad

- **HeapSort** – complejidad
  - buildHeap: N/2 llamadas a heapifyDown

$$N/2 \cdot O(t_{heapify})$$

  - extracción + ordenación: N llamadas a heapifyDown

$$N \cdot O(t_{heapify})$$

  - $t_{heapify}$: #comparaciones de clave ≤ profundidad del heap

$$O(\log N)$$

  → heapSort

$$N/2 \cdot O(\log N) + N \cdot O(\log N)$$

$$= O(N) \cdot O(\log N)$$

$$= \mathbf{O(N \log N)}$$

HeapSort. Complejidad

- **Eficiencia de HeapSort**
  - Ordenación con BubbleSort, InsertSort, SelectSort: $O(N^2)$
  - Ordenación con ABdB: O(N log N)
  - Ordenación con HeapSort: O(N log N)
    - Ventajas sobre ABdB:
      - Versión in-place: no es necesario crear una estructura compleja de árbol, ni pedir memoria adicional para nodos
      - No tiene el problema de árboles desbalanceados, en los que coste de ordenación en el caso peor es $O(N^2)$
  - ¿Es HeapSort el mejor método de ordenación?
    - No; QuickSort es mejor: O(N log N)
  - Los métodos de ordenación y su complejidad computacional se estudiarán en la asignatura de Análisis de Algoritmos