UNIVERSIDAD FRANCISCO DE VITORIA
VINCE IN BONO MALUM

Estructuras de Datos y Algoritmos

Tema 5.3. Árboles.
Colas con prioridad y Montículos

Prof. Dr. P. Javier Herrera

Contenido

- Colas con prioridad
  - Operaciones básicas
  - Especificación de colas con prioridad
- Árboles completos y semicompletos
- Montículos
  - Propiedades
  - Representación de un árbol semicompleto en un vector
  - Implementación de montículos
  - Convertir un vector en un montículo
  - Algoritmo de ordenación heapsort
  - Montículo con prioridades variables

Colas con prioridad

- En las colas “ordinarias” se atiende por riguroso orden de llegada (FIFO).
- También hay colas, como las de los servicios de urgencias, en las cuales se atiende según la urgencia y no según el orden de llegada: son **colas con prioridad**.
- Cada elemento tiene una prioridad que determina quién va a ser el primero en ser atendido; para poder hacer esto, hace falta tener un *orden total* sobre las prioridades.
- El primero en ser atendido puede ser el elemento con menor prioridad (por ejemplo, el cliente que necesita menos tiempo para su atención) o el elemento con mayor prioridad (por ejemplo, el cliente que esté dispuesto a pagar más por su servicio) según se trate de **colas con prioridad de mínimos** o **de máximos**, respectivamente.
- Para facilitar la presentación de las propiedades de la estructura de cola con prioridad, los elementos se identifican con su prioridad, de forma que el orden total es sobre elementos.

Operaciones básicas

- El TAD de las colas con prioridad contiene las siguientes operaciones:
  - crear una cola con prioridad vacía,
  - añadir un elemento,
  - consultar el menor elemento,
  - eliminar el menor elemento, y
  - determinar si la cola con prioridad es vacía.

Especificación de colas con prioridad

especificación *COLAS-CON-PRIORIDAD*[*ELEM*≤]
usa *BOOLEANOS*
tipos *colapr*
operaciones

cp-vacía	:		→	*colapr*	{ constructora }
añadir	:	*colapr elemento*	→	*colapr*	{ constructora }
mínimo	:	*colapr*	$\rightarrow_p$	*elemento*	
eliminar-mín	:	*colapr*	$\rightarrow_p$	*colapr*	
es-cp-vacía?	:	*colapr*	→	*bool*	

variables
e, *f* : *elemento*
cp : *colapr*

Especificación de colas con prioridad

- A diferencia de las colas ordinarias sin prioridad, ahora las constructoras no son libres, ya que el orden en el que se añaden los elementos no importa: el mínimo siempre será el mismo, independientemente del orden de inserción.

ecuaciones

$$\text{añadir}(\text{añadir}(cp, e), f) = \text{añadir}(\text{añadir}(cp, f), e)$$

$$\text{mínimo}(\text{cp-vacía}) = \text{error}$$

$$\text{mínimo}(\text{añadir}(\text{cp-vacía}, e)) = e$$

$$\text{mínimo}(\text{añadir}(\text{añadir}(cp, e), f)) = \text{mínimo}(\text{añadir}(cp, e)) \Leftarrow \text{mínimo}(\text{añadir}(cp, e)) \leq f$$

$$\text{mínimo}(\text{añadir}(\text{añadir}(cp, e), f)) = f \Leftarrow \text{mínimo}(\text{añadir}(cp, e)) > f$$

Especificación de colas con prioridad

eliminar-mín(cp-vacía) = error

eliminar-mín(añadir(cp-vacía, e)) = cp-vacía

eliminar-mín(añadir(añadir(cp, e), f)) = añadir(eliminar-mín(añadir(cp, e)), f)
$\Leftarrow$ mínimo(añadir(cp, e)) $\leq f$

eliminar-mín(añadir(añadir(cp, e), f)) = añadir(cp, e) $\Leftarrow$ mínimo(añadir(cp, e)) $> f$

