



INTRODUCCIÓN

0.1. Colas de prioridad

Definición: Es un TAD que consta de una información y una funcionalidad.

1. Información: Conjunto de valores G , cada uno de los cuales con una clave asociada a cada valor y que cumplen una relación de orden.

2. Funcionalidad:

i) Acceder al valor más alto (máximo)

$\text{Max}(G) \rightarrow$ Devuelve el valor de G con clave máxima

ii) Extraer valor

$\text{Extraer}(G) \rightarrow$ Devuelve el valor de G con clave máxima y lo elimina de G

iii) Insertar valor

$\text{Insertar}(G, x) \rightarrow$ añade x a G

0.2. Implementación EDD

Forma 1: Arrays

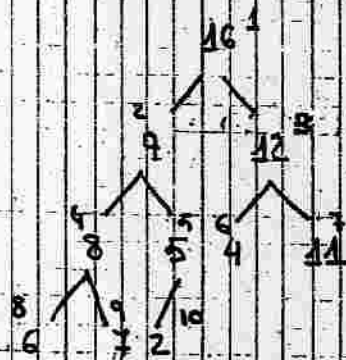
- 1) Max $O(1)$
- 2) Extraer $O(N)$
- 3) Insertar $O(N)$

En el caso de q los elementos de dentro del array estén ordenados.

- 1) Max $O(N)$
- 2) Extraer $O(N)$
- 3) Insertar $O(1)$

En el caso de q los elementos de dentro del array no estén ordenados.

Formulo 2: Heap. Donde el cuando es el valor más grande en todo momento. Es la forma más eficiente.



HEAP Definición: Un heap es un árbol binario en todos los niveles completos salvo como el último que todos los nodos cumplen la condición de heap.

| | | | | | | | | | |
|----|---|----|---|---|---|----|---|---|----|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 15 | 9 | 12 | 8 | 5 | 4 | 11 | 6 | 7 | 2 |

| | | |
|----------|---|---------------|
| hijo(i) | → | 2i |
| hijo(k) | → | 2k+1 |
| padre(i) | → | $\frac{i}{2}$ |

0.3. Primitivas

```

1) max(H)
   if tamaño_heap[H] = 0 then //control de heap vacío
       Error "underflow"
   else return H[1]
   // se devuelve el dato de la celda máxima
   // EL MÁXIMO SIEMPRE ESTÁ EN POS 1
  
```


2) Insert (H, x) \Rightarrow INSERTA NUEVA CLAVE X

if tamaño_heap[H] = longitud[H] (capacidad máxima)

Error "Overflow"

// SI HEAP LLENO

else tamaño_heap[H] ++ // AUMENTO TAMAÑO

i \leftarrow tamaño_heap[H] // NUEVA POS posición al fin del heap

while H[padre(i)] < x and i > 1

H[i] \leftarrow H[padre(i)]

// Mueve el elemento
// subiendo en el árbol

i \leftarrow padre(i)

H[i] \leftarrow x // GUARDAMOS ELEMENTO

$O(\lg N)$ \equiv Profundidad del árbol

3) Extract (H) \Rightarrow EXTRAER LA CLAVE MAXIMA (LA QUITA)

if tamaño_heap[H] = 0 then // SI HEAP VACIO

Error "Underflow"

max \leftarrow H[1] // MAXIMO

// Te devuelve el dato de
en clave máxima

H[1] \leftarrow H[tamaño_heap[H]] // EN POS 1 EL ULTIMO

tamaño_heap[H] -- // REDUZCO TAMAÑO

Heapify (H, 1)

// HAGO HEAPIFY DESD 1 // Te vuelve a construir el
árbol según sea la raíz

return max

// DEV MAX

del heap

$O(\lg N)$ \equiv Orden de Heapify

4) Heapify (H, i) \Rightarrow ORDENAR HEAP PARA CADA NODO COMO HEAD

```

l ← izquierda(i)
r ← derecha(i)
if (l ≤ tamaño_heap[H] and H[l] ≥ H[i]) then
    max ← l
else max ← i
if (r ≤ tamaño_heap[H] and H[r] ≥ H[max]) then
    max ← r
if max ≠ i then
    swap(H[i], H[max]) // swap
    Heapify(H, max) // REPIFY DESDE MAX
    
```

// Comparación de los hijos
con el padre y se elige
el mayor. Luego se
compara con el padre
si el hijo es mayor se
hace swap y se llama
a Heapify

$O(\log N)$ = Profundidad del árbol

0.4: Corrección de Heapify

Queremos demostrar:

$\left. \begin{array}{l} (H, \text{izq}(i)) \text{ es heap} \\ (H, \text{dcha}(i)) \text{ es heap} \end{array} \right\} \Rightarrow \begin{array}{l} \text{Después de Heapify}(H, i) \\ (H, i) \text{ es heap} \end{array} (*)$

Notación: $(H, \text{izq}(i)) \equiv$ el subárbol desde el nodo $\text{izq}(i)$ hacia abajo.

Demostración: Por inducción sobre la altura $h(H, i)$

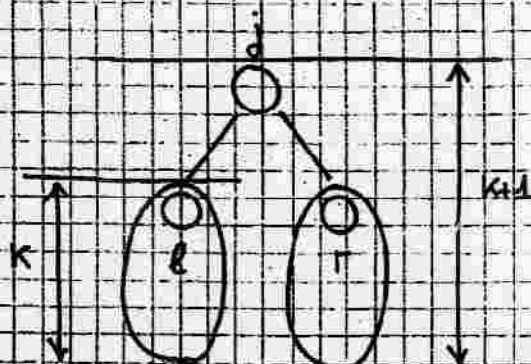
a) Para $h(H, i) = 1 \equiv$ árbol de altura 1 \Rightarrow (H, i) tiene un solo nodo \Rightarrow es un heap.

b) Supongamos que $(*)$ se cumple para todo i con $h(H, i) = k$ $(**)$

\hookrightarrow Hipótesis.

Vamos a demostrar que $h(H, j) = k+1 \Rightarrow$ Se cumple $(*)$

- Sea j tal que $h(H, j) = k+1$ (j es un nodo), y que $(H, \text{reg}(j))$ es heap y $(H, \text{dere}(j))$ es heap



1. Si $H[j] \geq H[l]$, $H[j] \geq H[r]$, entonces (H, j) es heap

2. Si $H[l] \geq H[r]$, $H[l] \geq H[j]$, entonces

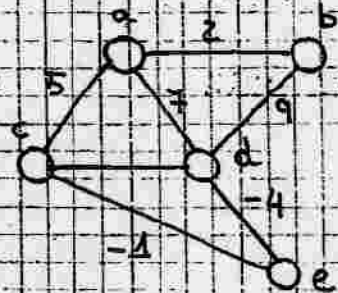
(- Después de swap $(H[j], H[l])$ y $H[j] \geq H[r]$
 $(*)$ (- (H, r) es heap \rightarrow no se toca
 - Después de Heapify (H, l) por hipótesis $(**)$
 $h(H, l) = k \Rightarrow (H, l)$ es heap.

$(*) \Rightarrow (H, j)$ es heap.

1

GRAFOS

Gráfico = Conjunto de vértices interconectados con aristas



$$A = \{(a,b), (a,c), (a,d), (b,d), (c,d), (c,e), (d,e)\}$$

$A = \text{Relación binaria en } V$
(conjunto de pares $\neq \emptyset$ de vértices entre sí)

$$V = \{a, b, c, d, e\} \rightarrow \text{Vértices}$$

\rightarrow Aristas

$$G = (V, A)$$

Definición: Un gráfico G es un par formado por un conjunto V de vértices y una relación binaria A en V .

4.1 Tipos de grafos

1. Dirigidos vs no dirigidos

D, ND

2. Con pesos vs sin pesos

CP, SP

$$w: A \rightarrow \mathbb{R}$$

3. Conexos vs no conexos

C, NC

4. Con vs sin ciclos

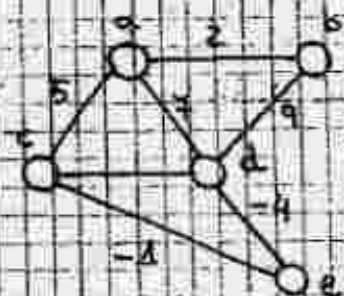
CC, SC

Conexo: Cualquier par de vértices (u, v) $u, v \in V$ que son mutuamente accesibles

Componentes conexos de G : Son los subconjuntos máximos de V formados por vértices mutuamente accesibles

1 GRAFOS

Gráfico = conjunto de uniones interconectadas con aristas



$$A = \{(a,b), (a,c), (a,d), (b,d), (c,d), (c,e), (d,e)\}$$

$$V = \{a, b, c, d, e\} \rightarrow \text{Vertices}$$

$$G = \{V, A\}$$

$A = \text{Rebaca } \text{Aristas}$
(conjunto de pares de V unidos entre sí)

\rightarrow Aristas

Definición: Un gráfico G es un par formado por un conjunto V de vértices y una relación bivariante A en V .

1.1 Tipos de grafos

1. Dirigidos vs no dirigidos

D, ND

2. con pesos vs sin pesos

CP, SP

$$w: A \rightarrow \mathbb{R}$$

3. Conexos vs no conexos

C, NC

4. Con vs sin ciclos

CC, SC

Conexo: Cualquier par de nodos mutuamente accesibles

2. $a \in V$ que son

Componentes conexos de G

Son los subgrafos

máximos de V formados por nodos mutuamente accesibles

1.2 Representación de grafos

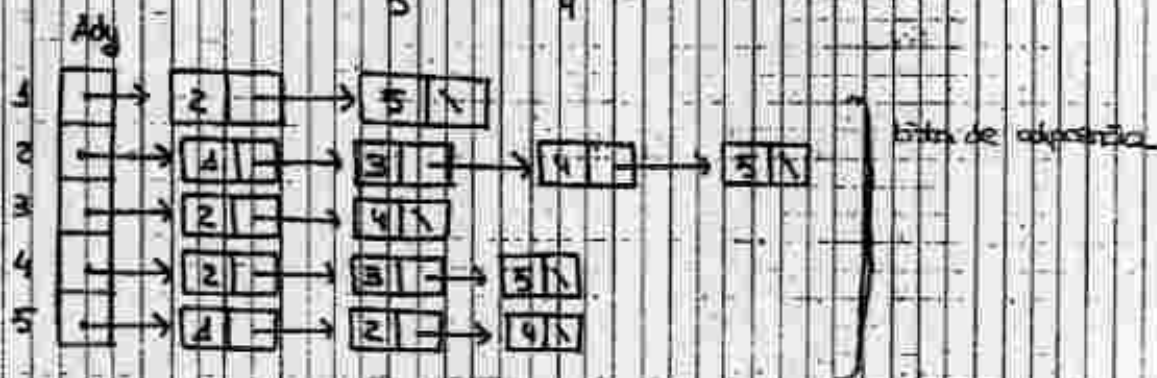
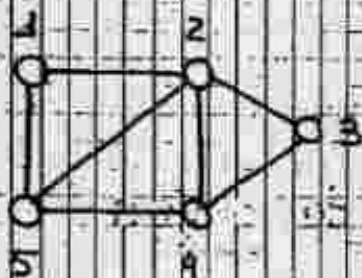
Forma 1

LISTAS DE ADYACENTES

```
struct nodo {
    int valor; // VALOR NODO
    nodo ** adyacentes; // PUNTERO ADYACENTES
    int nodosadyacentes; // N° ADYACENTES
}
```

nodo * grafo []; // grafo → array de nodo *

Ejemplo:



$Adj[N] \equiv$ lista de nodos

GASTO DE MEMORIA

1. Como mejor

0 aristas $\rightarrow O(|V|)$

espacio necesario de vértice
aristas

2. $O(|V| + |A|)$

\rightarrow transición + existencia del grafo

3. El máximo es

$O(|V| \times (|V| - 1))$

\rightarrow máximo de aristas por cada
nodo

Matriz de Adyacencia

struct nodo {

int valor // VALOR NODO

int adyacentes[N]; // 0 ó 1 // LISTA DE NODOS

}

1 → ADYACENTE
0 → NO ADY.

nodo x-grafo []; // grafo → array de nodos

$Adj[i, j] = \begin{cases} 1 & \text{si } (i, j) \in A \\ 0 & \text{si no es en caso contrario.} \end{cases} \Rightarrow \text{ADYACENTES } i, j$

Matriz $V \times V$

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 | 1 |
| 2 | 1 | 0 | 1 | 1 | 1 |
| 3 | 0 | 1 | 0 | 1 | 0 |
| 4 | 0 | 1 | 1 | 0 | 1 |
| 5 | 1 | 1 | 0 | 1 | 0 |

Desventajas : Perdida de espacio

Ventaja : acceso muy rápido **O(1)**

Si quiero saber todos los adyacentes de un nodo \Rightarrow **O(V)**
 \Rightarrow es más eficiente la otra forma

Por un grafo con pocas aristas, la otra forma es mejor.

4.3.2.

LD



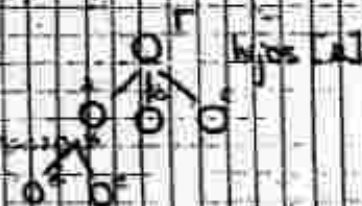
BFS (A)

 $\text{UserFor}(Q, r)$

White Q#0 Modific

$$\mu \leftarrow \text{extract}(\mathcal{Q})$$

```
for v in wjos[w] do
```

$$\text{user to } (a, w)$$


2)

DFS (A)

Insertor (P, max[A]) // Insert to max

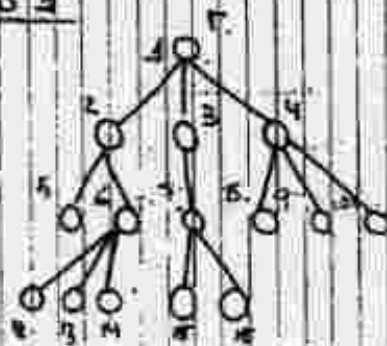
while $P \neq 0$ do $\forall p \in V$

$$u \leftarrow \text{extract}(\mathcal{P})$$

For $v \in \text{Lip}(\mathbb{R})$ do

usuario (P, v)

Example 4



u P

0 Sacuados un elemento

1 {1}

4 {2, 3, 4}

1 {2, 3, 4, 10, 11}

10 {2, 3, 4, 11}

9 {2, 3, 4}

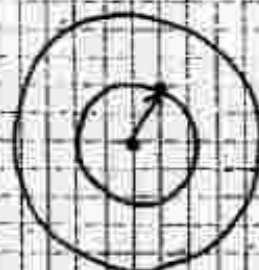
3 {2, 3, 4}

| | | | |
|----|----|----|----|
| 22 | P | | |
| 8 | 12 | 8 | |
| 10 | 12 | 15 | 16 |
| 15 | 12 | 15 | |
| 2 | 12 | | |
| 7 | 15 | | |

4.3.2 RECORRIDO DE GRAFOS

1) BÚSQUEDA EN ANCHURA BFS

Vamos a ir visitando las unidades adyacentes



Intentamos alejarlos lo más posible (Al contrario que en profundidad)

- A cada nodo asociamos un flag para saber si ha sido visitado o no (True / False)
- También podemos ir marcando arcos para luego obtener el recorrido mínimo

- atributos:**
- 1. Visitado o no.
 - 2. Antecesor para marcar el árbol
 - 3. Distancia entre arcos.

En anchura si el G no es conexo, los necesarios no serán visitados. En profundidad recorreremos todo el G.

El resultado en anchura del recorrido, será un árbol (en profundidad no se puede haber más de uno).

En anchura

- 1) Necesitamos saber por dónde habíamos pasado. Asociamos a cada nodo un flag: **N/V** (visitado/no visitado)
- 2) El marco y valor dejarlo, lo usamos Antecesor a cada nodo asignamos un antecesor al nodo anterior.
- 3) Distancia Nº de A que hemos ido recorriendo desde el nodo inicial.

- ① $\text{estado}[n]$, $N \text{ o } V$
 ② $\pi[n]$, antecesor
 ③ $d[n]$, n° de arcos recorridos desde s (modo inicial) hasta n .

SEUDOCÓDIGO

BFS (G, s)

Inicializar V { for $u \in V[G]$ do
 (i) $\text{estado}[u] \leftarrow N$ // $\text{estado}[u] \rightarrow \text{no visitado}$
 (ii) $\pi[u] \leftarrow \text{NULL}$ // $\text{ANTECESOR} \rightarrow \text{NULL}$
 (iii) $d[u] \leftarrow \infty$; los nodos q no son accesibles tienen distancia infinita.

Colocar en estado inicial Q { $\text{estado}[s] \leftarrow V$ // $s \text{ inicial} \rightarrow \text{visitado}$
 $d[s] \leftarrow 0$ // $n^\circ \text{ arcos} \rightarrow 0$
 $\text{insertar}(0, s)$ // insertamos en cola
 while $Q \neq \emptyset$ // Mientras la Cola Vacía

do $u \leftarrow \text{extraer}(Q)$ // sacamos u
 ; recorremos la adyacencia. Si el nodo ady no había sido visitado, entonces marcamos como V (y meterlo a la cola).

while $Q \neq \emptyset$ { for $v \in \text{Ady}[u]$ do // Para cada nodo adyacente a u
 if $\text{estado}[v] = N$ then // Si no visitado
 $\text{estado}[v] \leftarrow V$
 $\pi[v] \leftarrow u$ // PADRE $\rightarrow v$ (ANTECESOR)
 $d[v] \leftarrow d[u] + 1$ // AUMENTO DISTANCIA
 $\text{insertar}(Q, v)$ // INSERTO v en COLA

¿Que orden hay con respecto a nodos y arcos?

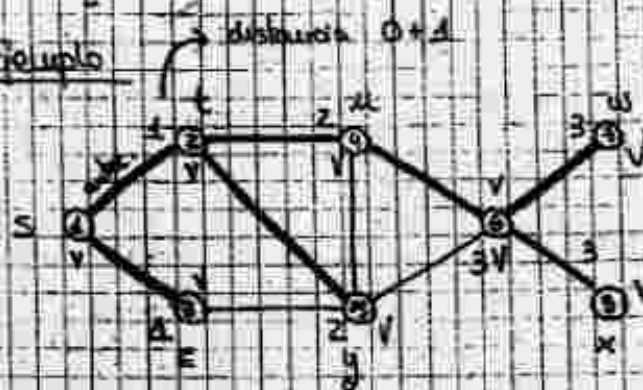
$O(N + A)$

VCA en el bucle superior

$O(N + A)$

la línea 1. dando igual si en vez de sacar el elemento de la cola, fue sólo a mirar.

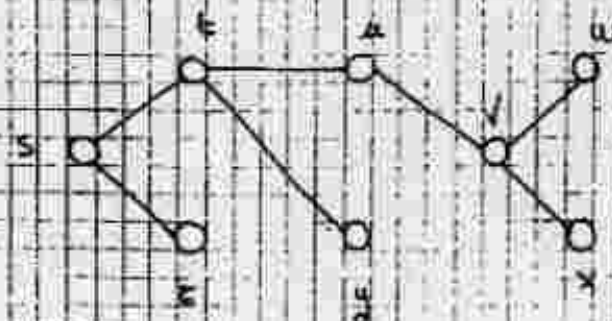
Ejemplo



$\{s\}$

| A | B |
|---|---------------|
| s | $\{t, u, y\}$ |
| t | $\{u, v, y\}$ |
| u | $\{y, v\}$ |
| y | $\{v\}$ |
| v | $\{w, x\}$ |
| w | $\{x\}$ |
| x | \emptyset |

= Solución



TEOREMA:

- a) BFS(G, s) recorre todos los nodos de G accesibles desde s y sólo estos.
- b) Después de BFS(G, s) se cumple $d[u] = d(s, u)$
 $\forall u \in V[G]$
- c) Después de BFS(G, s), para todo $u \neq s, u \in V[G]$ accesible desde s , si p es un camino de distancia mínima (CDM) de s a $\pi[u]$, entonces si seguimos el arco $(\pi[u], u)$ se obtiene un CDM de s a u .



Definición: La distancia mínima $d(s, u)$ entre dos nodos es el mínimo número de arcos de todas las caminoes de s a u en G .

Definición: Un CDM de s a u es un camino de s a u de longitud $d(s, u)$.

LEMAS PARA DEMOSTRACION DEL TEOREMA:

- 1) LEMA 1: Para todo $(u, v) \in A[G]$ se cumple $d(s, v) \leq d(s, u) + 1$.

Demo:



Supongamos que u es accesible desde s , entonces

Sea p un CDM de s a u , $\Rightarrow \text{long}(p) = d(s, u)$,
entonces la longitud p seguido de (u, v) , $= d(s, u) + 1$.

Camino de s a $v \geq d(s, v)$

2) **LEMA 2**: Durante $BFS(G, s)$, $\forall u \in V[G]$ se cumple
 $d[u] \geq d^*(s, u)$ (durante todo el proceso)

Demos: Por inducción sobre el n.º de nodos insertados en la cola.

i) Cuando se han insertado 0 nodos.

$$u \neq s \Rightarrow d[u] = \infty \geq d^*(s, u)$$

$$d[s] = 0 = d^*(s, s)$$

ii) Supongamos que se han introducido unos cuantos nodos y se cumple $d[u] \geq d^*(s, u)$.

Sea v el siguiente nodo insertado. Sea u_0 el último nodo extraído.

$$d[v] = d[u_0] + 1 \geq d^*(s, u_0) + 1 \geq d^*(s, v)$$

(hip. inducción) (lema 1)

porque $(u_0, v) \in A[G]$

3) **LEMA 3**: Sea $Q = \{u_1, u_2, u_3, \dots, u_n\}$ el estado de la cola en un momento cualquiera de $BFS(G, s)$. Se cumple siempre

① $d[u_i] \leq d[u_{i+1}] \quad \forall i = 1, \dots, n-1$

② $d[u_1] \leq d[u] + 1 \quad (\text{Escalón})$

Demos: Inducción sobre el n.º de operaciones de cola:

i) Cuando se han hecho 0 operaciones.

$Q = \emptyset \Rightarrow$ Se cumple todo.

ii) Supongamos que tras varias operaciones se cumple 1 y 2.

Tras la siguiente operación se cumple 1 y 2?

a) Extraer (Q) $\rightarrow u_1 \Rightarrow Q = \{u_2, \dots, u_n\}$

1. Se sigue cumpliendo desde $i=2, \dots, n-1$

2. Tenemos q demos $d[u_2] + 1 \geq d[u_n]$

$$d[u_2] + 1 \geq d[u_1] + 1 \geq d[u_n]$$

(hip 1) (hip 2)

$\Rightarrow d[u_2] + 1 \geq d[u_n]$, luego se cumplen 1 y 2

b) Sea u_{n+1} el siguiente nodo alcanzado.

$$\Rightarrow Q = \{u_1, u_2, \dots, u_n, u_{n+1}\}$$

$$1. d[u_{n+1}] = d[u_n] + 1 \geq d[u_n] \quad (\text{wp2})$$

$$\Rightarrow d[u_{n+1}] \geq d[u_n]$$

$$2. d[u_{n+1}] \leq d[u_n] + 1 \quad \text{porque de hecho se cumple}$$

$$d[u_{n+1}] = d[u_n] + 1$$

DEMOSTRACIÓN DEL TEOREMA

1) • Sea u no accesible desde s , entonces

$$d(s, u) = \infty \leq d[u] \Rightarrow d[u] = \infty = d(s, u) \quad (b)$$

(wp2)

Para demostrar (a), supongamos que u es visitado, entonces $\Pi(u)$ y $d[u] = d[\Pi(u)] + 1 = \infty$

Retrocediendo, todos los antecesoros tienen $d[\] = \infty$

En un n.º finito de pasos, se llega a $d[s] = \infty$!! $\Rightarrow (a)$

2) • Sea v accesible desde s . Por inducción sobre $n = d(s, v) < \infty$

$$i) d(s, v) = 0 \Rightarrow v = s \text{ es visitado (a), y } d[s] = 0 = d(s, s) \quad (b)$$

ii) Supongamos que a, b, c se cumplen para $d(s, v) \leq n$

Demstrar que a, b, c se cumplen para $d(s, v) = n+1$

Sea v con $d(s, v) = n+1$

Vamos a demostrar que v es visitado desde un nodo con $\delta^f(s, u) = n$

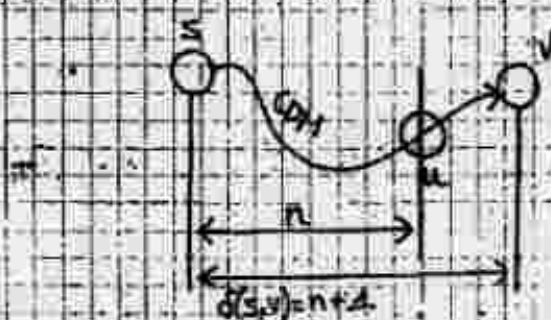
- Todos los u con $\delta^f(s, u) = n$ entran en Q antes que v .

$$\text{Porque } d[u] = \delta^f(s, u) = n < n+1 = \delta^f(s, v) \leq d[v] \quad \begin{matrix} \text{(lip b)} & \text{(lema 2)} \end{matrix}$$

Por el lema 3 u entra antes que v .

