

# Algorítmica: Heaps y heapsort

Conrado Martínez  
U. Politècnica Catalunya

Q1-2011-2012

Una **cola de prioridad** (cat: *cua de prioritat*; ing: *priority queue*) es una colección de elementos donde cada elemento tiene asociado un valor susceptible de ordenación denominado **prioridad**.

Una cola de prioridad se caracteriza por admitir inserciones de nuevos elementos y la consulta y eliminación del elemento de mínima (o máxima) prioridad.

```
template <typename Elem, typename Prio>
class ColaPrioridad {
public:
    ...
    // Añade el elemento x con prioridad p a la cola de
    // prioridad.
    void inserta(const Elem& x, const Prio& p) throw(error)

    // Devuelve un elemento de mínima prioridad en la cola de
    // prioridad. Se lanza un error si la cola está vacía.
    Elem min() const throw(error);

    // Devuelve la mínima prioridad presente en la cola de
    // prioridad. Se lanza un error si la cola está vacía.
    Prio prio_min() const throw(error);

    // Elimina un elemento de mínima prioridad de la cola de
    // prioridad. Se lanza un error si la cola está vacía.
    void elim_min() throw(error);

    // Devuelve cierto si y sólo si la cola está vacía.
    bool vacia() const throw();
};
```

```
// Tenemos dos arrays Peso y Simb con los pesos atómicos
// y símbolos de n elementos químicos,
// p.e., Simb[i] = "C" y Peso[i] = 12.2.
// Utilizamos una cola de prioridad para ordenar la
// información de menor a mayor símbolo en orden alfabético
ColaPrioridad<double, string> P;
for (int i = 0; i < n; ++i)
    P.inserta(Peso[i], Simb[i]);
int i = 0;
while(! P.vacia()) {
    Peso[i] = P.min();
    Simb[i] = P.prio_min();
    ++i;
    P.elim_min();
}
```

Se puede usar una cola de prioridad para hallar el  $k$ -ésimo elemento de un vector no ordenado. Se colocan los  $k$  primeros elementos del vector en una max-cola de prioridad y a continuación se recorre el resto del vector, actualizando la cola de prioridad cada vez que el elemento es menor que el mayor de los elementos de la cola, eliminando al máximo e insertando el elemento en curso.

- Muchas de las técnicas empleadas para la implementación de diccionarios puede usarse para implementar colas de prioridad (no las tablas de hash ni los tries)
- P.e., con árboles binarios de búsqueda equilibrados se puede conseguir coste  $\mathcal{O}(\log n)$  para inserciones y eliminaciones

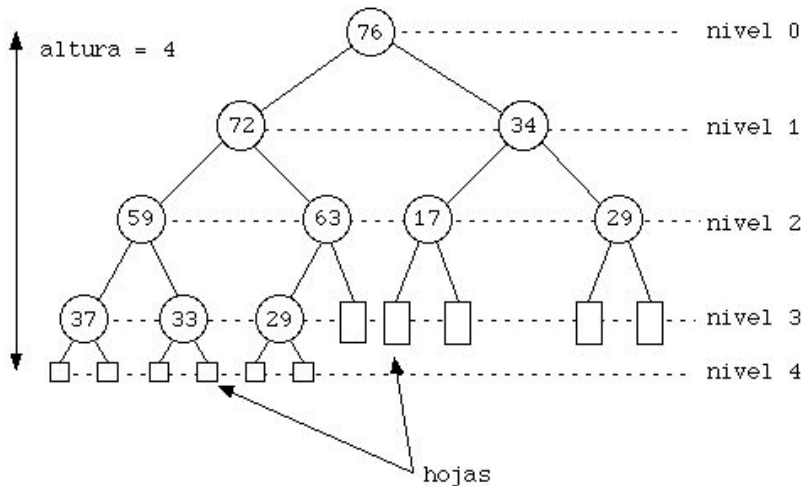
## Definición

Un *montículo* (ing: **heap**) es un árbol binario tal que

- 1 todos las hojas (subárboles son vacíos) se sitúan en los dos últimos niveles del árbol.
- 2 en el antepenúltimo nivel existe a lo sumo un nodo interno con un sólo hijo, que será su hijo izquierdo, y todos los nodos a su derecha en el mismo nivel son nodos internos sin hijos.
- 3 el elemento (su prioridad) almacenado en un nodo cualquiera es mayor (menor) o igual que los elementos almacenados en sus hijos izquierdo y derecho.

