# TAD MONTÍCULO

Al igual que los árboles de búsqueda, los montículos (heaps) son un tipo particular de árboles binarios, que pueden definirse cuando existe una relación de orden total en el tipo de los elementos del árbol.

Un **montículo** es un árbol casi completo en el que cada nodo es menor o igual que sus nodos hijos (montículo de mínimos).

Un árbol binario se llama **casi completo** si cada uno de sus nodos internos posee exactamente dos hijos, izquierdo y derecho, excepto posiblemente un nodo especial situado en el último nivel que puede tener un sólo hijo, el izquierdo. Además todas las hojas, o bien están en el último nivel, o bien están en el penúltimo nivel y, en este caso no estará nunca a la izquierda de un nodo interno de su mismo nivel.

Otra definición equivalente:

a) El árbol binario ha de ser casi completo.
b) El elemento raíz ha de ser menor o igual que el resto de los elementos del árbol. Además, los *subárboles izquierdo* y *derecho* deben ser montículos.

Las operaciones que va a tener a su disposición un usuario del tipo de dato son insertar, mínimo y eliminarMínimo. Es decir, la visión que un usuario tiene del montículo es la de un tipo de dato que permite añadir datos en cualquier orden y recuperarlos en orden creciente.

Si imaginamos que el valor de un elemento representa algún tipo de prioridad (la prioridad será tanto más alta cuanto más pequeño es el valor), este tipo de datos responde a lo que se conoce como **cola de prioridad**, en el que el primer elemento en salir es el de mayor prioridad (como algunas colas de impresión o colas de procesos batch).

# ESPECIFICACIÓN:

```
especificación TAD Montículo;
parámetros
  géneros
    telemento
  operaciones
    función esMenor(telemento d1,telemento d2) devuelve
            booleano
    {Devuelve el valor verdad si d1<d2, en caso contrario
    devuelve el valor falso }
    acción permutar(e/s d1:telemento, e/s d2:telemento)
    {Intercambia los valores de d1 y d2 }
géneros
  Montículo
operaciones
  acción iniciarMontículo( Sal Montículo M )
  { Inicia M como un montículo vacío }
  acción insertar(e/s Montículo M; ent telemento d)
  {Inserta en el montículo M el elemento d }
  función mínimo(Montículo M) devuelve telemento
  {Devuelve el elemento más pequeño del montículo M}
  acción eliminarMínimo(e/s Montículo M)
  {Elimina del montículo M el elemento mínimo }
  función montículoVacío(Montículo M) devuelve booleano
  {Devuelve verdad si M está vacío y falso en caso contrario}
  función alturaMontículo(Montículo M) devuelve entero
  {Devuelve la altura del montículo M }
```

## Implementación.

### Representación

La implementación habitual de los montículos se apoya en el hecho de que, al tratarse de árboles casi completos, podemos almacenar ordenadamente (por niveles) sus nodos en un vector, sin que esto suponga "desperdiciar" memoria.

**tipodef** tElemento = tVector[MAX]

tMonticulo= **registro**
**tVector** datos
numDatos: entero
**finreg**

**Idea**: los nodos del árbol ocupan las posiciones [0..numDatos-1] del vector. Los hijos del que ocupa la posición i, están en 2i+1 y 2i+2; el padre del que ocupa la posición k, está en (k-1) **div** 2.

### Operadores

```
acción iniciarMontículo( Sal Montículo M )
  { Inicia M como un montículo vacío }
     principio
          M.numDatos:=0
     fin

función montículoVacío(Montículo M) devuelve booleano
  {Devuelve verdad si M está vacío y falso en caso contrario}
     principio
          devuelve (M.numDatos==0)
     fin


función alturaMontículo(Montículo M) devuelve entero
    {Devuelve la altura del montículo M. Se podría calcular
    directamente usando el logaritmo en base 2}
     variables
          a,i : entero
     principio
          a:=0
          i:=1
          mientras que (i<M.numDatos) hacer
               a++
               i:=2*i+1
          fmq
          devuelve(a)
```

```
     fin


función mínimo(Montículo M) devuelve telemento
  {Devuelve el elemento más pequeño del montículo M}

     principio
          devuelve (M.datos[0])
     fin

acción insertar(e/s Montículo M, ent telemento d)
  {Inserta en el montículo M el elemento d }
     principio
          si (M.numDatos<MAX) entonces
               M.datos[M.numDatos]:=d
               M.numDatos++
               flotar(M.datos, M.numDatos-1, M.numDatos)
          finsi
     fin
```

