Algoritmos y Estructuras de Datos

Tema 6-Bis: Monticulos (Heaps) y Heap-Sort

Grado Imat. Escuela ICAI

March 2024

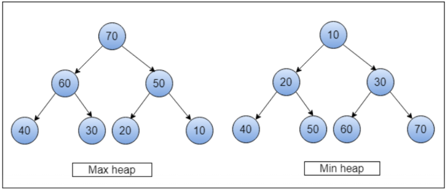
Part I

Monticulos a.k.a. Heaps

Arboles en Montón, Montículos, o Heaps

Se trata de un árbol binario con los nodos ordenados de tal manera que siempre el elemento mayor se encuentra en la raíz del árbol.

- **DEFINICIÓN:**
  - Un montículo de tamaño n se define como un árbol binario casi completo de n nodos, tal que el contenido de cada nodo es mayor[1] o igual que el contenido de sus hijos.
- **Un árbol binario COMPLETO**
  - es un árbol con todos los niveles llenos, con la excepción del último nivel, que se llena de izquierda a derecha. Esta estructura tiene la propiedad importante de que la altura de un árbol binario completo de n nodos es log2 n.



[1] o menor!

Heap Trees: Propiedades

1. Los montículos están organizados como **árboles binarios**, con lo cual sus operaciones tienen complejidades con cotas logarítmicas y además se puede representar fácilmente en una estructura tipo vector.
2. Están ordenados de manera que la **clave de cualquier nodo es mayor o igual a la de sus hijos (si tiene hijos)**. Por consiguiente, el nodo con mayor clave debe ser el nodo raíz del árbol. Por lo tanto, siempre "Buscar el máximo" tiene una complejidad constante, O(1).

   Sin embargo, no existe un orden entre las claves de los elementos que se encuentran en el mismo nivel.

Según se quiera utilizar el montículo, se elige que sea el elemento mayor el que se encuentre en la raíz o el menor. A lo largo de este tema será el mayor elemento el que se encuentre en la raíz.

HeapTrees (Montículos) USOS

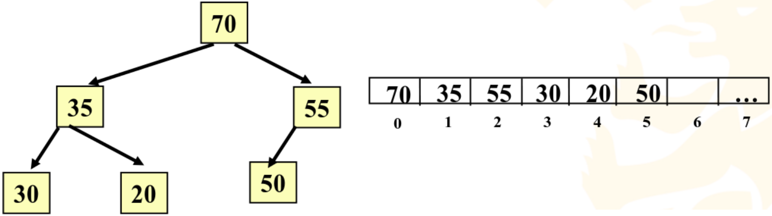
Estas propiedades hacen que los montículos sean una estructura muy adecuada para (entre otras):

1. Implementar COLAS DE PRIORIDAD ya que el elemento más prioritario siempre está en la raíz del árbol.
2. Se utilice para ORDENACIÓN de VECTORES, utilizando el ALGORITMO DEL HEAP- SORT.

Representación de un Montículo

Se utiliza una representación con una tabla, de manera que:

1. El nodo raíz está en la posición 0[2]
2. Para el nodo en la posición i, si existe, su hijo izquierdo está en la posición $2 * i + 1$ y su hijo derecho, están en la posición $2 * i + 2$, y son menores o iguales que el padre
3. Por lo tanto, para el nodo i, su padre se encuentra en la posición $(i - 1)//2$.



[2] python indexing

Monticulos: Condición de Ordenación

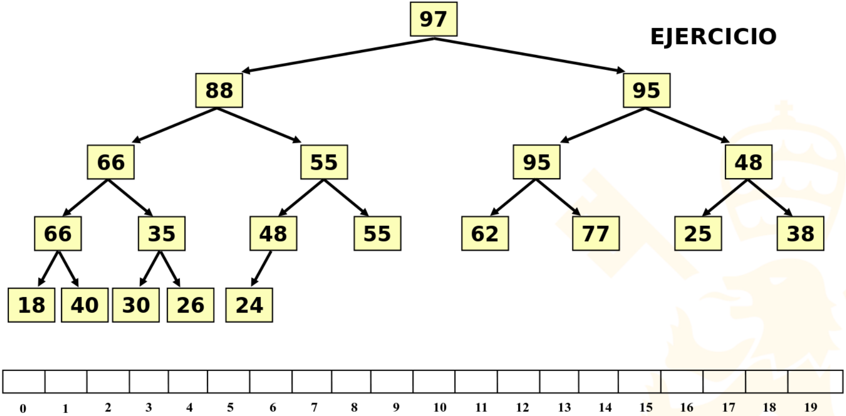
La ordenación parcial de los nodos de un montículo se conoce como CONDICIÓN DE MONTÍCULO, que permite encontrar con una sola operación el elemento máximo (se encuentra en la primera posición de la tabla en una representación secuencial).

