

3 - Indexación. Ficheros con organización secuencial indexada.

- 3.1 - Organizaciones para acceso directo: ficheros relativos.
- 3.2 - Concepto de índice. Técnicas de indexación de ficheros. Organización indexada.
- 3.3 - Organizaciones con varios índices secundarios: ficheros invertidos.
- 3.4 - Estructuras de árbol y su utilización en las organizaciones indexadas: árboles B y B*.
- 3.5 - Familia de árboles B+ y organización secuencial indexada.

3.1 - Organizaciones para acceso directo: ficheros relativos.

Objetivo: reducir (minimizar) el coste de las operaciones de *búsqueda*, implementando un acceso “*directo*” a los datos

tiene sentido en soportes “direccionables”

Solución: *establecer una conexión entre el valor del dato y su ubicación física*

clave: pequeña porción de información que **identifica** (nivel lógico) un dato

- Propia
- Añadida

es única para cada dato

¿ y si hay valores repetidos ?

fichero relativo ≡ fichero en el que existe una relación “predecible” entre la clave de un dato y su ubicación física.

$R(\text{clave}) \rightarrow \text{ubicación física}$, siendo R una función de cálculo de direcciones

Organización ligada a la clave

ficheros relativos: tipos y operadores básicos

la implementación de la función $R \rightarrow$ diferentes tipos de ficheros relativos :

- “mapeo” directo \rightarrow *ficheros directos*
- búsqueda en directorio \rightarrow *ficheros indexados*
- cálculo de direcciones \rightarrow *ficheros dispersos*

operadores básicos:

- *Asociar* (f , nombre)
- *Disociar* (f)

- *LeerDato* (f , clave, d , éxito)
- *EscribirDato* (f , clave, d , éxito)

- *EliminarDato* (f , clave, éxito)

¿ y para obtener todos los datos ? $\xrightarrow{\text{Op. secuenciales}}$ *PrimerDato*($f,d,exit$) + *SiguienteDato* ($f,d,exit$)

ficheros relativos: *ficheros directos con mapeo absoluto*

ficheros directos:

- “mapeo” directo absoluto: \Rightarrow clave \equiv dirección física

ventajas: • *simplicidad* (\neq cálculo direcciones) y *eficiencia*

- inconvenientes:**
- *dependencia* de la representación física \rightarrow registros “pinchados” (pinned)
 - \rightarrow **dificultad de reestructuración v gestión del espacio de almacenamiento**
 - la clave no es adecuada (no tiene relación con la información del dato)

Pb. semántico: ¿cómo se implementa la existencia de un dato ?

\rightarrow responsabilidad del usuario ?

no se suele usar

ficheros relativos: *ficheros directos con mapeo relativo*

ficheros directos:

- “mapeo” directo relativo: \Rightarrow clave \equiv dirección relativa (NRR)

ventajas: • *bastante simple y eficiente*

inconvenientes: • la clave puede no tener relación con la información del dato (suele ser algo añadido)
• implementación un poco más compleja

la no existencia de un dato \rightarrow { • marca de borrado (costoso) } en la op. de creación, o
{ • dirección no existente } responsab. del usuario

implementación: • contigua \rightarrow eficiente, pero muy poco adecuada (en general)
• “tabla” de traducción { • *directa* (pb. tamaño y gestión)
• *lista encadenada* (versátil, pero menos eficiente)
• *multinivel* (compromiso adecuado eficiencia-eficacia)

ficheros directos: *ejemplos de implementación*

☛ casi todos los gestores incluyen ficheros directos con operadores de “bajo nivel” (∄ relación con la información de los datos) e incluyen los operadores de tratamiento secuencial

en C
int **fseek** (FILE **fichero*, long *direcciónRelativa*, int *origen*)
siendo origen = SEEK_SET (=principio), SEEK_CUR (=actual), o SEEK_END (=final)

long **ftell** (FILE **fichero*) (*devuelve la posición actual, o -1L si hay error*)

int **fsetpos** (FILE **fichero*, const fpos_t **ptrDir*)
Sitúa el “fichero” en la posición almacenada en *ptrDir por la función fgetpos. Devuelve 0 si no error.

int **fgetpos** (FILE **fichero*, fpos_t **ptrDir*)
copia en *ptrDir la posición actual sobre el “fichero”, para uso posterior con fsetpos. Devuelve 0 si no error.

