Algoritmos y Estructuras de Datos II

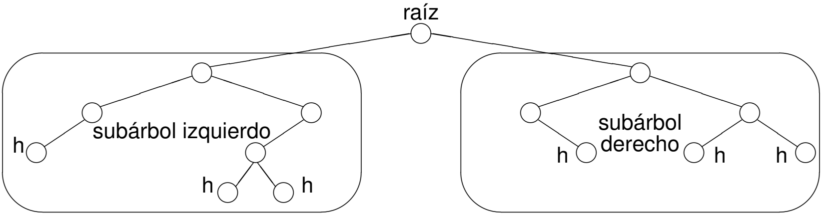
Heaps

20 de abril de 2016

Clase de hoy

1. Repaso
   - Árboles binarios
   - ABBs
   - TAD conjunto
2. Heaps
   - Ejemplos y definiciones
   - Implementación en un arreglo
   - Operaciones de heap
   - Implementación de cola de prioridades usando heaps
3. Heapsort

Intuición



Todos los árboles pueden construirse con los constructores

- $<>$, que construye un árbol vacío
- $< \_, \_, \_ >$, que construye un árbol no vacío a partir de un elemento y dos subárboles

Sobre los niveles

- En el nivel 0 hay a lo sumo 1 nodo.
- En el nivel 1 hay a lo sumo 2 nodos.
- En el nivel 2 hay a lo sumo 4 nodos.
- En el nivel 3 hay a lo sumo 8 nodos.
- En el nivel i hay a lo sumo 2^i nodos.
- En un árbol de altura n hay a lo sumo $2^0 + 2^1 + \ldots + 2^n = 2^{n+1} - 1$ nodos.
- En un árbol "balanceado" la altura es del orden del $\log_2 k$ donde k es el número de nodos.

Árboles binarios de búsqueda

Decimos que t es un **árbol binario de búsqueda** o **ABB**, cuando se cumplen las siguientes condiciones:

- los valores alojados en el subárbol izquierdo de t son menores que el alojado en la raíz de t,
- los valores alojados en el subárbol derecho de t son mayores que el alojado en la raíz de t,
- las dos condiciones anteriores se cumplen para todos los subárboles de t.

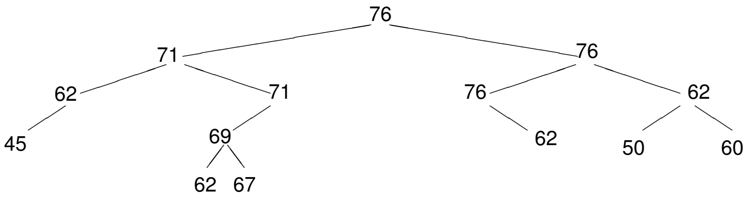
TAD conjunto

- Usando árboles binarios de búsqueda, hemos implementado el TAD conjunto con operaciones:
  - search (implementación de la operación está) de orden $\log n$,
  - insert (implementación de la operación agregar) de orden $\log n$,
  - delete (implementación de la operación borrar) de orden $\log n$,
  - las otras dos operaciones (empty, is_empty) son constantes.
- (asumiendo que el árbol binario de búsqueda está balanceado)

Heaps

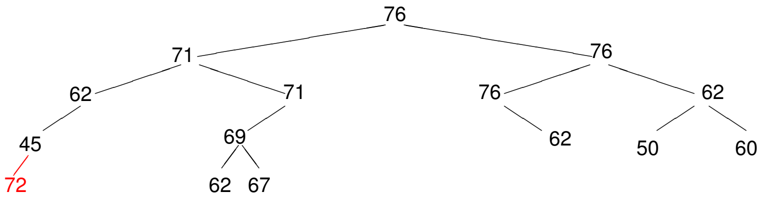
- Los heaps se asemejan a los ABBs en
  - que son árboles binarios
  - con una manera particular de organizar la información de sus nodos
- y se diferencia de los ABBs en
  - que admite repeticiones
  - la forma de organizar la información
    - en cada nodo del heap, la información es mayor o igual a la de sus descendientes
  - el heap se implementa muy convenientemente en arreglos