Utilizando la representación secuencial de los montículos, la propiedad de ordenación parcial de las claves se expresa:

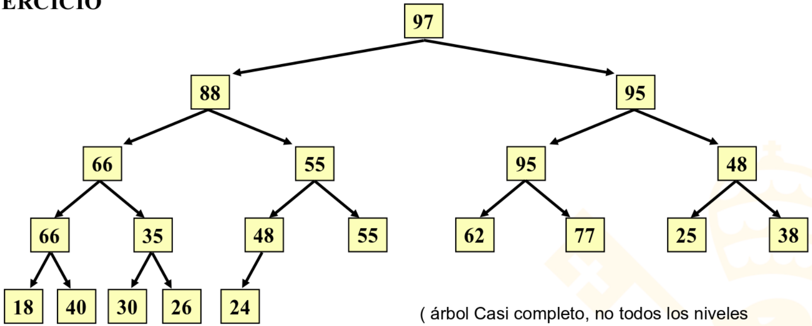
$$
\begin{aligned}
v[i] &\geq v[2*i+1] \; \textit{Hijo izquierdo} \\
v[i] &\geq v[2*i+2] \; \textit{Hijo derecho}
\end{aligned}
$$

Un nodo es siempre mayor o igual que sus hijos

Heaps Ejercicio



EJERCICIO



(árbol Casi completo, no todos los niveles han de estar llenos)

97	88	95	66	55	95	48	66	35	48	55	62	77	25	38	18	40	30	26	24
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Heaps: Operaciones

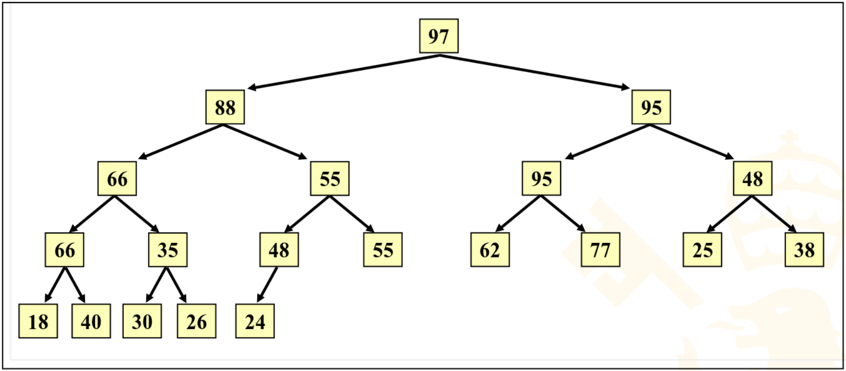
- **Operaciones**
  - **CrearMontículo()**: inicializa un montículo, en base a una lista de valores.
  - **EsVacío()** comprueba si el montículo está o no vacío.
  - **Insertar()** inserta un elemento en un montículo, conservando la condición de montículo.
  - **BuscarMáximo()** devuelve el elemento con la máxima clave de todas las almacenadas en el montículo (el nodo raíz).
  - **EliminarMáximo()** elimina el elemento de clave mayor (el que está en la raíz) conservando la condición de montículo.
- **Estructura de datos**
  - Lista o Array de elementos, opcionalmente de tamaño MAXNODOS, y con una longitud N $N \leq MAXNODOS$
  - En Python: Usar listas, o arrays (https://docs.python.org/es/3/library/array.html)

Inserción de un nuevo nodo en un montículo

- Los pasos a seguir son:
  1. Se inserta el nuevo nodo al final de la tabla.
  2. Si con ello no se rompe la condición de ordenación, el proceso ha terminado.
  3. Si se infringe, ya que el padre es menor que el hijo, se intercambian (el padre BAJA a la posición del hijo, el hijo FLOTA).
  4. En cada paso, se va realizando esta comprobación con el padre del nodo en estudio hasta encontrar la posición correcta del nodo (cuando ya no sea mayor que su padre).
- Por lo tanto, se puede decir que el nuevo nodo FLOTA hasta la posición que le corresponda.
- La máxima complejidad de esta operación (el número máximo de pasos) se da cuando se inserta una clave en el montículo que es el nuevo valor máximo. Ese número de pasos coincide con el número de niveles del árbol. Para un montículo de n nodos, el tiempo necesario en el peor de los casos es O(log2 n).

