Tema: Colas de prioridad y montículos

Asignatura: Métodos Algorítmicos de Resolución de Problemas
Profesor: Ricardo Peña

Curso 2013/14

1. Especificación

El tipo de datos *cola de prioridad* aparece con frecuencia en la programación, en particular en el diseño de sistemas operativos. Su comportamiento es el siguiente: al igual que en una cola FIFO, existe una operación *insert* que incorpora nuevos elementos a la cola, y operaciones que consultan y eliminan el primer elemento de la misma. Pero, a diferencia de las colas FIFO, aquí los elementos son recuperados en orden de prioridad, es decir, el primer elemento en salir es el de mayor prioridad. Normalmente se toma como prioridad el valor del elemento y se admite que cuanto menor es el valor tanto más alta es la prioridad. En definitiva, los elementos se recuperan de menor a mayor y la cola se llama *de mínimos*. En consecuencia, a la operación que accede al primer elemento la llamaramos *min*, y a la que lo suprime de la cola, *elimMin*. Si se cambia el convenio y los elementos se recuperan en orden decreciente, la cola se llamaría de máximos, y las respectivas operaciones, *max* y *elimMax*.

En la Figura 1 damos la especificación algebraica de las colas de prioridad. Observada atentamente, es fácil convencerse de que su modelo inicial corresponde a la idea matemática de *multiconjunto* o *bolsa*[1] de elementos, con la propiedad adicional de que existe una relación de orden en los elementos que permite preguntar por el mínimo de ellos. Nótese que, para especificar las operaciones min y $elimMin$ cuando hay dos o más elementos en la cola, sólo consideramos el caso $x \leq y$. El opuesto, es decir $x > y$, se obtiene aplicando previamente la primera ecuación de la especificación, que establece que la inserción de elementos es conmutativa.

Incluimos esta estructura en la sección dedicada a los árboles porque la implementación eficiente de una cola de prioridad requiere un tipo de datos denominado *montículo*[2] que es un tipo particular de árbol binario, no obstante lo cual, la visión que un usuario tiene de un montículo es la de una cola de prioridad. Es decir, la de un tipo de datos que le permite añadir elementos en cualquier orden y recuperarlos en orden creciente.

Al igual que para los árboles de búsqueda, para los montículos se puede definir un *invariante de la representación*, que denotaremos $mon(t)$, que caracteriza a los árboles binarios t que, además, son montículos.

Invariante de la representación Expresa que, o bien t es vacío, o bien su elemento raíz ha de ser menor o igual que el resto de los elementos del árbol. Adicionalmente, los subárboles izquierdo

[1] En inglés, respectivamente, *multiset* y *bag*.
[2] En inglés, *heap*.

y derecho han de ser montículos.

$$mon(t) \stackrel{\text{def}}{=} vacio?(t) \ \vee \ \begin{array}{l}(t = crear(i, x, d)) \wedge mon(i) \wedge mon(d) \wedge \\ (\neg vacio?(i) \Rightarrow x \leq raiz(i)) \wedge \\ (\neg vacio?(d) \Rightarrow x \leq raiz(d))\end{array}$$

*Ejercicio 1.

Especificar algebraicamente el invariante $mon(t)$ de los montículos. ■

El montículo así definido se denomina *de mínimos*, porque en su raíz se halla el mínimo del conjunto de elementos. Cambiando en el invariante la operación $\leq$ por $\geq$, se obtendría un *montículo de máximos*.

Al igual que sucede en los árboles de búsqueda, no se pretende que el usuario del tipo utilice directamente la operación *crear*, generadora de árboles binarios, ya que su uso podría dar lugar a árboles que no son montículos. Los usuarios del tipo manipulan montículos mediante el uso exclusivo de las operaciones mencionadas, *insert, min* y *elimMin*, junto con *cvacia*, que crearía el árbol binario vacío.