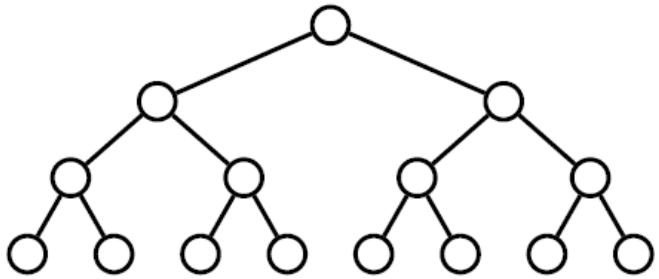
es-cp-vacía?(cp-vacía) = cierto

es-cp-vacía?(añadir(cp, e)) = falso

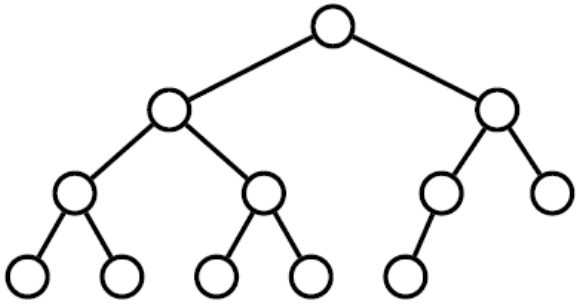
fespecificación

Árboles completos y semicompletos

- Un árbol binario de altura h es **completo** cuando todos sus nodos internos tienen dos hijos no vacíos, y todas sus hojas están en el nivel h.



- Un árbol binario de altura h es **semicompleto** si o bien es completo o tiene vacantes una serie de posiciones consecutivas del nivel h empezando por la derecha, de tal manera que al rellenar dichas posiciones con nuevas hojas se obtiene un árbol completo.



Árboles completos y semicompletos

especificación *ÁRBOLES-(SEMI)COMPLETOS*[*ELEM*]
usa *ÁRBOLES-BINARIOS*[*ELEM*], *BOOLEANOS*
operaciones

es-completo? : *árbol-bin* → *bool*
es-semicompleto? : *árbol-bin* → *bool*

variables

e : *elemento* ; *iz*, *dr* : *árbol-bin*

ecuaciones

es-completo?(árbol-vacío) = cierto
es-completo?(plantar(*iz*, *e*, *dr*)) = es-completo?(*iz*) ∧ es-completo?(*dr*) ∧
(altura(*iz*) == altura(*dr*))
es-semicompleto?(árbol-vacío) = cierto
es-semicompleto?(plantar(*iz*, *e*, *dr*)) = es-completo?(plantar(*iz*, *e*, *dr*)) ∨
(altura(*iz*) == altura(*dr*) ∧ es-completo?(*iz*) ∧ es-semicompleto?(*dr*)) ∨
(altura(*iz*) == altura(*dr*) + 1 ∧ es-semicompleto?(*iz*) ∧ es-completo?(*dr*))

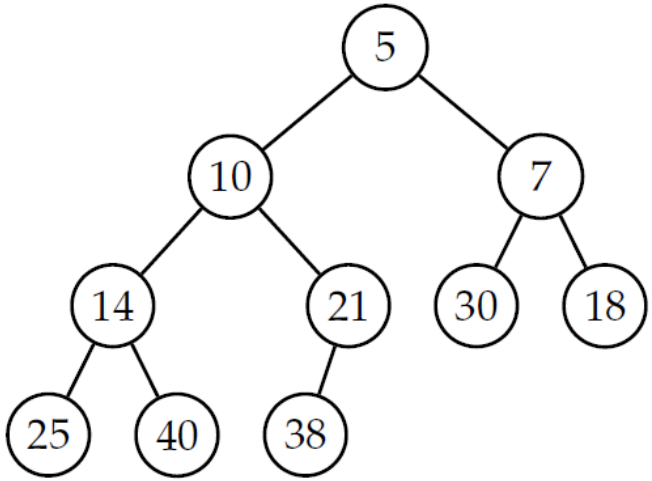
fespecificación

Montículos

- Los montículos son implementaciones eficientes de las colas de prioridad.

- Un **montículo de mínimos** es un árbol binario semicompleto donde el elemento en la raíz es menor que todos los elementos en el hijo izquierdo y en el derecho, y ambos hijos son a su vez montículos de mínimos.

