Tema 2. MONTÍCULOS BINARIOS

ESTRUCTURAS DE DATOS Y ALGORITMOS II

Grado en Ingeniería Informática

María José Polo Martín

mjpolo@usal.es

Curso 2015-2016

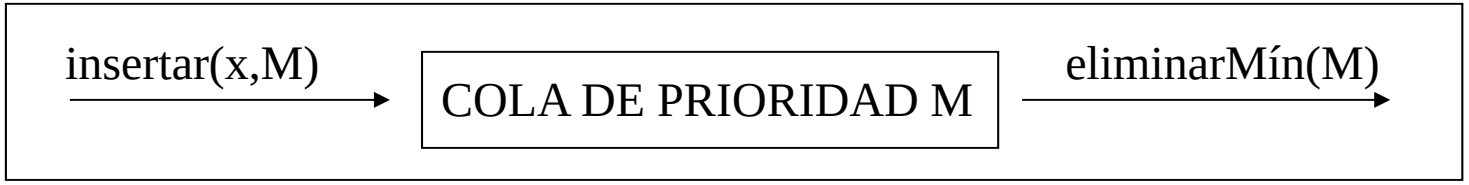
VNiVERSiDAD D SALAMANCA
CAMPUS DE EXCELENCIA INTERNACIONAL

DEPARTAMENTO DE iNFORMÁTiCA Y AVTOMÁTiCA

Tema 2. Montículos Binarios

1 INTRODUCCIÓN

- **Cola de PRIORIDAD.** Tipo abstracto de datos que almacena una colección de elementos que poseen una clave perteneciente a algún conjunto ordenado y que permite al menos las dos operaciones siguientes:
  - Insertar: inserta un elemento en la cola
  - Eliminar elemento mínimo: busca, devuelve y elimina el elemento mínimo de la cola



Aplicaciones y posibles implementaciones

- Aplicaciones:
  - Colas de impresión con prioridades
  - Planificador sistema operativo en entorno multiusuario
  - Implantación eficiente de algunos algoritmos de grafos (tema 4)
- Posibles implementaciones:
  - Listas enlazadas
  - Listas enlazadas clasificadas
  - Árbol binario de búsqueda
  - **Montículo binario ⇔ montículo**

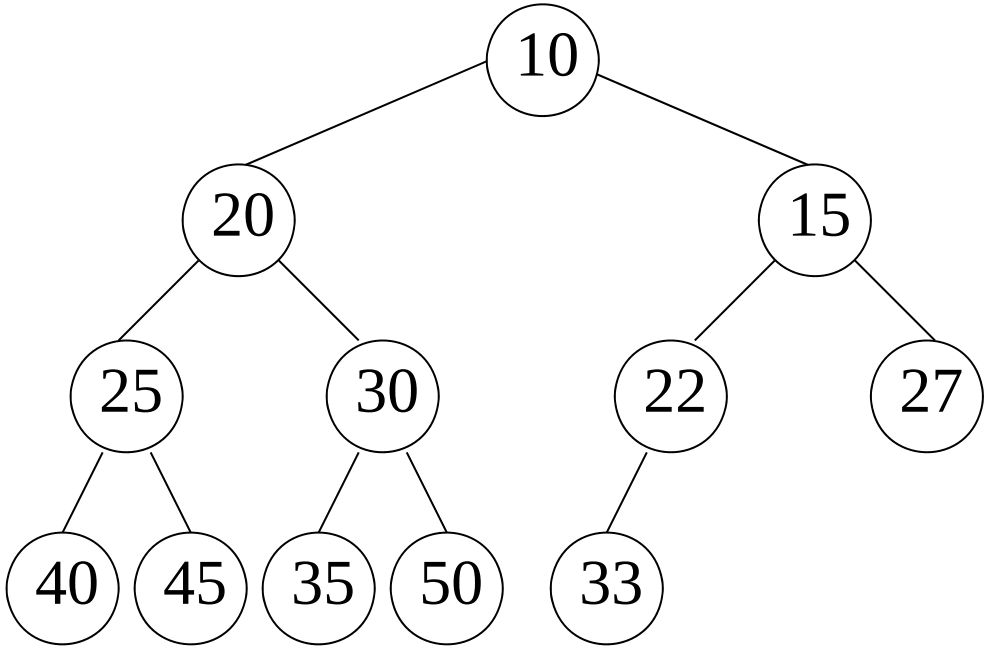
2 NIVEL ABSTRACTO O DE DEFINICIÓN

- **Definición:** árbol binario casi-completo (completo, con la posible excepción del nivel más bajo, el cual se llena de izquierda a derecha) en el cual, para todo nodo *n* se cumple la siguiente condición:

  "la clave en el padre de *n* es menor (o igual) que la clave de *n*, con la excepción obvia de la raíz (que no tiene nodo padre)"

- Observaciones:
  - Regularidad de un a.b. casi-completo ⇒ se puede representar mediante una matriz sin necesidad de recurrir a punteros
  - El elemento con valor mínimo siempre se encuentra en la raíz

Ejemplo



?	10	20	15	25	30	22	27	40	45	35	50	33			
0	1	2	3	4	5	6	7	8	9	10	11	12			

Propiedades

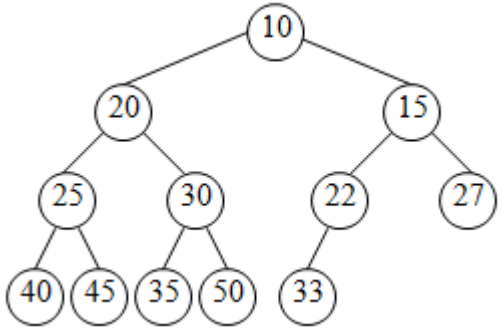
- Propiedad de orden

  El elemento con valor mínimo siempre se encuentra en la raíz

- Propiedad de la estructura

  Para cualquier elemento en la posición *i* de la matriz, el hijo izquierdo está en la posición *2i*, el hijo derecho está en la posición siguiente al hijo izquierdo (*2i* + *1*) y el padre está en la posición *[i/2]*

?	10	20	15	25	30	22	27	40	45	35	50	33			
0	1	2	3	4	5	6	7	8	9	10	11	12			



Operaciones

- **insertar(x,m)**: inserta un nuevo elemento con clave *x* en el monticulo *m*
- **eliminarMin(m)**: elimina y devuelve el elemento mínimo del monticulo m
- **decrementarClave(p,x,M)**: reduce el valor de la clave en la posición *p* del montículo *m* una cantidad positiva *x*
- **incrementarClave(p,x,m)**: aumenta el valor de la clave en la posición p del montículo m en una cantidad positiva x
- **construirMontículo(m)**: toma como entrada *n* claves y las coloca en un montículo vacío

3 NIVEL DE REPRESENTACIÓN

- Un montículo binario puede representarse mediante una matriz y un entero que indique el tamaño actual del montículo

declaraciones básicas

constante

MÁXIMO = 100

tipo

tipoMontículo= **registro**

elementos:matriz[0..MÁXIMO] **de** tipoElemento

tamaño: entero

fin registro

Inserción de un elemento en el montículo

- Añadir un elemento *x* al montículo sin violar las propiedades que los definen (propiedad de estructura y propiedad de orden)
- Proceso:

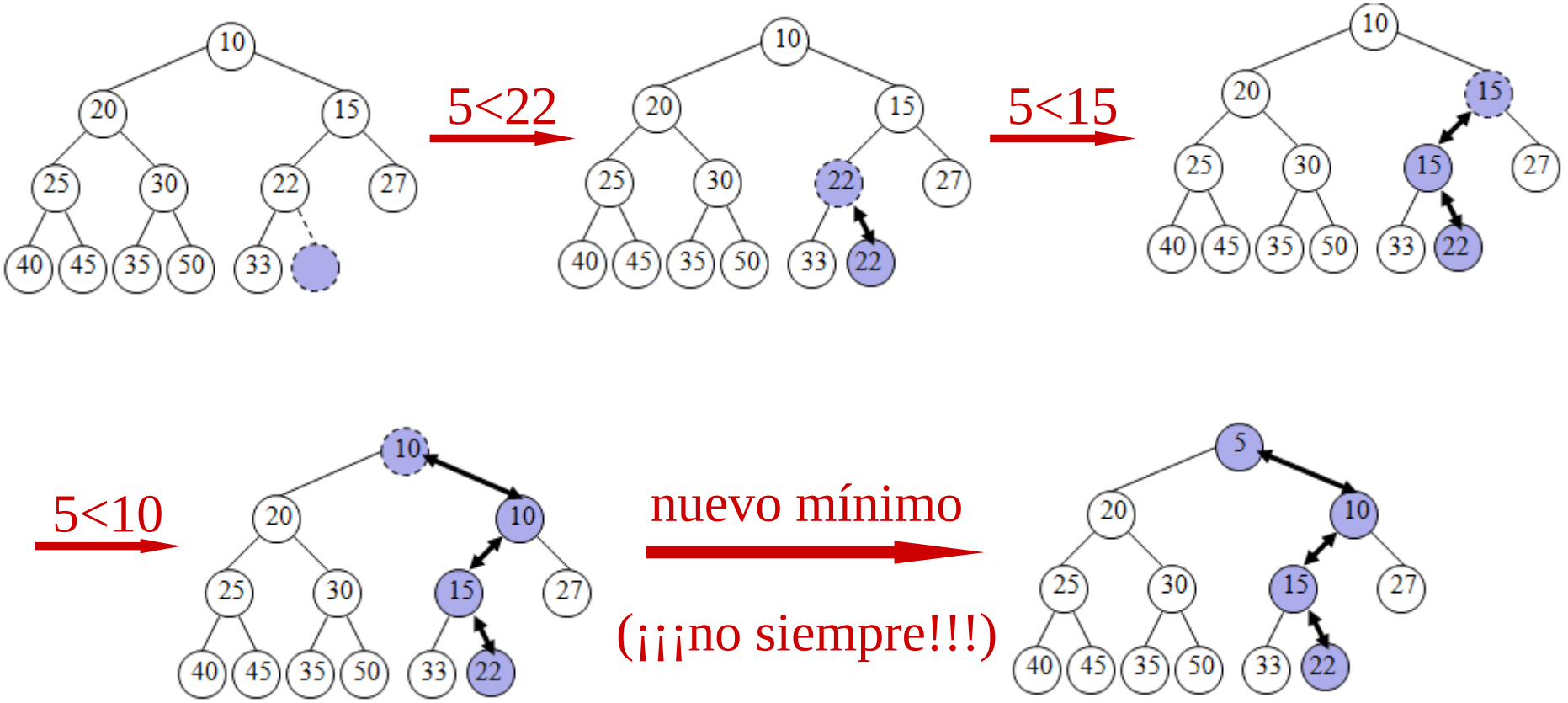
1 Añadir un nuevo nodo (hueco) en la siguiente posición disponible del a.b. casi-completo.

2 Distinguir los siguientes casos:

- Si *x* se puede asignar al hueco sin violar la propiedad de orden del montículo, se asigna y finaliza el proceso
- Si la asignación de *x* viola la propiedad de orden, se desliza el elemento del nodo padre al hueco, subiendo el hueco hacia la raíz, hasta poder asignar *x* al hueco

Estrategia de ***filtrado ascendente***: el nuevo elemento se filtra en el montículo hasta encontrar su posición correcta

Ejemplo filtrado ascendente: insertar un nuevo con clave 5 en el montículo



Algoritmo de INSERCIÓN

```
procedimiento insertar(x:tipoElemento, ref m:tipoMontículo)
1.  i : entero
2.  si m.tamaño = MÁXIMO entonces
3.     error ("Montículo lleno")
4.  sino
5.     m.tamaño ← m.tamaño + 1
6.     i ← m.tamaño
7.     mientras m.elemento[i div 2].clave > x.clave hacer
8.       m.elemento[i].clave ← m.elemento[i div 2].clave
9.       m.elemento[i].información ← m.elemento[i div 2].información
10.          i ← i div 2
11.    fin mientras
12.    m.elemento[i].clave ← x.clave
13.    m.elemento[i].información ← x.información
14. fin  si
```

Filtrado Ascendente

Observaciones:

- Si el elemento a insertar es el nuevo mínimo, el hueco debe subir hasta la raíz. En el momento que *i* tome el valor 1 habrá que romper el ciclo mientras
- Dos soluciones:
  1. Comprobación explícita en la condición del bucle (i ≠ 1 AND ...)
  2. Asignar un valor (centinela) en la posición 0 del array (menor o igual que cualquier elemento del montículo) que asegure que el bucle termina

Eliminación del elemento mínimo del montículo

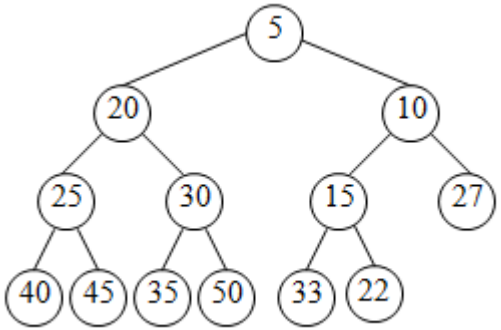
- Eliminar del montículo el elemento que contiene la clave mínima sin violar las propiedades que los definen (propiedad de estructura y propiedad de orden). Según la propiedad de orden será el nodo raíz del montículo.
- Proceso (Estrategia de ***filtrado descendente***):