Se dice que un montículo es un árbol binario quasi-completo debido a las propiedades 1-2. La propiedad 3 se denomina **orden de montículo**, y se habla de **max-heaps** o **min-heaps** según que los elementos sean  $\geq$  ó  $\leq$  que sus hijos.



$$n = 10$$


## Proposición

- 1 *El elemento máximo de un max-heap se encuentra en la raíz.*
- 2 *Un heap de  $n$  elementos tiene altura*

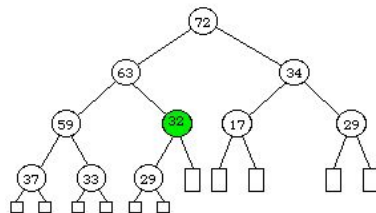
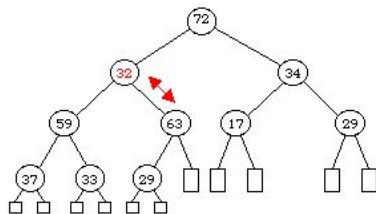
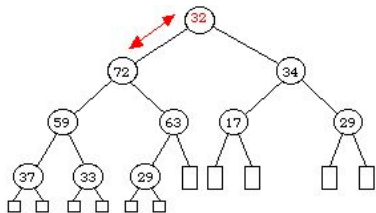
$$h = \lceil \log_2(n + 1) \rceil.$$

La consulta del máximo es sencilla y eficiente pues basta examinar la raíz.

## Cómo eliminar el máximo?

- 1 Ubicar al último elemento del montículo (el del último nivel más a la derecha) en la raíz, sustituyendo al máximo
- 2 Reestablecer el invariante (orden de heap) **hundiendo** la raíz.

El método `hundir` intercambia un nodo dado con el mayor de sus dos hijos si el nodo es menor que alguno de ellos, y repete este paso hasta que el invariante de heap se ha reestablecido



## Cómo añadir un nuevo elemento?

- 1 Colocar el nuevo elemento como último elemento del montículo, justo a la derecha del último o como primero de un nuevo nivel
- 2 Reestablecer el orden de monticulo **flotando** el elemento recién añadido

En el método `flotar` el nodo dado se compara con su nodo padre y se realiza el intercambio si éste es mayor que el padre, iterando este paso mientras sea necesario

Puesto que la altura del *heap* es  $\Theta(\log n)$  el coste de inserciones y eliminaciones será  $\mathcal{O}(\log n)$ .

Se puede implementar un *heap* mediante memoria dinámica, con apuntadores al hijo izquierdo y derecho y **también** al padre en cada nodo

Pero es mucho más fácil y eficiente implementar los *heaps* mediante un vector. No se desperdicia demasiado espacio ya que el *heap* es quasi-completo; en caso necesario puede usarse el redimensionado

Reglas para representar un *heap* en vector:

- 1  $A[1]$  contiene la raíz.
- 2 Si  $2i \leq n$  entonces  $A[2i]$  contiene al hijo izquierdo del elemento en  $A[i]$  y si  $2i + 1 \leq n$  entonces  $A[2i + 1]$  contiene al hijo derecho de  $A[i]$
- 3 Si  $i \geq 2$  entonces  $A[i/2]$  contiene al padre de  $A[i]$