void **rewind** (FILE **fichero*) { *rewind(f) = fseek(f,0L,SEEK_SET)*

3.2 - Concepto de índice. Técnicas de indexación de ficheros. Organización indexada

Objetivo: utilizar como *clave* una *abstracción* de la información del dato

búsqueda en directorio \equiv representar explícitamente la relación entre valores de la clave y la ubicación física del dato

INDICE \approx VECTOR (tabla) de registros

{ • absoluta
• relativa (NRR)

fichero indexado \equiv fichero cuya organización está basada en un índice (o más)

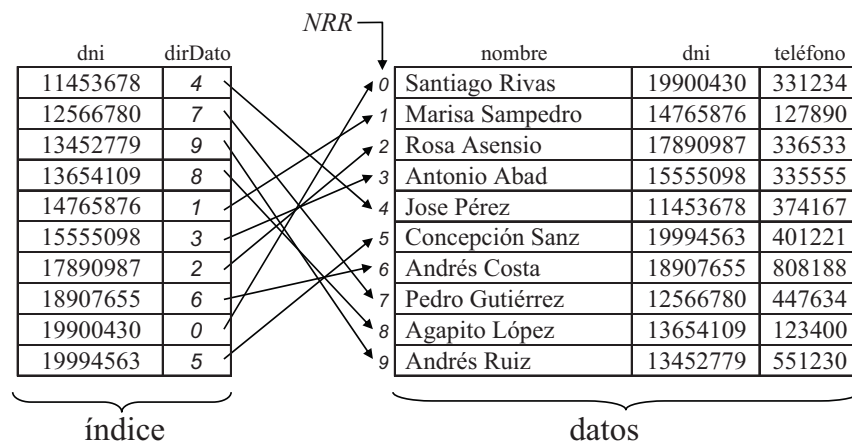
☛ acceso al dato \Rightarrow 2 pasos : **búsqueda en índice + acceso al dato**

☛ suelen implementarse en base a dos ficheros: **f. índice + f. datos**

\rightarrow pueden estar integrados en un único fichero (reg. diferentes)

Concepto de índice: ejemplo

☛ Información de personas. Como clave se usará el DNI (identifica una persona)



Concepto de índice: ventajas/inconvenientes. Operadores básicos.

- ventajas:**
- menor tamaño del índice \Rightarrow mayor eficiencia búsqueda (\uparrow factor de bloque) \rightarrow podría caber en memoria
 - registros del índice de long. fija (normalmente) \Rightarrow acceso + simple y eficiente (p.e. búsqueda binaria)
 - simplifica la utilización de registros de longitud variable (representados como tales)
 - permite acceso directo (eficiente) a los datos utilizando cualquier campo clave, incluso simultáneamente (varios índices)
 - permite mantener una (o varias) ordenación lógica de los datos sin moverlos \rightarrow el mantenimiento del orden es muy eficiente

- inconveniente:**
- operaciones de procesamiento secuencial (afectan a muchos datos) son más lentas

operadores básicos:

- Asociar (f, nombre)
- Disociar (f)

- LeerDato (f, clave, d, exito)
- EscribirDato (f, clave, d, exito)
- EliminarDato (f, clave, exito)

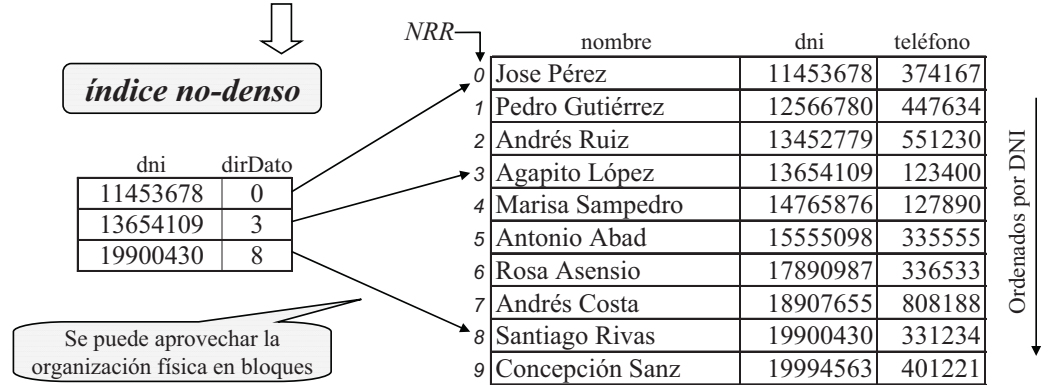
- PrimeraClave (f, clave, exito)
- SiguienteClave (f, clave, exito)