1 Se elimina el valor del elemento mínimo creando un **hueco** en la raíz

2 Como el tamaño del montículo se reduce en una unidad, para conservar la propiedad de la estructura el último elemento *x* del montículo debe moverse a otro lugar

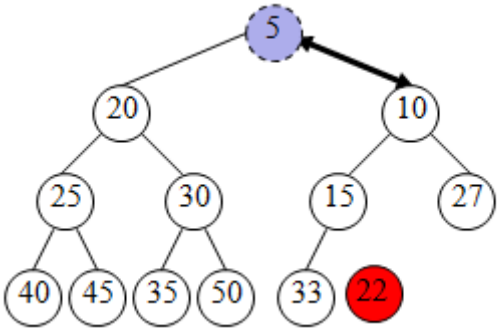
- Si *x* se puede asignar al hueco sin violar la propiedad de orden del montículo, se asigna y finaliza el proceso
- Si la asignación de *x* viola la propiedad de orden, se intercambia el hueco con el menor de sus dos hijos, empujando el hueco hacia abajo un nivel, hasta poder asignar *x* al hueco

Ejemplo filtrado descendente: eliminar el elemento mínimo del montículo

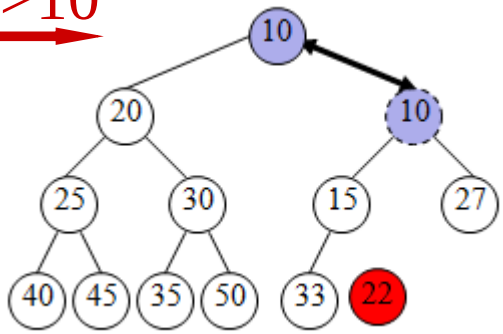


hueco en la raíz

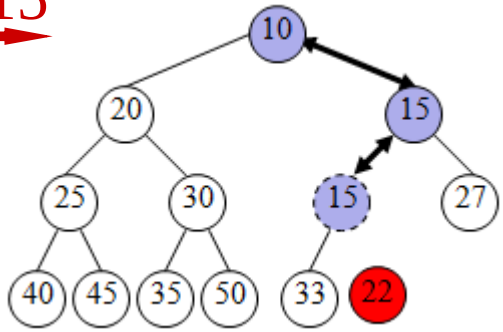
recolocar 22



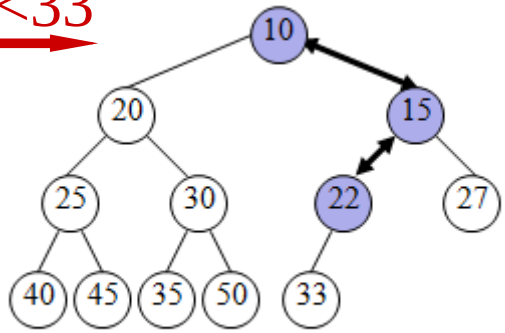
22>10



22>15



22<33



Algoritmo de ELIMINACIÓN

```
función eliminarMin(ref m: tipoMontículo):tipoElemento
1.  último, mínimo:tipoElemento
2.  si vacio(m) entonces
3.    error ( "Montículo vacío" )
4.  sino
5.      mínimo.clave ← m.elemento[1].clave
6.      mínimo.información ← m.elemento[1].información
7.      último.clave ← m.elemento[m.tamaño].clave
8.      último.información ← m.elemento[m.tamaño].información
9.      m.tamaño ← m.tamaño - 1
10.     i ← 1
11.     finFiltrado ← FALSO
12.     mientras (2*i ≤ m.tamaño AND NOT finFiltrado) hacer

                .......


15.     fin mientras
16.     m.elemento[i].clave ← último.clave
17.     m.elemento[i].información ← último.información
18.     devolver(mínimo)
19. fin si
```

Filtrado Descendente

IO
iCA
A

Filtrado descendente

```
12. mientras (2*i ≤ m.tamaño AND NOT finFiltrado) hacer
13.         hijo ← 2 * i
14.         si hijo ≠ m.tamaño entonces
15.           si m.elemento[hijo+1].clave<m.elemento[hijo].clave entonces
16.             hijo ← hijo + 1
17.           fin si
18.         fin si
19.         si último.clave > m.elemento[hijo].clave entonces
20.             m.elemento[i].clave ← m.elemento[hijo].clave
21.             m.elemento[i].información←m.elemento[hijo].información
22.             i ← hijo
23.         sino
24.           finFiltrado ← VERDADERO
25.          fin si
26.       fin mientras
```