Ejemplo

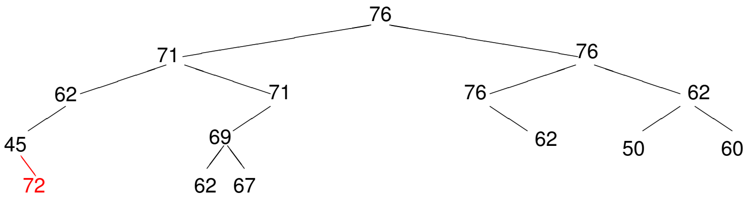


¿Es un heap?
Supongamos que queremos agregar el 72. ¿Dónde lo agregamos?

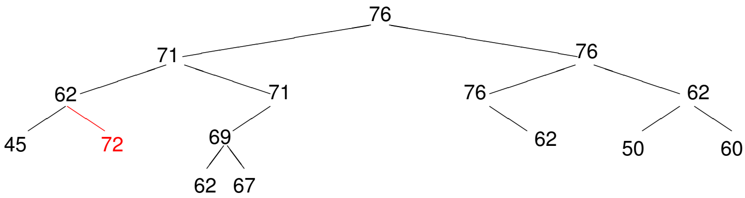
Ejemplo



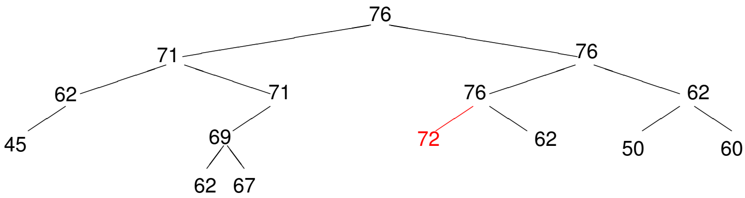
Ejemplo



Ejemplo

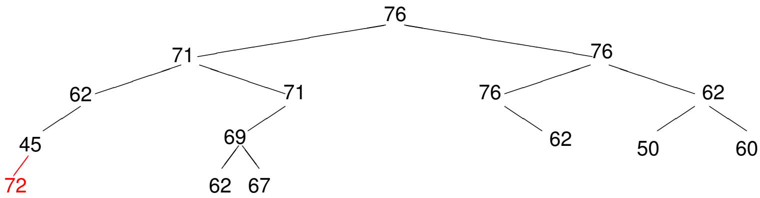


Ejemplo



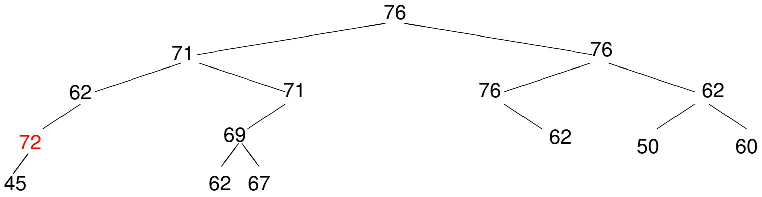
Éste es el único caso en que sigue siendo un heap.
Pero en general, puede que no exista esta posibilidad, por ejemplo, si el número a insertar es el 80.

Ejemplo



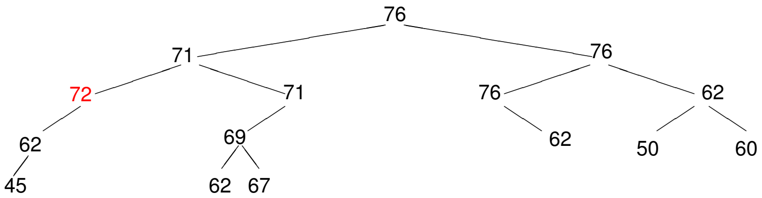
Probemos entonces insertar el 72 en “cualquier lado”.
No es un heap porque el 72 es mayor que su padre, el 45.
Los intercambiamos.

