Programación II

Tema 6. Colas de prioridad y Heaps

Rosa M. Carro, Ruth Cobos, Eduardo Serrano
Escuela Politécnica Superior
Universidad Autónoma de Madrid

Contenidos

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

Contenidos

- **El TAD Cola de prioridad**
- El Heap
- Implementación 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

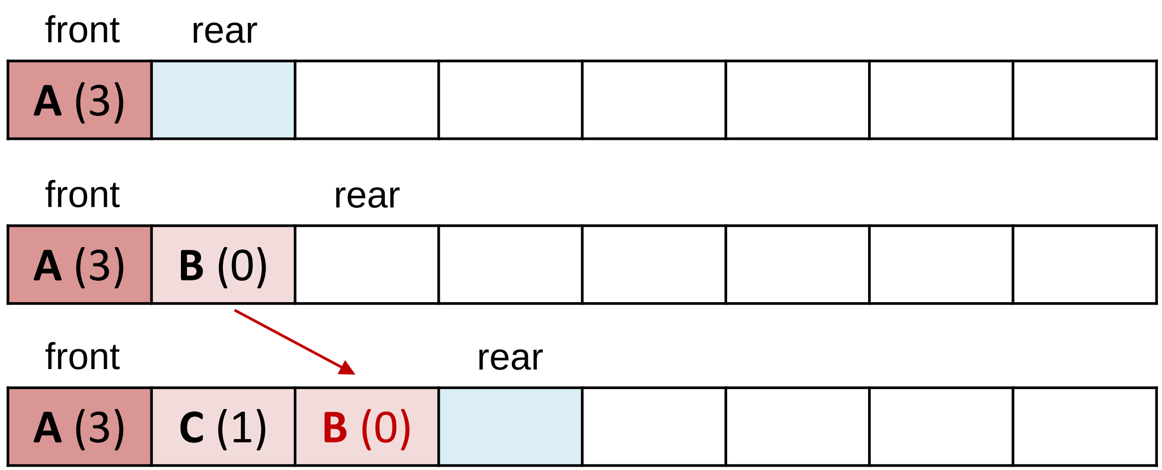
- **Solución 1:** usar la EdD del TAD Cola basada en array, almacenando los elementos no ordenados
  - **Inserción:** inserta un elemento en el rear; no tiene en cuenta las prioridades de los elementos → eficiente
    - Ejemplo: inserción de A (3), B (0), C (1) – prioridades entre paréntesis

front			rear				
A (3)	**B** (0)	**A** (1)					

  - **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

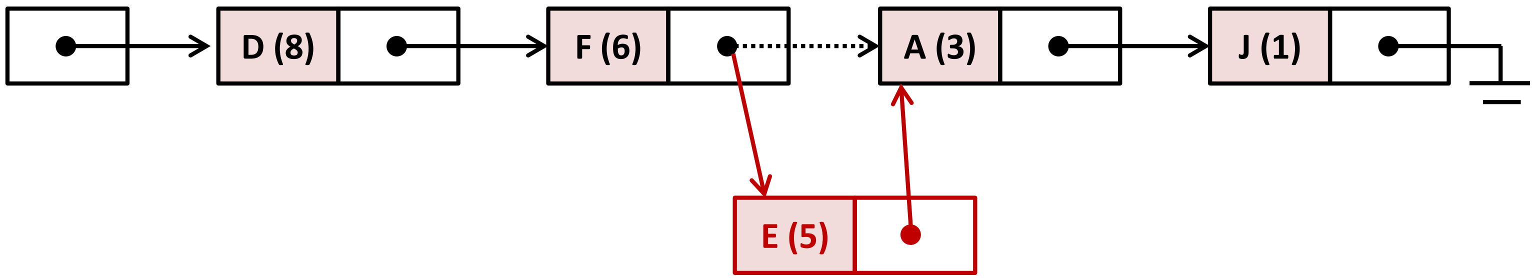
- **Solución 2:** usar la EdD del TAD Cola basada en array, manteniendo los elementos ordenados
  - **Inserción:** mueve los elementos pertinentes a la hora de insertar uno nuevo → ineficiente
    - Ejemplo: inserción de A (3), B (0), C (1) – prioridades entre paréntesis



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

El TAD Cola de prioridad. EdD con lista enlazada

- **<u>Solución 3</u>:** usar la EdD del TAD Lista Enlazada, manteniendo los <u>elementos ordenados</u>
  - **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 son N/2 comparaciones, siendo N el número de nodos de la lista→ O(N))
    - Ejemplo: inserción de E (5) – prioridades entre paréntesis



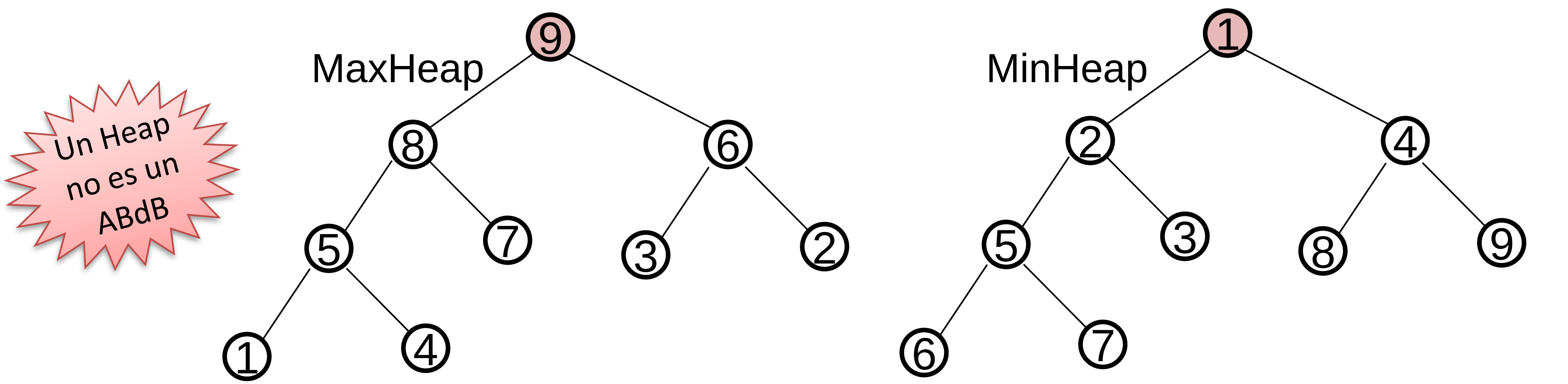
  - **Extracción:** devuelve el elemento del comienzo → eficiente

(Si extrajera del fin, sería eficiente usando la cola de forma circular)

El TAD Cola de prioridad. EdD con heap

- **<u>Solución definitiva</u>:** 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