- Equivalentemente, el elemento en cada nodo es menor que los elementos en las raíces de sus hijos y, por tanto, que todos sus descendientes; así, la raíz del árbol contiene el mínimo de todos los elementos en el árbol.



Montículos

especificación *ES-MONTÍCULO*[*ELEM*≤]
usa *ÁRBOLES-(SEMI)COMPLETOS*[*ELEM*≤], *BOOLEANOS*
operaciones

es-montículo? : *árbol-bin* → *bool*

operaciones privadas

menor-igual? : *elemento árbol-bin* → *bool*

variables

e, *f* : *elemento* ; *iz*, *dr* : *árbol-bin*

ecuaciones

es-montículo?(árbol-vacío) = cierto

es-montículo?(plantar(*iz*, *e*, *dr*)) = es-semicompleto?(plantar(*iz*, *e*, *dr*)) ∧ menor-igual?(*e*, *iz*) ∧ menor-igual?(*e*, *dr*) ∧ es-montículo?(*iz*) ∧ es-montículo?(*dr*)

menor-igual?(*e*, árbol-vacío) = cierto

menor-igual?(*e*, plantar(*iz*, *f* , *dr*)) = $e \leq f$ ∧ menor-igual?(*e*, *iz*) ∧ menor-igual?(*e*, *dr*)

fespecificación

Propiedades

- Un árbol binario completo de altura $h \geq 1$ tiene 2^{i-1} nodos en el nivel i, para todo i entre 1 y h.

- Un árbol binario completo de altura $h \geq 1$ tiene 2^{h-1} hojas.

- Un árbol binario completo de altura $h \geq 0$ tiene $2^h - 1$ nodos.

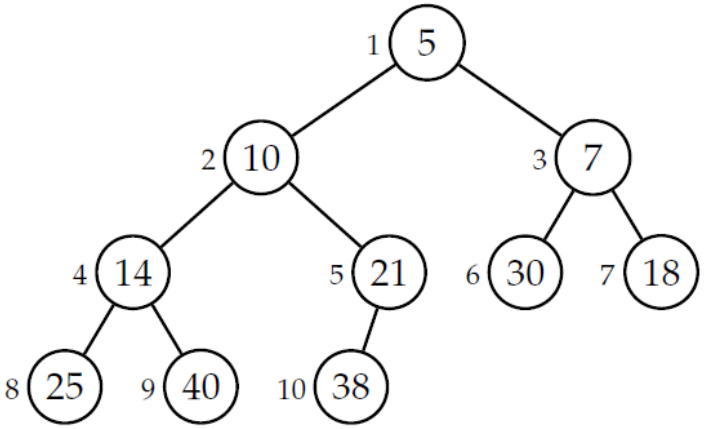
- La altura de un árbol binario *semicompleto* formado por n nodos es $\lfloor \log n \rfloor + 1$.

Representación de un árbol semicompleto en un vector

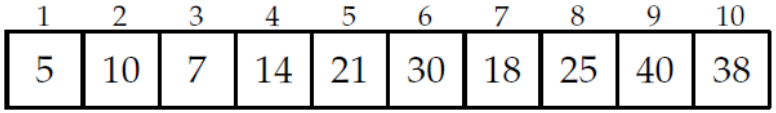
- Un árbol binario se puede representar fácilmente utilizando un vector $V[1..N]$, para N suficientemente grande, mediante la siguiente asignación de posiciones a nodos. La raíz del árbol se almacena en la primera posición del vector, y un nodo almacenado en la posición i tiene a su hijo izquierdo en la posición $2i$ y a su hijo derecho en la posición $2i + 1$. Como estas operaciones son ambas inyectivas y devuelven resultados en conjuntos disjuntos (números pares e impares respect.), cada nodo del árbol se almacena en una posición diferente del vector.

- Con esta representación también es fácil conocer al padre de un nodo, pues el padre del nodo almacenado en la posición i es el que se encuentra en la posición $i \text{ div } 2$.

Representación de un árbol semicompleto en un vector



- En general, esta representación puede dejar muchos huecos en el vector; sin embargo, cuando el árbol binario es *semicompleto*, como es el caso de un montículo, esta representación corresponde al recorrido del árbol por niveles y no deja huecos en el vector entre las posiciones 1 y N, donde N es ahora el número de nodos del árbol.