Ejemplo



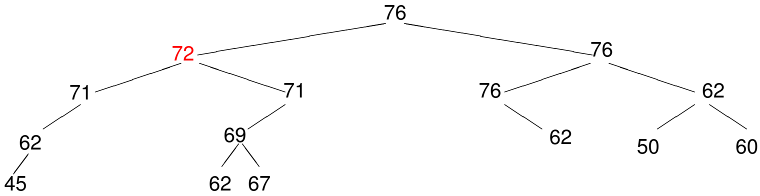
Sigue sin ser un heap porque el 72 es mayor que su padre, el 62.
Lo intercambiamos.

Ejemplo



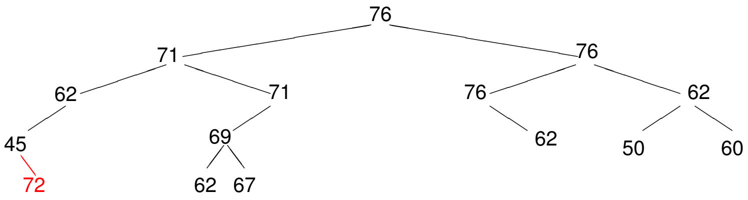
Sigue sin ser un heap porque el 72 es mayor que su padre, el 71.
Lo intercambiamos.

Ejemplo



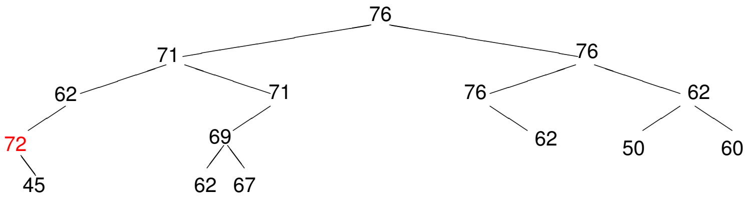
Ahora sí es un heap.

Ejemplo



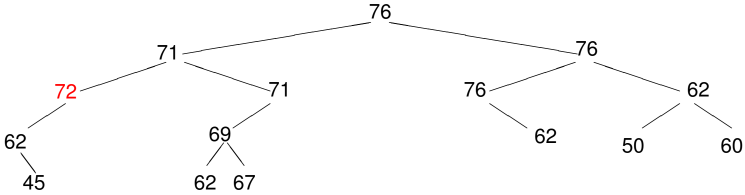
¿Qué pasaba si lo insertábamos acá?
No es un heap porque el 72 es mayor que su padre, el 45.
Los intercambiamos.

Ejemplo



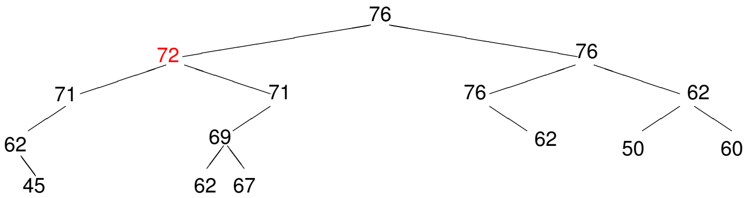
Sigue sin ser un heap porque el 72 es mayor que su padre, el 62.
Lo intercambiamos.

Ejemplo



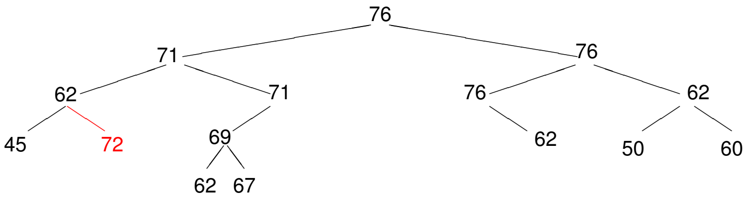
Sigue sin ser un heap porque el 72 es mayor que su padre, el 71.
Lo intercambiamos.