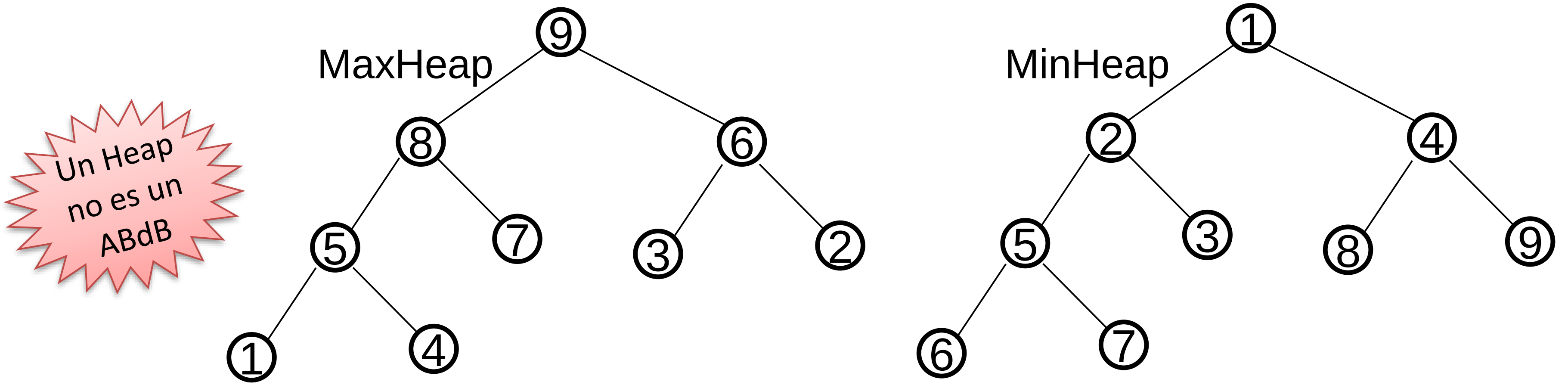


Contenidos

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

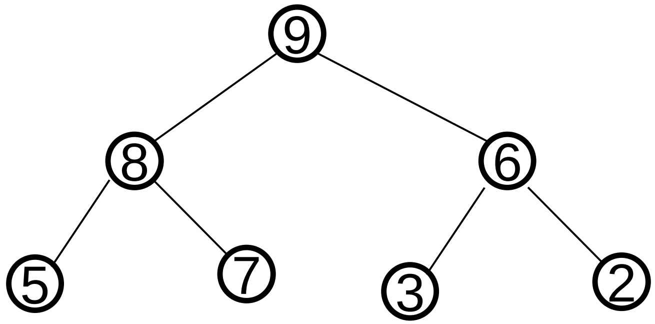
El TAD Heap. Definición

- Un **heap** (montón, montículo) es un <u>árbol binario</u> tal que:
  - Es <u>estrictamente casi completo</u>: 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 <u>condición de orden</u>:
    - Orden descendiente - MaxHeap: info(**n**) > info(**n'**), ∀**n'** descendiente de **n**
    - Orden ascendente - MinHeap: info(**n**) < info(**n'**), ∀**n'** descendiente de **n**

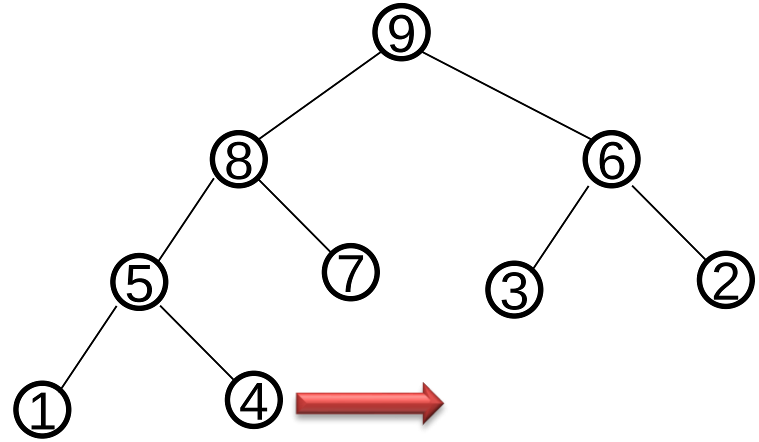


El TAD Heap. Definición

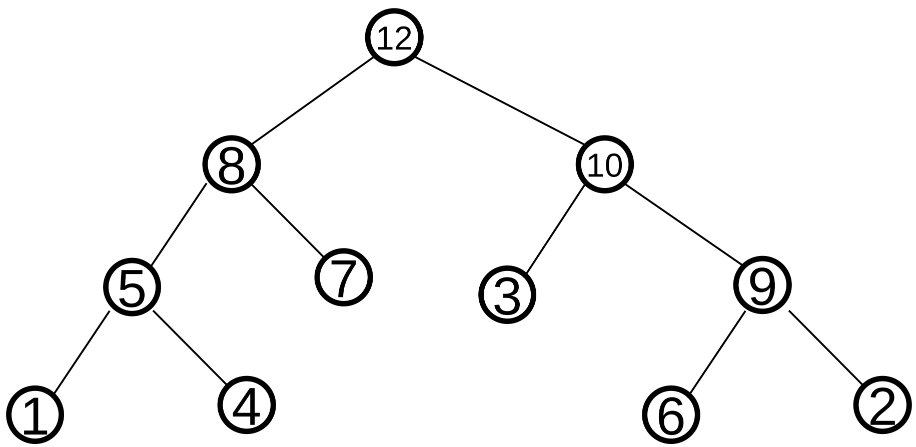
- **Completitud de árboles**



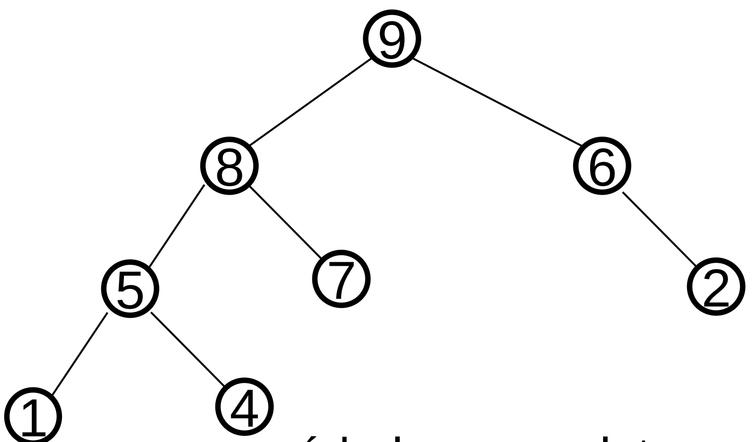
árbol completo



árbol (estrictamente) casi completo

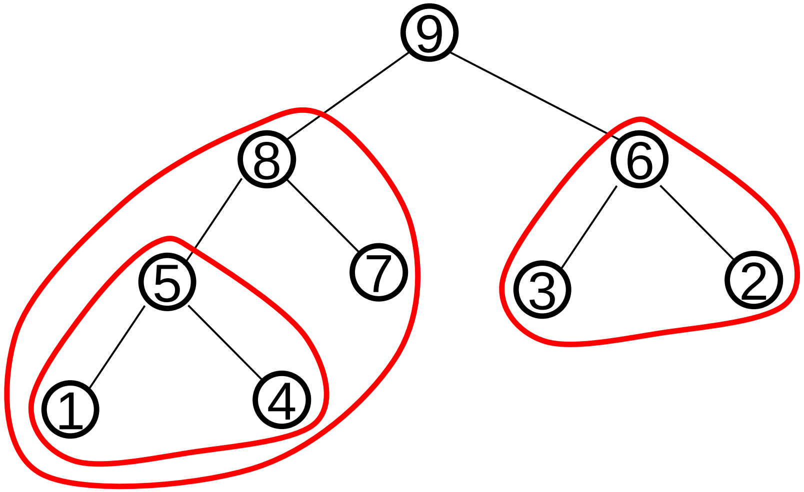


árbol casi completo
(no completo, faltan en el último nivel)



árbol no completo
(faltan en cualquier nivel)