- El primer nodo del nivel j se coloca en la posición 2^{j-1}.

- El número total de nodos en los niveles del 1 al $j-1$ es igual a $\sum_{k=1}^{j-1} 2^{k-1} = 2^{j-1} - 1$
  Así, no puede haber huecos hasta la posición 2^{j-1}. Además, los elementos del nivel j del árbol se encuentran de forma consecutiva entre las posiciones 2^{j-1} y $2^j - 1$.

Implementación de montículos

- El tamaño del vector N está fijo e indica el número máximo de elementos del montículo (nº de nodos del árbol). Utilizamos también un número natural *último* para indicar el número de elementos del montículo en un momento dado (equivalentemente, la última posición ocupada en el vector). Así, el elemento en la posición i tiene hijo izquierdo si $2i \leq \textit{último}$ y tiene hijo derecho si $2i + 1 \leq \textit{último}$.

tipos

montículo = **reg**
V[1..*N*] **de** elemento
último : 0..N
freg

ftipos

- Nótese que el requisito de orden entre los elementos que se debe cumplir en un montículo no se refleja en el tipo, por lo que este realmente representa árboles semicompletos. El requisito de orden se logrará por cómo se añaden y eliminan elementos.

Implementación de montículos

- El montículo vacío no tiene elementos.

fun montículo-vacío() **dev** M : montículo $\{O(1)\}$
    $M.último := 0$
ffun

- Solamente hay una forma de añadir un nodo a un árbol semicompleto manteniendo esta propiedad: si "faltan" nodos en el último nivel, el nuevo nodo será el siguiente (en el recorrido por niveles) al último nodo de ese nivel, y si el árbol de partida es completo el nuevo nodo será el de más a la izquierda en un nuevo nivel.

Implementación de montículos

- En la representación como vector de un árbol semicompleto esto se refleja en que el nuevo elemento se coloca en la posición siguiente al *último*. Pero este elemento no tiene por qué estar bien colocado en esa posición con respecto a la propiedad de orden en el montículo; para mantener dicha propiedad, el elemento tiene que "**flotar**", mediante la operación auxiliar flotar que veremos a continuación.

proc añadir(M : montículo, **e** e : elemento) $\{O(\log M.último)\}$

  si $M.último = N$ **entonces** error(Espacio insuficiente)

  si no

    $M.último := M.último + 1$

    $M.V[M.último] := e$

    flotar($M.V$, $M.último$)

  fsi

fproc

- El coste de añadir es proporcional al de flotar que como veremos el **logarítmico** respecto al número de elementos del montículo.

Implementación de montículos

- El elemento a flotar no está bien colocado si es menor que su padre, por lo que hay que intercambiarlo con este. El proceso se repite hasta el que el padre sea menor o igual, o lleguemos a la raíz.

proc flotar(V[1..N] **de** elemento, **e** j : 1..N) $\{O(\log j)\}$

$i := j$

mientras $i \neq 1 \wedge_c V[i] < V[i \text{ div } 2]$ **hacer**

$\{ V[i], V[i \text{ div } 2] \} := \{ V[i \text{ div } 2], V[i] \}$

$i := i \text{ div } 2$

fmientras

fproc

- El coste es proporcional a la altura del montículo porque en cada iteración del bucle se sube un nivel en el árbol, y como se indica en las propiedades, la altura es logarítmica respecto al número de elementos en el montículo.

Implementación de montículos

- Para eliminar el mínimo hay que quitar el elemento en la primera posición del vector y seguir manteniendo un montículo en el vector. Para conseguir que siga siendo semicompleto, el hueco que aparece en la primera posición del vector al borrar el mínimo se cubre con el elemento en la última posición (*último*). Para mantener el orden correcto entre los elementos, el elemento movido tiene que "**hundirse**", lo que hace la operación auxiliar hundir.

proc eliminar-mín(M : montículo) $\{O(\log M.último)\}$

    si $M.último = 0$ **entonces** error(Montículo vacío)

    si no $M.V[1] := M.V[M.último]$

        $M.último := M.último - 1$

        hundir($M.V$, 1, $M.último$)

    fsi

fproc

- El coste de eliminar-min es proporcional al de hundir que también es **logarítmico** respecto al número de elementos en el montículo.