Como se verá en la Sección 2 la inserción y el borrado de un elemento, restaurando en cada caso la propiedad de montículo, pueden hacerse con un coste en $\Theta(\log n)$. Ambas operaciones recorren, en el caso peor, una altura del árbol binario. La operación *min* tiene obviamente un coste constante ya que simplemente accede a la raíz del árbol.

2. Implementación de un montículo sobre un vector

La implementación de una cola de prioridad mediante un montículo se basa habitualmente en la implementación del montículo mediante un árbol binario casi completo, utilizando la representación vectorial para árboles casi completos. Esta versión fue ideada por J. W. J. Williams en 1964.

*Ejercicio 2.

Especificar algebraicamente la operación $casiCompleto$ que nos indica si un árbol binario es o no casi completo. ■

De este modo, si el montículo está previsto para un máximo de n elementos, la representación del mismo es:

rep
  tipo $monticulo =$ **reg**
    v : **vector** $[1..n]$ **de** $elem$;
    k : nat
  freg
frep

donde $k, 0 \leq k \leq n$, señala que los elementos válidos del montículo son los elementos $v[1], \ldots, v[k]$. Al ser un árbol casi completo, no puede haber elementos ausentes en el rango $1..k$. Con esta representación, los elementos hijos de $v[i]$, si existen, son $v[2i]$ y $v[2i + 1]$, y el padre de $v[i]$, si $i > 1$, es el elemento $v[i$ **div** $2]$.

espec *COLAS_DE_PRIORIDAD*

parametro

generos $elem$

operaciones

$\_ \leq \_\ ,\ \_ < \_\ ,\ \_ > \_ : elem\ \ elem \longrightarrow bool$

$\_ == \_ : elem\ \ elem \longrightarrow bool$

var

$x, y : elem$

ecuaciones

$\ldots \leq$ relación de orden total$\ldots$

$\ldots x == y$ equivale a $x \leq y \wedge y \leq x \ldots$

$\ldots >$ y $<$ se definen a partir de $\leq$ y de $==\ldots$

fparametro

generos $colap$

operaciones

$cvacia : colap$

$insert : colap\ \ elem \longrightarrow colap$

parcial $min : colap \longrightarrow elem$

parcial $elimMin : colap \longrightarrow colap$

$vacia? : colap \longrightarrow bool$

var

$c : colap; x, y : elem$

ecuaciones de definitud

$\neg vacia?(c) \Longrightarrow Def\ (min(c))$

$\neg vacia?(c) \Longrightarrow Def\ (elimMin(c))$

ecuaciones

$insert(insert(c, x), y) \stackrel{e}{=} insert(insert(c, y), x)$

$min(insert(cvacia, x)) \stackrel{e}{=} x$

$x \leq y \Longrightarrow min(insert(insert(c, x), y)) \stackrel{e}{=} min(insert(c, x))$

$elimMin(insert(cvacia, x)) \stackrel{e}{=} cvacia$

$x \leq y \Longrightarrow elimMin(insert(insert(c, x), y)) \stackrel{e}{=}$
$\qquad insert(elimMin(insert(c, x)), y)$

$vacia?(cvacia) \stackrel{e}{=} T$

$vacia?(insert(c, x)) \stackrel{e}{=} F$

fespec

Figura 1: Especificación algebraica del tipo *cola de prioridad*

```
accion insert (m : ent/sal monticulo; x : elem)
con m hacer
    si k = n entonces error
    si no k := k + 1; v[k] := x; flotar (v, k)
    fsi
fcon
faccion

accion flotar (v : ent/sal vector; k : nat)
var j : nat fvar
    j := k;
    mientras j ≠ 1 ∧_c v[j div 2] > v[j] hacer
        permutar (v, j div 2, j);
        j := j div 2
    fmientras
faccion
```

Figura 2: Implementación de *insert* y *flotar* de montículos

Invariante de la representación La propiedad $mon(t)$ enunciada abstractamente en la sección anterior se traduce, para esta representación, en el siguiente predicado:

$$I_R \stackrel{\text{def}}{=} 0 \leq k \leq n \wedge \forall \alpha \in \{2..k\}.v[\alpha \textbf{ div } 2] \leq v[\alpha]$$

En base a esta representación, las operaciones *cvacia*, *min* y *vacia?* de las colas de prioridad se implementan en un tiempo constante: la condición $k = 0$ corresponde a un montículo vacío y, si no es vacío, en $v[1]$ se encuentra el elemento mínimo.