Ejemplo



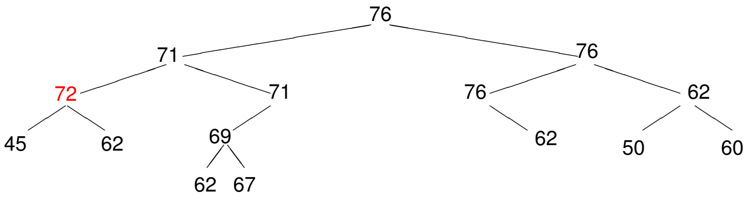
Ahora sí es un heap.

Ejemplo



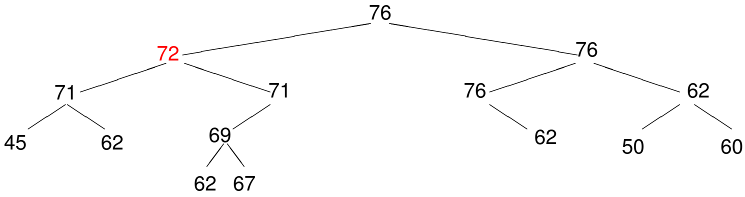
¿Qué pasaba si lo insertábamos acá?
No es un heap porque el 72 es mayor que su padre, el 62.
Los intercambiamos.

Ejemplo



Sigue sin ser un heap porque el 72 es mayor que su padre, el 71.
Lo intercambiamos.

Ejemplo



Ahora sí es un heap.

Conclusión

- En todos los casos se logra **restablecer la condición de heap** en log n intercambios.
- **Se puede elegir** libremente donde comenzar.
- Eso **determina la forma** del heap resultante.
- Luego hay que hacer **flotar** el nuevo elemento realizando los intercambios que sean necesarios, la forma del heap ya no cambia.
- Idea: elegir de modo de que se mantenga balanceado, llenando **nivel por nivel**.

Ejemplo

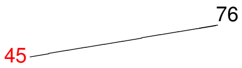
- A continuación mostraremos con un ejemplo cómo se puede ir llenando nivel por nivel.
- Sea la siguiente secuencia de números que se insertan en un heap inicialmente vacío.
- 76, 45, 80, 60, 69, 78, 40, 78, 73