Implementación de montículos

- El elemento a hundir no está bien colocado si es mayor que alguno de sus hijos. El problema se resuelve intercambiando el elemento con el menor de los hijos, y repitiendo el proceso hasta que el elemento que se hunde sea menor o igual que sus hijos, o lleguemos a una hoja.

- El coste es proporcional a la altura del montículo porque en cada iteración del bucle se baja un nivel en el árbol, y como sabemos la altura es logarítmica con respecto al número de elementos en el montículo.

Implementación de montículos

$\{ j \leq k \}$
proc hundir(V[1..N] **de** elemento, **e** j, k : 1..N) $\{ O(\log k) \}$
    fin := falso ; $i := j$
    mientras $2 * i \leq k \wedge \neg fin$ **hacer** { la posición i no es una hoja }
        { mínimo de los hijos }
        si $2 * i + 1 \leq k \ \wedge_c \ V[2 * i + 1] < V[2 * i]$ **entonces** $m := 2 * i + 1$
        si no $m := 2 * i$
        fsi
        { si es necesario, se intercambia con el mínimo de los hijos }
        si $V[m] < V[i]$ **entonces** $\{ V[i], V[m] \} := \{ V[m], V[i] \}$; $i := m$
        si no fin := cierto
        fsi
    fmientras
fproc

Implementación de montículos

- En un montículo, el mínimo se encuentra en la raíz, esto es, en la primera posición del vector.

fun mínimo(M : montículo) **dev** e : elemento $\{O(1)\}$
    si $M.último = 0$ **entonces** error(Montículo vacío)
    si no $e := M.V[1]$
    fsi
ffun

- Un montículo es vacío cuando no tiene elementos.

fun es-montículo-vacío?(M : montículo) **dev** b : bool $\{O(1)\}$
    $b := (M.último = 0)$
ffun

Convertir un vector en un montículo

- La primera versión utiliza flotar. Recorre el vector de izquierda a derecha, y mantiene que las posiciones por las que ya ha pasado forman un montículo, representado en dichas posiciones del vector. Cuando se avanza a la siguiente posición, se aplica la misma idea que para añadir un elemento a un montículo: el nuevo elemento se incorpora a los que forman el montículo haciéndolo flotar entre estos hasta que ocupe el lugar apropiado.

```
proc monticulizar1(V[1..N] de elemento)
    para j = 2 hasta N hacer
        flotar(V, j)
    fpara
fproc
```

nivel	nodos	flotan
2	2	cada uno 1
3	4	cada uno 2
	⋮	
i	2^{i-1}	cada uno $i-1$
	⋮	
h	2^{h-1}	cada uno $h-1$

$$\sum_{i=2}^{h}(i-1)2^{i-1}=\sum_{j=1}^{h-1}j2^{j}=(h-2)2^{h}+2=(\lfloor\log N\rfloor-1)2^{\lfloor\log N\rfloor+1}+2\in O(N\log N)$$

Convertir un vector en un montículo

- La segunda versión utiliza hundir. Tenemos que reorganizar los elementos de forma que los descendientes de cada uno sean mayores o iguales. Esto se consigue por niveles comenzando por las hojas. Por tanto, se realiza un recorrido de derecha a izquierda en el vector y, como las hojas de un árbol semicompleto de tamaño N se encuentran a partir de la posición N div $2 + 1$ en el vector, basta hacer ese recorrido empezando en la posición N div 2.

```
proc monticulizar2(V[1..N] de elemento)
    para j = N div 2 hasta 1 paso −1 hacer
        hundir(V, j, N)
    fpara
fproc
```