Se usa el siguiente método, que permite llevar un nodo hacia niveles superiores hasta llegar a su sitio, es decir, subirlo hasta que sea mayor que su padre.

```
accion flotar(e/s  tVector v, ent entero i, ent entero tam)
{Lleva el nodo en la posición i hacia niveles superiores
hasta alcanzar su posición, para restablecer la propiedad de
montículo}
     variables
          j : entero
     principio
        si ((i<tam)and (i>0)) entonces
               repetir
                    j :=i
                    si esMenor(v[j],v[(j-1) div 2])
                         entonces
                              permutar(v[(j-1) div 2],v[j])
                              j:= (j-1) div 2
                    fsi

               mientras que ((j!=0) and (i!=j))
          finsi
     fin
```

```
acción eliminarMínimo(e/s Montículo M)
  {Elimina del montículo M el elemento mínimo }

     principio
          si (M.numDatos>0) entonces
               permutar(M.datos[0], M.datos[M.numDatos-1])
               M.numDatos--
               hundir(M.datos,M.numDatos,0)
          finsi
fin
```

Para eliminar usamos hundir, que hace descender un nodo hasta alcanzar su posición, es decir, llega a ser hoja o es menor que sus hijos.

```
accion hundir( e/s  tVector v, ent entero n, ent entero i)
{ Hunde el nodo en la posición i para restablecer la
propiedad de montículo}
     variables
          j : entero
     principio
          repetir
               j:=i
               { Buscar el hijo menor del nodo j }
               Si  2*j+1<n AND esMenor(v[2*j+1],v[i]) hacer
                    i:=2*j+1
               fsi
               Si  2*j+2<n AND esMenor(v[2*j+2],v[i]) hacer
                    i ← 2*j+2
               fsi
               permutar(v[i],v[j])
          mientras que i != j
     fin
```

## Ordenación por el método del montículo

Las propiedades de los montículos los hacen interesantes para basar en ellos el algoritmo de ordenación en memoria interna conocido como ordenación por el método del montículo, algoritmo de Williams o **heapsort**.

Dado un vector de n elementos, en una primera fase se construye con ellos un montículo de mínimos. A continuación, se extrae sucesivamente el mínimo del montículo hasta dejar éste vacío, y los elementos extraídos del montículo se van copiando en posiciones consecutivas del vector, quedando éste finalmente ordenado. Este programa necesita un espacio adicional de $\Theta(n)$ para la variable M de tipo `Montículo`. Su complejidad en tiempo es $\Theta$ (nlogn) dado que los costes de las operaciones insertar y eliminar son $\Theta$ (logn).

```
accion HeapSort1( e/s  tVector v, ent  entero n)
{ Ordena los datos del vector v de tamaño n en orden
creciente }
     variables
          i : entero
          M : Montículo
     principio
          iniciarMontículo(M)
          para  i ← 0  hasta n - 1 hacer
               insertar(M,v[i])
          fpara
          para  i ← 0  hasta n - 1 hacer
               v[i] ← mínimo(M)
               eliminarMínimo(M)
          fpara
     fin
```

Es posible mejorar el coste en tiempo y en espacio del algoritmo heapsort si trabaja directamente con la implementación del montículo en términos de un vector. El propio vector a ordenar sirve para este propósito, con lo que no se consume espacio extra. (El algorimto mergesort necesita un espacio adicional de $\Theta$ (n) para el vector donde se almacena la fusión de las dos porciones de vector ordenadas previamente. El algoritmo quicksort necesita un espacio en el peor caso de $\Theta$ (n) para la pila de activación de las llamadas recursivas.)

En la nueva versión, un primer bucle se dedica a convertir el vector de entrada en un montículo de mínimos.

```
accion HeapSort2( e/s  tVector v, ent  entero n)
{ Ordena los datos del vector v de tamaño n en orden
decreciente }
     variables
          i : entero
     principio
          { Crear montículo }
          para  i ← n div 2 - 1 descendiendo hasta 0 hacer
               hundir(v, n, i)
          fpara
          para  i ← n - 1  descendiendo hasta 0 hacer
               permutar(v[0], v[i])
               hundir(v, i, 1)
          fpara
     fin
```