Observaciones

- En el filtrado descendente se compara la clave del último elemento con la del menor de los hijos del hueco. Debe tenerse en cuenta que el último nodo interno puede tener un único hijo
- Las operaciones decrementarClave e incrementarClave necesitarán las estrategias de filtrado ascendente y descendente, respectivamente, para mantener la propiedad de orden si esta se pierde
- Dos soluciones para la operación construirMontículo que toma como entrada *n* claves y las coloca en un montículo vacío

  Solución 1: Realizar *n* operaciones insertar sucesivas⇒ O(nlogn)

  Solución 2: ⇒ O(n)

  - Colocar las *n* claves dentro del árbol en cualquier orden, manteniendo la propiedad de la estructura
  - Realizar un filtrado descendente para obtener un montículo

Algoritmo construirMontículo (solución 2)

procedimiento construirMontículo(**ref** m: tipoMontículo)

1. i : entero
2. n: entero
3. n ← m.tamaño
4. **para** (i = n div 2) **bajando hasta** 1 **hacer**
5. filtradoDescendente(m, i);
6. **fin para**

Análisis del algoritmo construir montículo

- Se realiza el filtrado descendente de todos los nodos internos:

  1 nodo de altura *h*

  2 nodos de altura *h-1*

  2^2 nodos de altura *h-2*

  ……

  2^{h-1} nodos de altura *1* (puede que no todos sean internos)

  2^h nodos de altura *0*

- En el peor de los casos, cada uno de estos nodos debe descender desde su nivel a un nivel hoja, es decir, como máximo su altura
- La suma de las alturas de todos los nodos de un árbol binario completo es

$$S = h + 2(h-1) + 4(h-2) + \ldots + 2^{h-1} \cdot 1$$

Análisis del algoritmo construir montículo

- Multiplicando por dos la ecuación anterior y restando ambas se obtiene

$$\left.\begin{aligned} 2S &= 2h+4(h-1)+8(h-2)+\ldots+2^{h-1}\cdot 2+2^h \\ S &= h+2(h-1)+4(h-2)+\ldots+2^{h-1} \end{aligned}\right\} \quad S=-h+2+4+\ldots+2^h$$

ya que ciertos términos se cancelan: $2h-2(h-1)=2$

$4(h-1)-4(h-2)=4$ …

$$S = -h + 2^h \cdot 2 - 2 = 2^{h+1} - (h+2) \in O(n)$$

- Puesto que un árbol completo tiene entre 2^h y 2^{h+1} nodos (transparencia 11, tema 1), la suma obtenida es $O(n)$ donde n es el número de nodos

4 ORDENACIÓN POR MONTÍCULOS

- Los montículos se pueden utilizar como método de ordenación interna con un comportamiento de *O(nlogn)*
- El algortimo basado en esta idea se denomina ordenación por montículo (*heapsort*) y se basa en:
  - Construir un montículo binario de *n* elementos
  - Efectuar *n* operaciones eliminarMin: los elementos más pequeños dejan primero el montículo
  - Cada operación eliminarMin contrae el montículo en uno. Si se aprovecha la que era última celda del montículo para almacenar el elemento eliminado, al final de la operación las *n* celdas del array contienen los elementos en orden decreciente

Análisis del algoritmo ordenación

- Crear montículo requiere un tiempo lineal (*O(n)*) y genera un montículo de altura *[logn]*
- La operación eliminarMin se ejecutará *n-1* veces y filtra la raíz a lo largo de un camino de longitud máxima *logn*, por tanto, en el peor caso requiere un tiempo que es *O(logn)*
- Habrá también *n-1* operaciones de asignación que requieren un tiempo constante (*O(1)*)
- Por tanto el tiempo, *t(n)*, necesario para ordenar la matriz de n elementos verifica

$$t(n) \in O(n)+(n-1)O(1)+(n-1)\log(n)=O(n\log n)$$

Observaciones

- Para obtener los elementos en orden creciente, basta con cambiar la propiedad de orden de forma que el padre tenga una clave mayor que los hijos y considerar la operación eliminarMax
- Definición alternativa de Montículo Binario: árbol binario casi-completo en el cual, para todo nodo *n* se cumple la siguiente condición

  "la clave en el padre de *n* es mayor (o igual) que la clave de *n*, con la excepción obvia de la raíz"
- Cambia la propiedad de orden del montículo $\Rightarrow$ el elemento con valor máximo siempre se encuentra en la raíz