Ejemplo

76

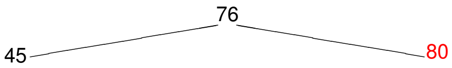
76, 45, 80, 60, 69, 78, 40, 78, 73

Ejemplo

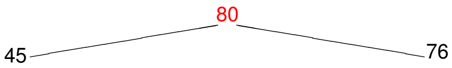


76, 45, 80, 60, 69, 78, 40, 78, 73

Ejemplo

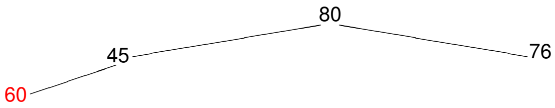


Ejemplo

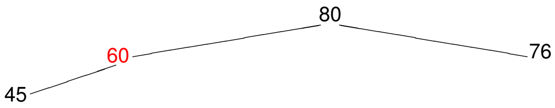


76, 45, 80, 60, 69, 78, 40, 78, 73

Ejemplo

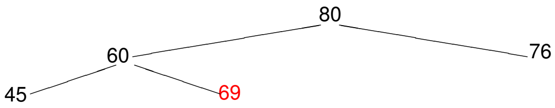


Ejemplo

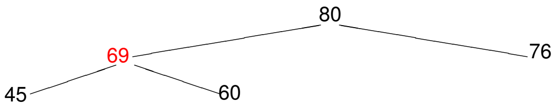


76, 45, 80, 60, 69, 78, 40, 78, 73

Ejemplo

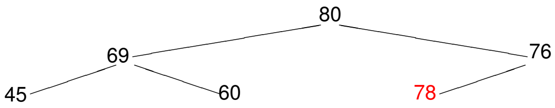


Ejemplo

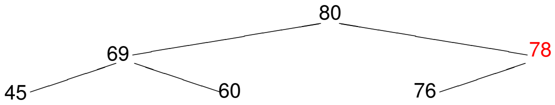


76, 45, 80, 60, 69, 78, 40, 78, 73

Ejemplo

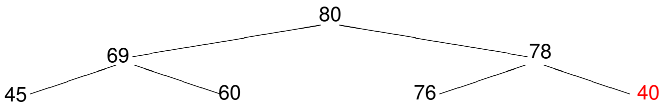


Ejemplo



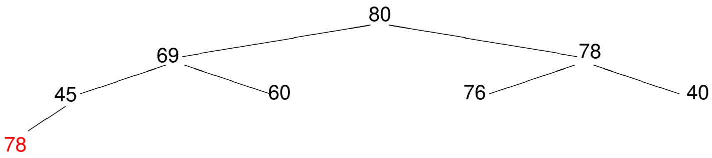
76, 45, 80, 60, 69, 78, 40, 78, 73

Ejemplo

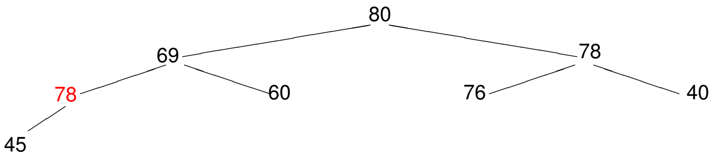


76, 45, 80, 60, 69, 78, 40, 78, 73

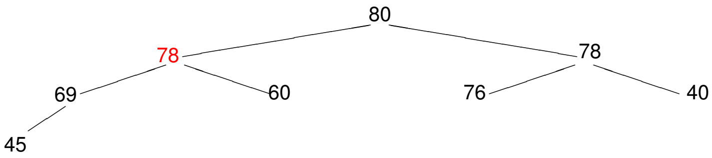
Ejemplo



Ejemplo

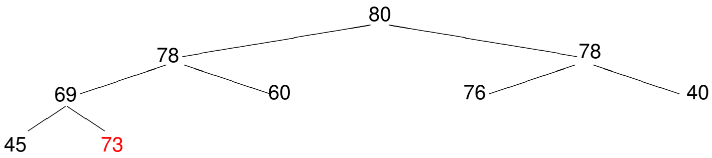


Ejemplo

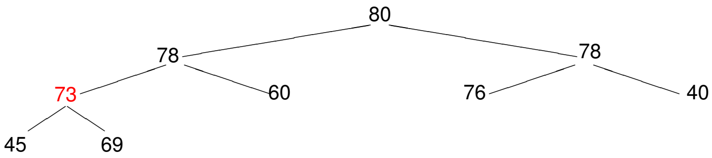


76, 45, 80, 60, 69, 78, 40, 78, 73

Ejemplo



Ejemplo



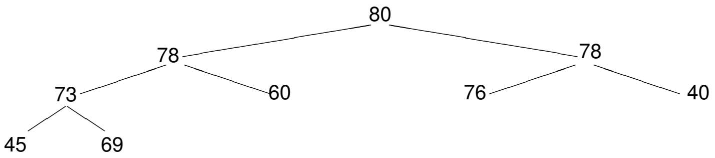
Conclusión

- Tenemos una forma de ir insertando elementos de modo que el árbol quede perfectamente balanceado.
- El algoritmo de inserción de cada elemento es $\log n$.

Implementando cola de prioridades

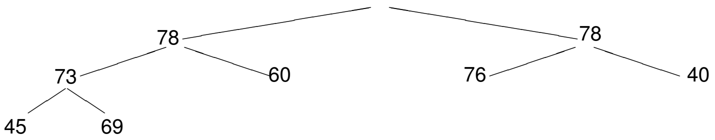
- Esto mejora las implementaciones anteriores de colas de prioridades.
  - inserción era constante
  - ver el primero y eliminar el primero eran lineales
  - o viceversa
- ahora inserción es log n
- ver el primero es constante
- ¿y borrar el primero?
  - veremos que se puede hacer log n