Las reglas anteriores implican que los elementos del heap se ubican en posiciones consecutivas del vector, colocando la raíz en la primera posición y recorriendo el árbol por niveles, de izquierda a derecha.

```
template <typename Elem, typename Prio>
class ColaPrioridad {
public:
    ...
private:
    // la componente 0 no se usa; el constructor de la clase
    // inserta un elemento ficticio
    vector<pair<Elem, Prio> > h;

    int nelems;

    void flotar(int j) throw();
    void hundir(int j) throw();
};
```



```
template <typename Elem, typename Prio>
bool ColaPrioridad<Elem,Prio>::vacia() const throw() {

    return nelems == 0;
}

template <typename Elem, typename Prio>
Elem ColaPrioridad<Elem,Prio>::min() const throw(error) {

    if (nelems == 0) throw error(ColaVacía);
    return h[1].first;
}

template <typename Elem, typename Prio>
Prio ColaPrioridad<Elem,Prio>::prio_min() const throw(error) {

    if (nelems == 0) throw error(ColaVacía);
    return h[1].second;
}
```

```
template <typename Elem, typename Prio>
void ColaPrioridad<Elem,Prio>::inserta(const Elem& x,
                                     const Prio& p) throw(error) {
    ++nelems;
    h.push_back(make_pair(x, p));
    flotar(nelems);
}

template <typename Elem, typename Prio>
void ColaPrioridad<Elem,Prio>::elim_min() const throw(error) {

    if (nelems == 0) throw error(ColaVacía);
    swap(h[1], h[nelems]);
    --nelems;
    hundir(1);
}
```

```

// Versión recursiva.
// Hunde el elemento en la posición j del heap
// hasta reestablecer el orden del heap; por
// hipótesis los subárboles del nodo j son heaps.

// Coste: O(log(n/j))
template <typename Elem, typename Prio>
void ColaPrioridad<Elem,Prio>::hundir(int j) throw() {

    // si j no tiene hijo izquierdo, hemos terminado
    if (2 * j > nelems) return;

    int minhijo = 2 * j;
    if (minhijo < nelems &&
        h[minhijo].second > h[minhijo + 1].second)
        ++minhijo;

    // minhijo apunta al hijo de minima prioridad de j
    // si la prioridad de j es mayor que la de su menor hijo
    // intercambiar y seguir hundiendo
    if (h[j].second > h[minhijo].second) {
        swap(h[j], h[minhijo]);
        hundir(minhijo);
    }
}

```

```

// Versión iterativa.
// Hunde el elemento en la posición j del heap
// hasta reestablecer el orden del heap; por
// hipótesis los subárboles del nodo j son heaps.

// Coste:  $O(\log(n/j))$ 
template <typename Elem, typename Prio>
void ColaPrioridad<Elem,Prio>::hundir(int j) throw() {

    bool fin = false;
    while (2 * j <= nelems && !fin) {
        int minhijo = 2 * j;
        if (minhijo < nelems &&
            h[minhijo].second > h[minhijo + 1].second)
            ++minhijo;
        if (h[j].second > h[minhijo].second) {
            swap(h[j], h[minhijo]);
            j = minhijo;
        } else {
            fin = true;
        }
    }
}

```

```
// Flota al nodo j hasta reestablecer el orden del heap;  
// todos los nodos excepto el j satisfacen la propiedad  
// de heap
```

```
// Coste:  $O(\log j)$ 
```

```
template <typename Elem, typename Prio>  
void ColaPrioridad<Elem,Prio>::flotar(int j) throw() {
```

```
    // si j es la raíz, hemos terminado
```

```
    if (j == 1) return;
```

```
    int padre = j / 2;
```

```
    // si el padre tiene mayor prioridad
```

```
    // que j, intercambiar y seguir flotando
```

```
    if (h[j].second < h[padre].second) {
```

```
        swap(h[j], h[padre]);
```

```
        flotar(padre);
```

```
    }
```

```
}
```

# Heapsort

**Heapsort** (Williams, 1964) ordena un vector de  $n$  elementos construyendo un *heap* con los  $n$  elementos y extrayéndolos, uno a uno del *heap* a continuación. El propio vector que almacena a los  $n$  elementos se emplea para construir el *heap*, de modo que **heapsort** actúa *in-situ* y sólo requiere un espacio auxiliar de memoria constante. El coste de este algoritmo es  $\Theta(n \log n)$  (incluso en caso mejor) si todos los elementos son diferentes.