- Entre todos los u con $\delta^f(s, u) = n$, al menos uno tiene a v por adyacente.

Por lo menos el antecesor u inmediato a v en un CAM de s a v , tiene $\delta^f(s, u) = n$.



- Sea u_0 el primer nodo con $\delta^f(s, u_0) = n$ y $(u_0, v) \in A[E]$ que sale de la cola.

Cuando u_0 sale de la cola, estado $[v] = N$

Si hubiese sido visitado antes, sería desde u' con $\delta^f(s, u') < n$. Pero entonces

$$d[v] = d[u'] + 1 = \delta^f(s, u') + 1 < n + 1$$

lip

$$n+1 = \delta^f(s, v) \leq d[v] \quad \begin{matrix} \text{(lema 2)} \\ \text{Contradicción} \end{matrix}$$

$\Rightarrow v$ es visitado (a) desde u_0 y además

$$\underline{d[v]} = d[u_0] + 1 = \delta^f(s, u_0) + 1 = n+1 = \underline{\delta^f(s, v)} \quad \text{(b)} \quad \text{(lip b)}$$

Para demostrar el apartado (c) $\pi[v] \leftarrow u$

Si p es CDM cualquiera de s a u (es decir a $\pi[v]$) entonces su distancia $\text{long}(p) = d(s, u) = n$
 p seguido de $(\pi[v], v)$ tiene longitud
 $\text{long}(p) + 1 = n + 1 = d(s, v)$

$\Rightarrow p$ seguido de $(\pi[v], v)$ es un CDM de s a v (c)

Definición: Dado un grafo $G = (V, A)$ y un nodo $s \in V$, un subgrafo $G' = (V', A')$ de G es un árbol BF para (G, s) , $\Leftrightarrow V'$ está formado por todos los nodos de V accesibles desde s y existe un único camino de s a u en G' para todo $u \in V'$, que además es un CDM de s a u .

TEOREMA: Dado un grafo $G = (V, A)$, sea $G_\pi = (V_\pi, A_\pi)$ el subgrafo definido por:

$$V_\pi = \{ u \in V \mid \pi[u] \neq \text{NIL} \}$$

$$A_\pi = \{ (\pi[u], u) \in A \mid u \in V_\pi - \{s\} \} \text{, lo definimos como un dirigido}$$

G_π es un árbol BF para G

$G_\pi = (V_\pi, A_\pi)$ es un árbol BF para $G = (V, A)$ con s como nodo inicial.

DEMOSTRACIÓN:

Si $u \in V_\pi \Rightarrow \pi[u] \neq \text{NIL} \Rightarrow \text{BFS}(G, s)$ visita u
 $\Rightarrow u$ es accesible desde s (*)

$\Rightarrow V_\pi$ formado por los nodos de V accesibles desde s en G (**)

G_π es no dirigido xq (**)
 G_π es conexo xq (*) y (**)
 $|A_\pi| = |V_\pi| - 1$

$\Rightarrow G_\pi$ es un árbol
 \Downarrow
 Hay un único camino desde s hasta cada nodo de V_π

Por inducción, vamos a demostrar que el camino de s a v en G_π es un CDH $\forall v \in V_\pi$ (***)

Por inducción en $n = d[u]$

i) Para $n=0 = d[u] = d(s, u)$ (Th.b)

$\Rightarrow u=s$

ii) Supongamos que (***) se cumple $\forall u$ con $d[u]=n$

Sea v con $d[v] = n+1$ (antecesor $+1$)

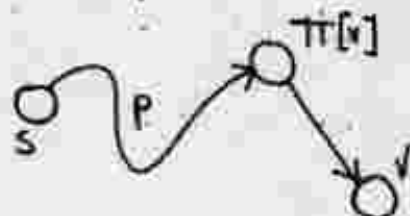
$d[\pi[v]] = n \Rightarrow$ (antecesor)

\Rightarrow Existe un camino (p) de s a $\pi[v]$ en G_π que es un CDH

\downarrow
 Por apartado (b) quedaría demostrado que es CDH también.

La longitud de p es n porque

$$d(s, \pi[v]) = d[\pi[v]] = n$$



p seguido de $(\pi[v], v)$

- ⊙ $\begin{cases} \bullet \text{ tiene longitud } n+1 \\ \bullet \text{ es un camino de } s \text{ a } v \text{ en } G_\pi \\ \bullet d(s, v) = d[v] = n+1 \end{cases}$
 (Th.b)

⊙ $\Rightarrow p$ seguido de $(\pi[v], v)$ es un CDH de s a v .

MÉTODO QUE IMPRIMA EL RECORRIDO CON BFS DESDE S HASTA UN NODO CUALQUIERA

Print_path (G, s, u)

if $u = s$ then print s (Condición de parada)
else if $\pi[u] = \text{Nil}$ then print u "no es accesible"
desde "s"

// Caso de inicio en misma

// (Pero en BFS no se ve a dar)



else print_path (G, s, $\pi[u]$)
print u

② BÚSQUEDA EN PROFUNDIDAD DFS

Recorrido en profundidad

- No damos nada inicial
- Si ^{no} es árbol, se recorren recorriendo todos los nodos (no como en anchura)

Ejemplo:



B en anchura da lugar al árbol



B en profundidad



Para el algoritmo también tendríamos que ir marcando el camino. En árboles utilizábamos pilas, pero podríamos usar recursión.

A diferencia tb. de árboles, aquí no todas las aristas serán recorridas.

En **DFS** la distancia no es importante, pero el tiempo hasta llegar a un nodo sí que lo será.

Por otra parte, necesitaremos 3 estados

- 1. NO VISITADO
- 2. VISITADO
- 3. (Adyacencia agotada) (Procesado)

PSEUDOCÓDIGO

DFS (G)

- ① for $u \in V[G]$ do
 $estado[u] \leftarrow N$
 $TT[u] \leftarrow nil$

 $t \leftarrow 0$

② for $u \in V[G]$ do // hace q visita si queda algun nodo sin visitar
 if $estado[u] = N$ then visitar(u)

Visitar(u)

$estado[u] \leftarrow V$ // a Visitado
 $d[u] \leftarrow ++t$ // sumando a otras
for $v \in Adj[u]$ do // no tiene adyacencia
 if $estado[v] = N$ then // si no visitado
 $TT[v] \leftarrow u$ // Antecesor de v es u
 visitar(v) // visita v

 $estado[u] \leftarrow P$ // a Procesado
 $p[u] \leftarrow ++t$ // momento a salir // tiempo salido

COMPLEJIDAD DEL ALGORITMO

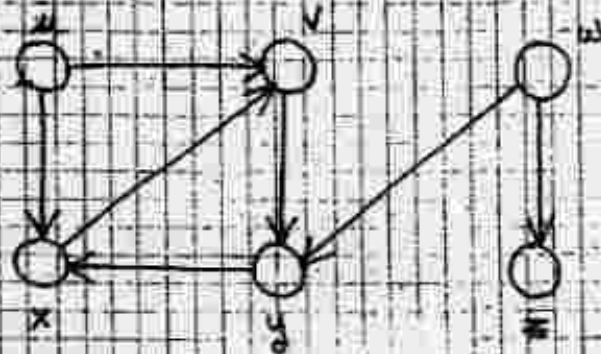
- ① Se recorre una vez por cada vértice $O(|V|)$
- ② Se recorre una vez por cada vértice $O(|V|)$

Visitar, para cada nodo, recorre su adyacencia (sus arcos). Si un nodo ha sido visitado, no volvemos a entrar en Visitar con él. Por tanto, en Visitar entraremos una vez por cada nodo como mucho. Y tampoco entraremos menos de una vez en Visitar por cada nodo.

- ③ Se recorre como el máximo entre $|A|$ y $|V|$ ($O(|V| + |A|)$)

↓
2º Bucla Complejidad \Rightarrow $O(V + A)$

Ejemplo:



DFS y BFS se pueden aplicar en cualquier tipo de grafo:
con/sin pesos, dirigidos/no dirigidos.

| t | u |
|----|----------|
| 0 | Entrar u |
| 1 | Entrar v |
| 2 | Entrar y |
| 3 | Entrar x |
| 4 | |
| 5 | salir x |
| 6 | salir y |
| 7 | salir v |
| 8 | salir u |
| 8 | Entrar w |
| 9 | Entrar z |
| 10 | |
| 11 | salir z |
| 12 | salir w |

(termina el for)

TEOREMA III DEL PARENTESIS

Después de DES(6), dados $u, v \in V[G]$ una y sólo una de las siguientes afirmaciones se cumple:

Abreviatura:

$$I_u = [d[u], p[u]] = \text{Intervalo}$$
$$I_v = [d[v], p[v]]$$

Afirmaciones:

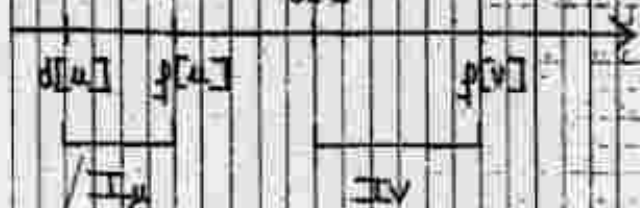
- ① $I_u \subset I_v$ y u descubre de v en el bosque DF
- ② $I_v \subset I_u$ y v descubre de u en el bosque DF
- ③ $I_u \cap I_v = \emptyset$

\Rightarrow No puede haber intervalos solapados

DEMOSTRACIÓN:

- ① Supongamos $d[u] < d[v]$

1) Si $p[u] < d[v]$, entonces, caso 3. $I_u \cap I_v = \emptyset$



2) Si $d[v] < p[u]$, entonces



Cuando se entra en $Visitar(v)$ ya se ha entrado en $Visitar(u)$ pero no se ha salido de $Visitar(u)$

\Rightarrow $Visitar(v)$ tiene que salir antes que $Visitar(u)$

$\Rightarrow p[v] < p[u]$

$\Rightarrow I_v \subset I_u$ Caso 2

v se alcanza desde u a través de una sucesión de nodos intermedios $u = u_0, u_1, u_2, \dots, u_n = v$ con $\pi[u_i] = u_{i-1} \Rightarrow v$ - descendiente de u .



© Para $d[v] < d[u]$ similar.

PROBLEMA Después de BFS(G), dadas $u, v \in V[G]$, se cumple que v descendiente de u en el Bosque DF $\iff I_v \subset I_u$.

DEMOSTRACIÓN: Bosque DF en $G_n = (V, A_n)$ donde $A_n = \{ (\pi[u], u) \mid u \in V \text{ y } \pi[u] \neq \text{NIL} \}$

(*) Se deduce del Teorema (Caso 2), de forma inmediata.

(**) $\exists u_i$ con $u = u_0, u_1, \dots, u_n = v$ y $\pi[u_i] = u_{i-1}$

Por inducción en i

1. $i=1$ tenemos $\pi[u_1] = u$, u_1 se visita desde $u \Rightarrow d[u] < d[u_1] < p[u_1] < p[u] \Rightarrow I_{u_1} \subset I_u$

(Visitar(u_1)) se llama directamente desde Visitar(u)

2. Supongamos $I_{u_i} \subset I_u$

u_{i+1} se visita desde u_i

Visitar(u_{i+1}) se llama directamente desde Visitar(u_i)

$\Rightarrow d[u_i] < d[u_{i+1}] < p[u_{i+1}] < p[u_i]$

$\Rightarrow I_{u_{i+1}} \subset I_{u_i} \subset I_u$
(hip)

En particular, $I_{u_n} = I_v \subset I_u$

Si en el instante $d[u]$...

- estado $[v] = V \Rightarrow u$ descubre de v en el bosque DF porque
 $d[v] < d[u] < p[v] \Rightarrow I_u \subset I_v$
th parentesis
- estado $[v] = N \Rightarrow u$ no descubre de v
- v adyacente a $u \Rightarrow v$ descubre de u en el bosque DF
- v accesible desde $u \Rightarrow$ no necesariamente descubre de u , sólo si los intermedios son estado $[] = N$



LEMA DEL PARENTESIS : (DADOS $u, v \in V[G]$)

v descubre de $u \Leftrightarrow$ En el instante $d[u]$, v es accesible desde u por un camino formado por nodos de estado N .

DEMOSTRACIÓN

\Rightarrow Sean $u = u_0, u_1, \dots, u_{n-1}, u_n = v$ con $\Pi[u_i] = u_{i-1}$ como u_i descubre de u ,

$$d[u] < d[u_i] < p[u_i] < p[u] \quad i \neq 0$$

\Downarrow

En el instante $d[u]$, u_i estaba en su descubridor (estado N) \Rightarrow camino de nodos en estado N

\Leftarrow Sean u_1, u_2, \dots, u_n los nodos del camino en estado N

\xrightarrow{t}
 $d[u]$



$I_{u_2} \subset I_u$, descubre de u

estado $[u] = N$ cuando $d[u] = 0$, $(u, u_1) \in A[G]$

$\Rightarrow u_1$ desciende de $u \Rightarrow I_{u_1} \subset I_u$

(4. porulesis)

estado $[u_2] = N$ cuando $d[u] > d[u_2] > d[u]$

Repetimos el razonamiento con u_2, u_3, u_4, \dots sucesivamente
hasta $I_{u_n} \subset I_u = v$ desciende de u

$\frac{1}{I_{u_1}}$

(ii) $(u_1, u_2) \in A[G]$ y estado $[u_2] = N$ cuando $d[u_1]$

u_2 desciende de $u_1 \Rightarrow I_{u_2} \subset I_{u_1}$!!

PRIMERA APLICACION DE BFS EN PROFUNDIDAD: CLASIFICACION DE ARCOS

Clasificación de arcos

Ejemplo:



Tipos de arcos:

Del origen:

$(u, v), (u, x), (v, y), (y, e)$

Hacia destino:

(u, y)

Hacia otros:

(e, v)

De cruce:

(x, y)

DEFINICIONES:

$(u, v) \in A[G]$ es un arco...

- ① del árbol si $\text{LCA}(u, v) = u$
- ② hacia atrás si u desciende de v en el bosque DF.
- ③ hacia adelante si v desciende de u en el bosque DF.
- ④ de cruce en otro caso.

Dada $(u, v) \in A[G]$ si en el momento de crear el arco (u, v)

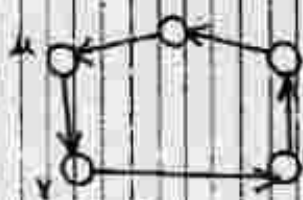
- Estado $|V| = N$, entonces (u, v) es del árbol
- Estado $|V| = V$, entonces (u, v) es hacia atrás (u desciende de v)
- Estado $|V| = P$, entonces si $d(v) > d(u)$ (u, v) es hacia adelante si $d(v) < d(u)$, (u, v) es de cruce

En caso de ambigüedad (p.e. cuando el grafo no es dirigido o arcos (u, u)) se toma el primer caso que se cumple.

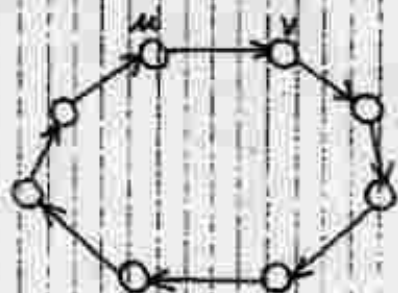
TEOREMA Un grafo es acíclico \Leftrightarrow no tiene ningún arco hacia atrás

DEMOSTRACIÓN:

\Rightarrow) Sea (u, v) arco hacia atrás, u desciende de v en el bosque DF. La raíz del árbol que va de v a u , seguido de (u, v) es un ciclo.



\Leftarrow) Sea C un ciclo en G



Sea v el primer nodo de C que se visita.

Sea u el nodo que preceda a v en el ciclo C .

En el instante $d[v]$, existe un camino desde v hasta u en G (el resto del ciclo C) formado por nodos en estado N .

$\Rightarrow u$ descende de v en el bosque $DF \Rightarrow (u, v)$ hacia atrás (H. parent)

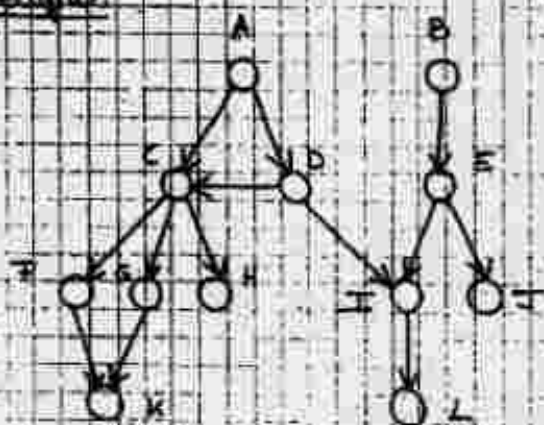
SEGUNDA APLICACIÓN DE DFS: ORDENACIÓN TOPOLOGICA

Un **DAG**, es un grafo acíclico dirigido.

DEFINICIÓN: Una ordenación topológica de un DAG $G=(V,A)$ es una relación de orden $<$ sobre V que cumple:

$$(u,v) \in A \Rightarrow u < v$$

Ejemplo:



No es OT, ya $(D,C) \in A$ pero $C < D$



\equiv OT.

Tiene que ir desde el de salida al de entrada.

Método

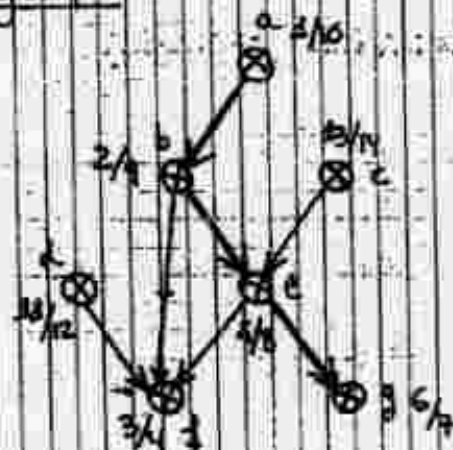
Orden Topológico (G)

variante de DFS(G). Al asignar

$p[u]$ a cada nodo v , introducir u al principio de una lista enlazada. Al final, devolver la lista de nodos como ordenación.

Así se ordenan los nodos por orden decreciente de $p[u]$.

Ejemplo:



10 12 11 9 8 7 6 5 4
c d a b e g f
→ Orden p decreciente



luego se la invierte

TEOREMA

Orden Topológico (G)

produce una D.T. de G

si G es un D.A.G

demostración: Hay que demostrar que todo $(u,v) \in A[G]$

$p[u] > p[v]$

① Imposible: (Haciendo $I_u \subset I_v \Rightarrow u$ descend de $v \Rightarrow (u,v)$ hacia o $\Rightarrow G$ tiene un ciclo γ

② $p[u] > p[v]$



③ Imposible

si $d[v] > d[u] \Rightarrow$ estado $[v] \neq N$
cuando $d[u]$ y v adyacente a $u \Rightarrow$
 $\Rightarrow v$ descend de $u \Rightarrow I_v \subset I_u$!!

1.4 Caminos de coste mínimo

Definición 1: Grado con peso es un par (G, w) donde

$$w: A[G] \rightarrow \mathbb{R}$$

Definición 2: Coste de un camino $p = (u_0, u_1, \dots, u_n)$ es

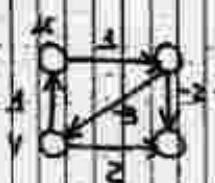
$$w(p) = \sum_{i=0}^{n-1} w(u_i, u_{i+1})$$

Definición 3: Coste mínimo entre dos nodos u y v es

$$d(u, v) = \begin{cases} \min \{ w(p) \mid u \xrightarrow{p} v \} & \text{si existe un camino de } u \text{ a } v \\ \infty & \text{en otro caso} \end{cases}$$

Definición 4: Un camino de coste mínimo (CCM) de u a v es un camino p de u a v con $w(p) = d(u, v)$

Ejemplo:



→ No habría CCM. No \exists la solución $\&$ pide

— Especial: Peso negativo.

PROBLEMA DE LOS CAMINOS DE COSTE MÍNIMO

- CONDICIÓN: grafos sin ciclos de coste negativo
- VARIANTES:

- ① Fuente única: encontrar un CCM desde un nodo inicial hasta cada uno de los demás nodos del grafo
- ② Destino único: o contrario.
- ③ Único par: dos nodos y devolver un único CCM
- ④ Todos los pares.

LEMA 1 Si $p = (u_1, u_2, \dots, u_i, \dots, u_{n-1}, u_n)$ es un CCN entre u_1 y u_n , entonces el subcamino $p' = (u_i, u_{i+1}, \dots, u_{n-1}, u_n)$ es un CCN entre u_i y u_n .

DEMOSTRACION:



Si p' no es un CCN de u_i a u_n entonces $\exists p''$ con $w(p'') < w(p')$.

Entonces substituyendo p' por p'' en p obtendríamos un camino de u_1 a u_n de coste menor que p . Pero p era un CCN \Rightarrow Imposible.

LEMA 2 Si $p = (s, \dots, u, v)$ es un CCN de s a v y u es antecesor de v , entonces $d(s, v) = d(s, u) + w(u, v)$.

DEMOSTRACION:



$w(p) = w(p') + w(u, v)$ (x def de coste de 1 camino: suma de pesos donde $p' = (s, \dots, u)$)

$$w(p) = w(p') + w(u, v) = \underbrace{d(s, u)}_{\substack{\uparrow \\ \text{(Lema 1: } p' \text{ CCN de } s \text{ a } u)}} + w(u, v)$$

$\rightarrow d(s, v)$

Demostrado

LEMA 3 Para todo arco (u,v) de un grafo G , y todo nodo s , se cumple $d(s,v) \leq d(s,u) + w(u,v)$

DEMOSTRACIÓN: Sea p un CMH de s a u ,



el coste de p seguido de (u,v) es $w(p) + w(u,v) = d(s,u) + w(u,v)$

p' es un camino de s a v . Su coste sea menor o igual a $d(s,v)$, es decir,

$$d(s,u) + w(u,v) \geq d(s,v)$$

4.4.1 OPERACIONES Y ALGORITMOS

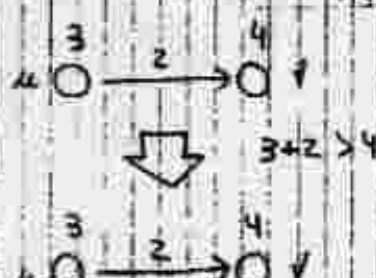
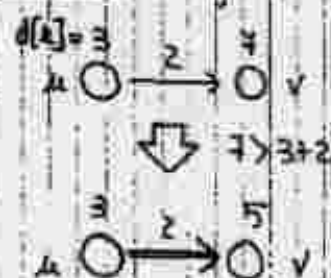
1) Inicializar (G, s)

for $u \in V[G]$ do // Para todo vertice
 $d[u] \leftarrow \infty$ // $d[u] = \text{infinito}$
 $\pi[u] \leftarrow \text{NIL}$ // Antecesor $\rightarrow \text{nulo}$
 $d[s] \leftarrow 0$ // distancia fuente=0

2) Relajar (u,v)

if $d[v] > d[u] + w(u,v)$ then // si $d[v] > d[u] + w(u,v)$
 $d[v] \leftarrow d[u] + w(u,v)$ // $d[v] = d[u] + w(u,v)$
 $\pi[v] \leftarrow u$ // Antecesor de $v \rightarrow u$

Ejemplo (Relajación):



1.4.2. ALGORITMO DE DIJKSTRA

Dijkstra (G, s)

Inicializar (G, s)

$\text{PQ} \leftarrow V[G]$ // Todos los nodos a la cola de prioridad.

- ① while $\text{PQ} \neq \emptyset$ // Mientras Cola Prior. no vacía
- ② $u \leftarrow \text{Extraer_mínimo}(Q)$ // $u = \text{Saco mínimo}$
- ③ for $v \in \text{Ady}[u]$ do // para todo adyacente
- ④ $\text{Relajar}(u, v)$ // $\text{Relajar}(u, v)$

② Se repite $|A|$ veces. Además hay que tener en cuenta que Relajar puede cambiar los valores, con lo que hay que reordenar la cola ④ $\log(v)$ cada llamada a Relajar . Como llamamos A veces: $A \log V$

③ Hay que contar el rendimiento del Heap.
 $\log |V|$

Como llamamos V veces (por el bucle de la cola)

$$V \cdot \log |V|$$

TOTAL $O((V+A) \log V)$

LEMMA 2: Después de llamar a Relajar (u, v) , para cualquier $u \in V[G]$ se cumple $d[u] \geq \delta(s, u)$ (*)

Además, si $d[u] = \delta(s, u)$ en algún momento, $d[u]$ ya no cambia más.