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

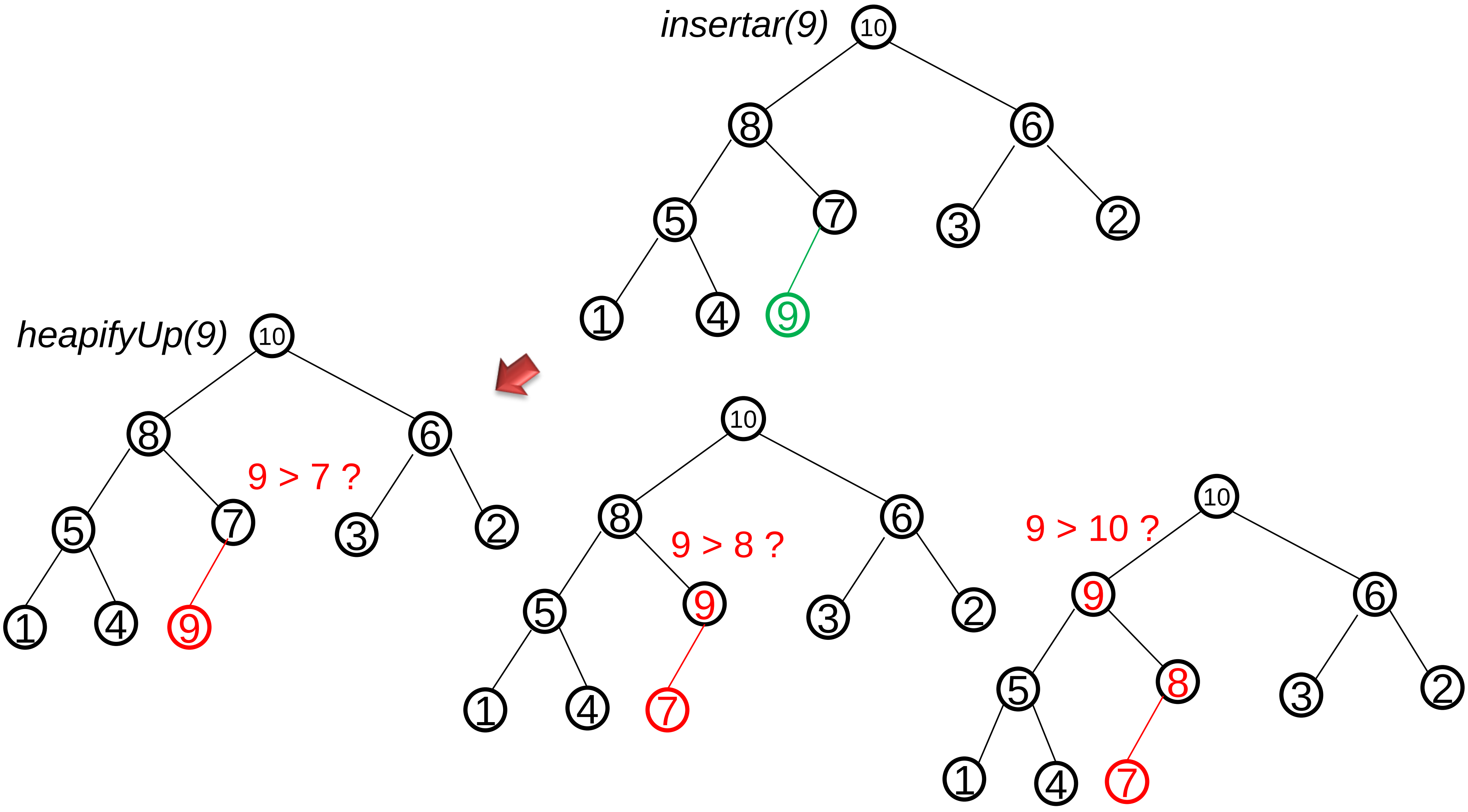


  - En un heap cualquier recorrido desde la raíz hasta una hoja proporciona un vector ordenado de elementos
    - 9 → 4: [9, 8, 5, 4]
    - 9 → 3: [9, 6, 3]

- **Heap** – inserción de un elemento
  1. Se coloca el elemento en un nuevo **"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 intercambian el elemento y el padre.
     - La recursión se detiene cuando se cumple la propiedad de MaxHeap o bien el elemento se encuentra en el nodo raíz.

El TAD Heap. Inserción de un elemento

- **Heap** – inserción de un elemento



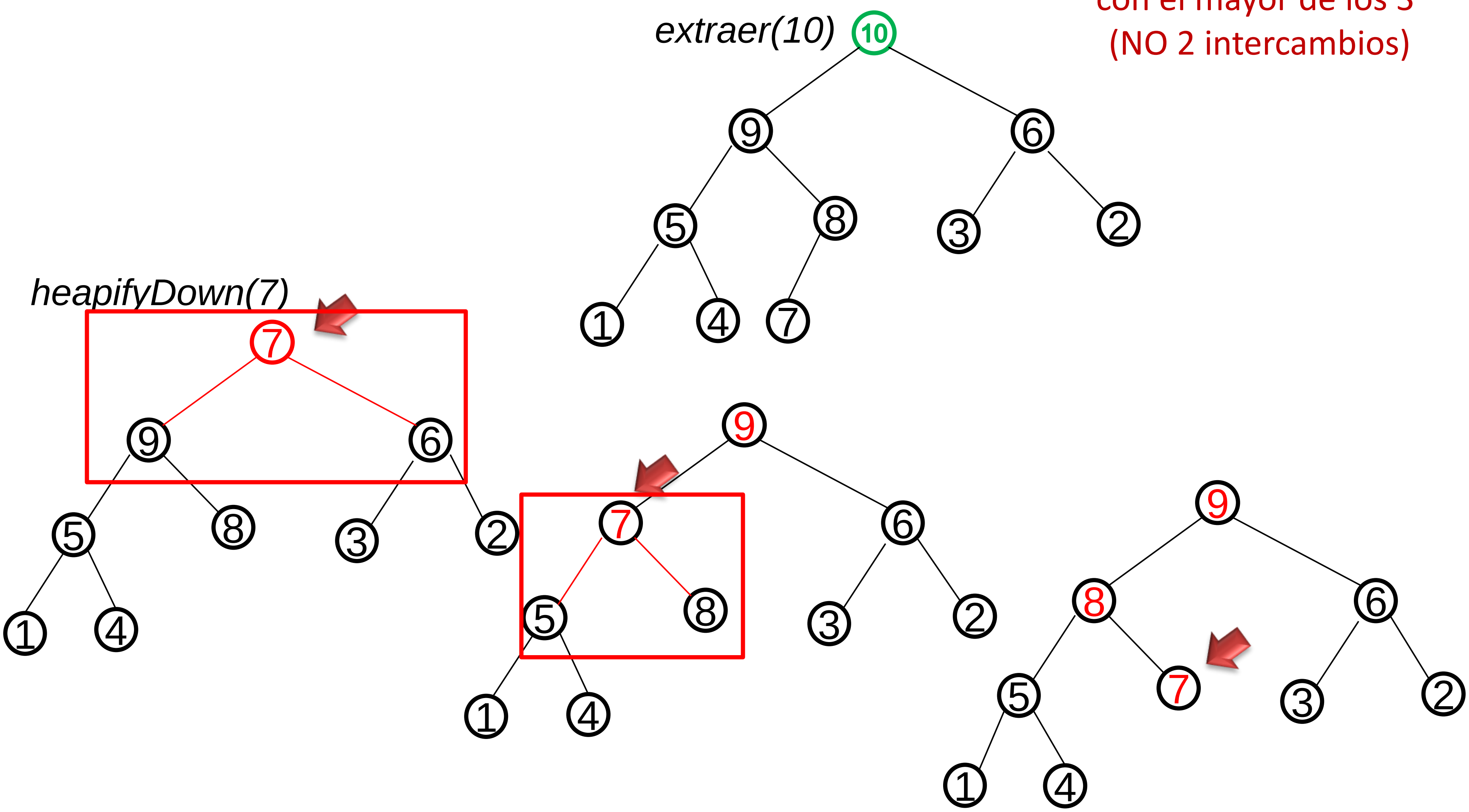
El TAD Heap. Extracción del elemento raíz

- **Heap** – extracción del elemento raíz
  1. Se extrae (para devolverlo) 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 recursiva el elemento se compara con sus hijos, y si no se cumple la condición de MaxHeap (o MinHeap), el elemento se intercambia con el hijo de mayor valor (MaxHeap)
     - La recursión se detendrá cuando se satisfaga la condición de MaxHeap (o MinHeap).

El TAD Heap. Extracción del elemento raíz

- **Heap** – extracción del elemento raíz

Importante: intercambia con el mayor de los 3 (NO 2 intercambios)

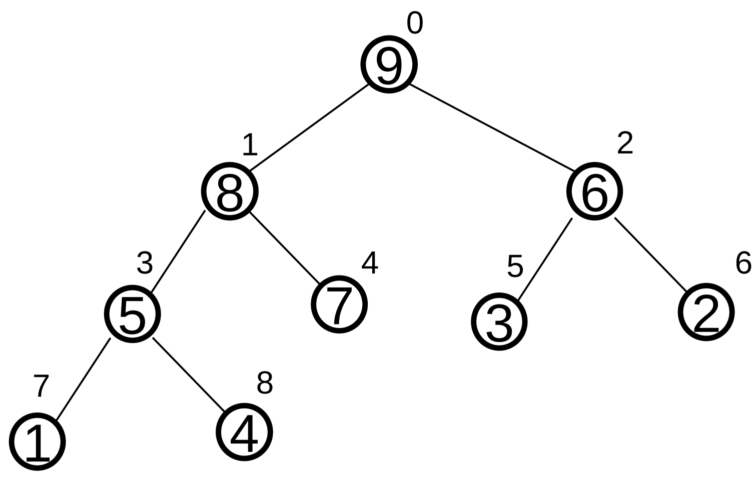


Contenidos

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

Implementación de Heap. EdD

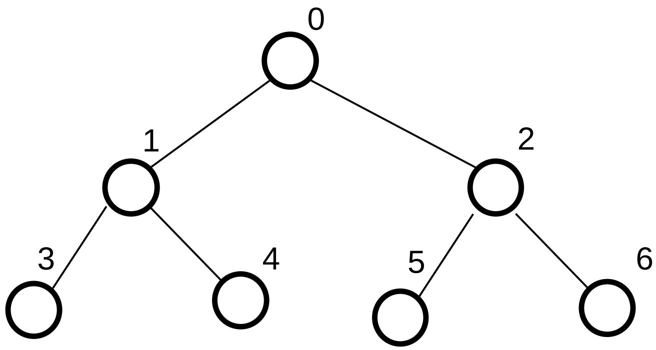
- **Heap** – EdD
  - heap = array en el que los nodos del heap se identifican con los índices del array
    - raiz = T[0]
    - izq(n) = 2n+1
    - der(n) = 2n+2
    - pad(n) = $\lfloor(n-1)/2\rfloor$