Ejemplo



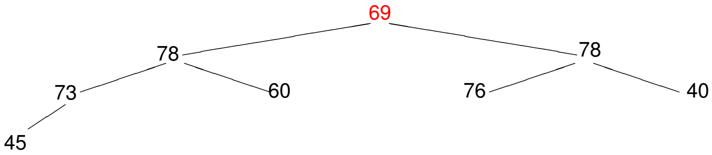
borremos el primero, o sea el 80

Ejemplo



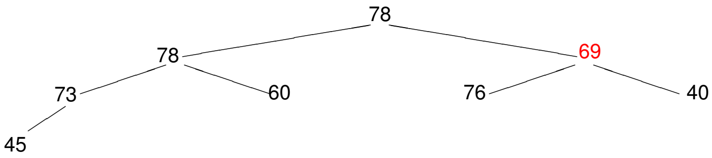
¿cómo hacemos para que nos quede un heap?
llevamos una hoja arriba

Ejemplo



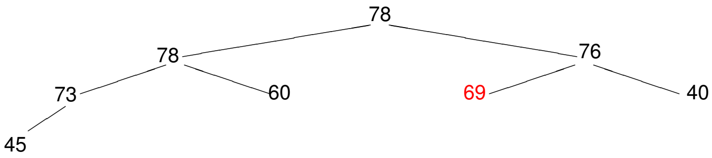
¿cómo hacemos para que nos quede un heap?
la **hundimos** intercambiandola con el mayor de sus hijos

Ejemplo



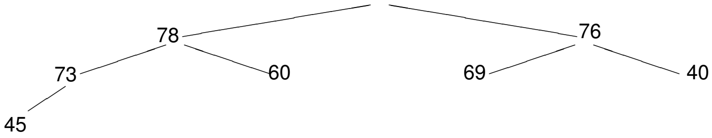
la **hundimos** intercambiandola con el mayor de sus hijos

Ejemplo



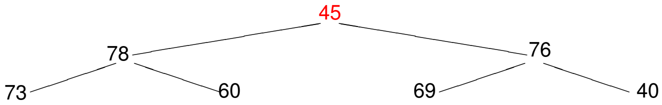
¡listo!
hundir es log n
el primero ahora es 78, borrémoslo

Ejemplo



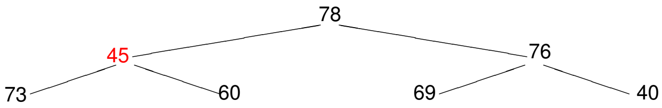
¿cómo hacemos para que nos quede un heap?
llevamos una hoja arriba

Ejemplo



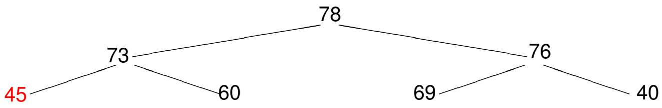
¿cómo hacemos para que nos quede un heap?
la **hundimos** intercambiandola con el mayor de sus hijos

Ejemplo



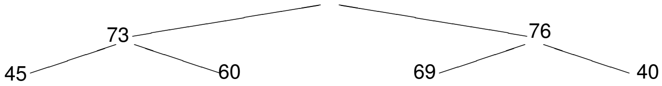
la **hundimos** intercambiandola con el mayor de sus hijos

Ejemplo

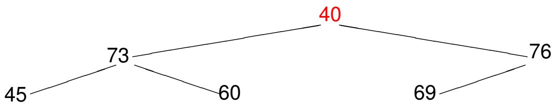


¡listo!