DEMOSTRACIÓN: Por inducción sobre el número de llamadas a relajar

$$i) \emptyset \text{ llamadas} \rightarrow d[u] = \infty \geq \delta(s, u) \quad u \neq s \\ d[s] = 0 = \delta(s, s)$$

ii) Supongamos que tras n llamadas se cumple (*) para todos los nodos. Veamos que después de la siguiente llamada Relajar (u, v) , se sigue cumpliendo $d[x]$ no cambia $\forall x \neq v$.

Si $d[v]$ no cambia, $d[v] \geq \delta(s, v)$

$$\text{Si } d[v] \text{ cambia, } d[v] = d[u] + w(u, v) \\ \geq \underset{\substack{\text{(hip. induc)}}}{\delta(s, u)} + w(u, v) \geq \underset{\text{(lema 3)}}{\delta(s, v)}$$

LEMMA 3: Si p es un CM de s a v , y (u, v) es el último arco de p , entonces después de Inicializar (G, s) y un indeterminado de llamadas a Relajar que incluya Relajar (u, v) si antes de esta llamada se tiene $d[u] = \delta(s, u)$, después de la llamada se tiene $d[v] = \delta(s, v)$.

DEMOSTRACIÓN: Después de Relajar $(u, v) \Rightarrow d[v] \leq d[u] + w(u, v)$
↑
(se ve por de Relajar)

$$\begin{array}{ccc} & \delta(s, u) + w(u, v) = \delta(s, v) & \\ \uparrow & \uparrow & \\ \text{lemas 1} & \text{lema 2} & \\ d[u] = \delta(s, u) & & \end{array}$$

$$\delta(s, v) \leq d[v] \leq \delta(s, v) \\ \uparrow \\ \text{lema 4}$$

fin

a) v no accesible desde $s \Rightarrow \delta(s, u) = \infty \not\leq d[v] \Rightarrow$
(lema 4)



Sea p CCM de s a v

Sea y el primer nodo de p que todavía está en \bar{Q} .

Sea x el antecesor inmediato a y en p .

- Veremos que $d[y] = \delta(s, y)$.

Cuando salió x se hizo $\text{Relajar}(x, y)$.

El camino de s a y es un CCM de s a y (lema 4), y $d[x] = \delta(s, x)$ antes de $\text{Relajar}(x, y)$ (hip. inducción).

\Rightarrow Después de $\text{Relajar}(x, y)$ $d[y] = \delta(s, y)$
(lema 5)

Además tenemos:

$$\Leftarrow \begin{cases} - d[y] \geq d[v] \text{ pq } v \text{ es el siguiente en salir de } PQ \\ \parallel \\ - \delta(s, y) \leq \delta(s, v) \leq d[v] \end{cases} \text{ lema 4}$$

\hookrightarrow ¡pq los arcos son positivos!

$\Rightarrow d[v] = \delta(s, v)$

TEOREMA: Después de Dijkstra (G, s) , donde los arcos de G no tienen pesos negativos, el subgrafo definido por $\Pi(u)$ para los $u \in V[G]$ es un árbol formado por CCM's.

1.4.3 ALGORITMO BELLMAN-FORD

(Permite pesos y ciclos negativos: No funciona, pero los detecta).

• Relaja arcos

- 1º Coge todos los arcos y los relaja una vez para cada uno
 - 2º Segunda pasada
 - 3º Tercera pasada
 - ⋮
 - 4º Tenemos un CCM.
- } Repite tantas veces como $|V|-1$

Hay mucho trabajo innecesario: lo lógico sería relajar cada pasada tan sólo aquellos nodos que han cambiado el valor de sus nodos.

PSEUDOCÓDIGO

Bellman-Ford (G, s)

 Inicializar (G, s)

 for $i=1$ to $|V[G]|-1$ $O(V \cdot A)$
 for $(u, v) \in A[G]$ do
 Relajar (u, v)

// ahora comprobamos si hay ciclos de coste negativo

for $(u, v) \in A[G]$

 (*) if $d[v] > d[u] + w(u, v)$ then return TRUE
return FALSE.

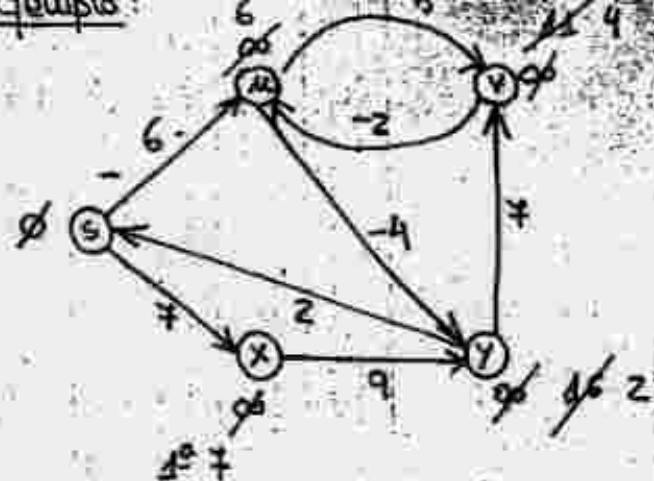
→ peso de la arista.

(*) Intuitivamente, si $d[v] > d[u]$ es que aun hay distancias mejorables → es decir, que se podría relajar algún arco más.

Esto es señal de que algo no ha funcionado.



Ejemplo:



Relajamos en cada paso en el orden que sea.

- Cada pasada los q tienen qdado serán los todos q han cambiado su valor

Esta versión 4:

- Trabajo mucho
- El orden de relajación se debería tener en cuenta.

VERSION OPTIMIZADA

Metemos en cola cuando en un nodo ha habido un cambio

(Bellman-Ford-Optimizado) (G, s)

Inicializar (G, s)

Insertar (Q, s) // metemos el nodo inicial

/* Iteramos hasta vaciar la cola y en cada una sacamos un nodo y relajamos todos los nodos que salen para cada arco */

While Q ≠ ∅ do

u ← extraer (Q)

for v ∈ Adj[u] do

if d[v] > d[u] + w(u, v) then

d[v] ← d[u] + w(u, v)

π[v] ← u

if v ∉ Q then

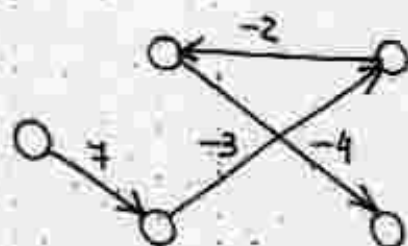
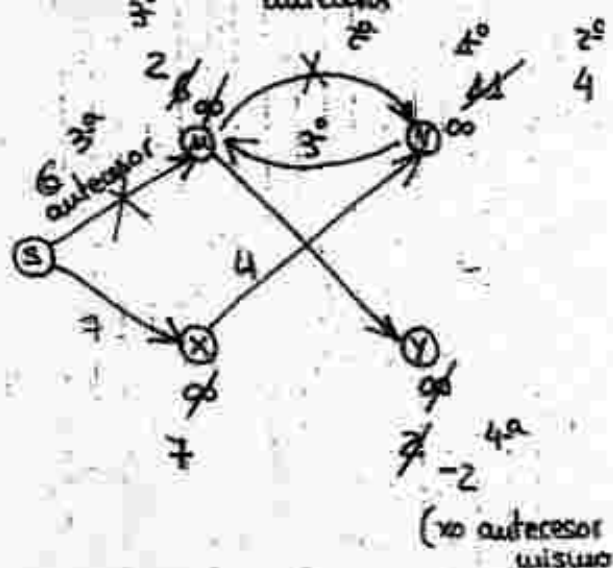
Insertar (Q, v)

Relajar

ejemplo

$Q \equiv G_0$; Quien sale

$\{(s, 0, NUL)\}$
 $\{(u, 6s)(x, 7s)\}$
 $\{(x, 7s)(v, 4u)(y, 2u)\}$
 $\{(v, 4x)(y, 2u)\}$
 $\{(y, 2u)(u, 2v)\}$ \rightarrow Acabo por
 $\{(u, 2, v)\}$
 $\{(y, 2, u)\}$
 $\{\} = \emptyset$



\equiv Arbol final generado.

Pero esta función en caso de ciclos negativos se funcionaría y además entraría en bucle infinito.

- Si un nodo entra más de $|V|-1$ veces en la cola \Rightarrow es que hay ciclos negativos.
Se podría solucionar haciendo una comprobación.

- En la versión mejorada la complejidad del algoritmo puede llegar a ser $O(V \cdot A)$ pero por regla general es menor.

TEOREMA Si G no tiene ciclos de coste negativo, entonces después de Bellman-Ford (G, s) , se tiene que \forall los nodos $v \in V[G]$ $d[v] = d^*(s, v)$ y el algoritmo devuelve FALSE.

Si G tiene algún ciclo de coste negativo, entonces el algoritmo devuelve TRUE.

DEMONSTRATION:

1. Sea G sin ciclos de coste negativo
- Sea $u \in V[G]$

a) Si u no es accesible desde s

$$d[u] \geq \delta(s, u) = \infty \geq d[u] \Rightarrow d[u] = \delta(s, u)$$

(Lemma 4)

b) Si u es accesible desde s , sea $p = (u_0, u_1, \dots, u_n)$ un CAM de s a u .

Vamos a demostrar que después de k iteraciones
(relajar k veces los arcos) se tiene

$$d[\mu_K] = \delta(s, \mu_K)$$

Por inducción en k .

- $i = \emptyset \rightarrow d[u_0] = d[s] = \emptyset = \delta(s, s)$
 - Supongamos que se cumple hasta i : es decir, después de i iteraciones

$$d[u_i] = \delta(s, u_i)$$
- $(u_0, \dots, u_i, u_{i+1})$ es CCN de s a u_{GH} (lema 1)
- En la iteración $i+1$ se hace Relajar(u_i, u_{GH})

(Lewin)

⇒ Después de la iteración $i+1$ se tiene
 $d[u_{i+1}] = \delta(s, u_{i+1})$



- Hemos demostrado hasta n que se cumple pero, ¿Se cumple para $n+1$?

→ Si, se couple.

Se cumple porque $n+1 \leq |V|$

$$n \leq |V| - 1$$

\Rightarrow Va a haber más que n iteraciones, luego el algoritmo iterará $|V|-1$ veces, es decir, más igual que n , luego en la iteración n se tiene

$$d[u_n] = \delta(s, u_n) \Rightarrow d[u] = \delta(s, u) \\ \forall u \in V[G]$$

Por otra parte, tenemos:

```
for  $(u, v) \in A[G]$ 
  if  $d[v] > d[u] + w(u, v)$  then return TRUE
return FALSE
```

- Si G no tiene ciclos de costo negativo ...
(1) Bellman-Ford(G, s) devuelve FALSE
- Si G tiene algún ciclo de costo negativo ...
(2) Bellman-Ford(G, s) devuelve TRUE

$$(1) (u, v) \in A[G] \quad \delta(s, v) \leq \delta(s, u) + w(u, v) \\ \parallel \text{ (lema 3)}$$

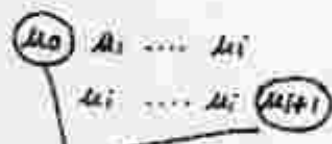
$$d[v] \leq d[u] + w(u, v) \quad \forall (u, v) \in A \\ \Rightarrow \text{devuelve FALSE.}$$

(2) Supongamos que no devuelve TRUE.

Sea $C = (u_0, u_1, \dots, u_{n-1}, u_n)$ con $u_0 = u_n$

$$w(C) = \sum_{i=0}^{n-1} w(u_i, u_{i+1}) \quad \text{ahora a } w(C) \text{ le sumo y resto lo mismo}$$

$$w(C) = \sum_{i=0}^{n-1} w(u_i, u_{i+1}) + d[u_i] - d[u_i] =$$



$$\begin{aligned}
&= \sum_{i=0}^{n-1} (w(u_i, u_{i+1}) + d[u_i]) - \sum_{i=0}^{n-1} (d[u_{i+1}]) \\
&= \sum_{i=0}^{n-1} (d[u_i] + w(u_i, u_{i+1}) - d[u_{i+1}]) \geq 0
\end{aligned}$$

$\nearrow 0$
 porque

$d[u_{i+1}]$ tiene q ser $< d[u_i] + w(u_i, u_{i+1})$ xq

"if $d[v] > d[u] + w(u, v)$ return true"

no se cumple para ningún arco.

(No devuelve true)

4.5. Arboles abarcadores mínimos

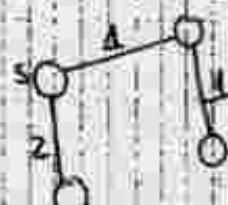
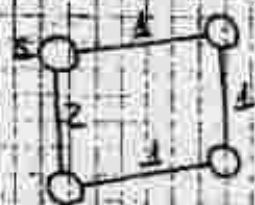
G CND

Definición: Dado un grafo G conexo y no dirigido, un árbol abarcador mínimo (AAM) es un subconjunto $T \subset A(G)$ acíclico que conecta todos los nodos de G y minimiza el coste total.

$$w(T) = \sum_{(u,v) \in T} w(u,v)$$

Es un algoritmo que tiene condiciones como OT o Dijkstra.

AAM $\left\{ \begin{array}{l} \text{Grafo conexo } C \\ \text{No dirigido } ND \end{array} \right.$ Condiciones



$CT = 4$



4.5.1 ALGORITMO DE PRIM

Necesitamos una clave que represente el peso más ligero que une un nodo de dentro de la cola con otro nodo de fuera de la cola.

- 1) Cogemos Δ nodo inicial \Rightarrow Parte del árbol q se va a construir
- 2) $Clave[u] = \infty$ \forall nodos inicialmente, exceptuando la raíz.

PSEUDOCÓDIGO

- Prim (G, r)

for $u \in V[G]$ do clave[u] $\leftarrow \infty$ // INI CLAVE
clave[r] $\leftarrow 0$ // CLAVE RAIZ

$\pi[r] \leftarrow NIL$ // no tiene antecesor por ser la raíz

// metemos todos los nodos en la cola

$Q \leftarrow V[G]$

while $Q \neq \emptyset$ do // MIENTRAS COLA NO VACIA

$u \leftarrow \text{extraer}(Q)$ ① // SACO

for $v \in \text{Adj}[u]$ do ② // PARA TODA ADY

if $v \in Q$ and clave[v] $> w(u, v)$ then

clave[v] $\leftarrow w(u, v)$ ③

$\pi[v] \leftarrow u$

COPIA
DE
PRIORIDAD
ORDENAR

Al final de este algoritmo tendremos un AMN

¿COSTE? En el bucle while

① Extraer de Q es proporcional a $O(\log V \cdot V)$

② Se repite 2A veces (xq es no dirigido) (tantas como arcos)

③ Si hay que reemplazar el heap y en el peor caso hay que subir hasta arriba del todo $O(\log V)$

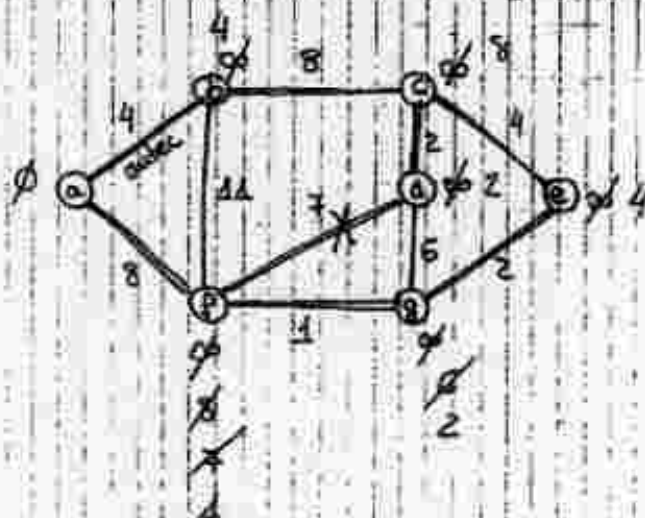
③③ $O(A \log V)$

En total
vamos a
decir complejo

$O((V+A) \log V)$

y en este caso es b
xq el grafo G es

Ejemplo:



$Q = \{(a, \emptyset, NIL), (b, \infty, NIL), (c, \infty, NIL), (d, \infty, NIL), (e, \infty, NIL), (f, \infty, NIL), (g, \infty, NIL)\}$

$Q = \{(b, q, a), (c, \infty, NIL), (d, \infty, NIL), (e, \infty, NIL), (f, 8, a), (g, \infty, NIL)\}$

Sale el q tenga la clave más baja.

$Q = \{(c, 8, b), (d, \infty, NIL), (e, \infty, NIL), (f, 8, a), (g, \infty, NIL)\}$

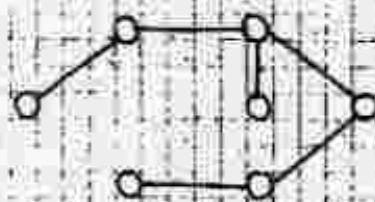
$Q = \{(d, 7, c), (e, 4, c), (f, 8, a), (g, \infty, NIL)\}$

$Q = \{(e, 4, d), (f, 7, d), (g, 6, d)\}$

$Q = \{(f, 7, d), (g, 2, e)\}$

$Q = \{(g, 1, g)\}$

$Q = \emptyset$



TEOREMA: Después de $\text{Prim}(G, r)$ donde G es conexo y no dirigido, el conjunto $\{(\pi[u], u) \mid u \in V[G] - \{r\}\}$ es un AAN.

DEMO: Sea $A = \{(\pi[u], u) \mid u \in V - PQ - \{r\}\}$ // pares en los que 2 vértices están fuera de Q .

Como el algoritmo termina cuando PQ está vacío, al final tenemos el mismo conjunto del enunciado del teorema.

En todo momento de la ejecución de $\text{Prim}(G, r)$, A es un subconjunto de algún AAN.

Si conseguimos demostrar esto, ya tenemos el resultado (pq A al final es lo mismo del enunciado).

Por inducción sobre el n° de nodos que salen de PQ

i) Cuando va a salir el 1° nodo, $A = \emptyset \subset$ cualquier AAM

(todos los nodos están en la cola $\Rightarrow V-Q = \emptyset$)

ii) Supongamos que $A \subset T$ con T AAM en un momento de PQ .

Sea V el siguiente nodo que va a salir de Q . Queremos demostrar que cuando V salga de Q , seguirá existiendo un AAM que incluya a A .

Sea $T' = T \cup \{(\pi[v], v)\} = \{(\pi[x], x)\}$

• $(\pi[v], v) \in T \Rightarrow A \cup \{x, y\} \subset T$

• $(\pi[v], v) \notin T$

$\exists p$ desde $\pi[v]$ hasta v en T (T es conexo)

$\exists (x, y) \in p$ con $x \notin Q$ y $y \in Q$

$A \cup \{(\pi[v], v)\} \subset T'$

$(x, y) \notin A$

\uparrow
 $y \in Q$

T' es AAM? Tenemos que:

• T conecta todos los nodos...

• $w(T') = w(T) + w(\pi[u], u) - w(x, y) \leq$

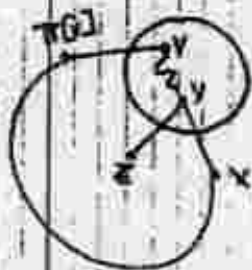
$< w(T) \Rightarrow T'$ es AAM

$w(\pi[u], u) = clave[u]$

$w(x, y) \geq clave[y] \geq clave[u]$

\uparrow
pq si $clave[y]$ fuera mayor, el algoritmo hubiese cambiado el padre.

\uparrow pq la salida antes de la cola.



2 ESTRUCTURAS DE FICHEROS

2.1 Conceptos Fundamentales

Los datos están en uno de estos en RAM, van a estar en disco consecutivamente:

- 1) El coste de acceso a los datos será superior.
- 2) El coste de las operaciones básicas también serán mayores en disco que en RAM. Son mucho más lentos.
Esto hará que algunos algoritmos de ordenación ya no sean muy buenos.

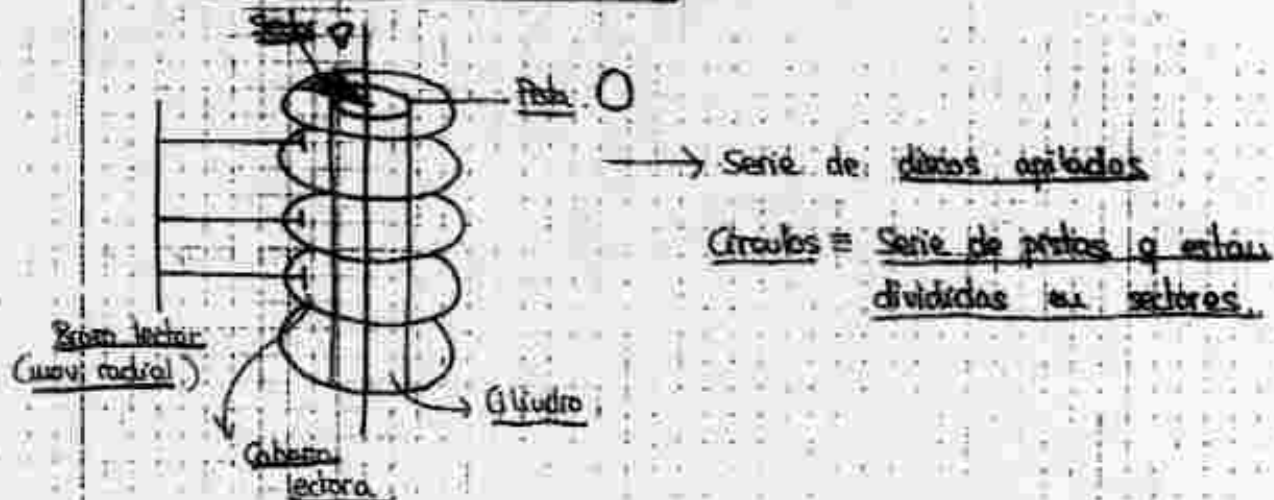
- 3) Coste de distintos puntos de un fichero es mucho más elevado que leer datos contiguos.

En operaciones en disco no vamos a tener structs y arrays, tenemos que controlarlo en bytes (con el tamaño de los tipos de datos primitivos, como int o long, x ejemplo)

Para almacenar todos estos datos se usan Bases de datos y sistemas de gestión (de B de Datos)

- Sistema de gestión de BD: software que nos permite hacer varias operaciones con las tablas de las BD.

2.1.1 ESTRUCTURA DEL DISCO



Cada sector es una unidad mínima de datos.
El archivo puede estar fragmentado en sectores
no-contiguos, pero la idea es que estén lo más
 juntos posibles.

MS2 - Plaster } Conjunto de sectores (u. fijo de sectores)
UNIX - Bloque }

Bloque = Es la unidad mínima de acceso a un
archivo. la unidad mínima por tanto no es un sector,
sino un conjunto.

Fragmentación = Espacio desaprovechado.
Los bloques y clusters pueden dar lugar a ella.

Para leer un dato se coloca la cabeza sobre la
pista, se espera a que rota sobre el sector y
se lee.

2.1.2. TIPOS DE ACCESO

- ① ACCESO (movimiento radial) Situar la cabeza sobre la
pista (cilindro) cuando está el dato a leer.
- ② ROTACION (movimiento giratorio) Giro del disco
Para situarse en el sector.
- ③ RECUPERADO DE LOS DATOS giro hasta leer / escribir
los datos.

San los ③ movimientos que se realizan se suma a
la hora de calcular el coste del tiempo de acceso
a archivos.

Características del disco

- ① depende de la rapidez del eje.
- ② y ③ dependen de la rapidez de giro del disco.

2.
Es preferible tener un archivo dividido en cilindros que dividido en un mismo disco (pq cambiar la cabeza lectora que lee en ese momento, lleva un tiempo despreciable)

También el costo de acceso dependerá de la suerte (del punto donde se encuentre la cabeza lectora en el momento de acceso, que es aleatorio). Por esto se suelen hacer promedios para calcular el costo

El seek es más costoso (pq lleva que cambiar paraudo) mientras que la rotación no gana.

Ejemplo:

- Seek promedio 9,5 ms
- Rotación 41 ms (5400 rpm)
- Sector 512 bits
- Pista 63 sectores
- Bloque 8 sectores

¿Cuanto tarda en leer un fichero de 2Mb con 8000 registros de 256 bits cada uno?

- 1ª forma: leer los registros de forma secuencial (todo seguido)
- 2ª forma: forma saltada

④ Lectura registro a registro → ALEATORIO

⑤ (Si quiero leerlo todo, es mejor de esta forma xq) evitamos así saltos.

Para cada registro:

- ① Seek: movimiento radial 9,5 ms
- ② Rotación: promedio $\frac{41}{2}$ ms
→ 20 ms por el promedio.