\rightarrow o dato (pero a través de la clave)

Concepto de índice: índice denso y no-denso

una entrada de índice por cada dato \equiv **índice denso**
 \rightarrow el índice puede ser excesivamente grande

Solución: elegir un subconjunto de claves \Rightarrow ordenación (parcial) de los datos por la clave



Concepto de índice: densidad de un índice. Ejemplos

densidad del índice = n° entradas / n° total de datos

el tratamiento de los datos se hace en bloques \Rightarrow en el índice la ref. del bloque

\rightarrow Bloque "ancla" \equiv primer bloque de la subsecuencia

FICHERO

		ordenado	no ordenado
denso	ordenado	posible	IS3
	no ordenado		fichero
no denso	ordenado	VSAM ISAM UFAS	
	no ordenado	posible	

- IS3 \equiv índice secuencial tipo AS400
- ISAM \equiv secuencial indexado IBM
- VSAM \equiv secuencial indexado regular IBM
- UFAS \equiv secuencial indexado BULL

Implementación de una Organización Indexada simple

implementación: operaciones de mantenimiento \Rightarrow **compromiso** $\left\{ \begin{array}{l} \bullet \text{ funcionalidad} \\ \bullet \text{ eficiencia} \\ \bullet \text{ seguridad} \end{array} \right.$

\rightarrow acceso al índice y a los datos

\rightarrow más complejas y menos eficientes (a veces) con índices no-densos

\Rightarrow el coste de la búsqueda (aun binaria) no tiene por qué ser despreciable

Ejemplo: aplicación al diseño de una organización indexada sencilla

• • • •

Concepto de índice: tipos de índices

la extensión de la idea de indexación → diferentes tipos de índices

- índice *primario* ≡ se utiliza la *clave (primaria)* del dato →
- índice *de agrupamiento* ≡ se utiliza un *campo de ordenación* del dato →
- índice *secundario* ≡ se utiliza la *cualquier otro campo* (clave o no) →

Si todavía es muy grande el índice → nuevo índice sobre el índice →

índice *multinivel*

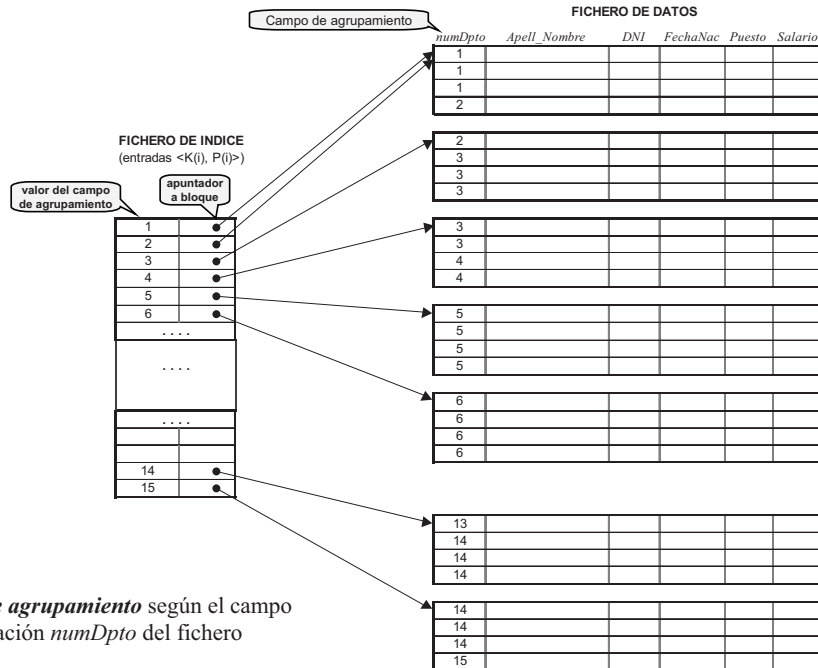
aumenta la eficiencia de la búsqueda, pero complica la gestión