La operación *insert* sigue la siguiente secuencia:

- Añade, siempre que $k < n$, un nuevo elemento en la posición $k + 1$ del vector, posición que corresponde a la hoja más a la derecha posible del último nivel del árbol.
- Incrementa en uno el valor de k. De este modo, el árbol conserva la propiedad de ser casi completo.
- Para preservar también la propiedad de montículo, se hace *flotar* el elemento en el montículo, intercambiándolo repetidamente con su padre hasta conseguir que, o bien el elemento es mayor o igual que su padre, o bien el elemento se ha convertido en la raíz.

Este proceso puede llevarse a cabo en un tiempo en $\Theta(\log k)$, ya que la operación *padre* se realiza en tiempo constante y, en el peor caso, ha de recorrerse el camino completo desde la hoja a la raíz. La implementación iterativa de ambas operaciones *insert* y *flotar* se muestra en la Figura 2. El invariante que proponemos para el bucle de *flotar* es el siguiente:

$$P \stackrel{\text{def}}{=} \forall \alpha \in \{2..k\}.\alpha \neq j \rightarrow v[\alpha \textbf{ div } 2] \leq v[\alpha]$$

que, como puede apreciarse, es un debilitamiento de la postcondición: la condición de montículo se satisface excepto quizás para la relación que guardan el elemento j y su padre.

*Ejercicio 3.

Con dicho invariante, y sabiendo que la función ha de preservar I_R, verificar formalmente la corrección de *flotar* e *insert*. ¿Cuál sería una función limitadora apropiada? ■

La operación *elimMin* ha de suprimir la raíz del montículo. Como resultado, se obtienen dos montículos. Para unirlos de nuevo, se realiza la siguiente secuencia de operaciones:

- Siempre que se cumpla $k \geq 1$, se asigna a $v[1]$ el contenido de $v[k]$. Ello equivale a convertir en raíz la hoja más a la derecha del último nivel.
- Se decrementa en uno el valor de k. Con ello se ha conseguido un árbol casi completo con los elementos remanentes del montículo.
- Para preservar el invariante I_R, se *hunde* el elemento raíz, intercambiándolo repetidamente con el menor de sus hijos (siempre que éstos existan), hasta conseguir que, o bien el elemento se ha situado de modo que es menor o igual que sus dos hijos, o bien el elemento es una hoja.

Esta secuencia de operaciones puede realizarse también en un tiempo en $\Theta(\log k)$. Para precisar el efecto de la operación *hundir*, la hemos especificado algebraicamente en la Figura 3 como una operación que, aplicada a un árbol binario, da otro árbol binario. Sólo está definida, o bien cuando el árbol es trivial (observar los cuatro casos especificados), o bien cuando no es trivial y el resto del árbol, excluida la raíz, es un montículo. Para precisar el dominio de definición hemos usado las operaciones auxiliares, *mon* y *casiCompleto*, que fueron especificadas en las ejercicios 1 y 2. En la Figura 4 se proporcionan versiones iterativas de las operaciones *elimMin* y *hundir*. El invariante que proponemos para el bucle de *hundir* es el siguiente:

$$P \stackrel{\text{def}}{=} \forall \alpha \in \{2..k\}.\alpha \,\mathbf{div}\, 2 \neq j \rightarrow v[\alpha \,\mathbf{div}\, 2] \leq v[\alpha]$$

que, al igual que sucedía en el bucle de *flotar*, es un debilitamiento de la postcondición: la condición de montículo se satisface excepto quizás para la relación que guardan el elemento j y sus dos hijos.