① Lectura: tenemos que leer el bloque entero contenido de un clúster.

bytes a leer
bytes de una pista

factores
a considerar al encontrar una pista

$$\frac{8 \cdot 512 \text{ b}}{63 \cdot 512 \text{ b}} = 14 \text{ us} = \frac{88}{63} = 1,39 \text{ us}$$

tiempo en leer un sector (unida de medida)

TOTAL: $\left(9,5 + \frac{1}{2} + 1,39 \right) \cdot 8000 = 434120 \text{ us} = 434,12 \text{ s}$
16,39 por registro
número de registros

② Lectura secuencial: (Fodo seguido)

Para cada clúster: $9,5 \text{ us} + 5,5 \text{ us} + \frac{1}{2} \text{ us} = 16,39 \text{ us}$

TOTAL: $16,39 \text{ us} \cdot \frac{8000 \cdot 256 \text{ b}}{8 \cdot 512 \text{ b}} = 819 \text{ s}$
nº veces total

La lectura secuencial sería preferible a la aleatoria en velocidad pero:

- Si hay que leer un archivo completo = secuencial
- Si hay que leer solo una parte = aleatoria

A veces para los cálculos nos dan la:

TASA DE TRANSFERENCIA = $\frac{\text{bytes a leer}}{\text{tiempo de transferencia}}$

Por tanto, el tiempo de transferencia de un bloque sería:

$\frac{\text{bytes a leer}}{\text{tasa de transferencia}}$

2.2 Registros

2.2.1 Campos y Registros

Delimitación de campos

- Por caracteres especiales entre campos
- Tamños fijos de campos
- Indicando el comienzo del campo al inicio de este.
- Usando pares: nombre campo | valor
(desperdiciamos espacio pero ganamos flexibilidad)

Delimitación de registros

- Longitud fija
- Longitud variable
 - Con delimitadores
 - Indicador de longitud
 - Nº fijo de campos

Las longitudes fijas son muy repetidas de procesos pero desperdician espacio

Las longitudes variables son tellos pero aprovechan bien el espacio

2.2.2 GESTIÓN DE FICHEROS DE REGISTROS

• Búsqueda

- Por posición, acceso directo
- Por clave (valor de un campo), acceso secuencial, otras mejores (p.e. índices)

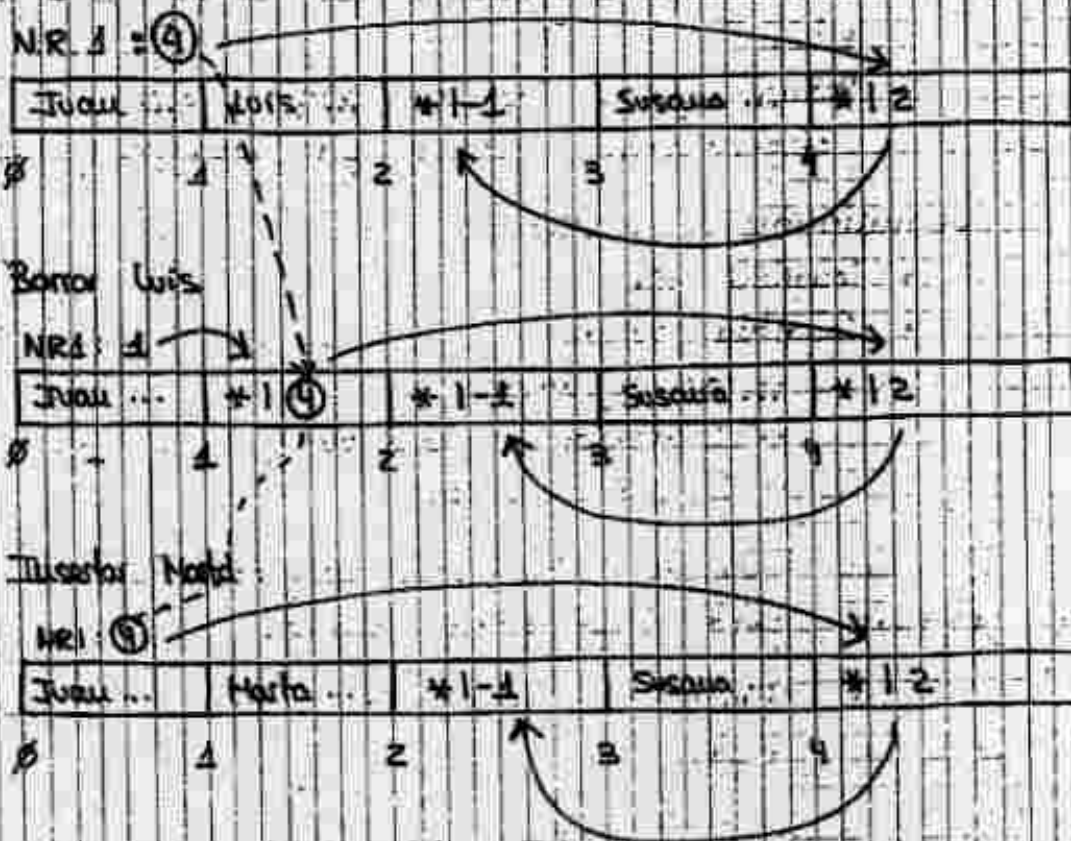
• ACTUACIÓN:

- Orden:
 - al final
 - en lugar
- Utilidad: usar como borrado y compactar periódicamente

2.3. Aprovechamiento del espacio de disco

- Aprovechar espacios de los borrados
- Mantener lista de borros
- Lista añadida a las propias borros

Ejemplo (suponiendo taquilla fija)



ACCIONES: con borros fija

• Borrado:

- Marcar como borrado
 - 2º campo del borros (el puntero) ← NR 1
 - NR 1 ← NR del registro borrado
- Borrar Luis

• Inserción:

- Si $NR1 = -1 \Rightarrow$ está sin huecos, escribir al final de Píndero.

- Si $NR1 \neq -1$

- $NR \leftarrow$ 2º campo del hueco en $NR1$
- Escribir registro en la posición $NR1$
- $NR1 \leftarrow NR$

• Campos con longitud variable:

los campos, en vez de ser índices, son offsets en bytes.

A la hora de insertar, si no cabe en el primer hueco, varios recomendados hacer subir el índice o lugar al final.

• MÉTODO DE INSERCIÓN (long variable)

- first + off: en el primer hueco que diste los datos en el que quier añadir (huecos espacio libre). Habría q. suplementar la lista de posic ordenada para q. sea eficiente.

- last + off: el que está sobre. Puede ser bueno para registros que varían mucho de tamaño, pq así el espacio sobrante posiblemente podamos usarlo para otro registro. La lista también debe estar ordenada.

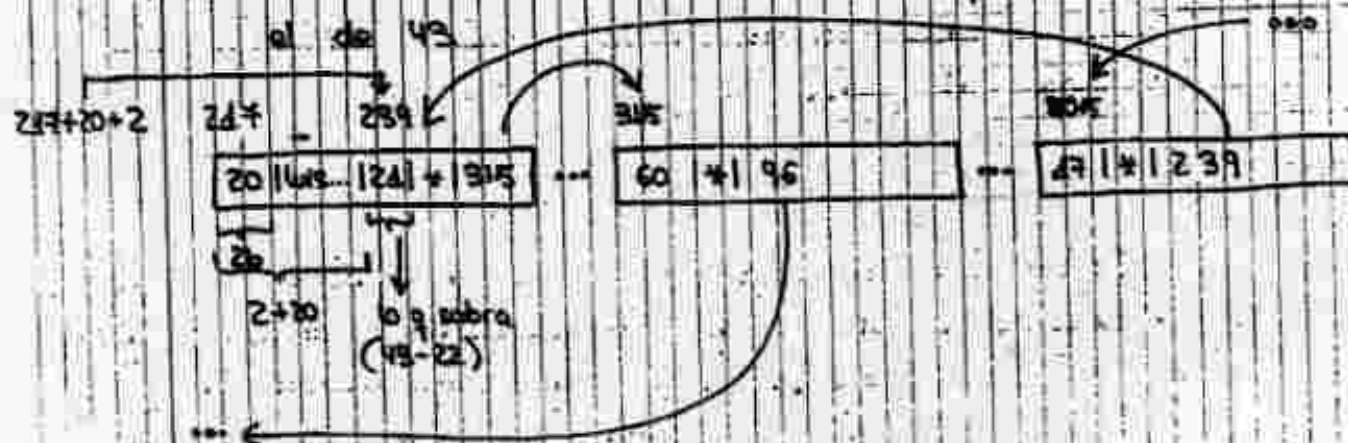
Ejemplo:



Si toda la lista esta ordenada en orden decreciente

⇒ Insertar

- Vamos a insertar un registro de 20 bytes: "luis..."
No cabe en el hueco de 47 bytes pero si en el de 43



$$43 - 20 - 2 - 2 = 21$$

Otras estrategias de ahorro espacio en ficheros son las de compresión de datos.

Hacer compresión significa que la misma información de un fichero se puede representar con o sin pérdida de la información.

Ej: Compresión de imágenes: pérdidas de resolución.

③ Algoritmo de Huffman

Es un método muy general y funciona para muchos tipos de archivos.

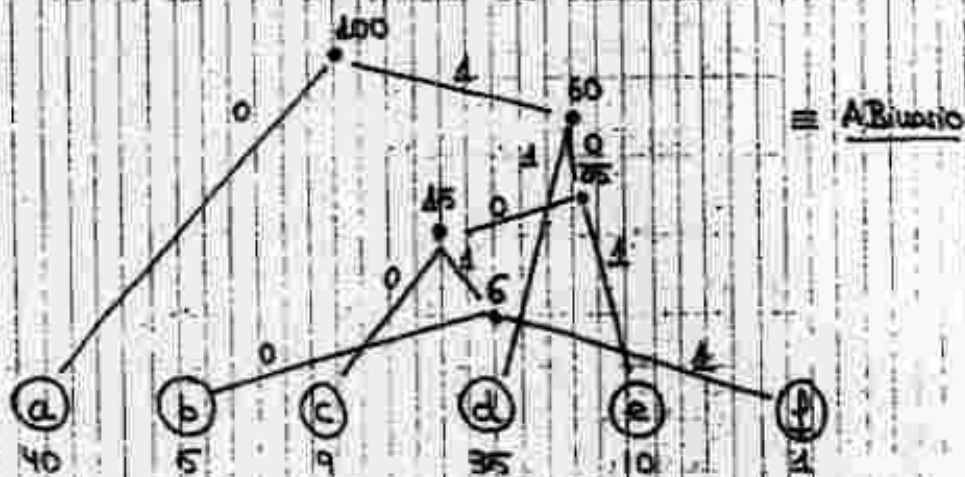
A cada valor de bytes que aparece en un fichero se les asigna una cadena de 0 y 1.

1. Valores frecuentes: cadenas más cortas.
2. Valores poco frecuentes: cadenas más largas.

Ejemplo:

Frecuencia: un fichero contiene los valores a, b, c, d, e, f

| | | | | | | |
|-------------|-----|----|----|-----|-----|----|
| Carácter: | a | b | c | d | e | f |
| Frecuencia: | 40% | 5% | 9% | 35% | 10% | 1% |



1. Nodo padre: 6
2. Itero
3. Pongo 0's y 1's: da igual el orden.

a → 0
 b → 10010
 c → 1000
 d → 11
 e → 101
 f → 10011

② NOTACIÓN COMPACTA

Ejemplo: provincias \rightarrow 15 bytes
52 provincias

¿Cuánto espacio necesitamos para guardar provincias?

• El nombre más largo: 45 caracteres \Rightarrow 15 bytes para ese grupo

• Hay 52 provincias \rightarrow Cu 2^6 puedo representar 64 valores distintos \Rightarrow utilizar un byte y ahorrar

Formato comprimido: 1 byte

Formato no comprimido: 45 bytes x prov.

③ COMBINACIÓN VAL-UNIDAD

Se codifica mediante:

Formato Valor-UNIDAD

① Valor único: (que no se repite)
de que el valor sea único

Tienes distinción en el resto de

② Valor repetido

③ Nº de veces q se repite

Ejemplo:

Original 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
Comprimido 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
#2302

Se usa en mapas de bits

Que un carácter tenga mayor frecuencia que otro, no implica que su frecuencia sea más alta que la del otro, lo q si q implica es que no será más larga (será menor o igual)

A la hora de leer el fichero comprimido, no habrá ambigüedad

a c d d e 40 bits

01000111101 22 bits

Para descomprimir, vamos cogiendo bits y seguimos al final desde la raíz. Cada vez que llegamos a una hoja, señalamos el carácter, y volvemos a la raíz.

01000111101

a c d d e

2.3.2. BÚSQUEDA Y ORDENACIÓN

- Búsqueda por clave
- Búsqueda secuencial $O(n)$
- Búsqueda binaria $O(\lg n)$
 - Secuencial
 - Binaria ordenada
 - Binaria sin ordenar el fichero
- Binaria ordenada:
 - Si RAH si cabe, pero no se a ser lo habitual
 - Selección, heapsort, burbuja son bastante malos en esta
 - Quicksort, que nos llevará a estudiar los índices
 - MergeSort va a ser el mejor, dividiendo el fichero en K partes que entren en RAH

- ② Leer fichero → crear array de registros ordenados en RAM
- ③ Ordenar el array en RAM
- ④ Recorrer el array, leer registros en las posiciones indicadas por los cores clave ordenados.
- ⑤ Escribir los registros en el fichero de salida ordenado.

Cada registro se lee 2 veces y se escribe 1 sola vez.

• 1ª lectura es secuencial (Hasta al final del fichero completo)

• 2ª lectura es registro por registro (Los números leyendo según diga la tabla ordenada).

- 1 seek por registro

• Escritura parece secuencial, ya se va escribiendo una columna después de otra.

• Sin embargo, es por saltos, ya entre cada escritura hay muchos que no se leen.

La 2ª lectura y la escritura, mejoraría este método.

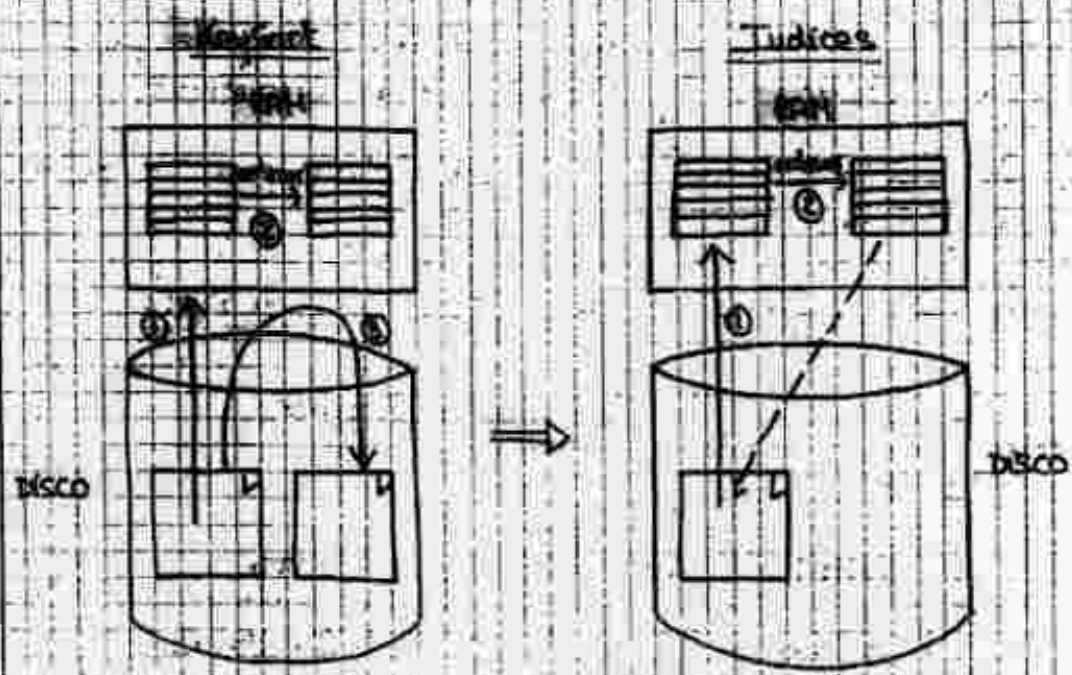
INCONVENIENTES DE LA ORDENACIÓN DE FICHeros PARA FACILITAR BÚSQUEDAS (en general)

- La ordenación es muuy costosa por 2 los accesos a disco.
- Actualización del fichero: coste de mantener el orden (hay q guardar el fichero ordenado).
- Búsqueda binaria → O(log₂n) datos.
- Se viene a ordenar por un solo campo.

Lo que podemos hacer es mantener la tabla con
pares clave/posición en RAM y locar las búsquedas
sobre esa tabla. Después accedemos al archivo original
directamente.

Nos acercamos a 2^a letra y a escritura del nuevo
archivo.

Esta tabla en RAM es lo que se llama índice.



22. Índices

Que se concuerden:

- Búsqueda en sus acceso a disco.
- Se crea múltiples ordenaciones para los múltiples datos.
- Permiten no buscar en todo el archivo (en términos de acceso a disco)

Índice simple si cabe en RAM, índice en pares clave / posición. Si no cabe, otras alternativas.

Se busca el índice en RAM y se busca el dato en disco.

23. INDICES SIMPLES Y COMPLEJOS

① INDICES PRIMARIOS (no se repite)

Ejemplo:

| | ISBN | TÍTULO | AUTOR |
|---|----------|---------------------|----------------|
| 0 | 450677 X | Lejos de Venecia | Vila-Helms |
| 1 | 246521 F | Rapela | Cortázar |
| 2 | 027159 Y | 100 años de soledad | García-Marquez |
| 3 | 925167 K | El amor en los... | García-Marquez |

↓
Clave prima
(no se repite)

↓
Secundaria

↓
Secundaria

④ Tienen longitud fija

Es importante que exista una clave primaria. Si no existe, se puede crear una o se postulan uno de los campos.

Array → Índice

CLAVE PRIMARIA | Posición

0247259 Y | 2

150627 X | 0

236524 F | 1

925467 K | 3

← Es fundamental que las claves estén ordenadas

Este índice se guarda en un fichero. ¿Cómo se gestiona ahora?

OPERACIONES CON INDICES PRIMARIOS

① CREAR

- Leer el fichero de datos.
- Crear el array en RAM, ordenar.
- Guardar índice en un fichero

② BUSCAR

- Buscar la clave en índice (búsqueda binaria en RAM) (Se devuelve una posición)
- Leer registro en la posición que acompaña a la clave encontrada.

③ INSERTAR

- Insertar registro en fichero de datos (cualquier sitio)
- Insertar nueva entrada en el índice con la clave del registro y la posición, manteniendo el orden.

Por ejemplo añadir 4. 342008 L El viaje...

⇒ En el índice

236524 F | 1

342008 L | 4

925467 K | 3

② INDICES SECUNDARIOS

Si general, no se tiene más de un índice primario.

Ejemplo:

| CLAVE SECUNDARIA | CLAVE PRIMARIA |
|------------------|----------------|
| García | 236521F |
| García - Márquez | 027159 Y |
| García - Márquez | 923467 K |
| Vila - Nata | 450677 X |
| Vila - Nata | 372006 L |

Ordenado

Ordenado por Busc
binaria

Ind. Secund

| | |
|--|--|
| | |
| | |
| | |
| | |
| | |

Ind. Prim.

| | |
|--|--|
| | |
| | |
| | |
| | |
| | |

Fichero
datos

Para evitar actualizaciones de los índices secundarios
ponemos la clave primaria y no las posiciones.

De hecho, cuando borramos algún registro, no es imprescindible que sea borrado en el índice secundario (actualizámoslo sólo al índice primario).

Con esto no hay problemas xq si buscamos algo que no existe, nos daremos cuenta al no encontrarlo en el índice primario.

OPERACIONES CON INDICES SECUNDARIOS

① BUSCAR REGISTRO/S

Búsqueda en
BDB

en Disco

- Buscar clave en índice secundario → clave/s
- ← primaria/s
- Buscar clave/s primaria/s en índice primario → posición de registro/s
- ← Leer registro/s en posición/es obtenidas

② AGREGAR REGISTRO Igual que en índice primario

③ ELIMINAR REGISTRO

- Responsables eliminación de la entrada correspondiente en el índice secundario
- Periódicamente limpiar el índice secundario (mirar en ind. primario si está, y si no, se borra)
 - Miramos la tabla de índices secund. enteros → comprobamos en el índice primario si está. Si no está, se borra del secundario.
- Si hacemos una compactación de un fichero de datos ⇒ tan sólo habría que cambiar el índice primario (el ind. secundario se quedaría igual)

④ CAMBIA UN CAMPO DE UN REGISTRO

- Si como consecuencia del cambio, el registro (sólo en caso de longitud variable) tiene que desplazarse a un lugar más grande, no hay que hacer nada (ya cambian sólo las posiciones y el ind. secundario no depende de las posiciones).
- Si cambia el campo indicado por el ind. secundario, actualizar la clave secundaria en la entrada correspondiente del índice y mantener al orden.
- Si cambia el campo indicado por el ind. primario, actualizar la clave primaria en el ind. secundario. Si se repite la clave secundaria, mantener al orden (hay que ordenar).

2.4.2 FICHEROS DE LISTAS INVERTIDAS

Vamos a solucionar el problema de qué repetimos en los índices secundarios.

Vamos a tener una sola entrada para la clave, en la que el índice asociado es una lista.

Interprete

Código

Kiko Varela

BCA 2759

COL 1526

RCA 0992

Paco de Luca

ANG 9418

BCA 5612

Ramón

COL 0425

RCA 1769

WAR 9221

7102199

Introducimos la lista en un fichero.

Posición

Clave prim

Rutero

Clave sec

Rutero

| | | |
|---|----------|----|
| 0 | BCA 2759 | 3 |
| 1 | COL 0425 | 6 |
| 2 | ANG 9418 | 5 |
| 3 | COL 1526 | 8 |
| 4 | 710 2190 | -1 |
| 5 | BCA 5612 | -1 |
| 6 | RCA 1769 | 7 |
| 7 | WAR 9221 | 4 |
| 8 | RCA 0992 | -1 |

| | |
|--------------|---|
| Kiko Varela | 0 |
| Paco de Luca | 2 |
| Ramón | 1 |

Índice secundario
abreviado



Fichero de listas invertidas

No truen por qué estar ordenado el fichero completo, pero sí las listas independientes.

Las flechas pueden ir hacia arriba, lo importante es que al recorrer la lista sacar ordenadas las claves secundarias.

Estos índices ahorran espacio y necesitas actualizar menos los índices en las actualizaciones.

Si tenemos varios judices secundarios creamos una lista en el archivo de listas invertidas. Esto se hace para evitar actualizaciones en los judices secundarios.

Para búsquedas combinadas, buscamos los judices primarios y hacemos la intersección.

El hecho de que los judices estén ordenados nos da mayor eficiencia al hacer la intersección.

y si no cabe el indice en una 2 Arboles B. Pero antes vamos a conocer técnicas de ordenación en ficheros:

2.4.3. PROCESAMIENTO SECUENCIAL Y ORDENACIÓN DE FICHEROS

MATCH Y MERGE

Ejemplo:

Alvaréz
Fernández
García
González
González
Pérez
Suárez

Brown
García
López
Martínez
Pérez

MATCH: es hacer la intersección. Iteramos sobre las dos listas.

Si lista 2[j] < lista 1[i], avanzamos j,
si no, avanzamos i. Si lista 2[j] == lista 1[i],
copiamos el valor, cuando termina una lista, paramos.

MERGE: Es hacer la unión ordenada. Iteramos sobre las dos listas. Comparamos el mayor va a la nueva lista y avanzamos su puntero.
Si lista 1[i] > lista 2[j] : copiamos i++

Hasta que va terminando las dos listas no termina.

Ordenación por registros

(1) Ordenación en RAM → registro

(2) Fichero no cabe índice a caso → ordenamos sólo el índice (visto)

(3) No cabe fichero ni índice → ordenación en disco { arboles B (siguiente capítulo)

(4) Ordenación en RAM → registro

fichero en disco → array en RAM → array ordenado → fichero ordenado

Dividir array en bloques (buffers)

Solapar lectura con creación de heap

Solapar ordenación con escritura

{ añadir bloque al heap (heap_insert).

{ leer siguiente bloque.

{ colocar elementos ordenados al final del array.

{ bloque ordenado → a disco.

L
E
C
T
U
R
A

Construir heap



Leer registros

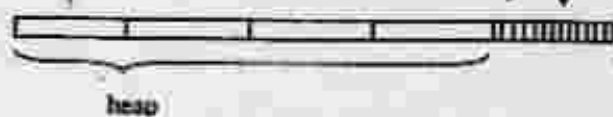
Leer registros

Leer registros



E
S
C
R
I
T
U
R
A

Ordenar



Ordenar



Escribir registros

Ordenar



Escribir registros

Así, lectura y escritura secuencial, al contrario que quicksort.