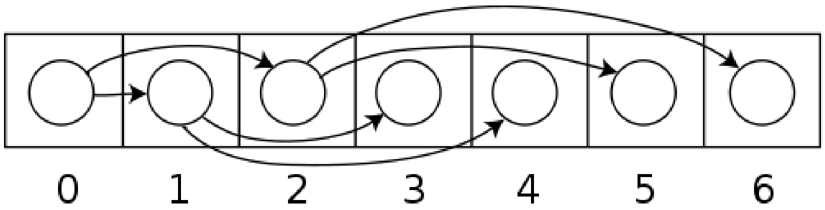


9	8	6	5	7	3	2	1	4	
0	1	2	3	4	5	6	7	8	

Implementación 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)**

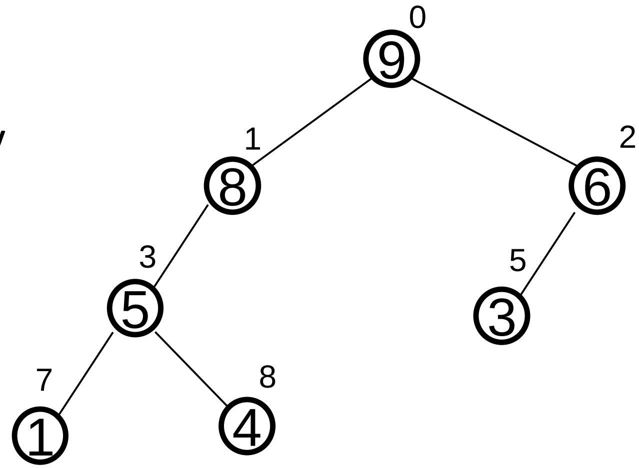




Implementación de Heap. EdD

- El índice del array se corresponde con la numeración de los nodos.
- Si tuviéramos árboles no completos
  - Representación ineficiente: huecos en el array

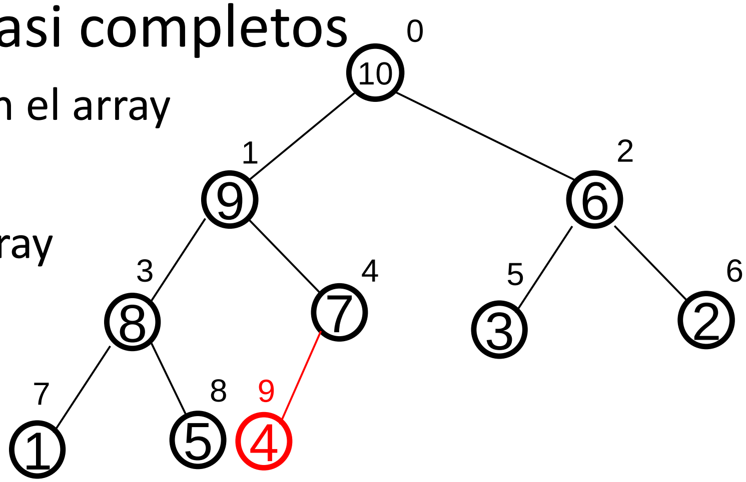
9	8	6	5		3		1	4	
0	1	2	3	4	5	6	7	8	



- 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 de Heap. EdD

- **Uso de un array como EdD para Heap**
  - **Ventajas**
    - Permite acceso aleatorio (O(1)) a los nodos mediante su índice [i]
    - No desperdicia memoria: el árbol está balanceado, al ser estrictamente casi completo.
    - Uso eficaz de la (reserva de) memoria cuando se conoce el número de elementos máximo que se va a manejar.
    - Representación y manejo de datos simple: no es necesario gestionar dinámicamente nodos y punteros (izq y der).
  - **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 de Heap. EdD (PSC)

- **Heap** (pseudocódigo):

Datos (a considerar para implementar la estructura):

```
MAX_HEAPSIZE → máximo número de elementos que puede
               albergar un heap
h[i]  → elemento que ocupa la posición i en el array de
        datos asociado al heap
Heapsize → tamaño actual del heap
```

Macros:

```
izq(p) → (2 * (p) + 1)
der(p) → (2 * (p) + 2)
pad(p) → ((int) (((p)-1)/2))
```

Primitivas:

```
Heap heapIni()
Bool heapVacio(Heap h)
Status heapIns(Dato D, Heap h) //Insertar elemento en el heap
Dato heapExt(Heap h)          //Extraer raíz del heap
Heap buildHeap(Vector v)       //Convertir un vector en heap
```

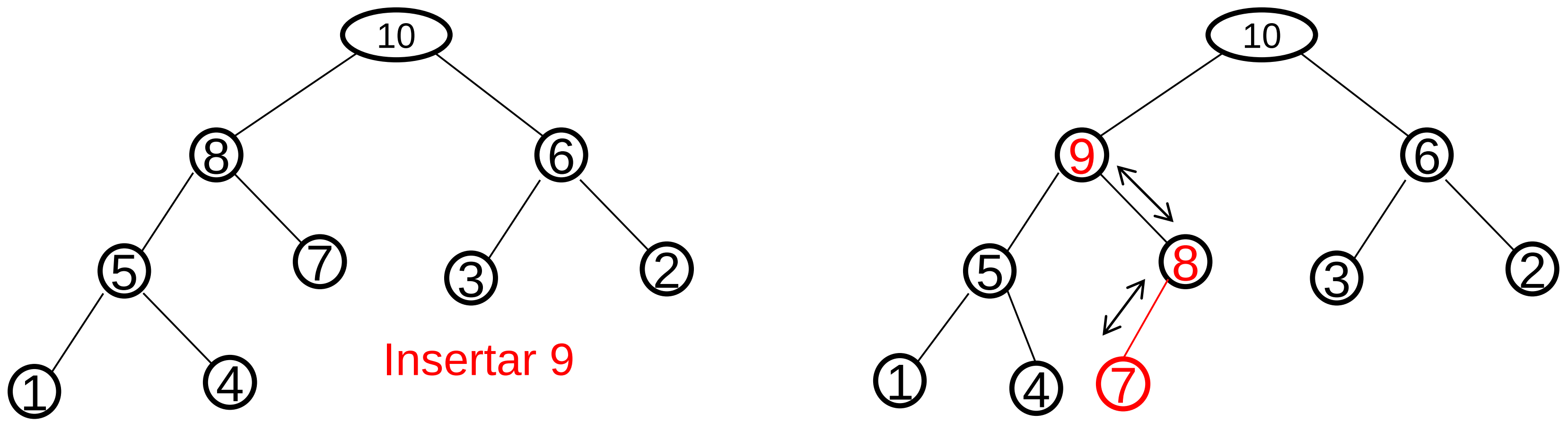
Implementación de Heap. Primitivas (PSC)

- **Heap** – Pseudocódigo

```
Heap heapIni()
   h=memoria
   heapsize= 0
   dev OK
```

```
Bool heapVacio(Heap h)
   si heapsize = 0:
      dev TRUE
   si no:
      dev FALSE
```

- Insertar
  - Se coloca el elemento en la última posición
  - Se hace un heapifyUp del elemento



Implementación de Heap. Primitivas (PSC)


```
Status heap_insertar(Heap h, Elemento ele)
       si heapsize >= MAX_HEAPSIZE  dev ERROR

       h[heapsize]= elemento_copiar (ele) //Poner último elemento
       incr heapsize                      //incrementar el tamaño del heap
       heapifyUp (h, heapsize-1)          // Hacer heapifyup
       dev OK


Status heapifyUp(Heap h, int k)
       p = pad(k)
       si p<0: dev OK // Si ya no hay padre -> condición de parada

       si  h[k] > h[p]
              h[p] ←→ h[k]  //intercambio
              dev heapifyUp(h, p)  // llamada recursiva

       //si no, no hay nada que intercambiar → salir de recursión
       dev OK
}
```