Apell Nombre	DNI	FechaNac	Puesto	Salario	Sexo
Abad, Adriana					
Abarca, Félix					
Acevedo, Irene				
Acosta, Beatriz					
Acosta, Roberto				
Aguilar, Amelia					
Aguilera, Héctor					
Aguilera, Santiago					
Albiol, Sonia				
Alcalá, Enrique					
Alcántara, Silvia					
Amaya, Francisco				
Aranda, María					
Atarés, Rosendo					
Azuara, Roberto				
Yáñez, Francisco					
Yáñez, Rita					
Zamora, Angel				
Zapata, Luis					
Zapatero, Antonio					
Zubiaurre, Abelardo				

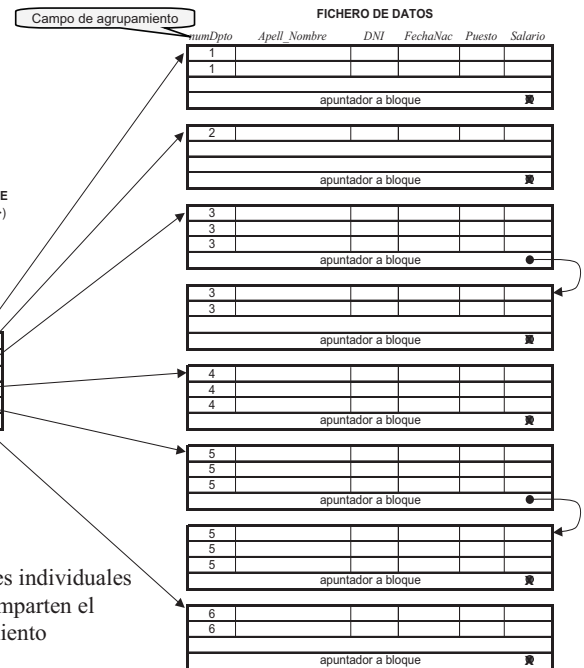
FICHERO DE INDICE (entradas <K(i), P(i)>)

Clave primaria del ancla del bloque	apuntador a bloque
Abad, Adriana	●
Acosta, Beatriz	●
Aguilera, Héctor	●
Alcalá, Enrique	●
Aranda, María	●
.....	
.....	
.....	
.....	
Yáñez, Francisco	●
Zapata, Luis	●
.....	

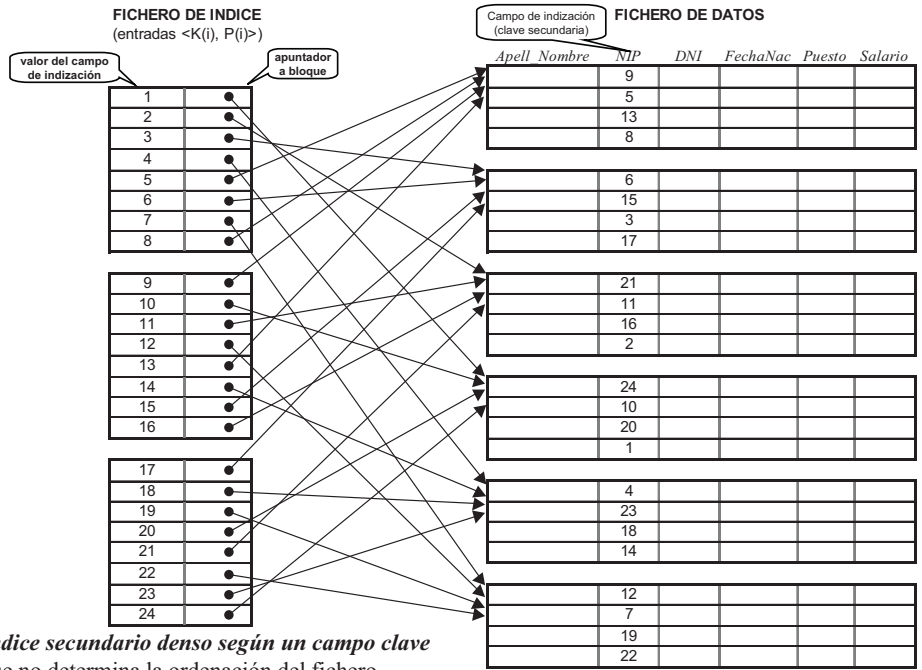
Índice *primario* según el campo de clave de ordenación del fichero (no-denso)