Ordenación en disco: mergesort

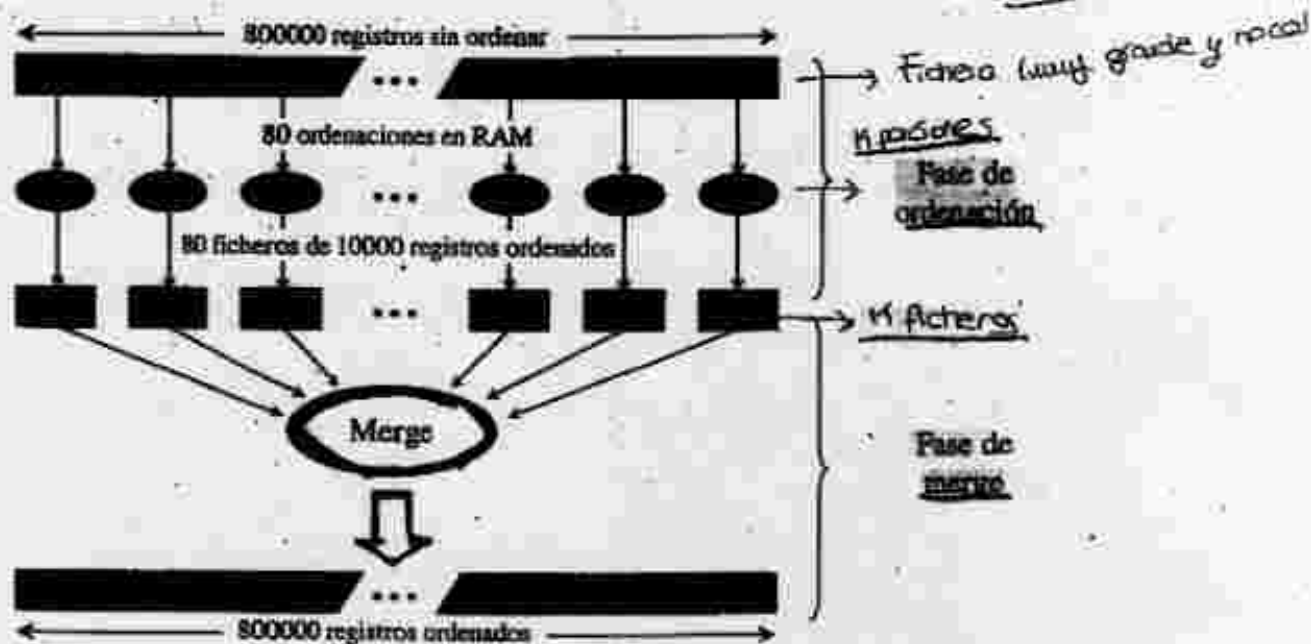
Quando ni siquiera el índice cabe en RAM.

- 800.000 registros de 100 bytes \rightarrow 80 Mb
- Claves de 10 bytes \rightarrow índice de más de 8 Mb
- 1 Mb de RAM

Mergesort

1. Dividir el fichero en k partes, que caben en RAM.
2. Ordenar cada división (p.e. heapsort) y guardar en un fichero.
3. Hacer merges de los k ficheros.

Ejemplo: en 1Mb de RAM caben 10000 registros \rightarrow hacemos $800000 / 10000 = 80$ trozos.



Vantajas:

- Ficheros de tamaño arbitrario.
- Lectura y escritura secuencial.
- En merge seeking sólo al alternar de fichero.

Coste de mergesort

Suponemos:

- Un solo seek por cada acceso secuencial.
- Ficheros almacenados de forma contigua.
- Ignoramos seek entre cilindros contiguos.
- Una sola rotación por acceso.
- 1 Mb de memoria disponible.

Características del disco:

- Tiempo de rotación: 11 ms
- Tiempo promedio de cada acceso: $9.5 \text{ ms (seek)} + 5.5 \text{ ms (rotación)} = 15 \text{ ms}$
- 1 sector \rightarrow 512 bytes
- 1 pista \rightarrow 63 sectores $\rightarrow 63 \cdot 512 \text{ bytes}$

$$\Rightarrow K_{un} \text{ trozos} = \frac{80 \text{ Mb}}{1 \text{ Mb}} = 80$$

* tiempo 1 acceso = seek + promedio rotación

$\text{tamaño trozo} = \text{tamaño disco} / M$ (en primer paso)
 $\text{tamaño trozo} = \text{tamaño disco} / \text{tamaño trozo}$
 $\text{tamaño buffer} = \text{tamaño RAM} / n \text{ trozos}$
 $\text{accesos por trozo} = \text{tamaño trozo} / \text{tamaño buffer}$
 $v \text{ de trans} = \text{tamaño pista} / \text{tiempo de 1 rotación}$

① Lectura del fichero inicial

- Accesos: para sacar cada trozo, un acceso $\Rightarrow 80 \text{ accesos} \times 15 \text{ ms} = 1.2 \text{ s}$

- Lectura de pistas: $n^\circ \text{ bytes del fichero} / n^\circ \text{ bytes de una pista} \cdot \text{tiempo de una rotación}$
 $80000000 / (63 \cdot 512) \cdot 11 = 27.2 \text{ s}$

Total: 28.4 s

$$\text{Si se dan el tamaño del fichero} = |F|$$

$$\Rightarrow \left[\frac{|F|}{\text{RAM}} \times \text{tamaño trozo} + \frac{|F|}{\text{tamaño buffer}} \right] \cdot K_{un}$$

también se dan el tiempo de acceso y la velocidad de rotación

② Escribir trozos a disco: igual que leer $\rightarrow 28.4 \text{ s}$

③ Lectura de trozos para merge: usar buffers para cada trozo en lugar de leer registros uno a uno

- Accesos:

- 1 Mb de RAM, 80 buffers $\Rightarrow 1 / 80 \text{ Mb por buffer}$

- Cada trozo tiene 10000 registros $\times 100 \text{ bytes} = 1 \text{ Mb} \Rightarrow \text{acceder 80 veces a cada trozo.}$

80 trozos $\times 80 \text{ accesos} = 6400 \text{ accesos}$

$\times 15 \text{ ms} = 96 \text{ s}$

- Lectura de pistas: igual que antes $\rightarrow 27.2 \text{ s}$

Total: 123.2 s

Cuando el tamaño crece, esta es la parte que determina el costo

④ Escritura del fichero final

No se puede usar RAM para buffers, por lo que se usa un buffer

Supongamos 2 buffers output del tamaño de 20000 bytes, $2 \cdot 10^4$

- $N^\circ \text{ de accesos: } 80000000 / 20000 = 4000 \text{ accesos}$

$\times 15 \text{ ms} = 60 \text{ s}$

- Escritura de pistas: 27.2 s

Total: 87.2 s

Total: 267.2 $\rightarrow 4' 27''$

Ordenar: 56.8 s

Merge: 210.4 s

\rightarrow Comparación keysort: un seek para cada registro del fichero $\rightarrow 800000 \text{ seeks (3h 20min)}$

Mejorar mergesort?

- Lectura de pistas: depende sólo del tamaño del fichero, independiente del troceo.

Inevitable, no se puede mejorar.

- Número de accesos

o Fase de ordenación:

- Lectura y escritura de cada trozo es secuencial.

- Intra-bloque

- Inter-bloque (salvo que otros procesos muevan brazo lector entre bloques y bloques).

- Número de seeks es mínimo, no se puede reducir.

o Fase de merge:

- Alternar entre ficheros: la lectura de cada 1/80 de fichero es secuencial, pero nada más.

- El número de seeks depende del número de trozos.

- Output: no se puede mejorar.

Análisis cuantitativo

Muchos registros \Rightarrow muchos trozos \Rightarrow buffers más pequeños \Rightarrow muchos más accesos

10×800.000 registros \Rightarrow 800 trozos, 800 accesos a cada uno \rightarrow 640.000 accesos (es decir $\times 10^3$)
Lo demás se multiplica por 10.
Total: 3h 10min

registros $\times 10 \Rightarrow$ tiempo $\times 40\pm$

En general, para k trozos:

Tamaño de cada buffer es: RAM/k tamaño trozo / k

Nº total de accesos \Rightarrow el coste es $O(k^2)$ en términos de número de accesos.

$k = \text{tamaño fichero} / RAM = n \text{ de registros} \times (\text{tamaño registro} / RAM) \Rightarrow \text{coste } O(n^2)$

Conclusión: crecimiento muy rápido del coste respecto al tamaño.

Soluciones

Hardware:

Más RAM: $RAM \times m \Rightarrow k/m$ trozo, k/m buffers, n^2 accesos en fase merge / m^2

Por ejemplo, con 4 Mb, para 8 millones de registros $\rightarrow \pm 38$ min

n^2 de accesos en merge $= O(n^2 / RAM^2)$

Otros: más unidades de disco, más canales de i/o \rightarrow solapamiento de i/o.

(Si fuese posible, un trozo en cada disco al hacer merge)

Software: disminuir k para poder usar buffers más grandes en fase merge.

Merge en varios pasos.

Selección conemplazo.

Merge multipaso

Queremos menos trozos \Rightarrow más grandes.

Por ejemplo, para un fichero de 8.000.000 registros:

Lo mejor que podemos hacer en un principio son trozos de 1 M \rightarrow 800 trozos.

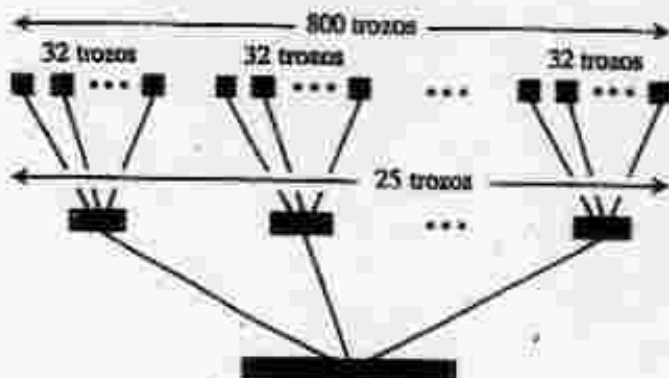
Después podemos formar 25 grupos de 32 ($32 \times 25 = 800$), hacer merge \rightarrow 25 trozos.

Merge final de los 25 trozos.

Formar trozos ordenados.

Hacer merge intermedio de grupos de trozos \rightarrow trozos mayores en menor número.

Hacer el merge final a un sólo fichero.



Algoritmo

Ordenación y formación de trozos mediante dos heaps

0. Inicialización: Suma a heap primario H_1 .

1. Heap secundario $H_2 = \emptyset$.

2. Algoritmo de Buble.

3. Escoger el elemento menor x de H_1 .

4. Leer el elemento x .

a) Si x es menor que H_2 .

b) Si x es mayor que H_2 .

5. Cerrar el ciclo de Buble.

$H_1 \leftarrow H_2$

Repetir hasta
que $H_1 = \emptyset$

Repetir hasta
eof (entrada)
y $H_1, H_2 = \emptyset$

Sale un trozo

Cada vez que se vacía H_1 , queda formado un trozo.

Tamaño promedio de los trozos

Si disponemos de un espacio de RAM de m registros para guardar los heaps, tenemos que:

Inicialmente:

Tamaño inicial de $H_1 = m$ registros.

Tamaño inicial de $H_2 = 0$.

H_1

En todo momento del proceso: tamaño de H_1 + tamaño de $H_2 = m$.

Es decir, el tamaño ocupado por los heaps se mantiene constante ya que cada vez que sale uno entra otro (independientemente de donde se inserte).

(Observar en el ejemplo del principio que la longitud de los dos heaps siempre suma 4.)



Si los registros están en orden aleatorio en el fichero de datos, en promedio de cada dos registros leídos, uno irá a H_1 y el otro a H_2 .

Luego en promedio, por cada dos elementos que salen de H_1 , entra uno en H_2 .

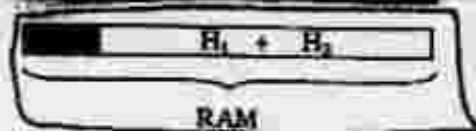
Así que al final, en promedio pasan por H_1 $2 \times$ tamaño inicial de $H_1 = 2 \times m$ registros.

Por lo tanto el tamaño promedio de los trozos = $2 \times m$.

Fichero de datos parcialmente ordenado \rightarrow trozos más grandes.

Los registros del fichero de datos no se deben leer de uno en uno necesitamos buffer I/O.

El espacio RAM dedicado a los heaps será el total de RAM menos el tamaño del buffer I/O.



Por lo tanto:

- Ventaja: trozos más grandes, tamaño promedio = $(RAM - \text{buffer I/O}) \times 2$

\Rightarrow poco más de la mitad de trozos que por el procedimiento estándar.

- Contrapartida: por el método estándar el buffer I/O era toda la RAM, ahora más pequeño

\Rightarrow aumentan los accesos en la fase de ordenación.

En total compensa.

25% RAM
25% buffer I/O

Coste de la selección con reemplazo

Supongamos buffer I/O de 2.500 registros (obs: siempre equilibrio input + output = 2.500).

Accesos:

- Fase de ordenación: $8.000.000 / 2.500 = 3.200$ accesos para lectura.
 $3.200 \times 2 = 6.400$ accesos lectura + escritura (frente a $800 \times 2 = 1.600$ accesos antes).
- Fase de merge:
 - Tamaño de trozos = $2 \times 7.500 = 15.000$ registros (promedio). 2×7.500
 - Nº de trozos = $8.000.000 / 15.000 = 534$ trozos \rightarrow 534 buffers.
 - Accesos por trozo = $\text{tamaño trozo} / \text{tamaño buffer} = (800M / 534 \text{ trozos}) / (1M / 534) = 800$ accesos por trozo.

Total merge: $800 \times 534 \text{ trozos} = 427.200$ accesos (frente a 640.000 accesos antes).

Conclusión: mejora moderada.

Selección con reemplazo + merge multipaso

La mejor opción:

Con selección con reemplazo formamos los trozos iniciales.

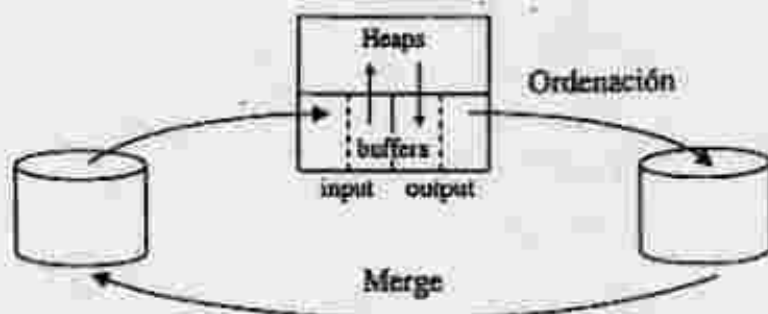
Después hacemos dos pasos de merge.

- El peso de la transferencia de datos (\pm la mitad del coste total) amortigua las mejoras (ver tabla).

Selección con reemplazo con dos unidades de disco

Se utiliza uno para input y otro para output.

- \Rightarrow Solapamiento de input y output \rightarrow tiempo de transferencia: reducción del 50%.
- \Rightarrow Solapamiento de entrada / salida con procesamiento CPU en la fase de ordenación (dibujo).
- \Rightarrow Reducción del número de accesos:
 - Ordenación: lectura casi totalmente secuencial (sólo contar rotación entre buffer y buffer).
 - Merge: output casi totalmente secuencial (disco output \leftrightarrow disco input).



Sumario

| | | Tamaño trozos | Accesos lectura en fase merge | Accesos total | Tiempo total (min) |
|-------------------------|-----------------|---------------|-------------------------------|---------------|--------------------|
| Mergesort | | 10.000 | 640.000 | 681.600 | 190 |
| mergesort multipaso | | 10.000 | 45.600 | 127.200 | 59 |
| | | 320.000 | (25 x 32) | | |
| Selección con reemplazo | orden aleatorio | 15.000 | 427.200 | 473.600 | 136 |
| | orden parcial | 40.000 | 160.000 | 206.400 | 69 |
| selección + | orden aleatorio | 15.000 | 38.038 | 124.438 | 58 |
| | | 420.000 | (19 x 28) | | |

| | | | | | |
|-----------|---------------|-------------------|---------------------|---------|----|
| multipaso | orden parcial | 40.000 800.000 | 24.000 (20 x 10) | 110.400 | 54 |
|-----------|---------------|-------------------|---------------------|---------|----|

ORDENACION POR FUSION

MERGESORT

80MB

Ejemplo: ordenar 800.000 registros de 100 bytes
 Tenemos 2Mb de RAM disponible.
 (m y k los tomamos como múltiplos de 10 y no potencias de 2).

Lo que vamos haciendo es coger trozos de 2Mb, cargarlos en RAM, ordenarlos y guardarlos en un fichero.

(los trozos que vamos cogiendo tienen que tener registros completos)

En RAM caben 20.000 registros ($\times 100 = 2 \cdot 10^6$ bytes = 2Mb)

Después ejecutamos Merge con todos los ficheros.

En la fase de ordenación hacemos 2 seeks por fichero (1 de lectura y otro de escritura).

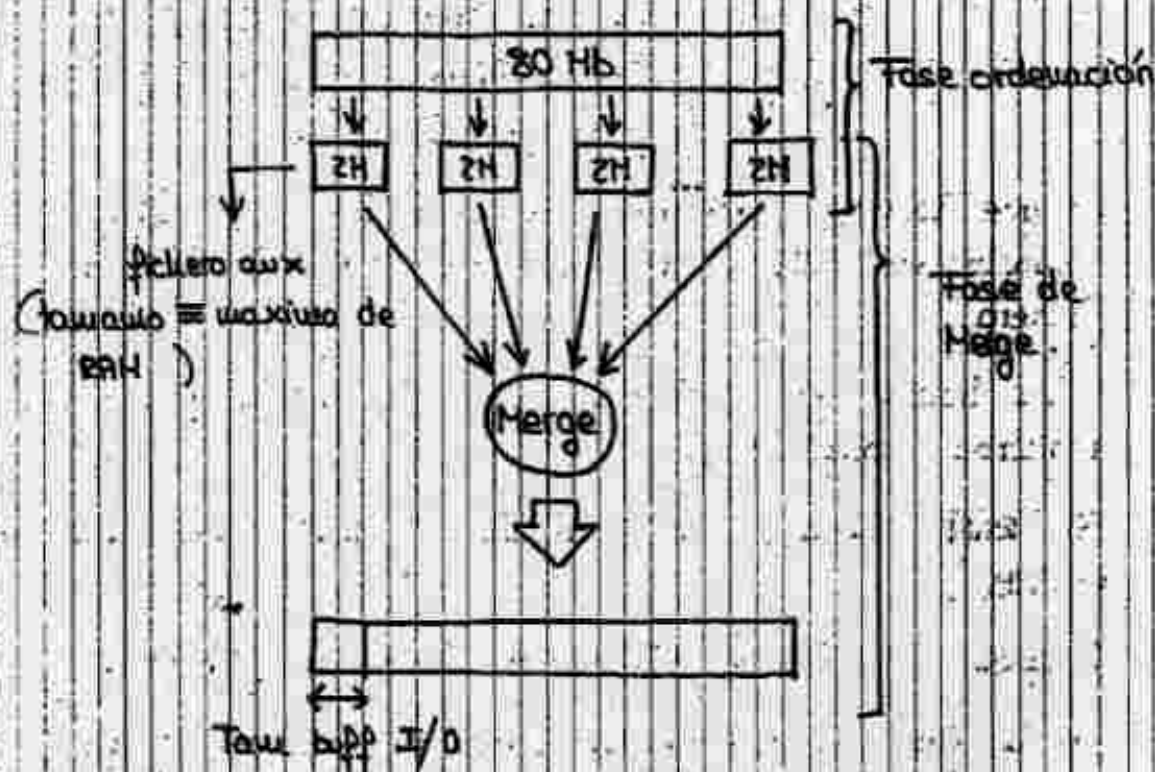
En la fase de Merge, en vez de leer un registro por fichero, iremos cogiendo varios a la vez.

ESTIMACION DE COSTE

Ejemplo: 800.000 registros de 100 b
 2Mb. de RAM disponible
 T. de rotación: 14 ms
 Seek promedio: 95 ms
 Tamaño pista: 59 x 512 b
 Buffer I/O Sistema Op: 20K
 $1K = 10^3 b$
 $1Mb = 10^6 b$

Calcular el tiempo necesario para ordenar el fichero (los registros)

MERGESORT



$$Taux f_{aux} = \left[\frac{RAN}{registro} \right] \cdot registro = \left[\frac{2 \cdot 10^6 b}{100 b} \right] \cdot 100 b = 2 \cdot 10^6 \text{ Bytes}$$

1) Fase ordenación:

Por cada fichero auxiliar

1. Lectura:

- N° accesos (saltos): 4 (nos colocamos al ppo y leemos secuencialmente)
- Coste de recorrido de datos:

$$\frac{2H}{63 \times 512b} \quad 44 \text{ ms} = 682'044 \text{ ms}$$

↓
T. rotación

$$\text{Coste recorrido datos} = \frac{\text{Tiempo a leer}}{\text{Tiempo p/s}} \cdot T_{\text{rotación}}$$

$$\text{Rotación promedio} = \frac{\text{Rotación}}{2} = \frac{11}{2} \text{ unidad } T_{\text{rotación}}$$

$$\bullet \text{ Coste n}^{\circ} \text{ accesos : } 1 \times (9,5 + 5,5) \text{ ms} = 15 \text{ ms}$$

→ promedio ⇒ media vuelta

② Ordenación en RAM: tiempo despreciable

③ Escritura en disco: lo mismo

$$0,68 + 0,015 \text{ s} = 0,695 \text{ s}$$

$$\text{TOTAL : } 0,695 \times 2 \times 40 = 55,6 \text{ s}$$

$$\text{N}^{\circ} \text{ faux} = \frac{\text{Tiempo total}}{\text{Tiempo RAM} = \text{tiempo faux}} = \frac{80 \text{ M}}{2 \text{ M}} = 40$$

② Base de Merge → 138,48 s.

① Lectura: Usamos un buffer RAM para leer cada página.

$$\text{Tiempo buffer para cada página} = \frac{2 \text{ Mb}}{40}$$

Por cada faux:

- Nº accesos: una cada vez que se llena el buffer.

$$\frac{\text{Tiempo faux}}{\text{Tiempo buffer}} = \frac{2 \text{ Mb}}{\frac{2 \text{ Mb}}{40}} = 40 \text{ accesos}$$

↓
40 · 15 ms = 600 ms

$$\text{Recorrido de dato} = \frac{2 \cdot 10^6 \text{ b} \times 11 \text{ ms}}{63 \cdot 512} = 0,683$$

$$\Rightarrow 0,68 \times 600 \text{ ms} = 1,28 \text{ ms}$$

$$\text{TOTAL : } 1,28 \text{ s} \times 40 = 51,2 \text{ s}$$

2. Escritura Usando el buffer I/O del Sist.
- Operativo

$$\text{Nº accesos} = \frac{\text{Tamaño total}}{\text{Tam. buffer I/O}} = \frac{80 \cdot 10^6 \text{ b}}{2 \cdot 10^4 \text{ b}} = 4000 \text{ accesos}$$

↓

$$\text{Recorrido datos} = \frac{80 \cdot 10^6 \text{ b}}{512 \text{ b}} \times 44 \text{ ms} = 24,28 \text{ s}$$

$$\text{TOTAL: } 84,28 \text{ s}$$

$$\text{TOTAL MERGESORT: } 56,6 + 138,48 = 295,08 \text{ s}$$

3

ARBOLES B

Con la solución a los índices cuando no caben en RAM.

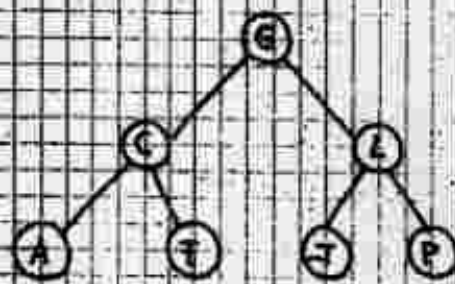
Si un índice simple no cabe en RAM (\Rightarrow estaría en disco):

1) Actualización tiene coste $O(n)$ ($n/2$ en promedio) en nº de accesos a disco (n es el nº de registros).

2) Búsqueda tiene coste $O(\log_2 n)$ en nº de accesos a disco.

SOLUCIÓN BWA 1)

Objetivo: Guardamos el índice en un árbol binario de búsqueda.