*Ejercicio 4.

Con dicho invariante, y sabiendo que *hundir* preserva I_R, verificar formalmente la corrección de *hundir* y *elimMin*. ■

Comparado con otras alternativas para implementar colas de prioridad con n elementos, los montículos resultan claramente ventajosos, tal como se expresa en la siguiente tabla:

	insert	*min*	*elimMin*
lista desordenada	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
lista ordenada	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
montículo	$\Theta(\log n)$	$\Theta(1)$	$\Theta(\log n)$

3. Ordenación por el método del montículo

Las propiedades de los montículos representados como vectores los hacen interesantes para basar en ellos el algoritmo de ordenación de vectores conocido como *ordenación por el método del montículo*, algoritmo de Williams (1964), o *heapsort*. La idea del mismo es muy sencilla: 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,

espec *MONTICULOS*

usa *ARBOLES_BINARIOS*

parametro

operaciones

$\_ \leq \_ : elem \;\; elem \longrightarrow bool$

ecuaciones

$\ldots \leq$ relación de orden total$\ldots$

fparametro

operaciones

$hundir : arbin \longrightarrow arbin$

operaciones auxiliares

$mon\ ,\ casiCompleto : arbin \longrightarrow bool$

var

$x, y, z : elem; i, i', d, d' : arbin$

ecuaciones de definitud

$Def\ (hundir\ (\triangle)), Def\ (hundir\ (crear\ (\triangle, x, \triangle)))$

$Def\ (hundir\ (crear\ (crear\ (\triangle, y, \triangle), x, \triangle)))$

$casiCompleto\ (crear\ (i, x, d)) \wedge mon\ (i) \wedge mon\ (d) \Longrightarrow$

$Def\ (hundir\ (crear\ (i, x, d)))$

ecuaciones

$hundir\ (\triangle) \stackrel{e}{=} \triangle$

$hundir\ (crear\ (\triangle, x, \triangle)) \stackrel{e}{=} crear\ (\triangle, x, \triangle)$

$x \leq y \Longrightarrow$

$hundir\ (crear\ (crear\ (\triangle, y, \triangle)), x, \triangle) \stackrel{e}{=} (crear\ (crear\ (\triangle, y, \triangle), x, \triangle)$

$x > y \Longrightarrow$

$hundir\ (crear\ (crear\ (\triangle, y, \triangle), x, \triangle)) \stackrel{e}{=} crear\ (crear\ (\triangle, x, \triangle), y, \triangle)$

$x \leq y \wedge x \leq z \Longrightarrow$

$hundir\ (crear\ (crear\ (i, y, d), x, crear\ (i', z, d'))) \stackrel{e}{=}$

$crear\ (crear\ (i, y, d), x, crear\ (i', z, d'))$

$y \leq x \wedge y \leq z \Longrightarrow$

$hundir\ (crear\ (crear\ (i, y, d), x, crear\ (i', z, d'))) \stackrel{e}{=}$

$crear\ (hundir\ (crear\ (i, x, d)), y, crear\ (i', z, d'))$

$z \leq x \wedge z \leq y \Longrightarrow$

$hundir\ (crear\ (crear\ (i, y, d), x, crear\ (i', z, d'))) \stackrel{e}{=}$

$crear\ (crear\ (i, y, d), z, hundir\ (crear\ (i', x, d')))$

fespec

Figura 3: Especificación algebraica de la operación *hundir*

```
accion elimMin (m : ent/sal monticulo)
con m hacer
    si k = 0 entonces error
    si no v[1] := v[k]; k := k − 1; hundir (v, 1, k)
    fsi
fcon
faccion

accion hundir (v : ent/sal vector; j, k : nat)
var m : nat; fin : bool fvar
    fin := F;
    mientras 2 ∗ j ≤ k ∧ ¬fin hacer
        caso 2 ∗ j + 1 ≤ k ∧c v[2 ∗ j + 1] < v[2 ∗ j] → m := 2 ∗ j + 1
        [] 2 ∗ j + 1 > k ∨c v[2 ∗ j + 1] ≥ v[2 ∗ j] → m := 2 ∗ j
        fcaso
        si v[j] > v[m]
            entonces permutar (v, j, m); j := m
            si no fin := T
        fsi
    fmientras
faccion
```