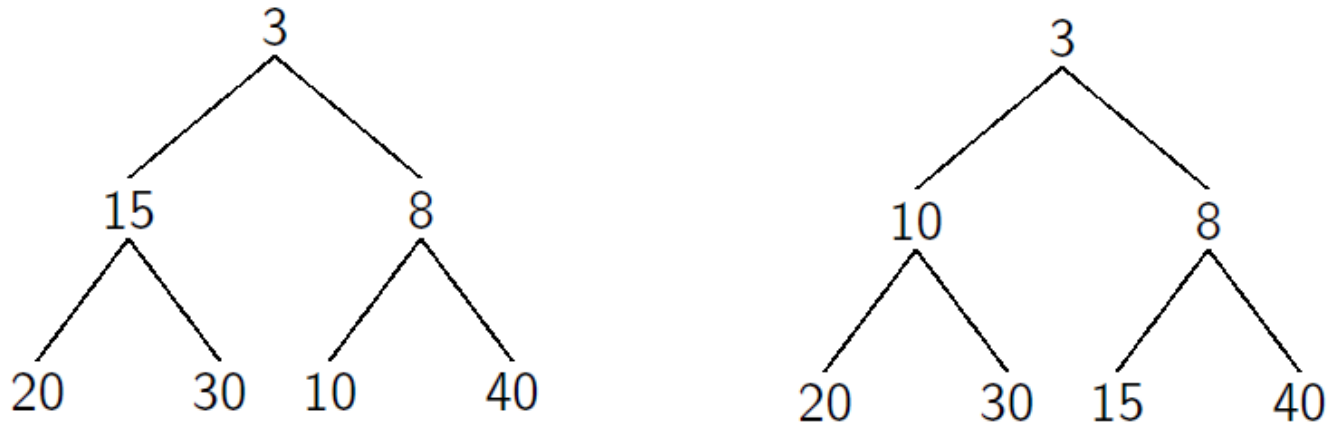
nivel	nodos	hunden
h	2^{h-1}	nada
$h-1$	2^{h-2}	cada uno 1
$h-2$	2^{h-3}	cada uno 2
	$\vdots$	
i	2^{i-1}	cada uno $h-i$
	$\vdots$	
1	1	$h-1$

$$\sum_{i=1}^{h-1}(h-i)2^{i-1} = \sum_{j=2}^{h}(j-1)2^{h-j} < \sum_{j=1}^{h} j2^{h-j} = 2^h \sum_{j=1}^{h} \frac{j}{2^j} = 2^h\left(2 - \frac{k+2}{2^k}\right) \leq 2^{h+1} = 2^{\lfloor \log N \rfloor + 2} \in O(N)$$

Convertir un vector en un montículo

- El montículo obtenido utilizando uno u otro método no tiene por qué ser el mismo.
- El montículo de la izquierda se obtiene a partir del vector V utilizando monticulizar1, mientras que el de la derecha se obtiene utilizando monticulizar2.

	1	2	3	4	5	6	7
V	15	10	8	20	30	3	40



Algoritmo de ordenación *heapsort*

- El algoritmo de ordenación por el **método del montículo** (*heapsort*) consta de dos fases. En la primera, todos los elementos del vector a ordenar se añaden uno a uno a un montículo inicialmente vacío. En la segunda fase, se va extrayendo sucesivamente el mínimo del montículo y se va colocando en la correspondiente posición del vector, que se recorre de izquierda a derecha.
- Implementación utilizando una cola de prioridad de forma abstracta:

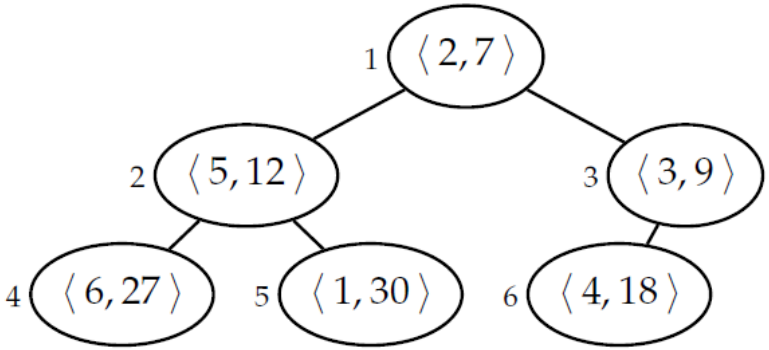
```
proc heapsort-abstracto(V[1..N] de elemento)
var C : colapr
    C := cp-vacía()
    para i = 1 hasta N hacer
       añadir(C,V[i])
    fpara
    para i = 1 hasta N hacer
       V[i] := mínimo(C)  ;  eliminar-mín(C)
    fpara
fproc
```