Implementación de Heap. Primitivas (PSC)

- **Heap** – Pseudocódigo extracción

```
Elemento heap_extraer(Heap h)
        si heapVacio(h)dev NULL
        ele= h[0]                       // Coger la raíz (para retorno)
        decrementar heapsize           // Decrementar nº de nodos
        h[0] ←→ h[heapsize]             // Intercambiar último con 1º

        heapifyDown(h, 0)               // heapify down desde la raíz

        dev ele
}
```

- Obtener el elemento en la raíz
- Copiar el último elemento a la raíz
- Hacer heapifyDown de la raíz

Implementación de Heap. Primitivas


```
Status heapifyDown(Heap h, int k)

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

      max = k

      si  izq < heapsize Y  h[k]< h[izq]: //max=índice del máx
              max= izq
      si  der < heapsize Y  h[max]< h[der]:
              max= der

      si  max ≠ k:
              h [max] <-> h[k]   //intercambia
              dev heapifyDown(h, max) // Llamada recursiva desde la
                                     // posición donde estaba el
                                     // máximo (el intercambiado)
      dev OK
```

Construir heap a partir de vector

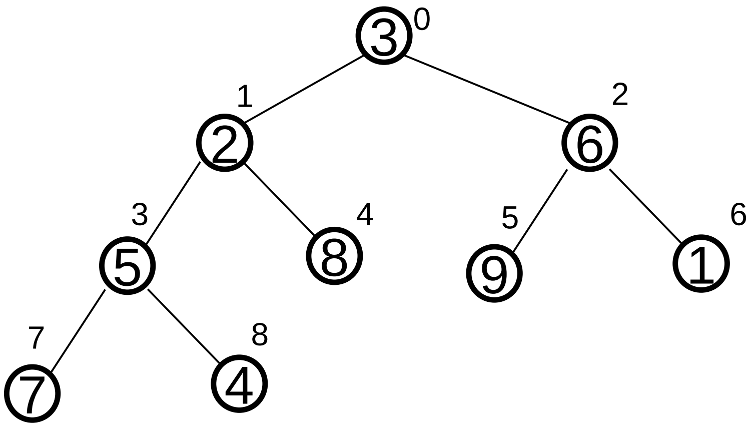
- **Construir un heap a partir de un vector:**
  - Utilizar directamente el vector de números desordenado para crear un heap: procedimiento <u>buildHeap</u>
    - Vector inicial desordenado = árbol. Reordenarlo con buildHeap.
    - buildHeap= sucesión de llamadas a heapifyDown desde el último padre de los nodos del heap
      - ¿Es necesario llamar a heapifyDown de todos los nodos? ¿Hojas?

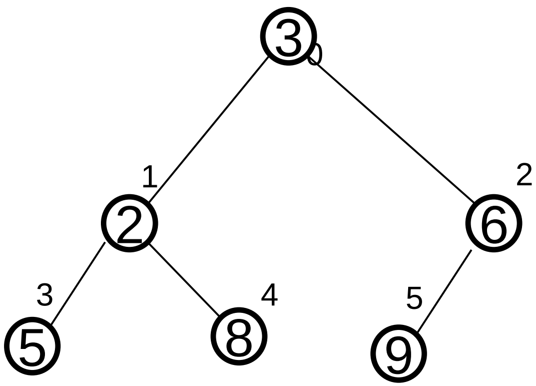
Construir heap a partir de vector

- **Construir un heap a partir de un vector:**
  - Utilizar directamente el vector de números desordenado para crear un heap: procedimiento <u>buildHeap</u>
    - Vector inicial desordenado = árbol. Reordenarlo con buildHeap.

¿Desde qué elemento (posición del array) hay que llamar a heapifyDown ?

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





Pos 3 = pad(8) = floor((8-1)/2) = 3

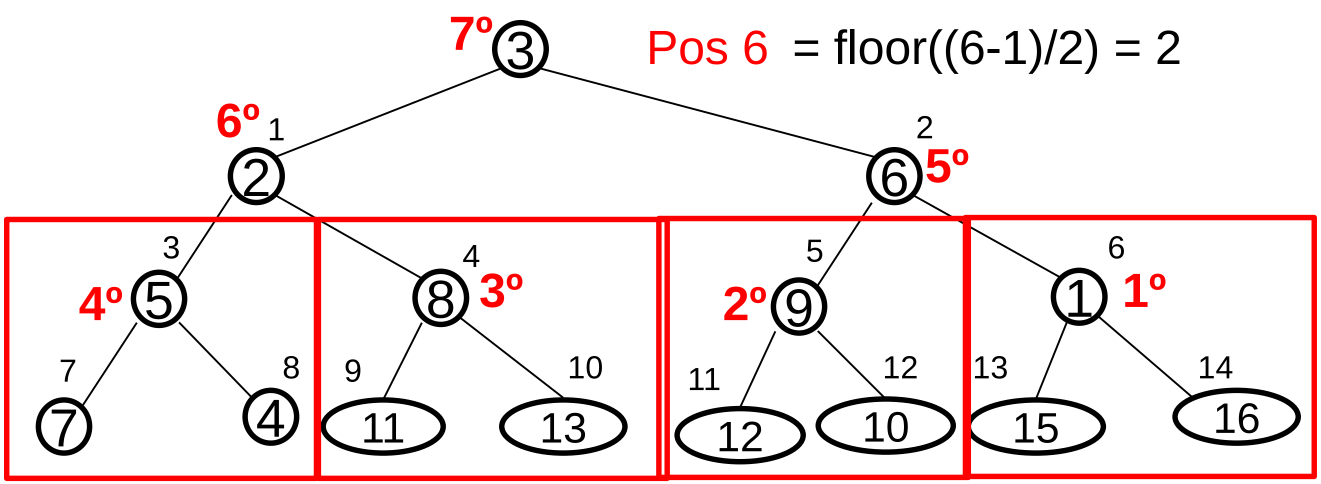
Pos 2 = pad(5) = floor((5-1)/2) = 2

Construir heap a partir de vector

Se comienza llamando a heapifyDown desde el último padre.

```
Heap buildHeap(Vector v)
   para i desde suelo(heapsize/2-1) hasta 0:
      heapifyDown(v,i)
   dev v      //Heap y array o Vector son lo mismo
```

3	2	6	5	8	9	1	7	4	11	13	12	10	15	16



Contenidos

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

Algoritmo de ordenación HeapSort

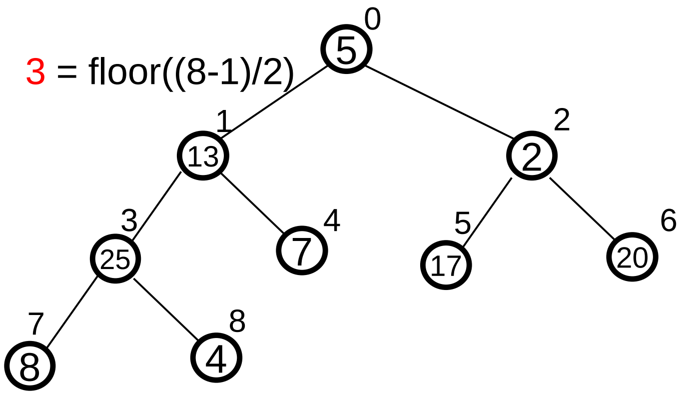
- **Algoritmo ordenación de datos basado en heaps**
- **Fase 1: construcción del heap *(buildHeap)***
  - Crea heap a partir del vector de elementos desordenados con **buildHeap**:
    - sucesión de llamadas a heapifyDown sobre los nodos (excepto las hojas) del heap, desde "el último padre" índice: floor((N-1)/2)
- **Fase 2: extracción iterativa de los elementos del heap**
  - Mientras heap no vacío, **heapExtr** (extrae la raíz) y colocar dato en el vector, desde el final hasta el principio (***in-place***)
    Obs: "separar" los elementos del heap de los elementos extraídos

HeapSort.