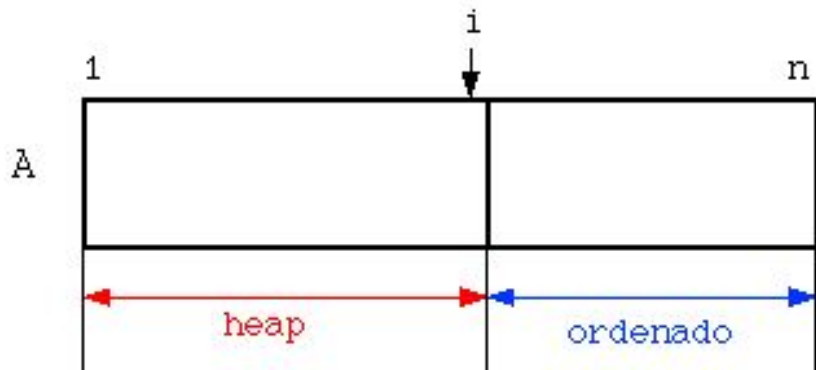
En la práctica su coste es superior al de quicksort, ya que el factor constante multiplicativo del término  $n \log n$  es mayor.

```
// Ordena el vector v[1..n]
// (v[0] no se usa)
// de Elem's de menor a mayor

template <typename Elem>
void heapsort(Elem v[], int n) {

    crea_max_heap(v, n);
    for (int i = n; i > 0; --i) {
        // saca el mayor elemento del heap
        swap(v[1], v[i]);

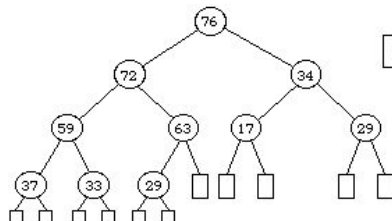
        // hunde el elemento de indice 1
        // para reestablecer un max-heap en
        // el subvector v[1..i-1]
        hundir(v, i-1, 1);
    }
}
```



$$A[1] \leq A[i+1] \leq A[i+2] \leq \dots \leq A[n]$$

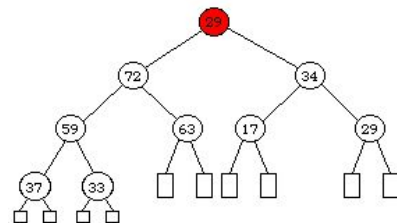
$$A[1] = \max_{1 \leq k \leq i} A[k]$$





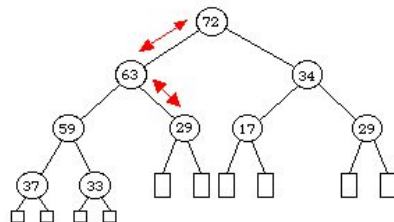
76 72 34 59 63 17 29 37 33 29

i = 10



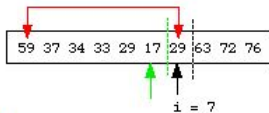
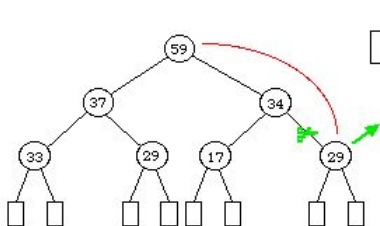
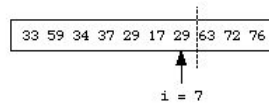
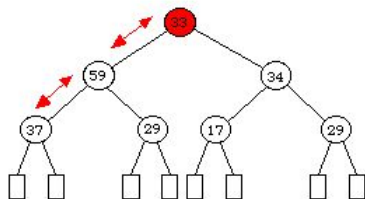
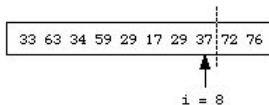
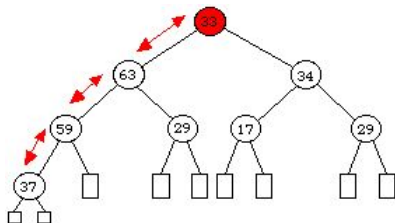
29 72 34 59 63 17 29 37 33 76

i = 9



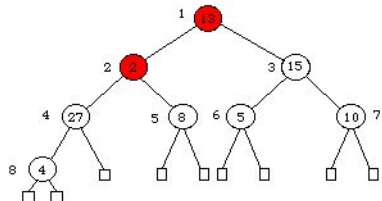
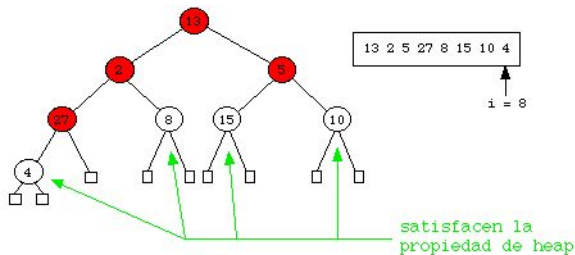
72 63 34 59 29 17 29 37 33 76