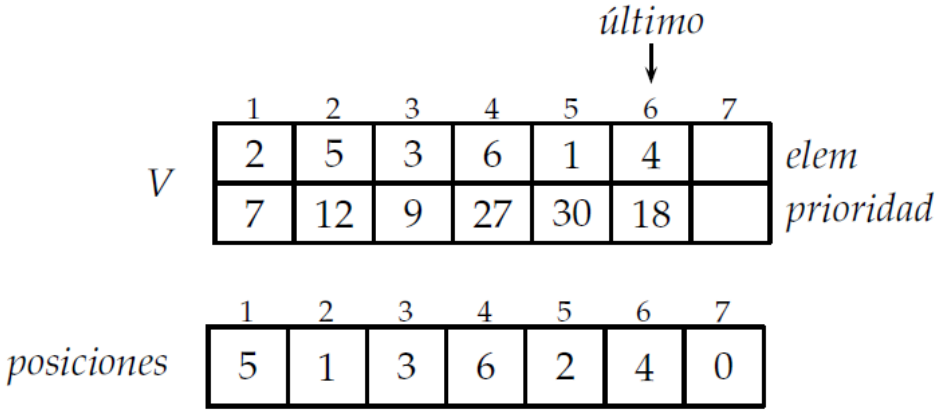
- El coste depende de la implementación concreta de la cola de prioridad. Supongamos que se utiliza un montículo. En la primera fase, el proceso es el mismo que en monticulizar1, por lo que su coste está en $O(N \log N)$. En la segunda fase, el coste de eliminar el mínimo en un montículo de tamaño j está en $O(\log j)$. Por tanto, el coste del algoritmo está en $O(N \log N)$.

Montículo con prioridades variables

- Supongamos que queremos utilizar un montículo para almacenar pares de la forma $\{ elem, prioridad \}$ donde *elem* es un número natural en el intervalo $1..N$, las prioridades son números reales, y el *elem* de todos los pares es diferente. El orden entre los pares viene inducido por el orden entre las prioridades.

- Implementar el montículo de tal forma que se pueda modificar la prioridad asociada a un elemento en el montículo y mantener las propiedades de la estructura en tiempo logarítmico.





Montículo con prioridades variables

- El montículo se representa en un vector $V[1..N]$, donde los elementos son pares $\{ elem, prioridad \}$. Para poder saber en tiempo constante donde está en el montículo un *elem* concreto, utilizaremos además un vector $posiciones[1..N]$ tal que $posiciones[i]$ es la posición de V donde se encuentra el elemento i, $V[posiciones[i]].elem = i$. Si el elemento i no está en el montículo, $posiciones[i] = 0$.

tipos

par = **reg**
elem : $1..N$
prioridad : real
freg

montículo-pares = **reg**
$V[1..N]$ **de** par
$posiciones[1..N]$ **de** $0..N$
último : $0..N$
freg

ftipos

Montículo con prioridades variables

- La implementación de las operaciones es muy similar a la vista para montículos simples, sólo que ahora cualquier modificación de V tiene que reflejarse también en *posiciones*.

- El montículo vacío no tiene ningún elemento.

fun montículo-vacío() **dev** M : montículo-pares $\{ O(N) \}$
    $M.último := 0$
    $M.posiciones := [0]$
ffun

- La función mínimo ahora devuelve un par, el que se encuentra en la posición 1 de V.

fun mínimo(M : montículo-pares) **dev** p : par $\{ O(1) \}$
    si $M.último = 0$ **entonces** error(Montículo vacío)
    si no $p := M.V[1]$
    fsi
ffun

Montículo con prioridades variables

- Al eliminar el mínimo hay que marcar que el *elem* en la posición 1 de V ya no estará, y que el último se ha movido a esa posición. Ahora la operación hundir-pares recibe el montículo completo, para que modifique tanto V como *posiciones*.