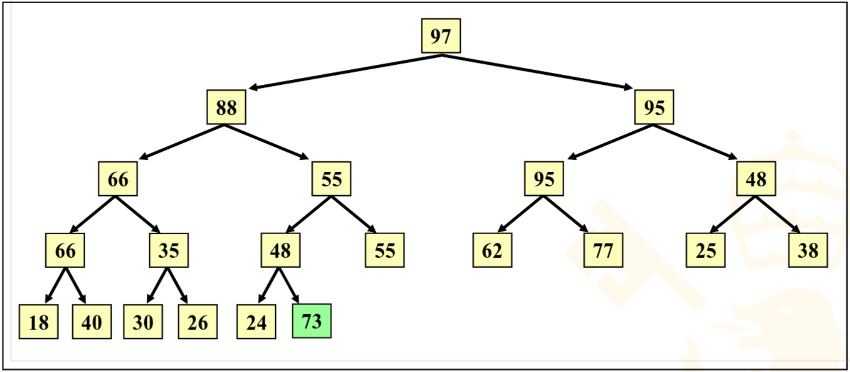
Heaps: Insert clave

INSERTAR NODO 73



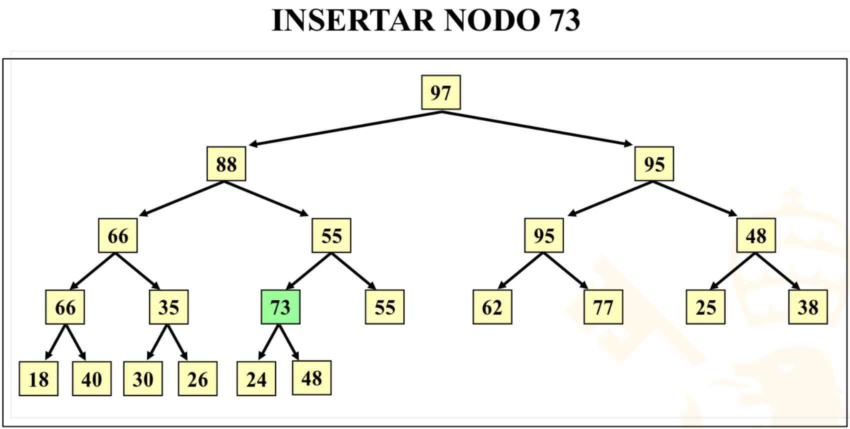
Heaps: Insert clave

INSERTAR NODO 73



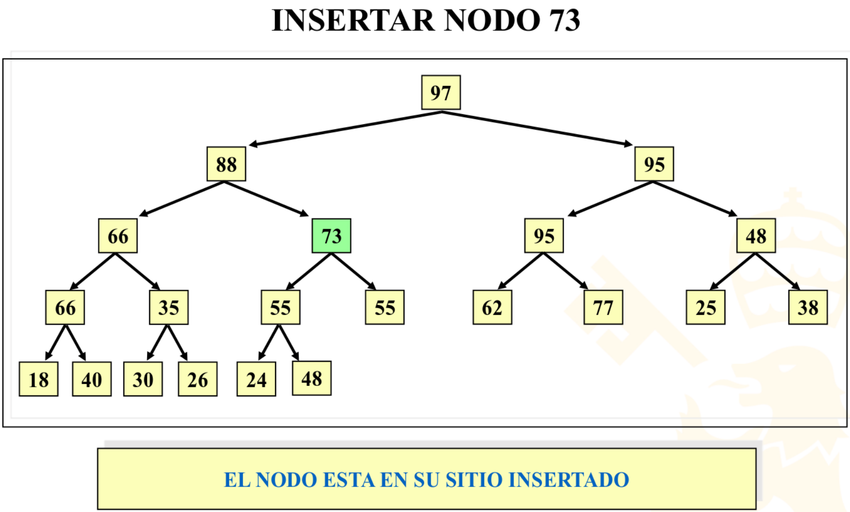
Provisionalmente se inserta al final (en el último nivel)