Figura 4: Implementación de *elimMin* y *hundir* de montículos

y los elementos extraídos del montículo se van copiando en posiciones consecutivas del vector, quedando éste finalmente ordenado. Esta versión, cuya implmentación puede verse en la Figura 5, necesita un espacio adicional en $\Theta(n)$ para la variable c del tipo *cola de prioridad*. Tiene la ventaja de que se puede razonar sobre su corrección utilizando tan sólo las propiedades de las colas de prioridad. Éstas son las que se deducen de la especificación de la Figura 1. El que c esté implementada o no mediante un montículo afecta al coste del algoritmo, pero no a su corrección. Dado que los costes de *insert* y de *elimMin* están en $\Theta(\log m)$, siendo m el número de elementos del montículo, el coste del programa de la Figura 5, utilizando un montículo para implementar c, es:

$$max\,(\sum_{m=0}^{n-1} \log m, \sum_{m=1}^{n} \log m) \in \Theta(n \log n)$$

Aunque la corrección del algoritmo es bastante evidente, para verificarlo formalmente necesitamos expresar con predicados su precondición Q, postcondición R y los invariantes P_1 y P_2 de ambos bucles. Proponemos los siguientes:

$$\begin{array}{lcl} Q & \stackrel{\text{def}}{=} & v = V \wedge n \geq 0 \\ R & \stackrel{\text{def}}{=} & bag(v[1..n]) = bag(V[1..n]) \wedge ord(v, 1, n) \end{array}$$

$$\begin{array}{lcl} P_1 & \stackrel{\text{def}}{=} & bag(c) \cup bag(v[i..n]) = bag(V[1..n]) \wedge 1 \leq i \leq n+1 \\ P_2 & \stackrel{\text{def}}{=} & bag(v[1..i-1]) \cup bag(c) = bag(V[1..n]) \wedge 1 \leq i \leq n+1 \\ & & \wedge ord(v, 1, i-1) \wedge (\neg vacia?(c) \rightarrow \forall \xi \in \{1..i-1\}.v[\xi] \leq min(c)) \end{array}$$

```
accion heapsort (v : ent/sal vector [1..n] de elem);
var c : colap; i : nat fvar
    c := cvacia;
    para i desde 1 hasta n hacer
        c := insert (c, v[i])
    fpara;
    para i desde 1 hasta n hacer
        v[i] := min (c); c := elimMin (c)
    fpara
faccion
```

Figura 5: Versión abstracta del algoritmo *heapsort*

donde $ord(a, i, j)$ expresa que el subvector $a[i..j]$ está ordenado, y bag es una función que, dado un subvector o un montículo, construye el multiconjunto o bolsa de sus elementos. Por ejemplo, la expresión $bag(v[1..n]) = bag(V[1..n])$ de la postcondición es equivalente a decir que el valor final de v es una permutación de los elementos existentes en el vector de partida V. Para expresar los invariantes se ha considerado que los bucles **para** se sustituyen por los bucles **mientras** equivalentes y, en consecuencia, el rango de i termina en $n + 1$.

La demostración de corrección, aligerada de formalismo, consiste en los siguientes razonamientos:

1. P_1 es trivialmente cierto con $i = 1$ y $c = cvacia$.

2. P_1 se mantiene invariante al incorporar el elemento $v[i]$ a c y sumar 1 a i.

3. Cuando $i = n+1$, $bag(v[i..n]) = \emptyset$, y todos los elementos están en c. Al hacer la asignación inicial $i := 1$ del segundo bucle, se cumplen trivialmente todas las conjunciones de P_2, por ser vacío el rango $\{1..i - 1\}$.