i = 9

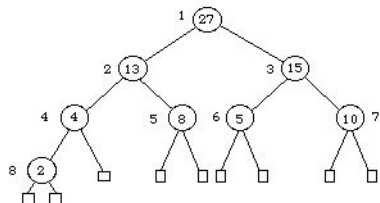


```
// Da estructura de max-heap al
// vector v[1..n] de Elem's; aquí
// cada elemento se identifica con su
// prioridad
template <typename Elem>
void crea_max_heap(Elem v[], int n) {

    for (int i = n/2; i > 0; --i)
        hundir(v, n, i);
}
```



`hundir(A,8,4)`  
`hundir(A,8,3)`



`hundir(A,8,2)`  
`hundir(A,8,1)`

Sea  $H(n)$  el coste en caso peor de `heapsort` y  $B(n)$  el coste de crear el *heap* inicial. El coste en caso peor de `hundir`( $v, i - 1, 1$ ) es  $\mathcal{O}(\log i)$  y por lo tanto

$$\begin{aligned} H(n) &= B(n) + \sum_{i=1}^{i=n} \mathcal{O}(\log i) \\ &= B(n) + \mathcal{O} \left( \sum_{1 \leq i \leq n} \log_2 i \right) \\ &= B(n) + \mathcal{O}(\log(n!)) = B(n) + \mathcal{O}(n \log n) \end{aligned}$$

Un análisis simple de  $B(n)$  indica que  $B(n) = \mathcal{O}(n \log n)$  ya que hay  $\Theta(n)$  llamadas a `hundir`, cada una de las cuales tiene coste  $\mathcal{O}(\log n)$

Por tanto  $H(n) = \mathcal{O}(n \log n)$ , de hecho  $H(n) = \Theta(n \log n)$  en caso peor

Podemos refinar el análisis de  $B(n)$ :

$$\begin{aligned} B(n) &= \sum_{1 \leq i \leq \lfloor n/2 \rfloor} \mathcal{O}(\log(n/i)) \\ &= \mathcal{O}\left(\log \frac{n^{n/2}}{(n/2)!}\right) \\ &= \mathcal{O}\left(\log(2e)^{n/2}\right) = \mathcal{O}(n) \end{aligned}$$

Puesto que  $B(n) = \Omega(n)$ , podemos afirmar que  $B(n) = \Theta(n)$ .

Demostración alternativa: Sea  $h = \lceil \log_2(n + 1) \rceil$  la altura del *heap*. En el nivel  $h - 1 - k$  hay como mucho

$$2^{h-1-k} < \frac{n+1}{2^k}$$

nodos y cada uno de ellos habrá de hundirse en caso peor hasta el nivel  $h - 1$ ; eso tiene coste  $\mathcal{O}(k)$

Por lo tanto,

$$\begin{aligned} B(n) &= \sum_{0 \leq k \leq h-1} \mathcal{O}(k) \frac{n+1}{2^k} \\ &= \mathcal{O} \left( n \sum_{0 \leq k \leq h-1} \frac{k}{2^k} \right) \\ &= \mathcal{O} \left( n \sum_{k \geq 0} \frac{k}{2^k} \right) = \mathcal{O}(n), \end{aligned}$$

ya que

$$\sum_{k \geq 0} \frac{k}{2^k} = 2.$$

En general, si  $|r| < 1$ ,

$$\sum_{k \geq 0} k \cdot r^k = \frac{r}{(1-r)^2}.$$



Aunque  $H(n) = \Theta(n \log n)$ , el análisis detallado de  $B(n)$  es importante: utilizando un *min-heap* podemos hallar los  $k$  menores elementos en orden creciente de un vector (y en particular el  $k$ -ésimo) con coste:

$$\begin{aligned} S(n, k) &= B(n) + k \cdot \mathcal{O}(\log n) \\ &= \mathcal{O}(n + k \log n). \end{aligned}$$

y si  $k = \mathcal{O}(n / \log n)$  entonces  $S(n, k) = \mathcal{O}(n)$ .