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

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

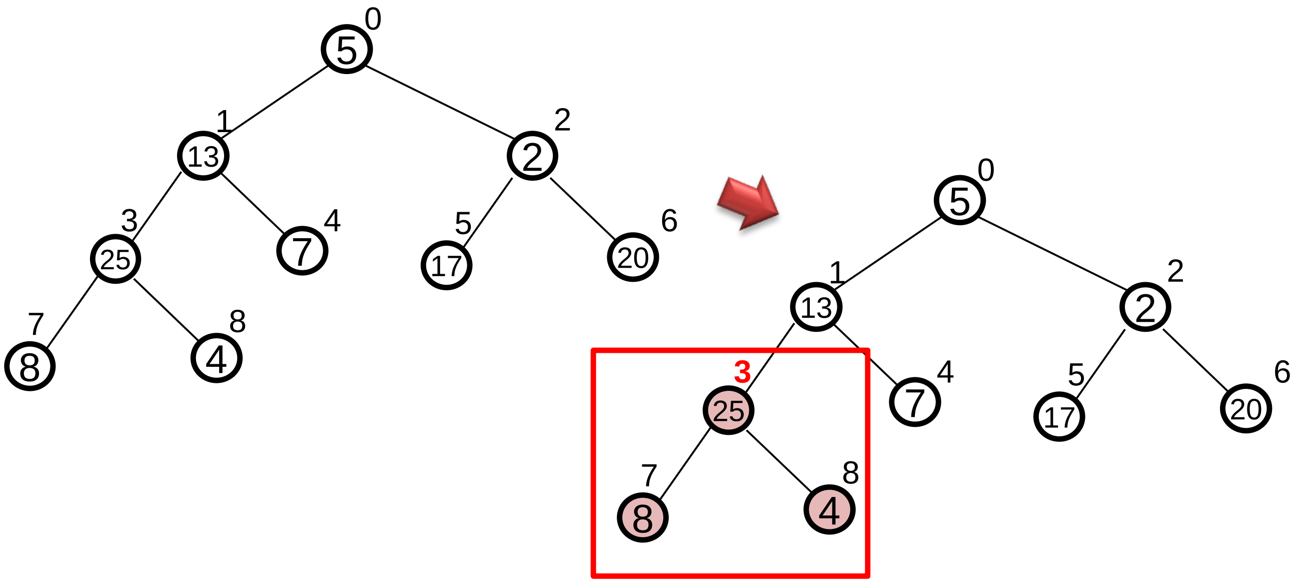
3 = floor((8-1)/2)



HeapSort.

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

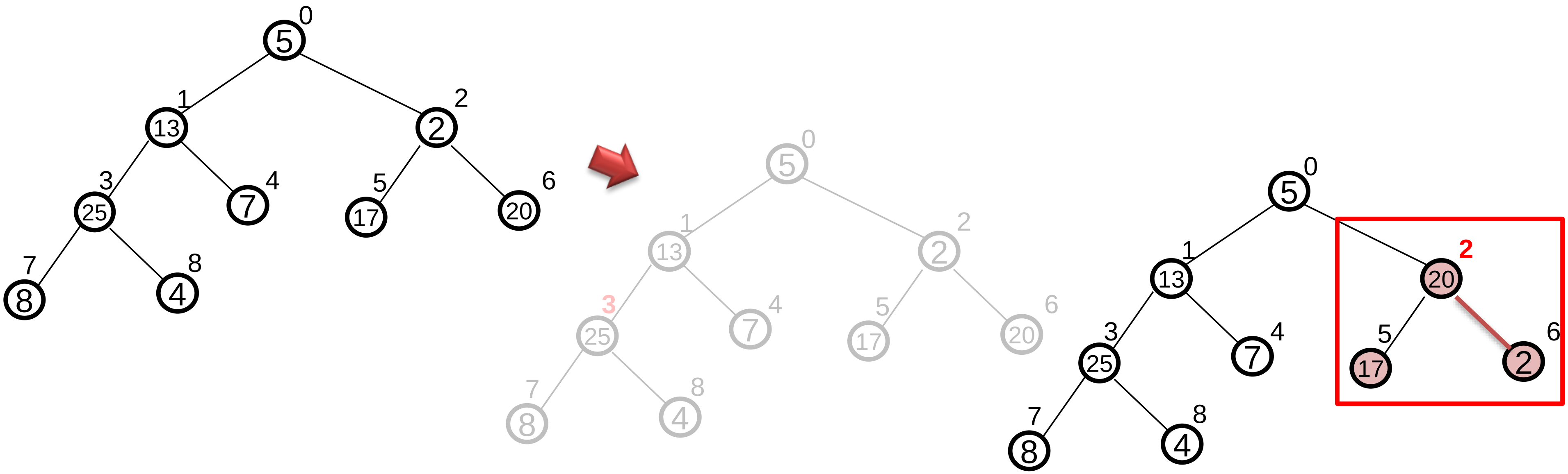
HeapSort	5	13	2	25	7	17	20	8	4
	0	1	2	3	4	5	6	7	8



HeapSort.

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

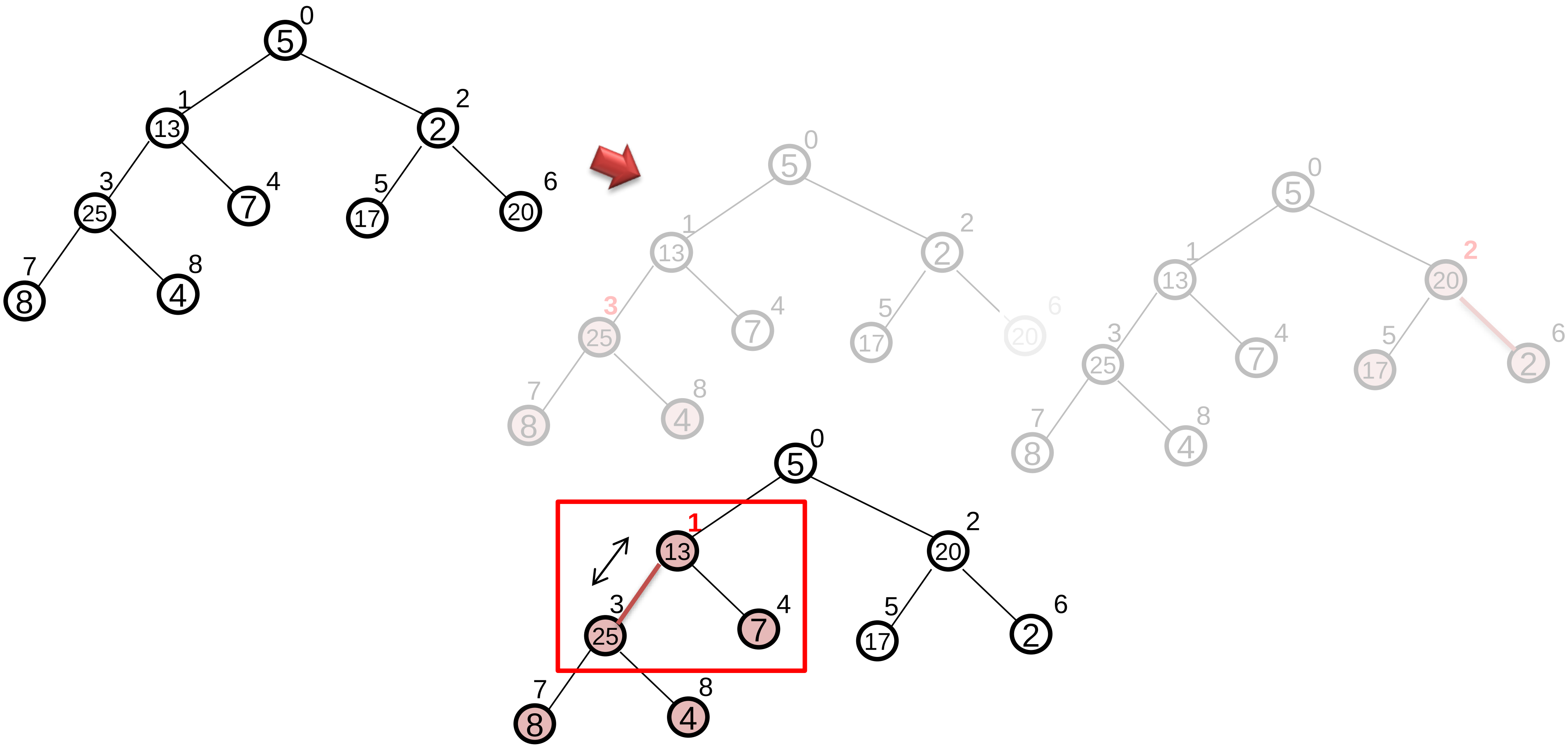
HeapSort	5	13	2	25	7	17	20	8	4
	0	1	2	3	4	5	6	7	8



HeapSort.