Podemos guardarlo en una tabla con 3 campos:

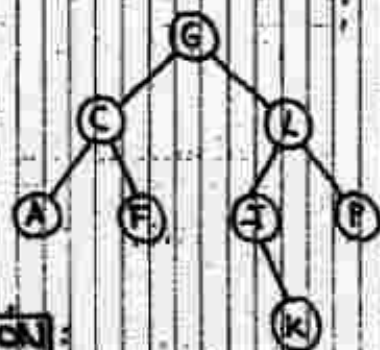
RowID: 3

| | Clave | Id | Dato |
|---|-------|----|------|
| 0 | F | -1 | -1 |
| 1 | L | 5 | 4 |
| 2 | A | -1 | -1 |
| 3 | G | 6 | 1 |
| 4 | P | -1 | -1 |
| 5 | I | -1 | -1 |
| 6 | C | 2 | 0 |
| 7 | K | -1 | -1 |

$\rightarrow 7$

\leftarrow Insertamos K

Una vez insertado K:



$$h \approx \log_2 n$$

INSERTION:

Nº discos: (a disco)

- ① Escribir nueva tabla
- ② Leer NUE
- ③ Recorrido árbol

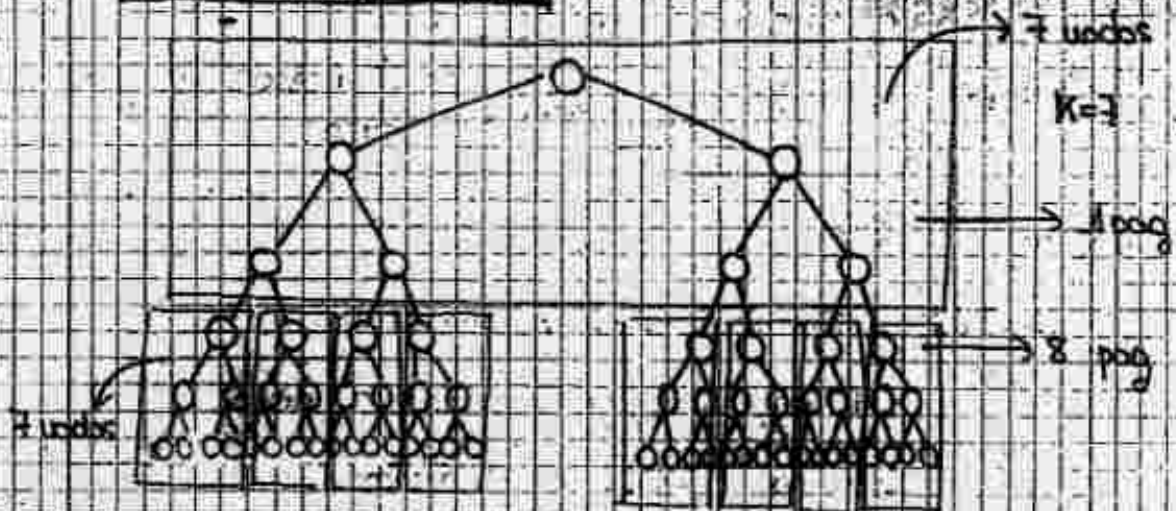
$$\Rightarrow \text{Nº discos} = \text{altura árbol} = \lceil \log_2 n \rceil + 2$$

(Siempre y cuando el árbol esté equilibrado)

BUQUEDA: $O(\log_2 n)$ Siempre y cuando el árbol esté equilibrado.

Problemas: { Eliminación
Desequilibrios

SOLUCIÓN PROBLEMA 2



todas las páginas tienen \Rightarrow 7 nodos (en este caso)

Se usa el índice para la búsqueda internos a través de los índices (recomendados)

Formación \rightarrow altura del árbol de páginas

$$K \cdot K \cdot K \cdot K \cdot K$$

| Nivel | Nº páginas | Nº claves |
|-------|----------------------|--------------------------|
| 1 | 1 | 1 · K |
| 2 | 1 · (K+1) | (K+1) · K |
| 3 | (K+1) · (K+1) | (K+1) ² · K |
| 4 | (K+1) ³ | (K+1) ³ · K |
| ... | ... | ... |
| h | (K+1) ^{h-1} | (K+1) ^{h-1} · K |

Sumando

$$K \cdot \sum_{i=0}^{h-1} (K+1)^i =$$

Total claves

$$= K \cdot \frac{(K+1)^h - 1}{(K+1) - 1} = \textcircled{2}$$

$$(k+1)^{n-1} - 1 = n$$

⇒

$$n = 1 + \log_{k+1}(n+1) = \boxed{\text{Nº accesos en peor caso}}$$

Ejemplo:

$$k = 511 \quad n = 134217727 \text{ registros}$$

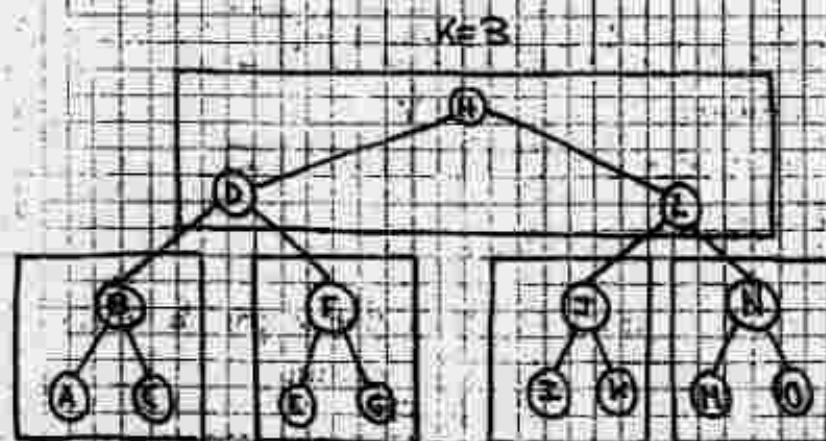
$$1 + \log_2(n+1) = 28 \text{ accesos a disco}$$

$$1 + \log_{512}(n+1) = 4 \text{ accesos}$$

Si k aumenta, será mejor ya la cantidad de accesos será menor, aunque k tiene sus límites.

Para insertar, crea una nueva página a medio leer (segura habiendo problemas de desequilibrio).

3.2 Representación del árbol de páginas en un fichero



Guardamos las páginas en cualquier orden : y asignamos los subíndices

| | | | | | |
|---|-----|----|----|----|----|
| 0 | EFB | -1 | -1 | -1 | -1 |
| 1 | ABC | -1 | -1 | -1 | -1 |
| 2 | JK | -1 | -1 | -1 | -1 |
| 3 | MNO | -1 | -1 | -1 | -1 |
| 4 | DAL | 1 | 0 | 2 | 3 |

Para solucionar el desequilibrio de momento usaremos una uoda.

Nos vamos a asegurar de q esto equilibrado al menos al 50% lo que nos interesa sobre todo es que las páginas estén lo más completas posibles (las páginas estarán llenas al 50%).

El procedimiento va a ser que antes de crear una nueva página, vamos a dividir su dos la anterior (que está completa).

Ejemplo: $K=5$, insertar E

Página 9

-1 A -1 B -1 C -1 D -1 F -1

el procedimiento
antiguo sería añadir
por aquí.



En vez de eso, dividiremos la página por la mitad
(auténor)

9 -1 A -1 B -1 C -1

23 -1 E -1 F -1

← ABC | D | EF →

y la nueva clave "promedio" a la página superior
(si no existe, se crea) (sube)

9 D 23

9 -1 A -1 B -1 C -1

23 -1 E -1 F -1

En vez de crecer el árbol por abajo, "crece por
arriba."

Para guardar las páginas en fichero, la estructura que
usaremos es:



Esto son los árboles B de orden K (K claves máximo por página)

4.

Aunque no se haya dicho, junto a cada clave aparece su posición en la bd. No olvidar que esto es un índice.

3.3 Búsqueda en Árboles B

Ejemplo: $K=3$, buscos



- ① Longitud página raíz
- ② Buscar ahí (puede ser binaria)
- ③ Si no lo encontramos, seguimos la página que esté entre las claves donde debería estar la clave que buscamos.

BUSCAR

→ la página

Buscar (clave, fichero, np)

if (np == -1) return NO ENCONTRADO // N^o PAG = -1

pag ← leer (fichero, np) // LEO PAGINA

pos ← buscar clave en pag // BUSCO CLAVE EN PAG

// pos es la clave de la 1ª clave \geq a la que busco

if (pag.claves[pos] = clave) // la clave está en esta pag.

return np, pos

// en realidad debería ser

la pos en la bd

else

return Buscar(clave, fichero, pag.hijas[pos]) // RECURSIVO

// BUSCO EN PAGINAS HIJAS

Si el grupo: = buscar II

np : 2
pos : 2

- la página raíz
- la posición de la 1



→ Descendemos por el hijo 2.

np : 4
pos : 4

- la página siguiente
- la posición de la 1



ENCUENTRO: 4, 1

- buscar 6

np : 2
pos : 1



np : 0
pos : 2



np : -1



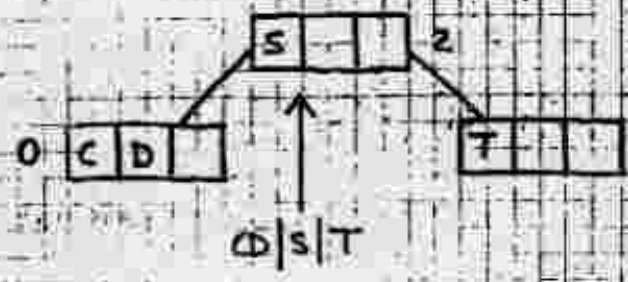
NO ENCONTRADO

B.4. Inserción en Árboles B

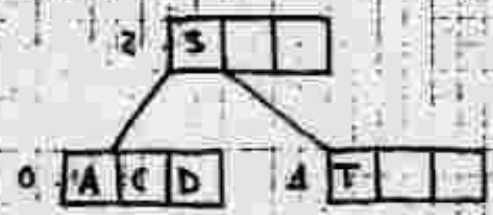
Ejemplo: $k=3$ insertar C S D T A N P I B W N G U R
K E H O L T Y Q E I X U

- ① Creamos la raíz: 0 [C] [] []
- ② Si cabe el siguiente, lo añadimos: 0 [C|S] [] []
- ③ Se inserta el siguiente: 0 [C|D] [] [] \Rightarrow 0 [C|D|S] [] []
(siempre por orden)

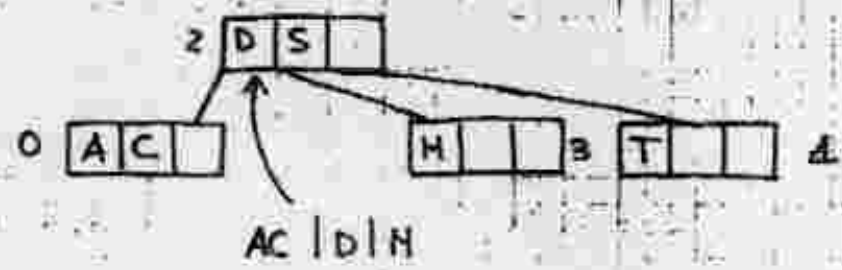
④ Como el siguiente va uno más, dividimos en dos:



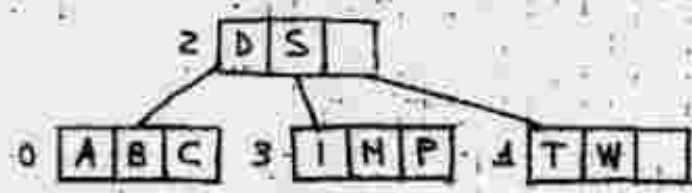
⑤ Como la A es menor, va para la izquierda:
(g, S)



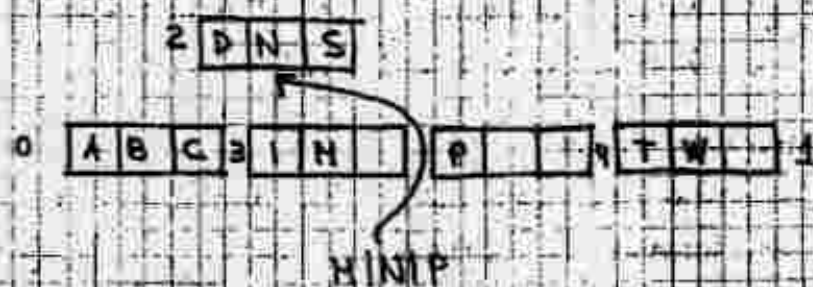
⑥ Al insertar la H hay que dividir el 0 y promociona arriba:



⑦ Insertamos B, I, P, W



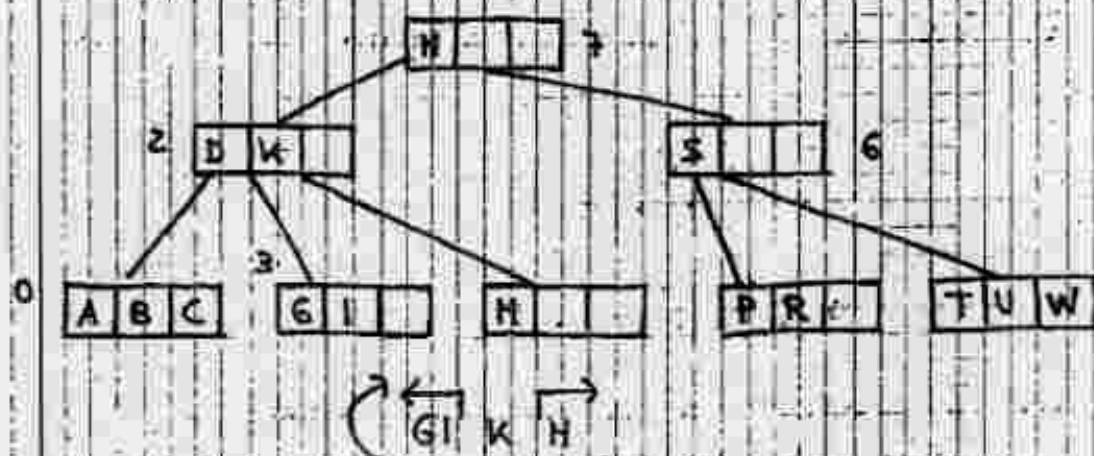
③ Al insertar la N nos para el usuario
(va al 3)



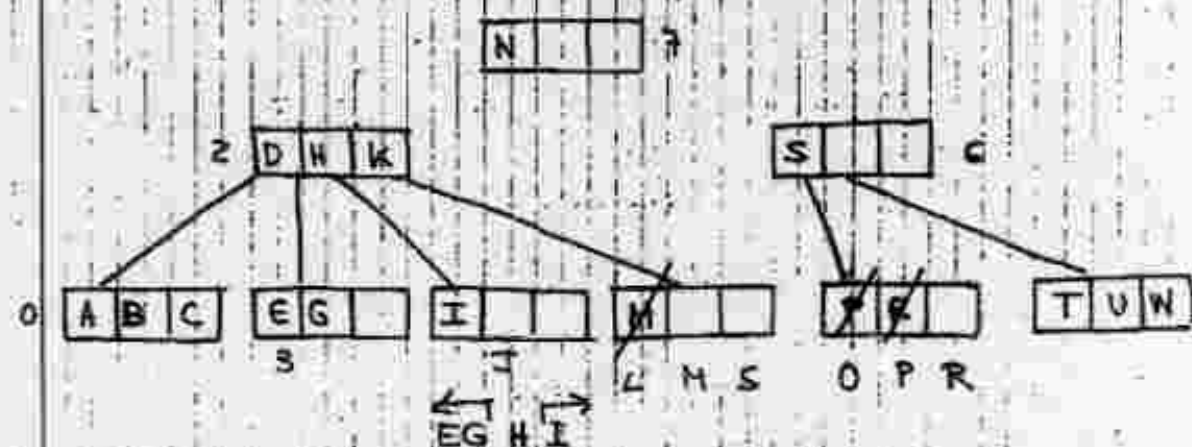
④ Insertamos N.O.U.R



⑤ Con K para una con N, pero además la raíz
está completa.

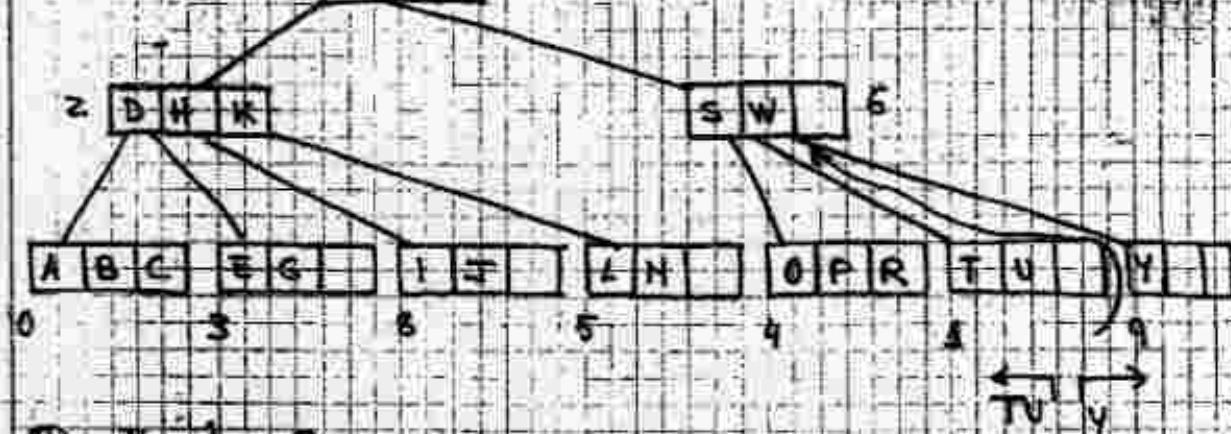


⑥ Moves la E, y la H nos divide el 3

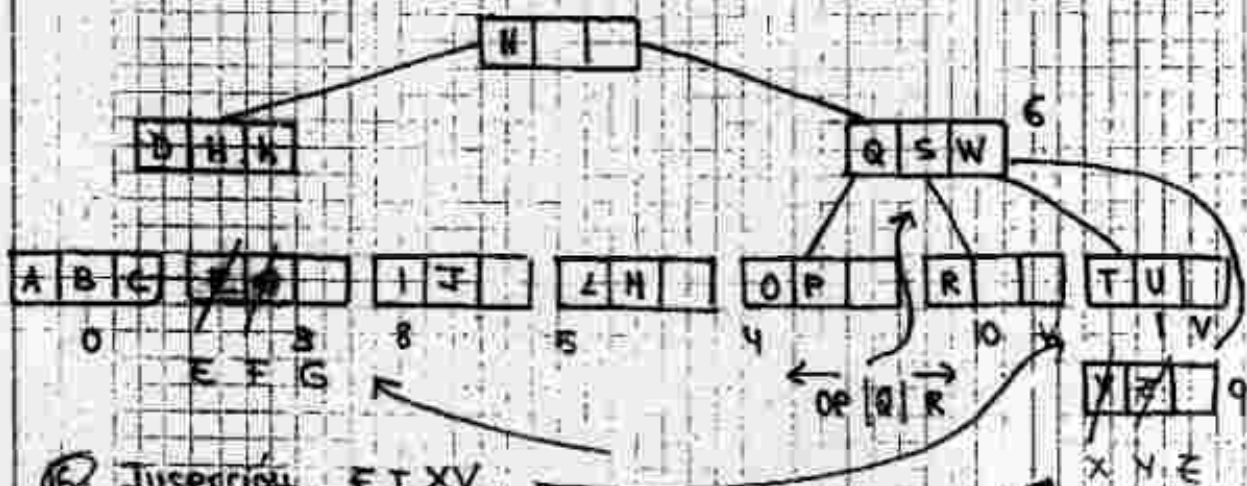


⑦ Insertión de O L I

13. Ins. N



14. Insertion Q



16. Insertion E I X V



Arboles B

Introducción

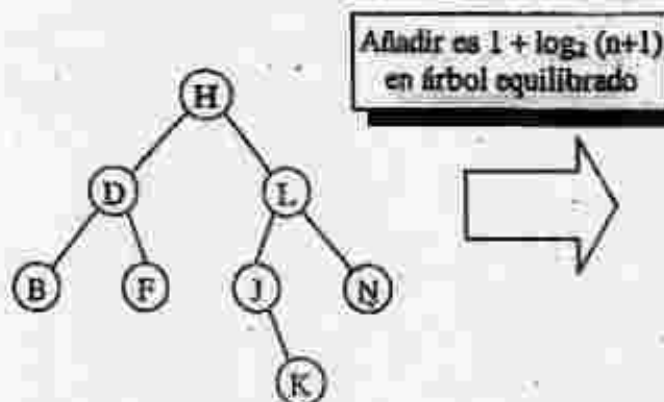
Índice no cabe en RAM, índice en disco.
Mergesort \rightarrow orden secuencial.
Arboles B \rightarrow estructura de árbol.
Más eficiente.

Inconvenientes de la búsqueda binaria en índice en disco:

1. Coste del mantenimiento de la ordenación (inviabile para p.e. miles de entradas).
Abrir hueco: $O(n)$.
2. Número de accesos para la búsqueda.

Solución para 1

- Generar un árbol binario de búsqueda (en lugar de ordenar por mergesort).
- Representarlo mediante punteros en un fichero de entradas sin ordenar.
- Para añadir nuevo registro, se sitúa al final, y basta cambiar el valor del puntero adecuado.
 - Encontrar posición es $1 + \log_2(n+1)$ si el árbol es totalmente equilibrado.
 - Cambiar puntero es 1.



Raíz: 5

| | Clave | izq. | dch. |
|---|-------|------|------|
| 0 | J | / | 7 |
| 1 | D | 2 | 3 |
| 2 | B | / | / |
| 3 | F | / | / |
| 4 | L | 0 | 6 |
| 5 | H | 1 | 4 |
| 6 | N | / | / |
| 7 | K | / | / |

Problema: desequilibrio del árbol (raíces mal escogidas).

Ejemplo: input preordenado.



Solución: equilibrar el árbol para limitar diferencias de longitud entre ramas.

Ejemplo: árboles AVL, rotaciones de elementos.

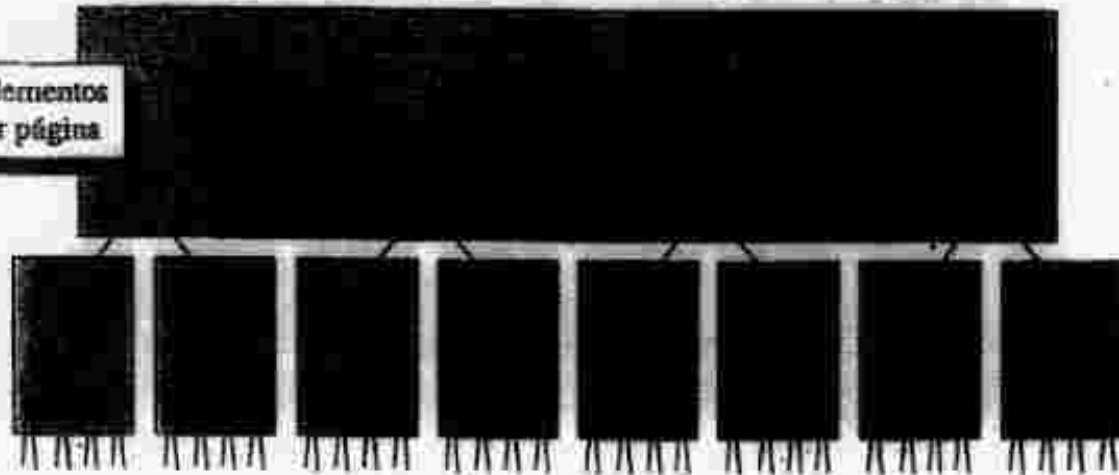
Aún así, demasiado profundos.

Solución para 2

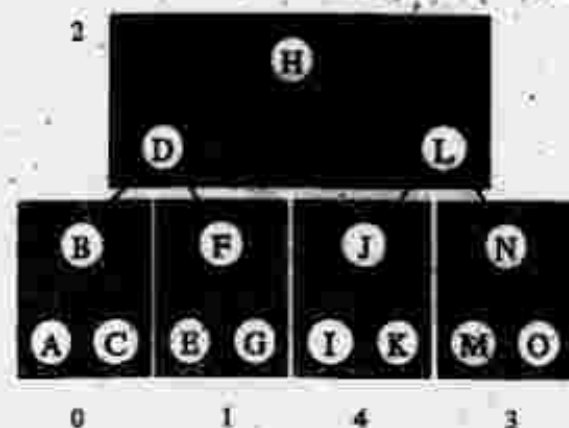
Descomponer el árbol en páginas de k nodos (explicar búsqueda).

Así se reduce también el número de accesos necesarios para añadir entradas.