4. Suponiendo que P_2 es cierto al comienzo de una iteración, extraer el elemento mínimo de c y copiarlo en $v[i]$, seguido de incrementar en uno i, mantiene invariantes los apartados de P_2:

   a) los elementos inicialmente en V continúan repartidos entre (la nueva) c y $v[1..i - 1]$ (con la nueva i)

   b) el elemento colocado en $i - 1$ (con la nueva i) mantiene la ordenación de $v[1..i - 1]$ porque los anteriores a él eran, según P_2, menores o iguales que cualquier elemento que estuviera en c

   c) el elemento colocado en $i-1$ (con la nueva i), es menor o igual que cualquier elemento que permanezca aún en c, ya que extrajimos el mínimo de ellos

5. La condición de terminación $i = n + 1$, junto con P_2, conducen directamente a R ya que, para conservar la cardinalidad de $bag(V[1..n])$, ha de cumplirse forzosamente $bag(c) = \emptyset$.

Es posible mejorar el coste en tiempo y en espacio del algoritmo *heapsort* si se trabaja directamente con la implementación del montículo en términos de un vector. De hecho, el propio vector a ordenar sirve para este propósito, con lo que el consumo en espacio del algoritmo pasa a ser

$\Theta(1)$. Ello marca una diferencia con respecto a otros algoritmos de ordenación interna con una eficiencia en tiempo en $\Theta(n \log n)$:

- El algoritmo *mergesort*, en su versión para ordenar vectores, necesita un espacio adicional en $\Theta(n)$ para el vector donde se almacena la *fusión* de las dos porciones de vector ordenadas previamente. En su versión para listas enlazadas, puede conseguirse un espacio constante del montón, pero no así el espacio de pila, que necesita un coste en $\Theta(\log n)$.
- El algoritmo *quicksort* necesita un espacio, en promedio en $\Theta(\log n)$ y en el caso peor en $\Theta(n)$, 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 máximos. Para ello se emplea la siguiente táctica: si n es el número de elementos del vector, es inmediato demostrar que los elementos

$$v[n \textbf{ div } 2 + 1], \ldots, v[n]$$

corresponden a hojas del futuro montículo. De hecho, son ya montículos de un elemento. Entonces, se procede a *hundir* los elementos $v[n \textbf{ div } 2]$ a $v[1]$, en ese orden, construyendo en sentido ascendente montículos de 2, 4, 8, etc. elementos, hasta llegar a formar un solo montículo. Se puede demostrar que el coste de este bucle está en $\Theta(n)$ mediante el siguiente razonamiento: considerando el peor caso, que sería un árbol completo de altura $k = \lfloor \log n \rfloor$, hundir los nodos interiores de profundidad $k-1$, ocasiona $2\frac{n}{4}$ comparaciones, hundir los de profundidad $k-2$, ocasiona $4\frac{n}{8}$ comparaciones, los de profundidad $k-2$, $6\frac{n}{16}$, etc. El número total de comparaciones es, pues:

$$\sum_{i=1}^{\lfloor \log n \rfloor - 1} n \frac{2i}{2^{i+1}} = n \sum_{i=1}^{\lfloor \log n \rfloor - 1} \frac{i}{2^i}$$

donde la cantidad $\sum_{i=1}^{\lfloor \log n \rfloor - 1} \frac{i}{2^i} \leq \sum_{i=1}^{\infty} \frac{i}{2^i}$ está acotada por una constante.