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

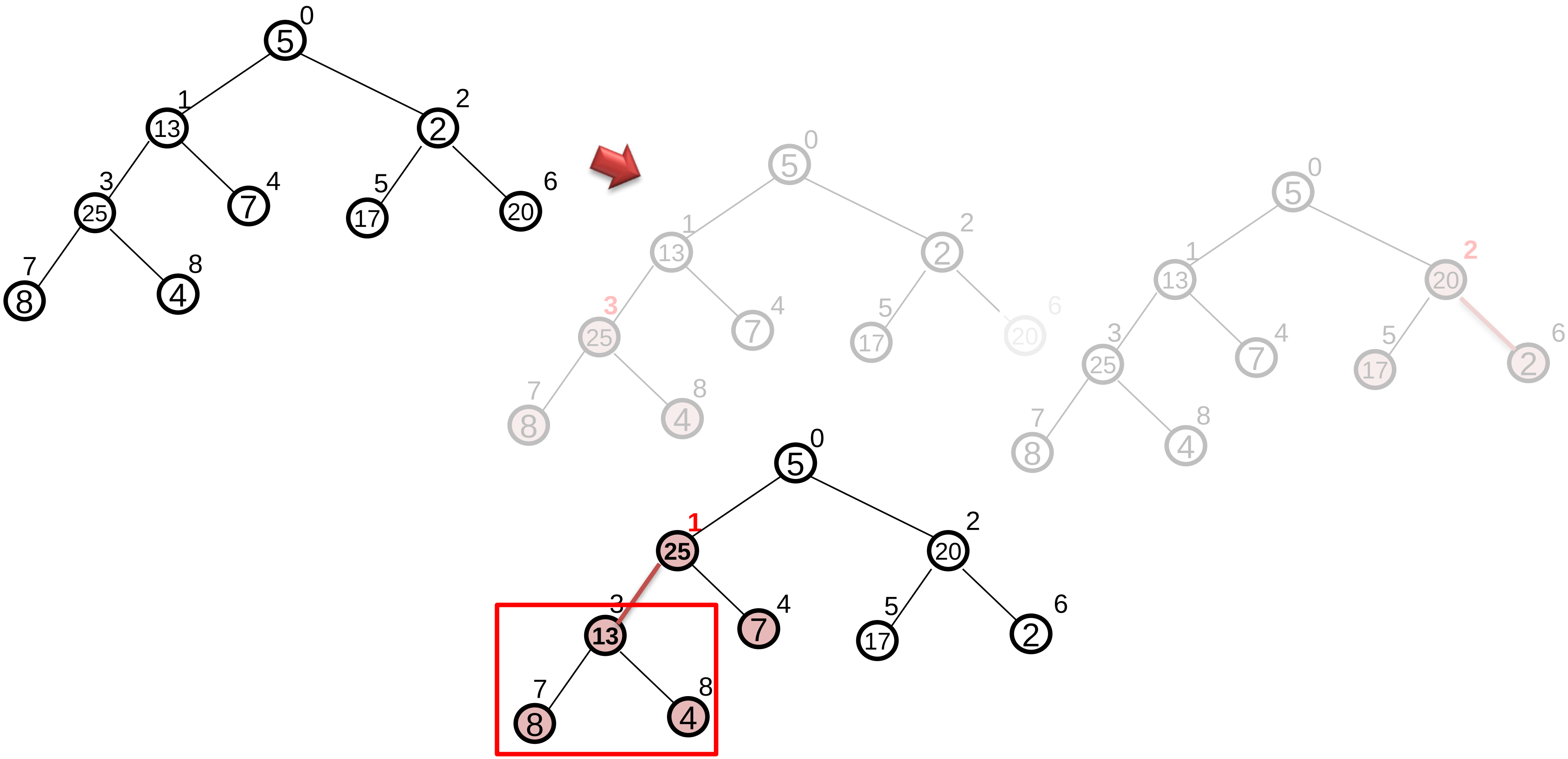
HeapSort	5	13	2	25	7	17	20	8	4
	0	1	2	3	4	5	6	7	8



HeapSort.

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

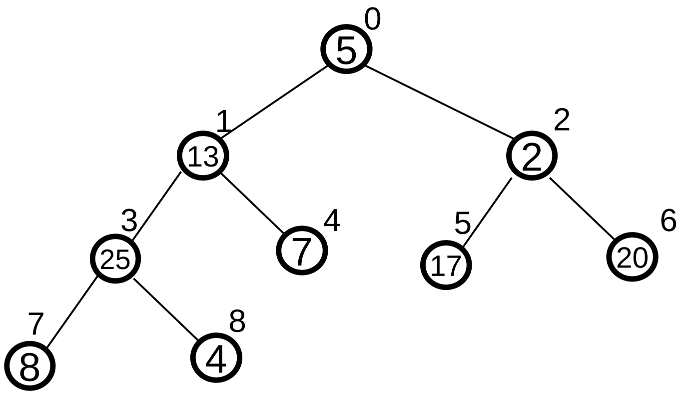
HeapSort	**5**	**13**	**2**	**25**	**7**	**17**	**20**	**8**	**4**
	0	1	2	3	4	5	6	7	8



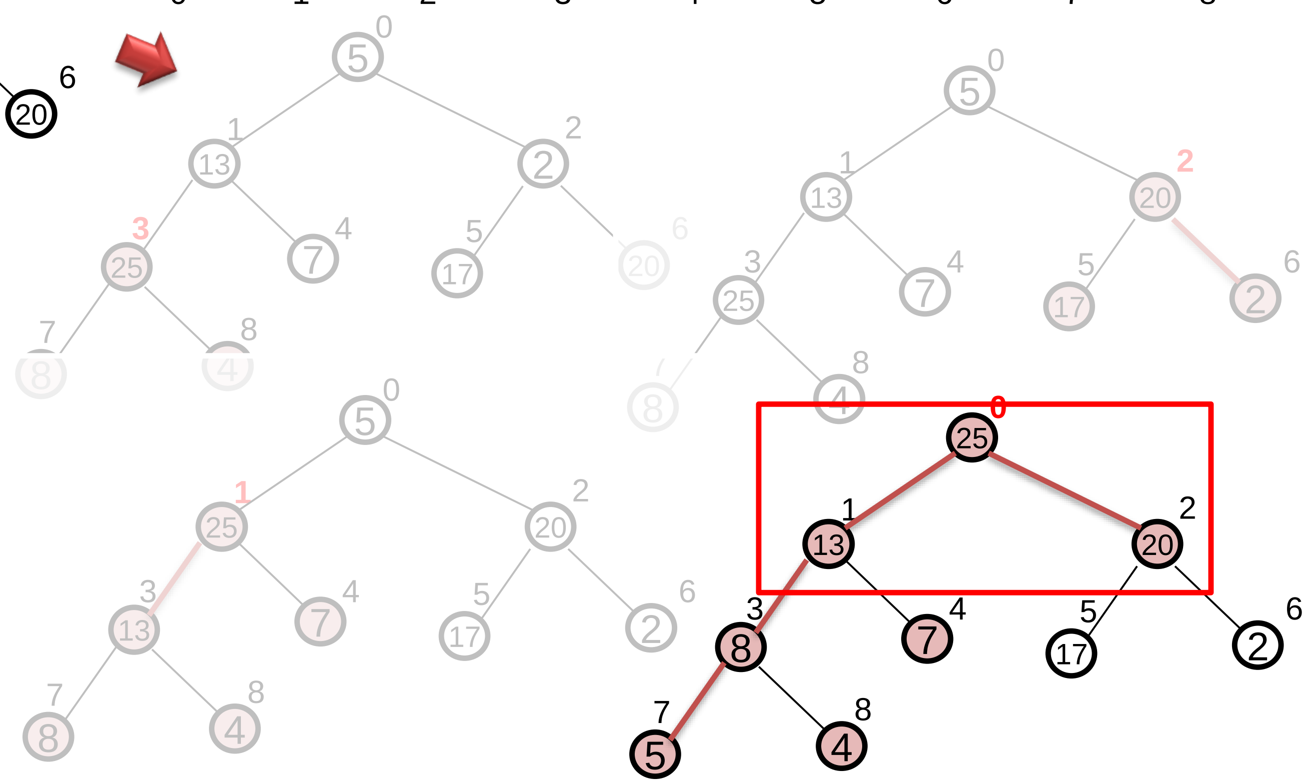
HeapSort.