Heaps: Insert clave



Se comprueba si está bien situado.
Como no lo esta, se intercambia con el padre

Heaps: Insert clave



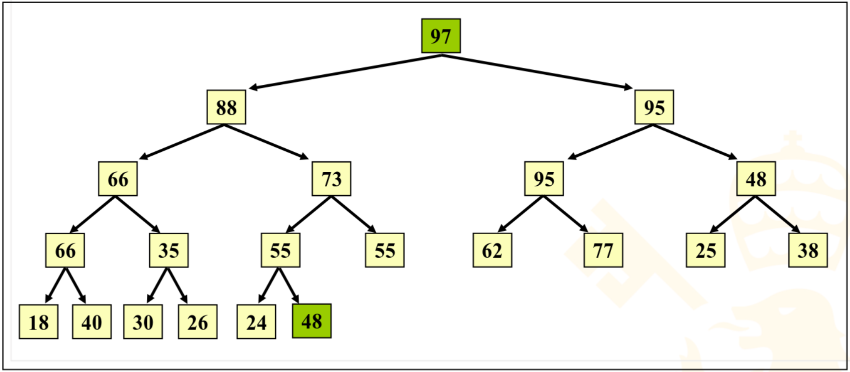
Heaps: Eliminar el nodo mayor: Raiz

- **Los pasos a seguir son:**
  1. Se extrae el elemento raíz del montículo.
  2. Se pone como raíz el último nodo del montículo.
  3. Se comprueba si se rompe la condición de ordenación, de manera que se compara el nodo con sus dos hijos.
  4. Si es menor que alguno de sus dos hijos, se intercambia por el mayor de ellos.
  5. Y así sucesivamente hasta que llegue a su posición correcta.
- Por lo tanto, se puede decir que el nodo que ha pasado a sustituir al raíz al extraerlo SE HUNDE hasta la posición que le corresponda.
- **Complejidad**

  La máxima complejidad de esta operación (el número máximo de pasos) se da cuando se sustituye la raíz por el menor elemento de todo el montículo, por lo que habrá que "hundirle" hasta el último nivel, por lo tanto realizando tantos intercambios como niveles haya menos 1.
  $\Rightarrow$ Por lo tanto, la complejidad es también logarítmica en el peor de los casos, O(log2 n).