El segundo bucle es muy similar al de la primera versión, con la diferencia de que ahora se extrae sucesivamente el máximo del montículo y se coloca al final del vector. En todo momento de la iteración el vector está dividido en dos porciones: la $v[1..i]$, que corresponde al montículo remanente, y la $v[i+1, ..n]$, que contiene los elementos extraídos del montículo y que está ordenada crecientemente. El coste de este segundo bucle está en $\Theta(n \log n)$. El algoritmo completo se presenta en la Figura 6.

La acción $hundir\,(v, j, m)$ supone que el subvector $v[j..m]$ está organizado como un (conjunto de) montículo(s), en este caso, de máximos, y que el único elemento que puede no cumplir la propiedad de ordenación es $v[j]$. Tras "hundirlo" hasta el nivel apropiado, todo el subvector $v[j..m]$ cumple la propiedad. Su especificación formal es la siguiente:

$$\begin{array}{l} \{MON(v, j+1, m) \wedge 1 \leq j \leq m \leq n \wedge v = V\} \\ \textbf{accion } hundir\ (v: \textbf{ ent/sal vector } [1..n] \textbf{ de } elem; j, m: nat) \\ \{MON(v, j, m) \wedge bag(v[j..m]) = bag(V[j..m])\} \end{array}$$

donde el predicado *MON* expresa la propiedad de ordenación:

$$\begin{array}{rcl} MON(v, j, m) & \stackrel{\text{def}}{=} & \forall \xi \in \{j..m \textbf{ div } 2\}. \\ & & v[\xi] \geq v[2\xi] \wedge 2\xi + 1 \leq m \rightarrow v[\xi] \geq v[2\xi + 1] \end{array}$$

La acción $permutar\ (v, i, j)$ intercambia los valores $v[i]$ y $v[j]$ en el vector v.

```
accion heapsort (v : ent/sal vector [1..n] de elem);
var i : nat fvar
    para i bajando desde n div 2 hasta 1 hacer
        hundir (v, i, n)
    fpara;
    para i bajando desde n hasta 2 hacer
        permutar (v, 1, i);
        hundir (v, 1, i − 1)
    fpara
faccion
```

Figura 6: Versión definitiva del algoritmo *heapsort*

Los invariantes propuestos para ambos bucles son los siguientes:

$$
\begin{array}{lcl}
P_1 & \stackrel{\text{def}}{=} & bag(v[1..n]) = bag(V[1..n]) \wedge 1 \leq i \leq n+1 \wedge MON(v, i+1, n) \\
P_2 & \stackrel{\text{def}}{=} & bag(v[1..n]) = bag(V[1..n]) \wedge 1 \leq i \leq n+1 \wedge MON(v, 1, i) \\
 & & \wedge ord(v, i+1, n) \wedge \forall \xi \in \{1..i\}.\forall \eta \in \{i+1..n\}.v[\xi] \leq v[\eta]
\end{array}
$$

El razonamiento de corrección sigue unas pautas muy semejantes a las que se siguieron para la versión abstracta del algoritmo. Se deja como ejercicio para el lector.

El coste del algoritmo *heapsort* en el caso peor está en $\Theta(n \log n)$. Su coste promedio está también en dicha clase de complejidad. Además, como se ha dicho, no necesita espacio adicional. Sin embargo, su constante multiplicativa es ligeramente superior a la del algoritmo *quicksort*. Dado que su primera fase tiene un coste lineal, es, no obstante, un algoritmo muy recomendable cuando sólo se pretenden obtener los k menores (o mayores) elementos de un vector, siendo k bastante menor que n.

Lecturas complementarias

Estas notas están tomadas literalmente de la Sec. 7.5 del libro [1]. En el Cap. 7 del mismo, pueden encontrarse también los llamados montículos *zurdos*, que no tienen un tamaño prefijado como los de Williams y además ofrecen una operación *combinar* que une dos montículos en un tiempo logarítmico.

Referencias

[1] R. Peña. *Diseño de Programas: Formalismo y Abstracción*. Tercera edición. Pearson Prentice-Hall, 2005.