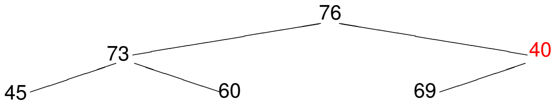
Ejemplo



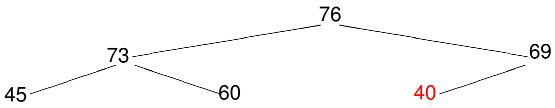
Ejemplo



Ejemplo



Ejemplo



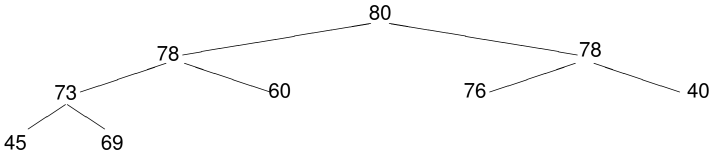
etcétera

Implementación en un arreglo

- Tener tanto control sobre la forma del heap,
- podemos asegurarnos de que se va llenando nivel por nivel,
- y se van borrando exactamente en orden inverso.
- Por ello, en todo momento se tienen los primeros $i - 1$ niveles llenos,
- y el nivel i llenándose de izquierda a derecha.

Esto permite implementar el heap en un arreglo.

Ejemplo

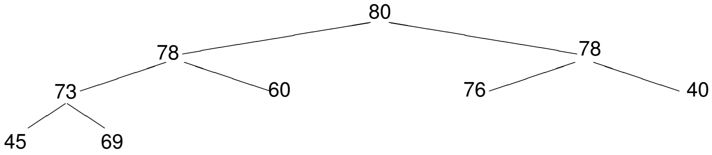


Se implementa en un arreglo de la siguiente manera:

1	2	3	4	5	6	7	8	9	10	11	12	13	14
80	78	78	73	60	76	40	45	69					

y hace falta un natural que informe el tamaño del heap.

Ejemplo



1	2	3	4	5	6	7	8	9	10	11	12	13	14
80	78	78	73	60	76	40	45	69					

- Observar que los hijos del elemento que se encuentra en la posición i del arreglo, se encuentran en las posiciones $2i$ y $2i + 1$.
- El padre del elemento que se encuentra en la posición i, se encuentra en la posición $i \div 2$.

Implementación de heap

```
type heap = tuple
              elems: array[1..n] of elem
              size: nat
            end
```

Funciones para encontrar los hijos

fun left(i:**nat**) **ret** j:**nat**
    j:= 2*i
end

fun right(i:**nat**) **ret** j:**nat**
    j:= 2*i+1
end

Función para encontrar el padre

fun parent(i:**nat**) **ret** j:**nat**
    j:= i $\div$ 2
end

Funciones booleanas

{Pre: 1 $\leq$i$\leq$ h.size}
fun has_children(h:heap, i:**nat**) **ret** b:**bool**
    b:= (left(i) $\leq$ h.size)
end
{Post: b = i tiene hijos en h}

fun has_parent(i:**nat**) **ret** b:**bool**
    b:= (i $\neq$ 1)
end

Máximo de los hijos

{Pre: $1 \leq$ i $\leq$ h.size $\wedge$ has_children(h,i)}
fun max_child(h:heap, i:**nat**) **ret** j:**nat**
    if right(i) $\leq$ h.size $\wedge$ h.elems[left(i)] $\leq$ h.elems[right(i)] **then** j:= right(i)
    else j:= left(i)
    fi
end
{Post: j = posición donde se encuentra el mayor de los hijos de i en h}

Ascenso de un elemento

{Pre: 1 $\leq$ i $\leq$ h.size $\wedge$ has_parent(i)}
proc lift(**in**/**out** h:heap,**in** i:**nat**)
    swap(h.elems,i,parent(i))
end

{Pre: 1 $\leq$ i $\leq$ h.size $\wedge$ has_parent(i)}
fun must_lift(h:heap, i:**nat**) **ret** b:**bool**
    b:= (h.elems[i] > h.elems[parent(i)])
end
{Post: b = i es mayor que su padre}

Flotar un elemento

{Pre: h (= H) es heap excepto tal vez porque el elem en h.elems[h.size] es grande}
proc float(**in**/**out** h:heap)
    var c: **nat**
    c:= h.size
    while has_parent(c) $\wedge$ must_lift(h,c) **do**
        lift(h,c)
        c:= parent(c)
    od
end
{Post: h es un heap con los mismos elementos que H}