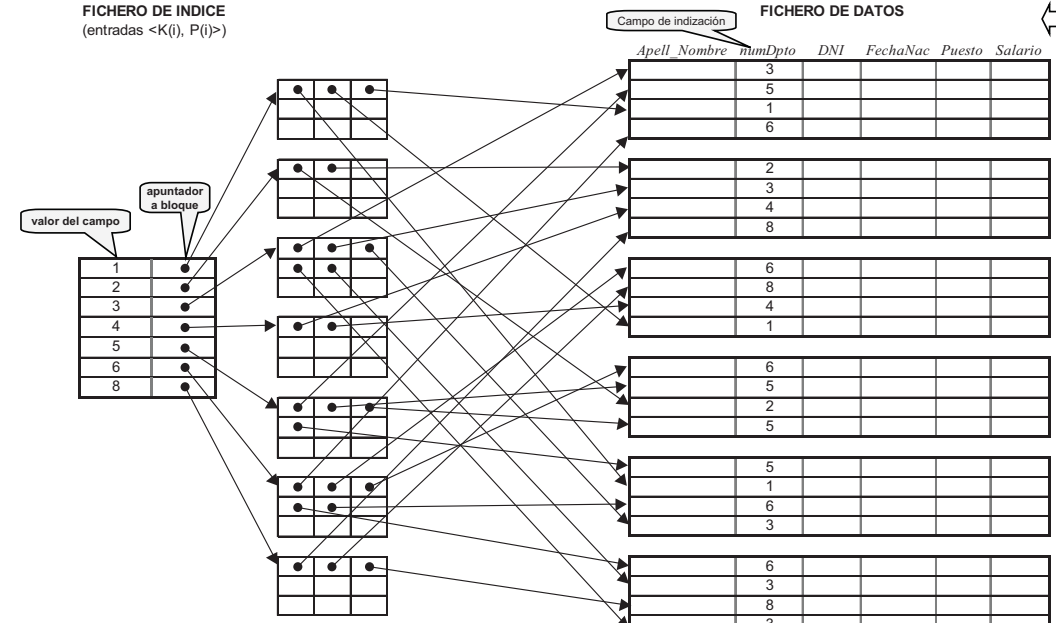
Índice *de agrupamiento* según el campo de ordenación *numDpto* del fichero



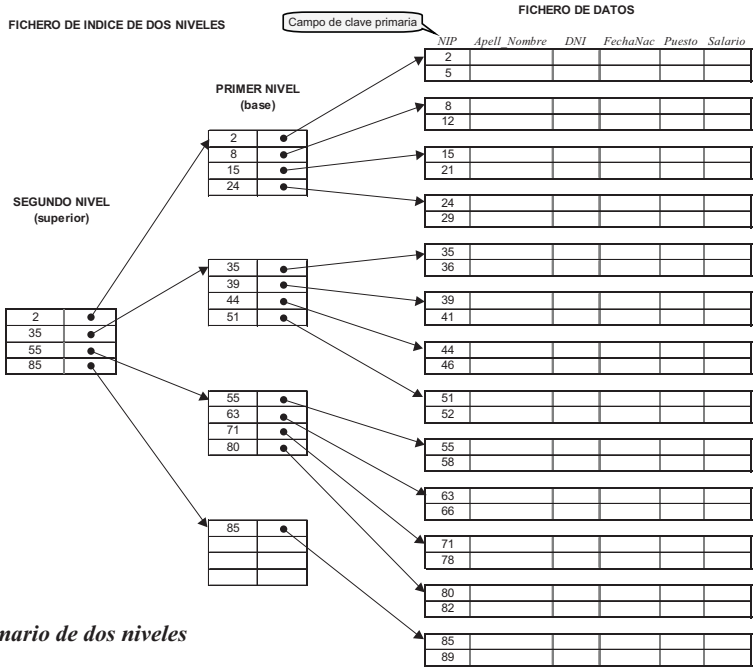
Índice *de agrupamiento* con bloques individuales para cada grupo de registros que comparten el mismo valor del campo de agrupamiento



Indice secundario denso según un campo clave que no determina la ordenación del fichero



Indice secundario según un campo no clave, implementado con un nivel de indirección (entradas del índice de longitud fija y valores únicos)