Delete nodo en Heap: Secuencia

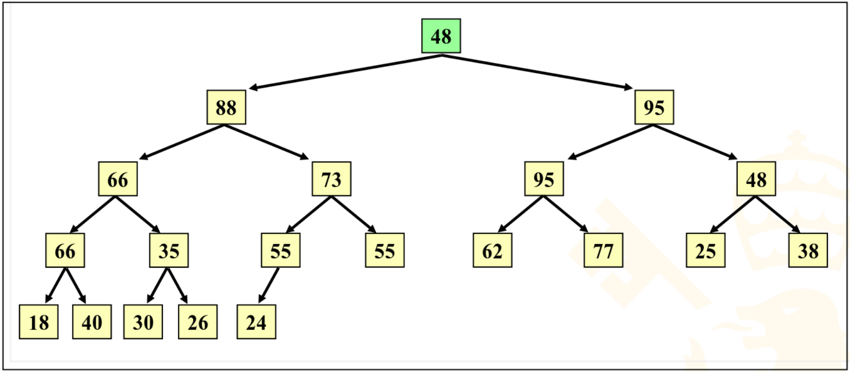
ELIMINAR NODO 97



El nodo Raíz se sustituye por el último nodo

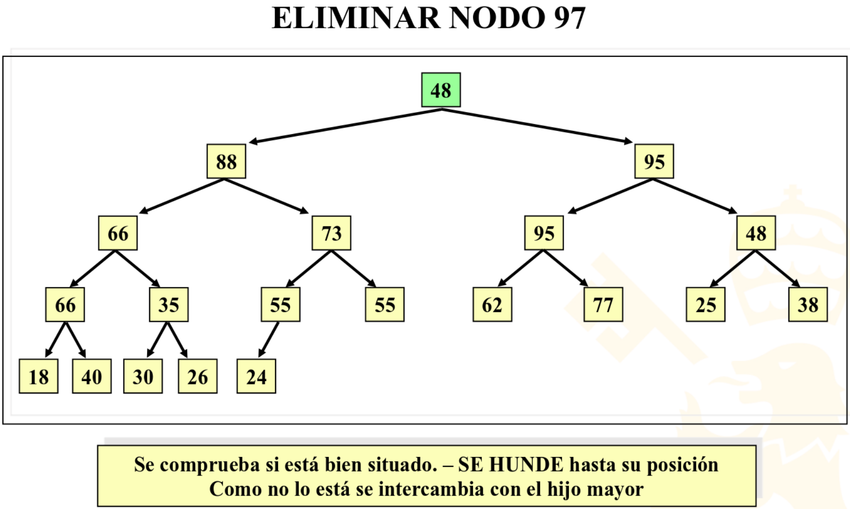
Delete nodo en Heap: Secuencia

ELIMINAR NODO 97



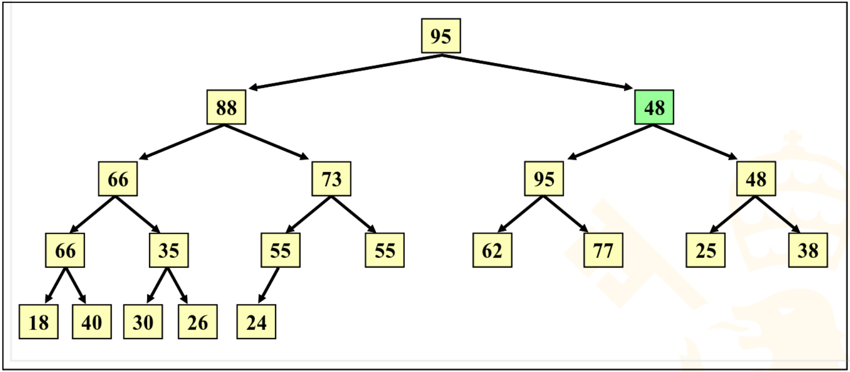
El nodo Raíz se sustituye por el último nodo

Delete nodo en Heap: Secuencia



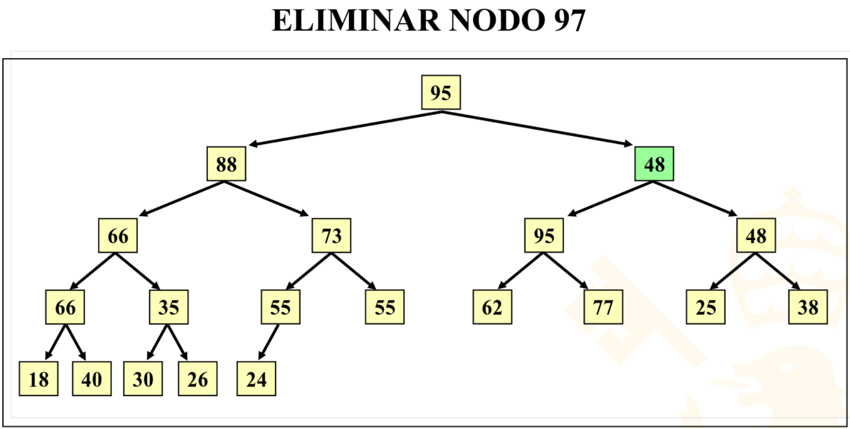
Delete nodo en Heap: Secuencia

ELIMINAR NODO 97



Se comprueba si está bien situado. – SE HUNDE hasta su posición
Como no lo está se intercambia con el hijo mayor

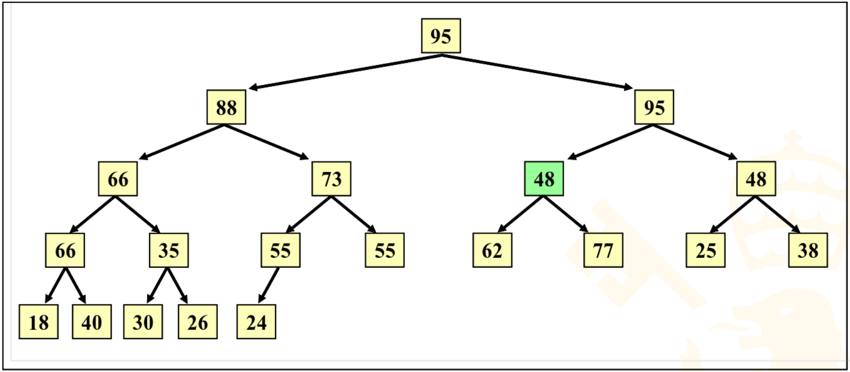
Delete nodo en Heap: Secuencia



Se comprueba si está bien situado.
Como no lo está se intercambia con el hijo mayor

Delete nodo en Heap: Secuencia

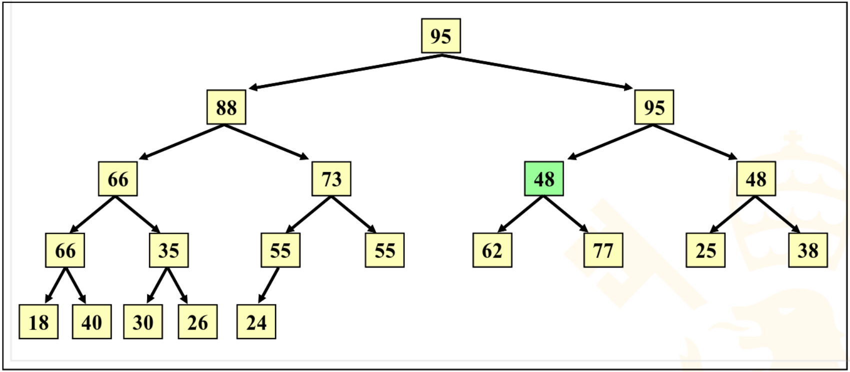
ELIMINAR NODO 97



Se comprueba si está bien situado.
Como no lo está se intercambia con el hijo mayor