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

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



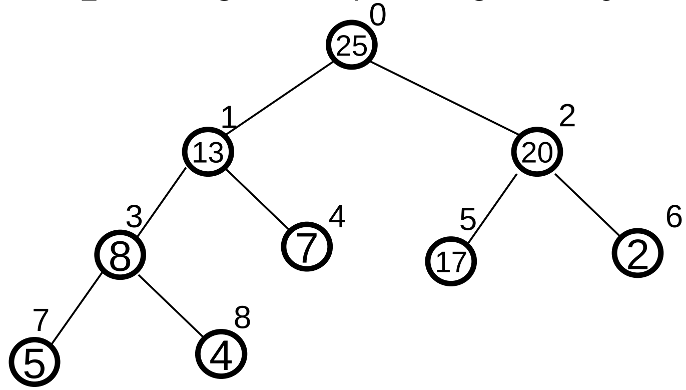
25	**13**	**20**	**8**	**7**	**17**	**2**	**5**	**4**
0	1	2	3	4	5	6	7	8



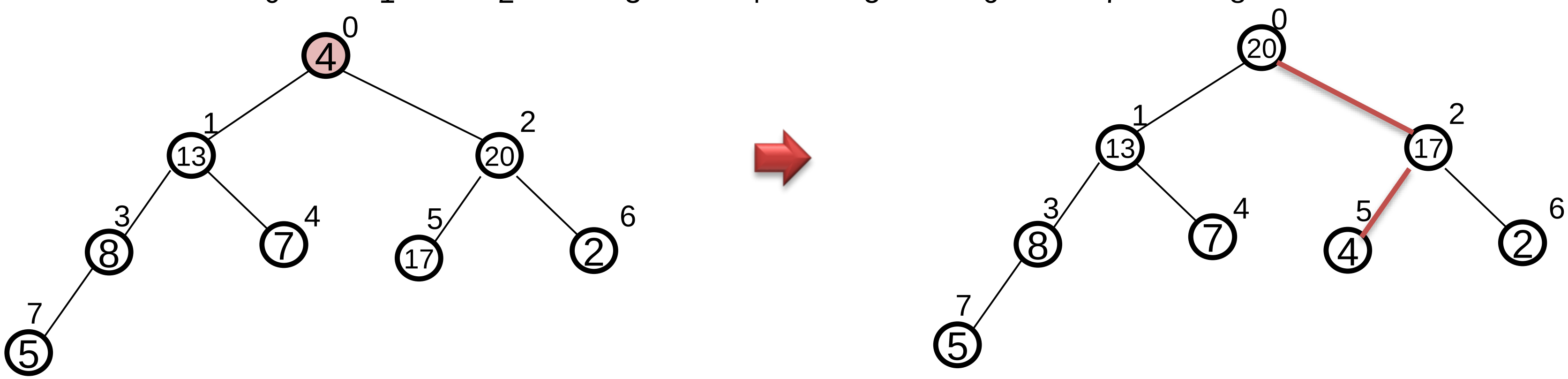
HeapSort.

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

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



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



HeapSort.

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

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

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

HeapSort.

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

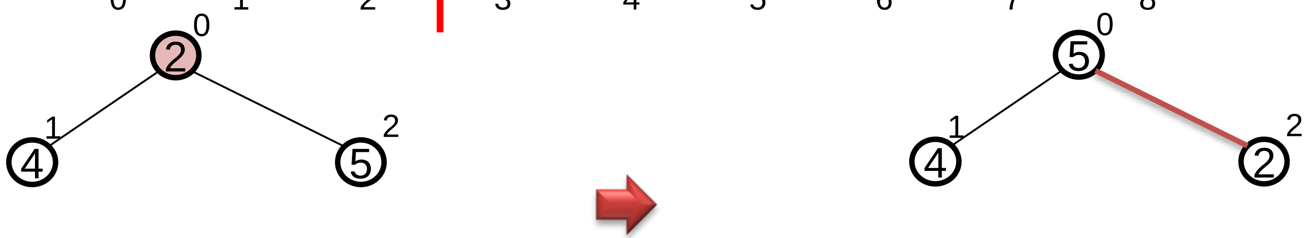
8	7	5	2	4	**13**	**17**	**20**	**25**
0	1	2	3	4	5	6	7	8

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

HeapSort.

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

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



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

HeapSort.

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

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

0 2 0 2

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

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 ABdB: O(N log N)   (insertar N elemtos)
  - 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 para nodos
      - No tiene el problema de árboles desbalanceados, en los que el coste de ordenación en el caso peor es $O(N^2)$

  - Más métodos de ordenación y su complejidad se estudiarán en Análisis de Algoritmos (quicksort, bubblesort, etc.)

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 de una cola de prioridad devuelve el elemento de mayor prioridad
  - Un **heap** puede almacenar una serie de elementos con prioridades. Si las prioridades se utilizan para satisfacer la condición de MaxHeap, la extracción de un maxheap devuelve el elemento con mayor prioridad (la raíz)

El TAD Cola de prioridad. Ejemplo

Queremos **ordenar los pacientes que van llegando a las urgencias conforme a la gravedad de sus dolencias**. Grados de gravedad:

0 (no es urgencia), 1 (urgencia muy leve), 2 (urgencia leve), 3 (urgencia moderada), 4 (urgencia notable), 5 (urgencia grave), 6 (urgencia muy grave).

El orden de llegada de los pacientes y la gravedad de su urgencia es:

Instante llegada	Nombre	Gravedad urgencia
1	Alicia	1
2	Darío	2
3	Edu	4
4	Eva	0
5	Manu	3
6	Lore	5

Anacleto ha ido anotando la información según han ido llegando los pacientes, sin considerar su gravedad.

¿Forma rápida de reorganizar la información para tener acceso inmediato al siguiente paciente a llamar en cada momento?

El TAD Cola de prioridad. Ejemplo

Datos de Anacleto (desordenados):

Alicia - 1	**Darío - 2**	**Edu - 4**	**Eva - 0**	**Manu - 3**	**Lore - 5**

Buildheap

Mismos datos reordenados (acceso inmediato al paciente más urgente):

El TAD Cola de prioridad. Ejemplo

Se queda libre médico → llamar al siguiente paciente
(y mantener datos garantizando acceso inmediato al siguiente)

Extraer raíz, llevar último a la raíz y heapifyDown

El TAD Cola de prioridad. Ejemplo

Se queda libre médico → llamar al siguiente paciente
(y mantener datos garantizando acceso inmediato al siguiente)

Extraer raíz, llevar último a la raíz y heapifyDown

El TAD Cola de prioridad. Ejemplo

Llega Oscar, gravedad 4. Incorporación al listado:

Insertar en última posición y heapifyUp

El TAD Cola de prioridad. Ejemplo

Ya no se pueden atender más pacientes por hoy ☹

→ Ordenar para obtener listado completo de pacientes a llamar en orden.

HeapSort: 1º BuildHeap (ya lo tenemos)

2º Extraer de uno en uno *in-place*

El TAD Cola de prioridad. Ejemplo

Cuestiones:

1) ¿Has creado un maxheap o un minheap? ¿Por qué?

Maxheap, Porque así siempre tenemos accesible para su extracción INMEDIATA el paciente de mayor prioridad (= gravedad) en cada momento, que es el primero al que hay que atender.

2) ¿Qué habías hecho si te hubieran dicho que 0 significa máxima urgencia y 6 la mínima? ¿Cuáles serían las diferencias principales?

Crear un minheap. Funciona todo igual, salvo que las condiciones < cambian a > y viceversa.

2) Sobre el mantenimiento de los datos que hizo Anacleto y tu forma de mantenerlos: ¿Qué tiene que hacer él para encontrar el siguiente paciente a llamar? ¿Y tú? ¿Cuál de las 2 opciones es más eficiente?

Él tenía que recorrer el array. Yo solo tengo que llamar directamente al que se encuentre en la raíz del heap (es inmediato). Después tendré que recolocar el heap, subiendo el último elemento a la raíz y haciendo heapifyDown del mismo. Como mucho, tardaré O(logN), mientras que el array de Anacleto habrá que recorrerlo hasta el final para buscar el paciente de mayor urgencia (O(N)). Por tanto, mi solución es más eficiente.