proc eliminar-mín(M : montículo-pares) $\{ O(\log N) \}$

  si $M.último = 0$ **entonces** error(Montículo vacío)

  si no

    $M.posiciones[M.V[1].elem] := 0$

    $M.V[1] := M.V[M.último]$; $M.posiciones[M.V[1].elem] := 1$

    $M.último := M.último - 1$

    hundir-pares(M, 1)

  fsi

fproc

Montículo con prioridades variables

- El procedimiento auxiliar hundir-pares es prácticamente idéntico, sólo que ahora la comparación es entre prioridades, y se utiliza un procedimiento intercambiar que tiene en cuenta las *posiciones*.

proc intercambiar(M : montículo-pares, i, j : $1..N$)

$\{ M.V[i], M.V[j] \} := \{M.V[j], M.V[i] \}$

$M.posiciones[M.V[i].elem] := i$

$M.posiciones[M.V[j].elem] := j$

fproc

Montículo con prioridades variables

proc hundir-pares(M : montículo-pares, **e** j : 1..N) { $O(\log N)$ }
  fin := falso ; $i := j$; $k := M.último$
  mientras $2 * i \leq k \ \wedge \ \neg fin$ **hacer**
    { mínimo de los hijos }
    si $2 * i + 1 \leq k \ \wedge_c \ M.V[2 * i + 1].prioridad < M.V[2 * i].prioridad$ **entonces**
      $m := 2 * i + 1$
    si no $m := 2 * i$
    fsi
    { si es necesario, se intercambia con el mínimo de los hijos }
    si $M.V[m].prioridad < M.V[i].prioridad$ **entonces**
      intercambiar(M, i, m) ; $i := m$
    si no fin := cierto
    fsi
  fmientras
fproc

Montículo con prioridades variables

- La operación añadir recibe ahora un elemento y una prioridad. El nuevo par se coloca en la posición siguiente a la última, apuntándolo en *posiciones*. Después se utiliza flotar-pares.

proc añadir(*M* : montículo-pares, **e** *v* : 1..*N*, **e** *p* : real) $\{O(\log N)\}$

```
    si M.posiciones[v] ≠ 0 entonces error(El elemento ya está)
    si no
        si M.último = N entonces error(Montículo lleno)
        si no
            M.último := M.último + 1
            M.V[M.último].elem := v  ;  M.V[M.último].prioridad := p
            M.posiciones[v] := M.último
            flotar-pares(M, M.último)
        fsi
    fsi
fproc
```

Montículo con prioridades variables

proc flotar-pares(M : montículo-pares, **e** j : $1..N$) $\{O(\log N)\}$

    $i := j$

    mientras $i \neq 1 \ \wedge_c \ M.V[i].prioridad < M.V[i \text{ div } 2].prioridad$ **hacer**

        intercambiar(M, i, i div 2)

        $i := i$ div 2

    fmientras

fproc

- La nueva operación modificar recibe el elemento del cual se quiere modificar su prioridad y la nueva prioridad, que puede ser mayor o menor. Para colocar correctamente el par modificado, se compara con su padre. Si se ha hecho menor que su padre, entonces se "flota". En caso contrario, o si no tiene padre, el par se "hunde". Si el elemento a modificar no está, entonces se añade por primer vez.

Montículo con prioridades variables

```
proc modificar(M : montículo-pares, e v : 1..N, e p : real )        { O(log N) }
    i := M.posiciones[v]
    si i = 0 entonces añadir(M, v, p)
    si no
        M.V[i].prioridad := p
        si i ≠ 1  ∧c  M.V[i].prioridad < M.V[i div 2].prioridad entonces
            flotar-pares(M, i)
        si no
            hundir-pares(M, i)
        fsi
    fsi
fproc
```

Bibliografía

- Martí, N., Ortega, Y., Verdejo, J.A. *Estructuras de datos y métodos algorítmicos*. Ejercicios resueltos. Pearson/Prentice Hall, 2003. Capítulo 8

- Peña, R.; *Diseño de programas. Formalismo y abstracción*. Tercera edición. Prentice Hall, 2005. Sección 7.5

(Estas transparencias se han realizado a partir de aquéllas desarrolladas por los profesores Clara Segura, Alberto Verdejo y Yolanda García de la UCM, y la bibliografía anterior)