Hundir un elemento

{Pre: h (= H) es heap excepto tal vez porque el elem en 1 es chico}
proc sink(**in**/**out** h:heap)
    var p: **nat**
    p:= 1
    while has_children(h,p) ∧ must_lift(h,max_child(h,p)) **do**
        p:= max_child(h,p)
        lift(h,p)
    od
end
{Post: h es un heap con los mismos elementos que H}

Implementación de cola de prioridades

```
type pqueue = heap

proc empty(out q:pqueue)
      q.size:= 0
end
{Post: q ~ Vacía}

{Pre: q ~ Q}
fun is_empty(q:pqueue) ret b:bool
    b:= (q.size = 0)
end
{Post: b ~ es_vacía Q}
```

Encolar

{Pre: q $\sim$ Q $\wedge$ q.size < n}
proc enqueue(**in**/**out** q:pqueue,**in** e:elem)
    q.size:= q.size+1
    q.elems[q.size]:= e
    float(q)
end
{Post: q $\sim$ encolar Q e}

Primero

$\{$Pre: q $\sim$ Q $\wedge \neg$is_empty(q)$\}$
fun first(q:pqueue) **ret** e:elem
    e:= q.elems[1]
end
$\{$Post: e $\sim$ primero Q$\}$

Decolar

{Pre: q $\sim$ Q $\wedge\neg$is_empty(q)}
proc dequeue(**in**/**out** q:pqueue)
    q.elems[1]:= q.elems[q.size]
    q.size:= q.size-1
    sink(q)
end
{Post: q $\sim$ decolar Q}

Conclusiones

Hemos implementado cola de prioridades con heaps:

- Vacía es constante,
- Encolar es log n,
- es_vacía es constante,
- primero es constante, y
- decolar es log n.

Heapsort

- Ya vimos un algoritmo de ordenación que llamamos pqueue_sort
- utilizaba una cola de prioridades para ordenar del siguiente modo
  - una primera fase en que todos los elementos del arreglo se introducen en la cola de prioridades
  - una segunda fase en que todos los elementos van saliendo de la cola y ubicándose en la celda correcta del arreglo
- ahora sabemos que una cola de prioridades se implementa eficientemente con un heap
- más aun, el heap_sort utiliza el propio arreglo a ordenar para ir guardando los elementos del heap
- por ello, no necesita espacio auxiliar.

Heapsort

Preliminares

- Separamos el heap en sus dos componentes: arreglo a y tamaño i (ó i-1).
- Por ello, los procedimientos float y sink recibirán las dos componentes por separado.
- No se usan arreglos auxiliares: se utiliza el mismo arreglo a ordenar como heap.

Heapsort: el algoritmo

```
proc heap_sort(in/out a:array[1..n] of elem)
    for i:= 1 to n do
        float(a,i)
    od
    for i:= n downto 1 do
        swap(a,1,i)                                  {a[i]:= first}
        sink(a,i-1)
    od
end
```

El heapsort suele presentarse sin funciones y procedimientos auxiliares, como en la filmina que sigue.

Heapsort

```
proc heap_sort(in/out a:array[1..n] of elem)
    var p,c,r: nat                         {p = parent, c = (left) child, r = right child}
    for i:= 1 to n do
        c:= i                                                   {comienza enqueue(a[i])}
        p:= i ÷ 2
        while c ≠ 1 ∧ a[c] > a[p] do
            swap(a,c,p)
            c:= p
            p:= p÷2
        od                                                       {termina enqueue(a[i])}
    od
    for i:= n downto 1 do
        swap(a,1,i)                                       {a[i]:= first, comienza dequeue}
        p:= 1
        c:= 2
        r:= min(3,i-1)
        while c < i ∧ (a[p] < max(a[c],a[r])) do
            if a[c] ≤ a[r] then swap(a,r,p)
                                p:= r
            else swap(a,c,p)
                    p:= c
            fi
            c:= 2∗p
            r:= min(2∗p+1,i-1)
        od
    od
end
```