Delete nodo en Heap: Secuencia

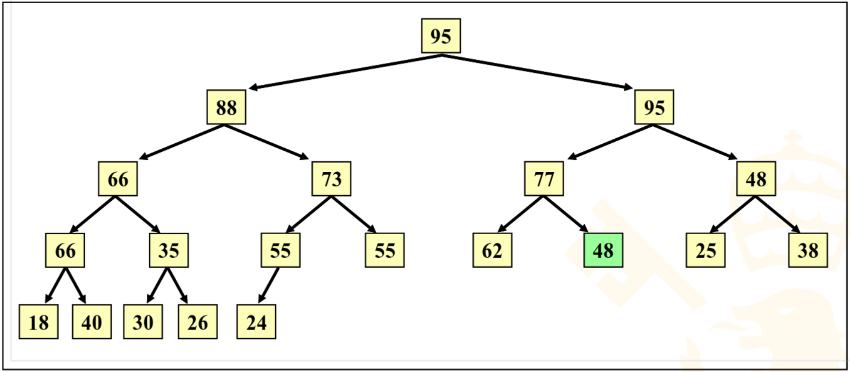
ELIMINAR NODO 97



Se comprueba si está bien situado.
Como no lo está se intercambia con el hijo mayor

Delete nodo en Heap: Secuencia

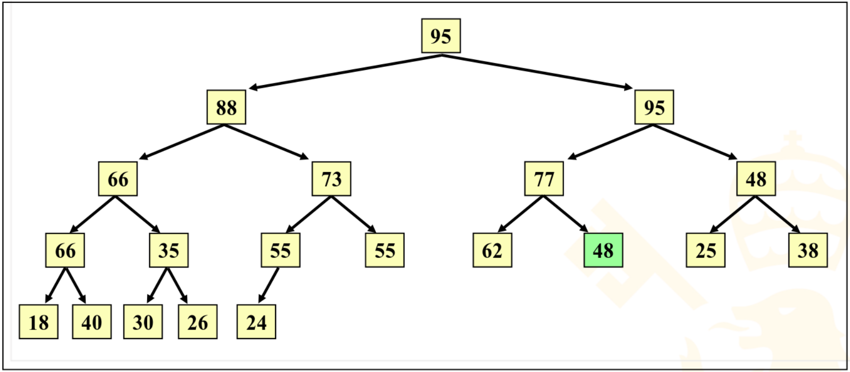
ELIMINAR NODO 97



Se comprueba si está bien situado.
Como no lo está se intercambia con el hijo mayor

Delete nodo en Heap: Secuencia

ELIMINAR NODO 97



EL NODO ESTA EN SU SITIO INSERTADO

Part II

HeapSort

HeapSort I. Ideas clave

HeapSort se base en tres ideas clave:

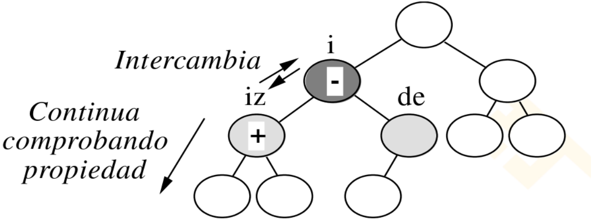
1. Construcción de una estructura de Heap in place, sobre un vector o array de datos. $\Rightarrow$ **BuildHeap** o **CrearMonticulo()**
2. Reconocer que, en todo momento, el máx de los elementos está en la raiz, si se cumple la condición de montículo
3. Manteninimiento de la condición de montículo, tras retirar el primer elemento (max), y ser reemplazado por (ej.) el último $\Rightarrow$ **Monticulizar(), o Heapify()**

COMILLAS ICAI

Mantenimiento de la propiedad de Heap

El poder mantener la propiedad de un montículo en el que únicamente se modifica el nodo raíz es una pieza clave en el algoritmo

- **IDEA:** considerar tres nodos cada vez



- **Suposición**: Subárboles hijos 'iz' y 'de' tienen que ser ya montículos
- **Versión:** recursiva o iterativa