7 elementos
por página



Representación de las páginas en fichero



Raíz: 2

Claves Páginas hijas

| 0 | A | B | C | -1 | -1 | -1 | -1 |
|---|---|---|---|-----|----|----|----|
| 1 | E | F | G | -1 | -1 | -1 | -1 |
| 2 | D | H | L | 0 | 1 | 4 | 3 |
| 3 | M | N | O | -1 | -1 | -1 | -1 |
| 4 | I | J | K | -1 | -1 | -1 | -1 |
| | k | | | k+1 | | | |

Coste de la búsqueda - Mayor coste árbol sin páginas que árbol con páginas

Nº de accesos para acceder a un elemento:

- Con páginas: $1 + \lceil \log_{k+1} (n+1) \rceil$ con $k = n^\circ$ de registros por página.
- Sin páginas ($k = 1$): $1 + \lceil \log_2 (n+1) \rceil$.

Por ejemplo: $1 + \log_2 (134.217.727 + 1) = 28$ accesos

$1 + \log_{511+1} (134.217.727 + 1) = 4$ accesos

Contrapartida: mayor página \Rightarrow $\begin{cases} \text{menos accesos} \\ \text{mayor coste de transmisión de datos} \\ \text{para cargar una página} \end{cases}$

Construcción de los árboles

Agregar elementos al árbol según van llegando \Rightarrow los primeros en llegar quedan cerca de la raíz.

Si llegan elementos muy pequeños o muy grandes pronto \Rightarrow árbol desequilibrado.

Ejemplo: C S D T A M P I B W ...



Equilibrar el árbol sobre la marcha:

Rotación de páginas \Rightarrow romper páginas \Rightarrow propagación de cambios, demasiados cambios.

Arboles B

Página de orden k :

- k claves (más otra información como referencias a registros) ordenadas.
- $k+1$ punteros a otras páginas.
- Contador del número de claves en la página.



Por ejemplo con 25-k bytes por página \rightarrow para $k = 4.000$ salen páginas de 100 Kb.

Algoritmo de B+.

Interés:

Equilibrio entre el número de páginas: lo más compacto posible.

Equilibrio en el número de páginas.

Método del árbol B: construcción de abajo hacia arriba.

1ª página: leer a RAM, ordenar.

Árbol B:

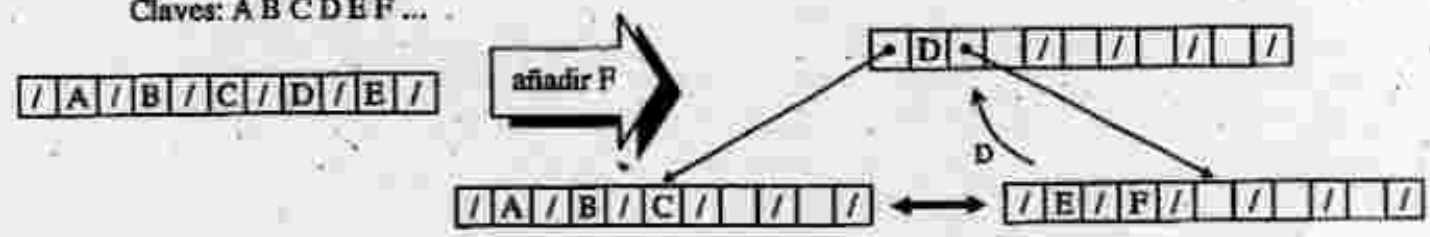
- Número de nodos.

- Promoción de una página.

Ejemplo 1

Páginas de 5 registros.

Claves: A B C D E F ...



Búsqueda en árboles B

Buscar (clave, fichero, nrr_actual)

if nrr = -1 then return NO ENCONTRADO

pag ← leer_pagina (fichero, nrr_actual)

pos ← buscar clave en pag

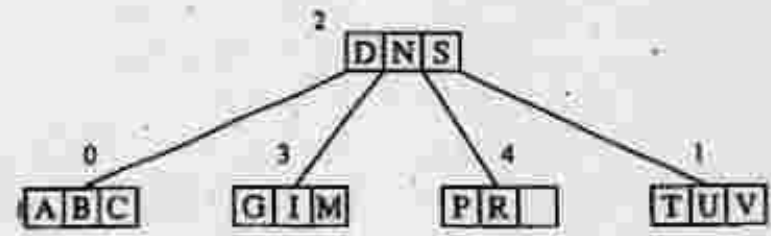
if pag.claves[pos] = clave then return ENCONTRADO, nrr, pos /* Clave encontrada

else return Buscar (clave, fichero, pag.hijos[pos])

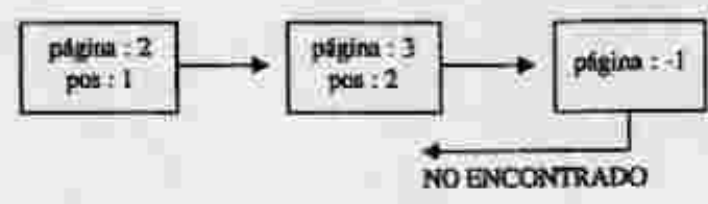
/* Venimos de una hoja */

/* Buscar más abajo */

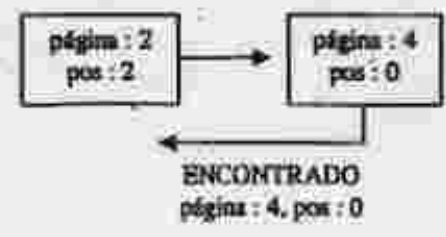
Ejemplo



Buscar K:



Buscar P:



OBS: varias páginas en RAM durante la búsqueda: tantas como niveles en el árbol.

Insertión en árboles B

Ejemplo 2

Páginas de 3 registros.

Insertar C S D T A M P I B W N G U R K E H O L J Y Q Z F X V.

(ver libro p. 350-351)

Insertar (clave, fichero, nrr)

if nrr = -1 then return PROMOCIONAR, clave, -1

Venimos de una hoja:
subir pidiendo promoción

pag ← leer_pagina (fichero, nrr)

pos ← buscar clave en pag

if clave encontrada return ERROR

Clave duplicada → error

resultado, promo_clave1, promo_nrr1 ← insertar (clave, fichero, pag, hijos[pos])

Seguir bajando

if resultado = PROMOCIONAR then

if cabe una clave en pag then

insertar promo_clave, promo_nrr en pag

escribir pag en fichero

return NO PROMOCIONAR

Hemos vuelto desde abajo
Nos viene clave a promocionar?

else promo_clave2, promo_nrr2 ← Partir (pag, promo_clave1, promo_nrr1)

return PROMOCIONAR, promo_clave2, promo_nrr2

else return resultado

resultado = ERROR o NO PROMOCIONAR

Fase
bajada

Fase
subida

Observar

- Fase de bajada, fase de subida.
- Valores de promoción recibidos y enviados.
- Puntos de retorno.

Tendremos páginas de orden $K \Rightarrow$ están K nodos en página como máximo.

Por la construcción del árbol B, las primeras claves nos caben en la página y las podemos ordenar. Llegará un momento en el que la clave no entra, entonces promocionamos el bloque y promovemos más hacia arriba. Por promoción, creamos el siguiente nodo a la izda del bloque que después de haberse podido e inserte a la izda de ella. Recuerda que las páginas tienen que estar ordenadas, y que por insertar hay que tener bien el sitio en el que se a de abajo.

Ejemplo
Pag. llena.

A B D

Añadimos

C

C no cabe en página \Rightarrow partir

A B

C D

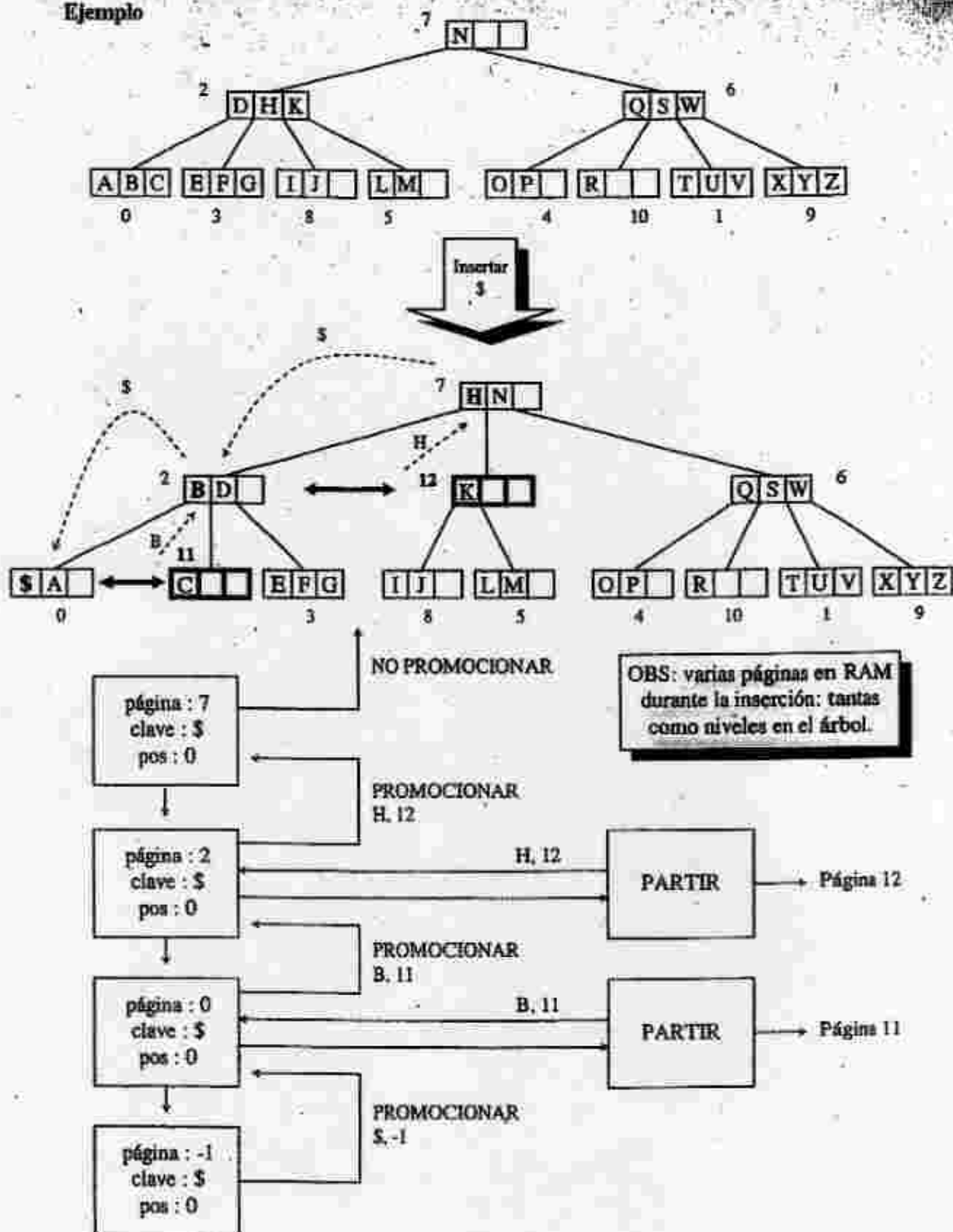
Promocionar (Carrito)

C

A B

D

Ejemplo



Partir (pag, nueva_clave, nuevo_nrr)
copiar pag a temp_pag, que tiene un espacio de más

insertar nueva_clave, nuevo_nrr en temp_pag

nueva_pag ← crear_pagina (fichero)

promo_clave ← clave central en temp_pag

promo_nrr ← nrr de nueva_pag

Partir se puede mejorar:
prescindir de temp_pag

copiar a pag claves y punteros de temp_pag anteriores a promo_clave

copiar a nueva_pag claves y punteros de temp_pag posteriores a promo_clave

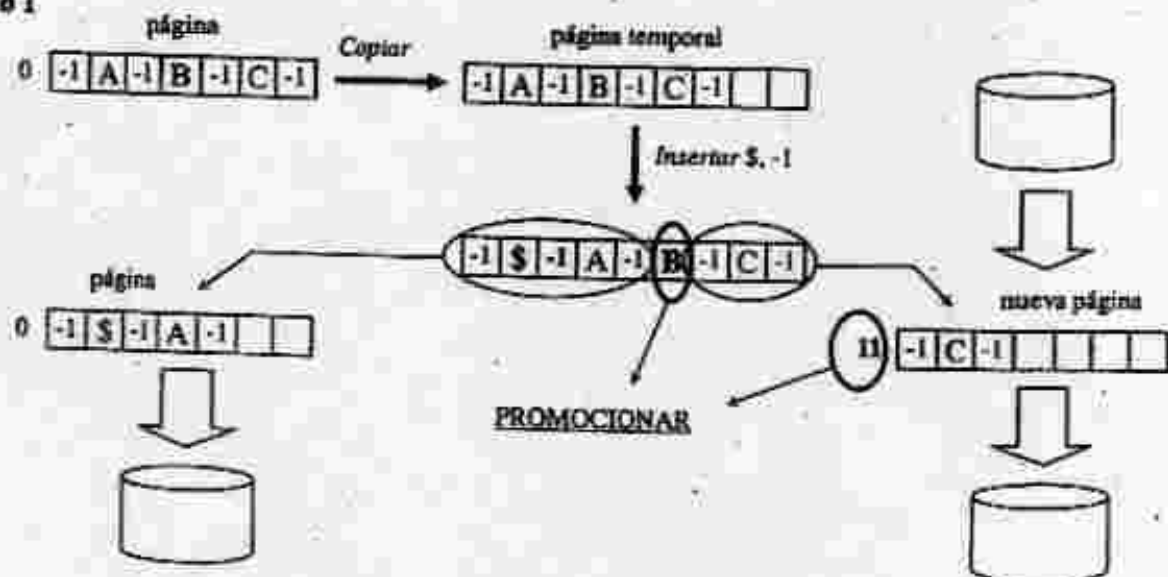
escribir pag y nueva_pag en el fichero

return promo_clave, promo_nrr

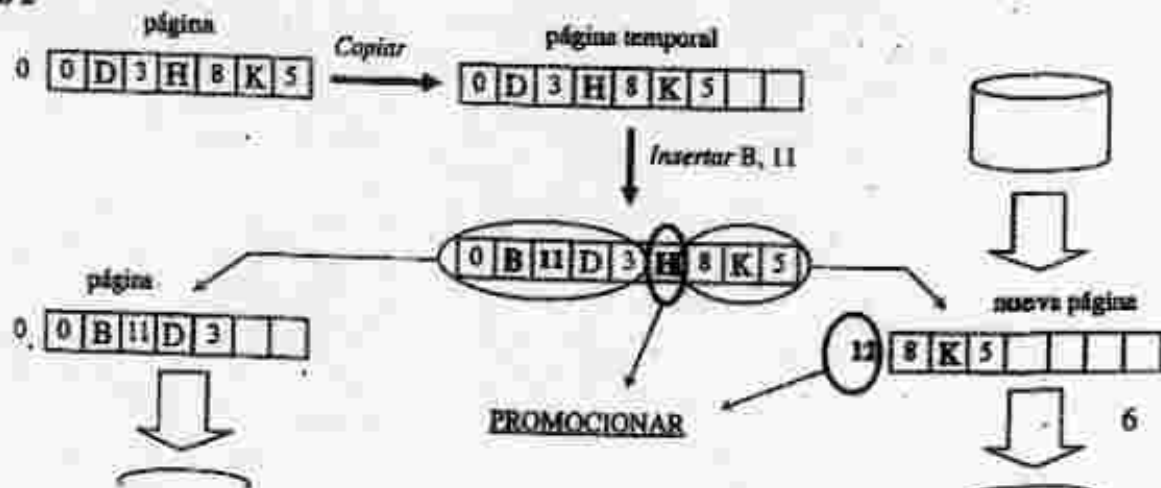
Observar:

- Partir recibe:
 - Página a dividir.
 - Clave + nrr a insertar.
- Partir construye:
 - Crea nueva página.
 - Reparte claves entre las dos páginas.
 - Guarda las páginas en el fichero.
- Partir devuelve:
 - Clave + nrr a promocionar

Ejemplo 1



Ejemplo 2



Insertar0 (clave, fichero)

nrr_raiz ← leer nrr_raiz de cabecera del fichero

RESULTADO, promo_clave, promo_nrr ← Insertar (clave, fichero, nrr_raiz)

If RESULTADO = PROMOCIONAR then

nueva_raiz ← crear_pagina (fichero)

guardar nrr_raiz + promo_clave + promo_nrr en nueva_raiz

escribir nueva_raiz en fichero

escribir nrr de nueva_raiz en cabecera del fichero

Por qué Insertar0: al insertar claves en árbol, código necesario para tratar las promociones que llegan más allá de la raíz

Construir (fichero, claves)

if no existe fichero then

crear_fichero fichero

raiz ← crear_pagina (fichero)

clave ← leer (claves)

guardar -1 + clave + -1 en raiz

escribir raiz en fichero

escribir nrr 0 en cabecera del fichero

while quedan claves do

clave ← leer (claves)

Insertar0 (fichero, clave)

Arboles B+

Propósito:

- Combinar organización indexada y secuencial.
- Arboles más anchos

Introducción

Vistas de un fichero de registros:

Indexado

- Acceso rápido a registros individuales.
- Mantenimiento eficiente.
- Procesamiento secuencial costoso (p.e. para procesamiento cosecuencial).

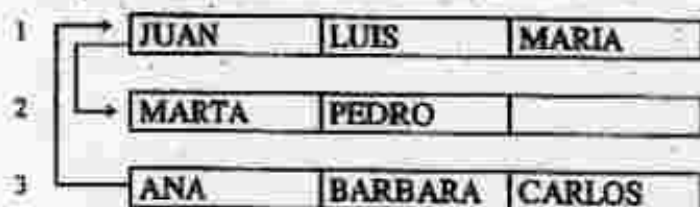
Secuencial: fichero ordenado.

- Búsqueda y mantenimiento costosos.
- Algoritmos para procesamiento secuencial.

Cómo combinar las dos vistas?

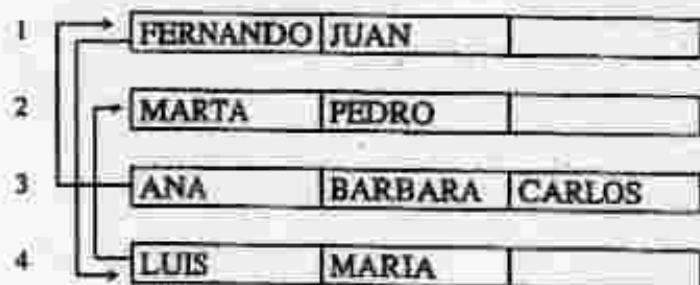
- Dividir fichero en bloques de k registros.
- Mantener orden dentro de bloques.
- Permitir que los bloques estén desordenados físicamente (bloques no ordenados físicamente).
- Procesar los bloques.

Ventaja: la concatenación sólo dentro de bloques → en RAM.



Añadir registros

Por ejemplo, añadir FERNANDO:



Partir

Concatenar

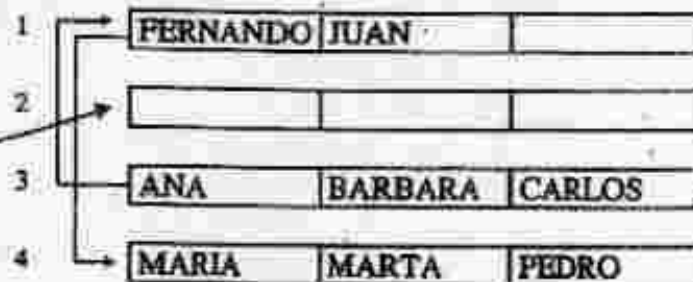
Eliminar registros

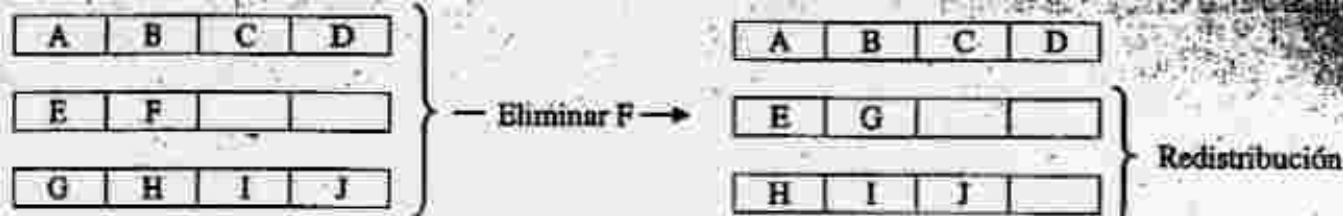
Si el bloque queda por debajo de la mitad:

- Redistribuir si hay bloque consecutivo con registros de sobra.
- Concatenar en otro caso.

Por ejemplo, eliminar LUIS:

Disponible para reuso





Inconvenientes:

- Fragmentación interna: los bloques no están del todo llenos.

Mejoras:

- Inserción con redistribución.
- División 2 → 3.
- Secuencialidad sólo a nivel de bloque.

Tamaño de los bloques:

En principio, cuanto más grande mejor. Pero:

- Limitaciones de RAM: tienen que caber varios bloques (2 para partir, 3 si se hace 2 → 3).
 - Contigüidad real de los datos del fichero: bloques del tamaño de los clusters.
- Cluster = unidad mínima de asignación: 8, 16 sectores...
- Tamaño máximo que garantiza lectura sin seek: evitar seek al leer un bloque.

Acceso indexado para secuencias:

Índice simple:

Array de punteros clave (separador) / ptr (ptr).

Qué es un separador:

| | | | | | | |
|------|-----|-------|------|-------|-----|-------|
| LUIS | ... | MARIA | MART | MARTA | ... | PEDRO |
|------|-----|-------|------|-------|-----|-------|

| | |
|------|---|
| F | 3 |
| L | 1 |
| MART | 4 |
| ∞ | 2 |



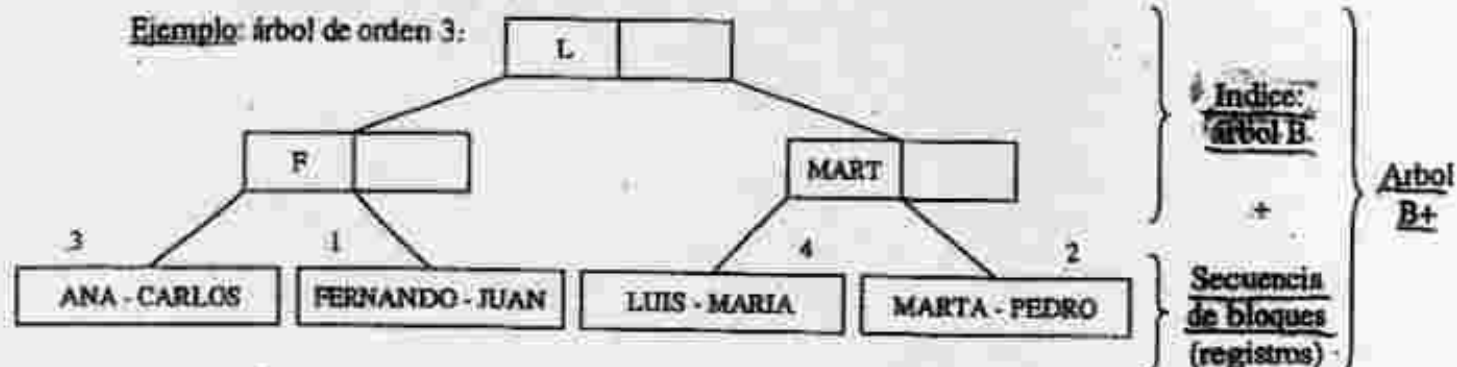
Cuando el

Organizar el índice de secuencias como árbol B:

Orden del árbol B?

En principio el que nos de la gana.
Más adelante vemos criterio.