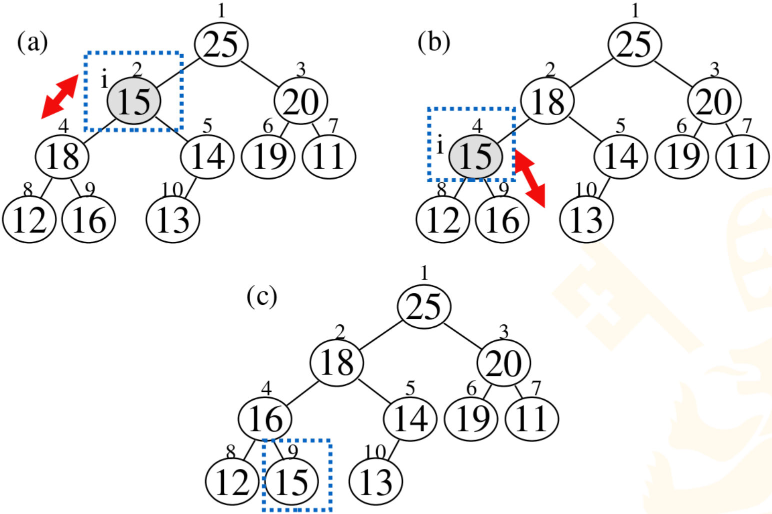
Heapf (a.k.a. "Monticulizar")

Monticulizar el subarbol de la posición i del array arr, de longitud total n

```
function HEAPIFY(arr, n, i)
    left ← 2 ∗ i + 1                                  ▷ Hijo Izquierdo
    right ← 2 ∗ i + 2                                 ▷ Hijo Derecho
    largest = i
    if (left < n) & (arr[largest] < arr[left]) then
        largest = left
    if (right < n) & (arr[largest] < arr[right]) then
        largest = right

    if largest ≠ i then                  ▷ Intercambiar hijo y padre, si necesario
        Swap arr[i] ↔ arr[largest]
        HEAPIFY(arr,n,largest)           ▷ Monticuliza en la nueva posición
    return
```

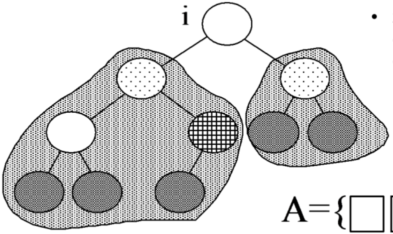
Heapify (Monticulizar) Ejemplo



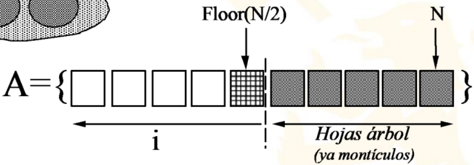
Heap: Build Algorithm

HeapBuild

function HEAPBUILD(arr,n)
    ▷ *Convertir arr en un monticulo, in place* ◁
    for $i = \lfloor n//2 \rfloor$ -1 to 0 by -1 **do** ▷ *empieza en el centro-izquierda*
        HEAPIFY(arr, n, i) ▷ *heapify starting at i hasta N*
    return

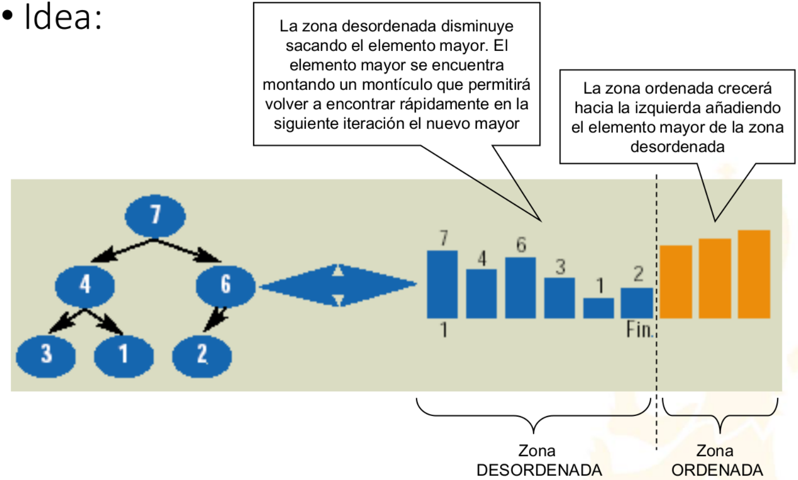


- Se construye de abajo a arriba (árbol) o de derecha a izquierda (array), garantizándose que los subárboles de los hijos son ya montículos



HeapSort: Idea

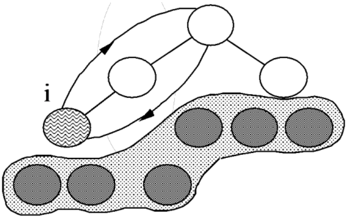
- Idea:

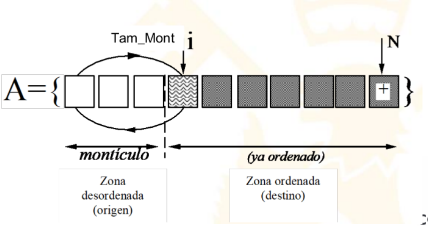


HeapSort: Algorithm

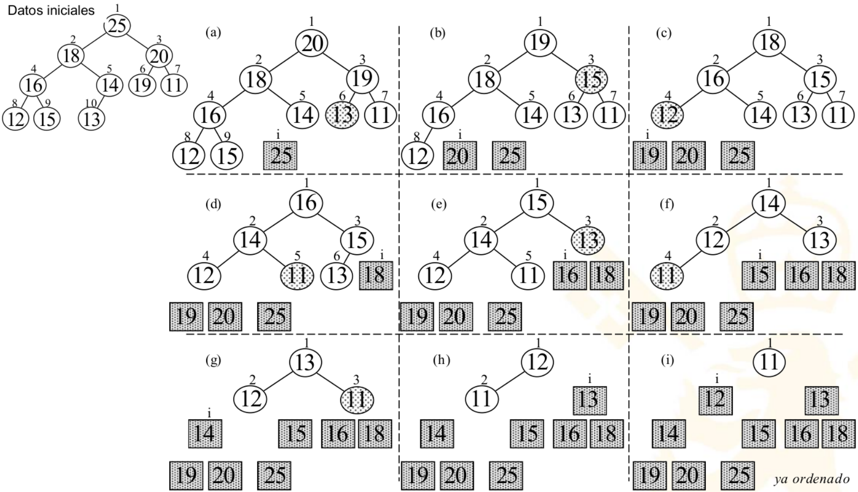
HeapSort

```
function HEAPSORT(arr)
    n ← length of arr
    ▷ Convertir arr en un monticulo, in place ◁
    HEAPBUILD(arr,n)
    ▷ One by one extract elements ◁
    for i = n − 1 to 1 by -1 do                 ▷ in reverse, until second element
        Swap arr[i] ↔ arr[0]                     ▷ swap el i con el 0
        HEAPIFY(arr, i, 0)                       ▷ heapify [0: i-1], empezando en 0
    return
```





HeapSort Example



HeapSort: Complejidad, Pros and Cons

Complejidad

Complejidad temporal: $\mathbf{O}(n\log(n))$
Complejidad Espacial:$\mathbf{O}(log(n))$, debido a la pila de llamadas recursiva.

- **Puntos importantes sobre Heap Sort:**
  - Su implementación típica no es, en general, un algoritmo estable
  - Normalmente es 2 o 3 veces más lento que QuickSort
- **Ventajas de la ordenación HeapSort**
  - **Complejidad temporal eficiente:** Heap Sort tiene una complejidad temporal de O (n log n) en todos los casos.
  - **Uso de la memoria:** el uso de la memoria puede ser mínimo (escribiendo un heapify() iterativo en lugar de uno recursivo).
  - **Simplicidad:** es más sencillo de entender que otros algoritmos al no ser recursivo
- **Desventajas de la ordenación HeapSort:**
  - **Costoso:** la ordenación Heapsort no es la más rápidad
  - **Inestable** la ordenación HeapSort no respeta el orden original a

https://www.youtube.com/watch?v=EreoMaOBTzE

HeapSort Práctica: Entrega Domingo 14/abril

Dado un vector de números de longitud arbitraria, procede en los siguientes paso:

1. Implementa la función **heapify**
2. Implementa la función **heapBuild**
3. Convierte el vector o array de números en un vector de tipo Montículo o heap. Visualiza como vector
4. **[Opcional]** Visualiza como árbol binario.[3]
5. Implementa la función **HeapSort**, utilizando las funciones anteriores
6. Aplícala sobre el vector y comprueba su funcionamiento
7. Aumenta el tamaño, de forma gradual, mide tiempos y evalua la escalabilidad del algoritmo heapSort. Verifica es que $\mathbf{O}(n \log(n))$
8. **[Opcional]** Implementa una cola de prioridad usando una estructura heap. [4]

[3] usa la función plot provista en prácticas anteriores

[4] Necesitarás implementar un insert generico y un remove del root, para implementar el push() y el pop()