Ejemplo: árbol de orden 3:



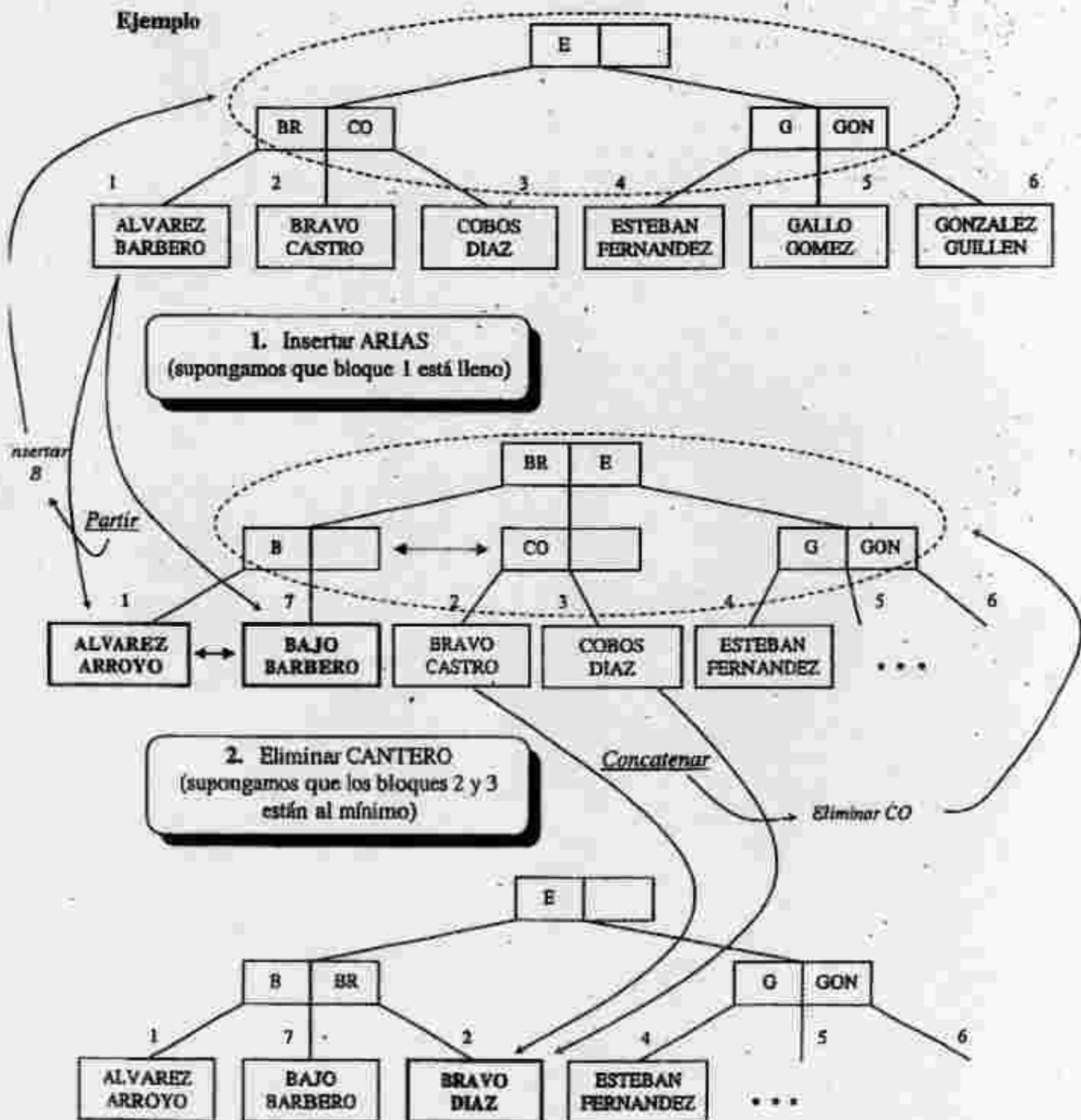
- las páginas del árbol contienen claves.
- las páginas contienen separadores (más cortos que las claves ⇒ caben más en cada página).

Ejemplo: buscar BARBARA

Mantenimiento de árboles B+ (prefijo simple)

- ① Buscar bloque en el que hay que insertar o eliminar registro.
- ② Inserción o eliminación no suponen partir ni redistribuir ni concatenar → el árbol no cambia.
- ③ Se parte un bloque → se crea un nuevo separador, se inserta en el árbol.
- ④ Se concatenan bloques → se elimina un separador del árbol.
- ⑤ Se redistribuyen registros entre bloques → generar nuevo separador, reemplazar el antiguo.

Ejemplo





Estructura de los nodos del árbol B+

Tamaño

Tamaño de página = tamaño de bloque

Los criterios para el tamaño de los bloques son válidos también para las páginas del árbol.

Homogeneidad entre páginas y bloques simplifica:

- Almacenamiento en el mismo fichero.
- Creación de un árbol virtual.

Estructura

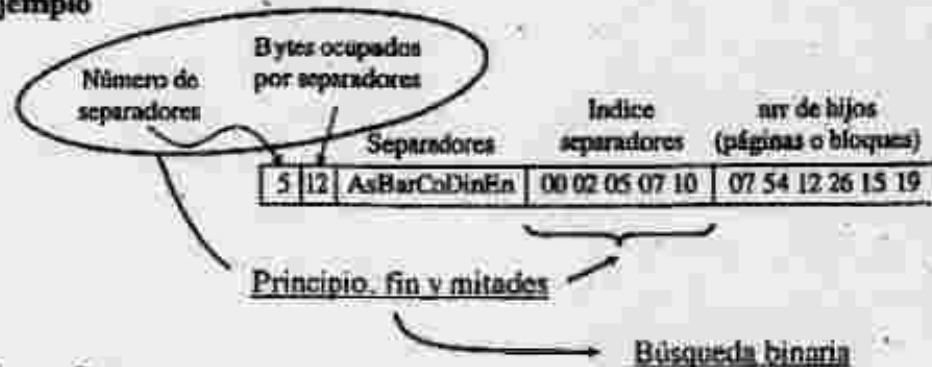
Nos interesa:

- Separadores de longitud más corta posible para meter tantos como quepan en cada página. (Orden variable de página)
- Búsqueda binaria para buscar claves dentro de una página.

Por tanto:

- Los separadores se guardan concatenados.
- Índice local en cada página que señale las posiciones en que empieza cada separador.

Ejemplo



Buscar Bravo:

- Búsqueda binaria:
 1. 05 → Co > Bravo ⇒ izquierda
 2. 02 → Bar < Bravo, fin.
- Bar es el separador en la posición 2 } → ir a bloque 12 (hijo en posición 3).
- Bravo > Bar
- El bloque 12 puede ser una página del árbol B o un bloque de registros.

Orden variable de los árboles

- Número de elementos por página depende del tamaño de las páginas, no de un orden prefijado.
- Estados como lleno, medio lleno, etc. para páginas y bloques se determinan en función de número de bytes en lugar de número de elementos (un poco más complicado que antes).

Construcción de un árbol B+

En lugar de construir a base de una llamada a insert por cada registro:

- Ordenar todo el fichero de registros.
- Ir sacando registros por orden, llenando y formando un bloque (en RAM).
- Cuando se llena un bloque, guardarlo y empezar el siguiente.
- Entre medias generar un separador.
- Ir metiendo los separadores generados en una página (en RAM) hasta llenarla.

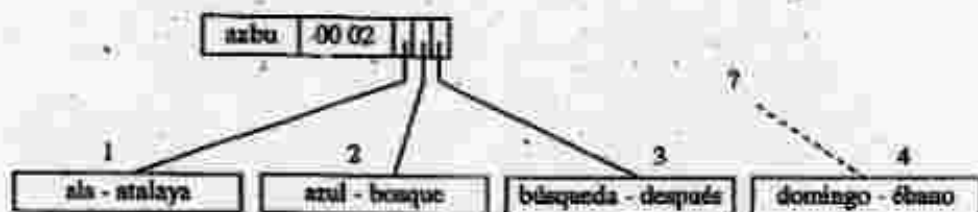
Ejemplo

Registros = palabras de un diccionario.

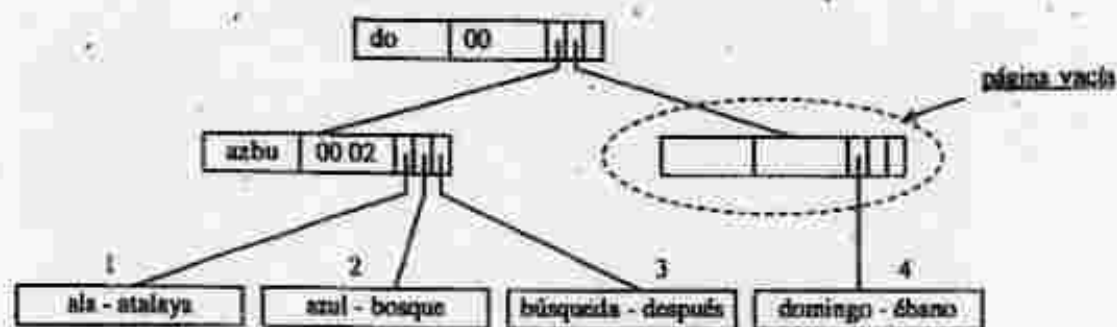
(En este ejemplo vamos a mostrar con más detalle la estructura interna de las páginas.)

Registros: ala - atalaya, azul - bosque, búsqueda - después, domingo - ébano, entrada - examen, ...

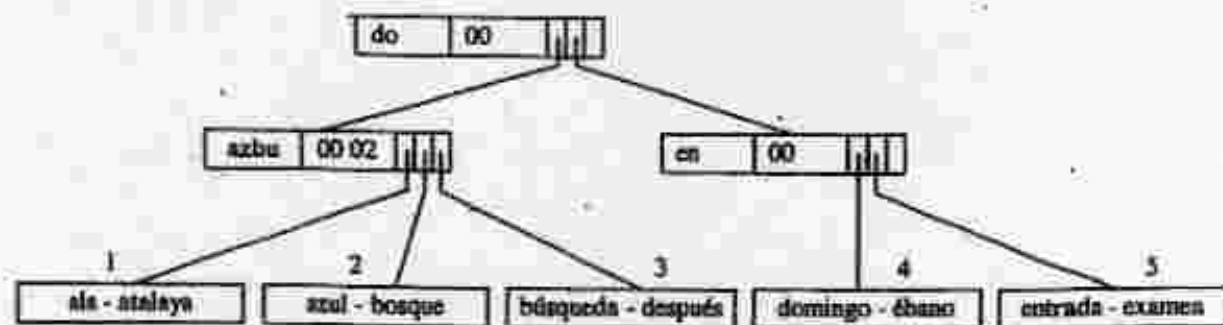
1. Hemos llenado tres bloques y una página, y completamos un cuarto bloque (domingo - ébano).



2. Creamos una página hermana → nos hace falta una página padre (esto no es partir ¿por qué?). La página hermana está vacía.



3. Llenamos otro bloque: entrada - examen.
La página que estaba vacía se empieza a llenar.



Páginas vacías: ¿desventaja?

Pueden quedar páginas vacías (como mucho una por nivel) justo cuando termina la construcción.
A la primera operación (insertar o eliminar) sobre esa página, se arregla el problema.

Ventajas

- Construcción más rápida:
 - Escritura secuencial.
 - Sólo una pasada sobre los datos (al ordenar) vs. varias con inserción aleatoria.
 - Los bloques no necesitan ser reorganizados en la construcción.
- Resultado mejor:
 - Bloques llenos al 100% vs. 67-80% con inserción aleatoria.
 - Páginas y bloques relacionados físicamente cercanos → seek más cortos.

Tipos de árboles, recapitulación

Similitudes

- Índice paginado → menos seeks.
- Árboles equilibrados en altura.
- Crecimiento de abajo hacia arriba: división, concatenación y redistribución para mantener el equilibrio.
- Técnicas de optimización:
 - Redistribución antes de partir
 - Partir 2 → 3
 - Buffer de bloques/páginas en RAM.
- Los nodos pueden tener contenido de longitud variable.

Diferencias

- Árboles B:
 - Pares clave/registro o clave/nrr: la información está en todo el árbol, no sólo en las hojas.
 - Acceso secuencial:
 - Se puede hacer recorriendo el árbol por orden de clave.
 - Funciona bien si se hace buffering de páginas.
 - Funciona bien si los registros están en el propio árbol.
- Árboles B+:
 - La información está en las hojas.
 - Procesamiento secuencial de verdad.
 - Caben más elementos por página: los separadores ocupan menos que los pares clave/registro o clave/nrr → árbol más ancho.
- Árboles B+ de mejor tipo simple:
 - Los separadores ocupan menos que claves enteras → más elementos aún por página.
 - Contrapartida: en cada página, índice para los separadores.

Propiedades y Costes

Propiedades

- Árbol B de orden k $\rightarrow \lceil k/2 \rceil \leq \text{claves por página} \leq k$
 $\rightarrow \lceil k/2 \rceil + 1 \leq \text{descendientes por página} \leq k+1$
- La raíz tiene al menos dos descendientes (salvo que la raíz sea hoja).
- Todas las hojas están en el mismo nivel.

Costes

El costo (nº de accesos) de búsqueda e inserción depende de la profundidad del árbol.

La profundidad se puede relacionar con el N total de claves en el árbol.

En el peor caso, mínimo número de claves en todas las páginas:

| Nivel | Descendientes | |
|----------|---|-----------|
| (raíz) 1 | 2 | k+1 |
| 2 | $2 \times (\lceil k/2 \rceil + 1)$ | $(k+1)^2$ |
| 3 | $2 \times (\lceil k/2 \rceil + 1) \times (\lceil k/2 \rceil + 1)$ | $(k+1)^3$ |
| | | |
| h | $2 \times (\lceil k/2 \rceil + 1)^{h-1}$ | $(k+1)^h$ |

Árbol con N claves $\rightarrow N+1$ descendientes después de las hojas.

$$p_h = c_h + p_{h-1} = c_h + c_{h-1} + p_{h-2} = \dots = c_h + c_{h-1} + \dots + c_2 + p_1 = c_h + c_{h-1} + \dots + c_2 + c_1 + 1 = N+1.$$

$$\Rightarrow N+1 \geq 2 \times (\lceil k/2 \rceil + 1)^{h-1} \Rightarrow h \leq 1 + \log_{(\lceil k/2 \rceil + 1)} ((N+1)/2)$$

Por ejemplo: árbol de orden 512 con 1.000.000 claves $\rightarrow h \leq 3 \cdot 36 \rightarrow h \leq 3$.

En general $h \leq 1 + \log_{(m+1)} ((N+1)/2)$ donde m es el mínimo número de claves por página.

Eliminación de claves

1. Buscar clave.
 2. Si no está en una hoja, intercambiar con la hoja adecuada.
 3. Eliminar la clave.
 4. Si la página no está bajo mínimo, fin.
 5. Si está bajo mínimo:
 - a) Redistribuir si hay una página hermana con una clave de sobra.
 - b) Concatenar en otro caso.
- La concatenación puede propagarse hacia arriba.

(ver ejemplo libro p. 368-369)

Optimizaciones

Redistribución en inserción

Antes de partir una página, intentar redistribuir con páginas hermanas.

Datos experimentales:

- Sin redistribución: páginas 69% llenas.
- Con redistribución: páginas 85% llenas.

Páginas más llenas
 \Rightarrow árbol más ancho

Arboles B*

División 2 → 3

- Hacer inserción con redistribución.
- Cuando se llenan dos páginas contiguas, partir las en tres.
- Raíz: no tiene hermanas.
 - Página raíz más grande para mantener la proporción ($1/2$ raíz = $2/3$ página estándar).
 - O bien división 1 → 2 ⇒ páginas bajo raíz pueden estar sólo a la mitad.

Páginas 2/3 llenas
en lugar de 1/2

Arboles B virtuales

Buffers de páginas: tener varias páginas en RAM, no sólo una.

Acceso a página:

- Mirar primero si está ya en RAM.
- Si no está en RAM, leer de disco, substituir una de las que estaban en RAM.

Criterios de substitución:

- La usada menos recientemente.
- La más profunda.

Por ejemplo, la raíz siempre en RAM.

Muchos menos
accesos a disco

Datos experimentales (árboles B+):

| | | | | | |
|--------------------------------------|-----------------|------|------|------|------|
| | Páginas en RAM: | 1 | 5 | 10 | 20 |
| Nº promedio de accesos por búsqueda: | | 3.00 | 1.71 | 1.42 | 0.97 |

2.400 claves

140 páginas

Profundidad: 3

Páginas con contenido de longitud variable

Nº variable de claves por página: las que quepan.

Permite páginas de orden
mayor en mismo espacio

Nota: meter registros en árbol ⇒ árbol de orden menor.

en la actualidad:

- Búsqueda en un acceso a disco.
- Múltiples ordenaciones para múltiples campos.
- Actualización no supone costo adicional.
- Funcionalidad mediante claves ordenadas.
- No se existe clave primaria si no se ordenan 2 campos

Indices Secundarios

Para evitar redundancia de los índices principales, se crea la clave primaria y no las secundarias. Si tenemos algún registro no tiene fecha modificarse el índice de orden y con el primer índice.

Operaciones:

1. Búsqueda:

- Buscar clave en índice primario → clave prim.
- Buscar clave en índice primario → posición.
- Leer de registro de fichero.

2. Añadir registro:

- Igual que en índices primarios.

3. Eliminar registro:

- Eliminar de fichero de datos.
- Eliminar en ~~primario~~ índice primario.
- Posteriormente ~~eliminar~~ en secundarios la tabla de índices secundarios para ver si sea el del primario.

4. Ficheros de listas invertidas:

- Se crea una sola entrada para cada clave en la que el índice correspondiente es una lista.
- No tiene pq estar ordenados el fichero completo pero sí los listas correspondientes.
- Al buscar en lista se tiene de sacar ordenadamente las claves secundarias.
- Se crea un espacio y se van actualizando nuevas los índices en las actualizaciones.

PROCEDIMIENTO CONVENCIONAL Y ORDENACIÓN DE FICHEROS

MATCH

es hacer la intersección sobre 2 listas ordenadas.

Match (lista1, lista2)

Inicializar

```
x ← input(lista1, -1)
y ← input(lista2, -1)
while (questar-derrotas)
  if (x < y) then x ← input(lista1, x)
  if (y < x) then y ← input(lista2, y)
  if (x = y) then
    write(salida, x)
    x ← input(lista1, x)
    y ← input(lista2, y)
```

Input (entrada, valor previo)

```
if (entrada) then
  valor-derrotas ← valor
  return valor
else read (entrada, valor)
  if (valor < valor-previo) then error
  else return valor
```

Indice primario

| clave primaria | posición (offset) |
|----------------|-------------------|
| 0217159V | 1 |
| 150627X | 0 |
| 276821F | 123 |
| ... | ... |
| ... | ... |
| 505167K | 415 |

orden

4. Eliminar:

- Eliminar registros de fichero
- Eliminar entrada en índice

5. Modificar un campo:

- Modificar en fichero
- Si el registro se mueve, modificar offset en índice
- Si se modifica el tiempo de la ~~entrada~~ entrada ~~entrada~~ entrada, no tiene

6. Cambiar un campo de un registro:

- Si es consecuencia del cambio el registro se desplaza, modificándose el índice primario involucrado.
- Si cambia el campo indicado por el índice primario / actualizar la clave primaria en el índice secundario.

- Si tenemos varios índices secundarios creamos una cadena en el fichero de listas invertidas.
- Para búsquedas combinadas, buscamos los índices primarios y hacemos la intersección

- Los ficheros de datos
 - Los arrays en RAM, ordenar
 - Guardar índice en fichero
2. Borrar
- Borrar la clave en el índice (Búsqueda binaria)
 - Se agota de la posición que sea en el fichero de CB.

3. Insertar

- Insertar registro en fichero de datos
- Insertar nueva entrada en el índice con la clave y posición correspondiente al orden.

6. Cargar y guardar el índice

- Algoritmo nuevo: como indicio de control del programa fichero.
- Al cargar el índice pensarlo con desestructurizado
- Si el campo nuevo de poder insertar sin ser necesario el índice si cambia desestructurizado.

MERGE

- es hacer los índices ordenados.

Merge(lista1, lista2)

Inicializar

```
x ← input(lista1, min, Null)
y ← input(lista2, min, Null)
while (questar-derrotas)
  if (x < y) then write(salida, x) x ← input(lista1, x)
  if (y < x) then write(salida, y) y ← input(lista2, y)
  if (x = y) then write(salida, x) x ← input(lista1, x)
  if (y = y) then write(salida, y) y ← input(lista2, y)
  if (x = y) then
    write(salida, x)
    x ← input(lista1, x)
    y ← input(lista2, y)
```

Input (entrada, valor previo, valor2)

```
if (entrada) then if (valor2 = Null) then valor-derrotas ← valor
  else read (entrada, valor)
  if (valor < valor-previo) then error
  else return valor
```

- Permite pesos y ciclos negativos, no funciona para los detectos.
- Relaxa todos los arcos ~~repetidamente~~ por cada pasada, hay mucho trabajo innecesario, sería mejor relaxar los que han cambiado el valor de sus vecinos.

Relaxar (G, s) OPTIMIZADO (G, s)

Inicializar (G, s) [V]

Insertar (G, s) [1]

while Q ≠ ∅ do

u ← extraer(Q)

for v ∈ Adj[u] do [1]

if d[v] > d[u] + w(u, v) do

d[v] ← d[u] + w(u, v)

π[v] ← u

if v ∉ Q

Insertar(Q, v)

- Elegir en la cola un nodo cuando ha habido cambio.

- En caso de ciclos negativos no funcionaría, se volvería en bucle infinito.

Relaxar

- Se podría comprobar que existen ciclos negativos si un nodo entra N-1 veces en la cola.

Bellman-Ford (G, s)

Inicializar (G, s) // Igual que inicializar de Dijkstra.

for i = 1 to |V[G]| - 1

for (u, v) ∈ A[G] do

Relaxar (u, v)

// Ahora se comprueba si hay datos de este negativo

for (u, v) ∈ A[G]

if d[v] > d[u] + w(u, v) then return ~~return~~ TRUE

return FALSE.

PROB:

- Trabajo inútil

- El orden de relajación es lo que debería tener en cuenta.

★ DESPUÉS DE BELLMAN FORD.

• Si G no tiene ciclos de costo negativo $\forall v \in V[G]$ d[v] = $\delta(s, v)$ y devuelve FALSE.

• Si G tiene ciclos negativos, el algoritmo devuelve TRUE.

• ARBOLES ABARCADORES MÍNIMOS

Un árbol abarcador mínimo (AAM) es un ~~subgrafo~~ subgrafo acíclico que conecta todos los nodos de G y minimiza el coste total de un grafo G conexo y no dirigido.

- CONDICIONES

- Grafo conexo
- Grafo no dirigido.

★ Algoritmo de Prim

Prim (G, r)

for u ∈ V[G] do clave[u] = ∞

clave[r] = 0 // clave = pesos para los vecinos

π[u] = NULL

Q ← V[G] // Inicializar TODOS los nodos en la cola

while Q ≠ ∅

u ← extraer(Q) ①

if v ∈ Adj[u] do ②

if v ∈ Q and clave[v] > w(u, v) then

clave[v] ← w(u, v) ③

π[v] ← u.

★ COSTE

① Seleccion de la cola $O(\log V \cdot V)$

② Se repite 2A veces por ser no dirigido. (todas como arcos)

③ Si hay que actualizar el heap de la cola $O(\log V)$

★ COSTE TOTAL

$O((V+A) \log V)$

• BFS (6,5)

- Visita los nodos adyacentes etiquetando el primer lo mas posible.
- Cada nodo se le asocia un path de unidades.
- Marcamos nodos con precedencias para obtener el recorrido minimo.

- Si G es no conexo, los nodos no accesibles no serán visitados.
- El resultado de BFS sobre un grafo será UN solo.
- Después de BFS(6,s) se cumple $d[u] = \delta(s,u) \forall u \in V[G]$
- ARBOR BF: Contiene todos los vertices accesibles desde s (del grafo donde se aplica BFS) y existe un unico camino de s a todo u, que define $\delta(s,u)$ de s a u. ($\delta(s,u)$).

★ PSEUDOCÓDIGO

```
Inicializa [N]
estadoInicial [N]
while Q != 0 [N]
  u ← Extrae(Q)
  for v ∈ Ady[u] do [IA]
    if estado[v] = N
      estado[v] = V
      π[v] = u
      d[v] = d[u] + 1
      insertar(Q, v)
```

$O(V+A)$

★ METODO DE IMPRESION

```
Print-Path(6,s,u)
if u == print s
else if π[u] = NULL then print "no accesible"
else print-path(6,s,π[u])
print u
```

DFS (6)

- Se danar estado inicial
- Si G es no conexo se recorren todos los nodos igualmente
- Se usa una pila para el camino, reemplazando recursión.

- En DFS no se cuenta la distancia sino el tiempo hasta llegar a un nodo.
- Necesitaremos 3 estados: No Visitado, Visitado y Procesado (Adyacentes visitados)
- A diferencia de BFS no se crea un unico arbol.

★ PSEUDOCÓDIGO

```
Inicializa [N]
for u ∈ V[G] do [IV]
  if estado[u] = N then
    visitar(u)
```

```
Visitar(u)
estado[u] = V
d[u] = t + 1
for v ∈ Ady[u] do
  if estado[v] = N then
    π[v] = u
    visitar(v)
estado[u] = P
δ[u] = t
```

★ COSTE
 $O(V+A)$

★ TERCERA DEL PROGRESO

```
Iu = [d[u], g[u]]
Iv = [d[v], g[v]]
Se cumple: (solo una de las dices)
Iu < Iv // u es padre de v en T
Iv < Iu // v es padre de u en T
Iu ∩ Iv = ∅ // no son parientes
```

FS y DFS se pueden aplicar en cualquier tipo de grafo: - CON O SIN CICLOS - DIRIGIDOS O NO DIRIGIDOS

u descendiente de v \Leftrightarrow en el momento d[u], u es accesible desde v por un camino formado por nodos de estado N.