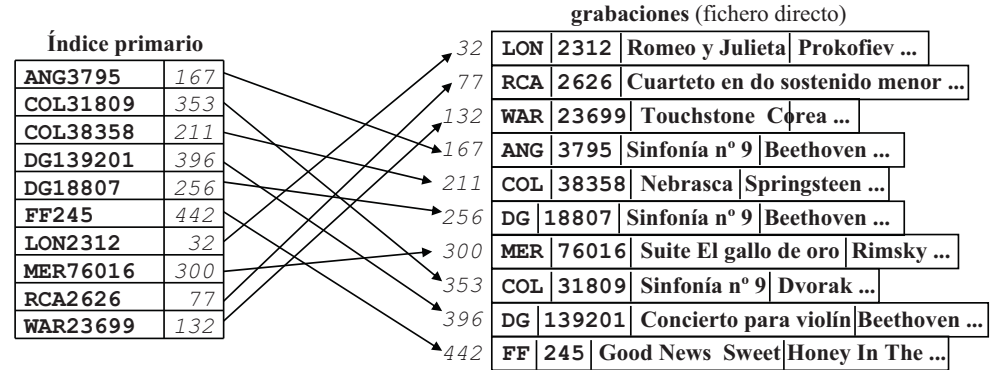


Indice primario de dos niveles

3.3 - Organizaciones con varios índices secundarios: Ficheros invertidos.

Objetivo: acceso eficiente ("directo") a los datos utilizando diversos criterios
no tienen por qué ser información clave

ejemplo: información de grabaciones musicales



Organizaciones con varios índices secundarios

Solución: utilizar varios índices sobre los datos

→ **Densos**, pues sólo puede haber una ordenación física

ejemplo: acceso a la información musical por compositor

Índice secundario	
BEETHOVEN	167
BEETHOVEN	396
BEETHOVEN	256
BEETHOVEN	77
COREA	132
DVORAK	353
PROKOFIEV	32
RIMSKY KORSAKOV	300
SPRINGSTEEN	211
SWEET HONEY IN THE R	442

→ también

Índice secundario	
BEETHOVEN	ANG3795
BEETHOVEN	DG139201
BEETHOVEN	DG18807
BEETHOVEN	RCA2626
COREA	WAR23699
DVORAK	COL31809
PROKOFIEV	LON2312
RIMSKY KORSAKOV	MER75016
SPRINGSTEEN	COL38358
SWEET HONEY IN THE R	FF245

NRR clave primaria

Organizaciones con varios índices secundarios: **Ficheros invertidos.**

Problemas:

- los índices secundarios tienen *valores repetidos* (salvo que sean clave)
- la actualización/inserción/eliminación ⇒ *reactualizar todos los índices*

↙ { • ref. absoluta
• NRR

↗ acceso a través de índice secundario ⇒ *inversión del proceso de obtención de información*

fichero invertido ≡ fichero organizado a partir de índices secundarios basados en la clave primaria

el fichero está invertido con respecto a la clave, para cada uno de los índices secundarios

tipos { • *totalmente invertido* (se utilizan todos los campos)
• *parcialmente invertido* (sólo se utilizan algunos campos)

Ficheros invertidos: ventajas

↗ la información del índice no tiene por qué estar en el dato (*es redundante*)

un fichero *totalmente invertido* puede implementarse *sólo con índices secundarios*

(no es interesante por razones de eficiencia y seguridad)

ventajas:

- acceso eficiente para diferentes criterios (campos)
- se pueden responder a preguntas complejas sin acceder a los datos (*operaciones con listas*)

Ficheros invertidos: **implementación (1)**

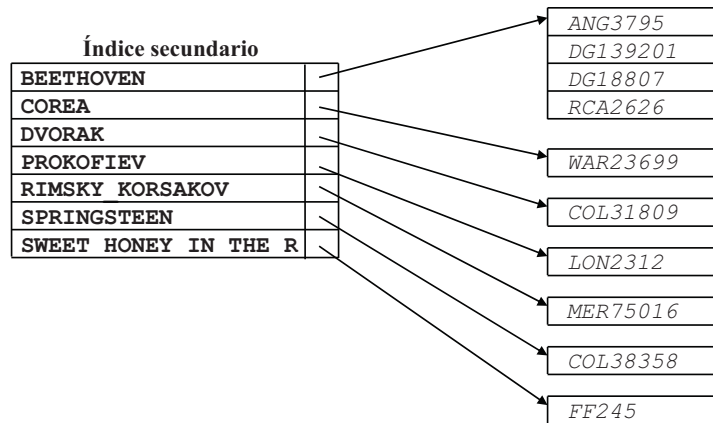
implementación:

• índice con valores no repetidos del campo ⇒ listas de claves primarias

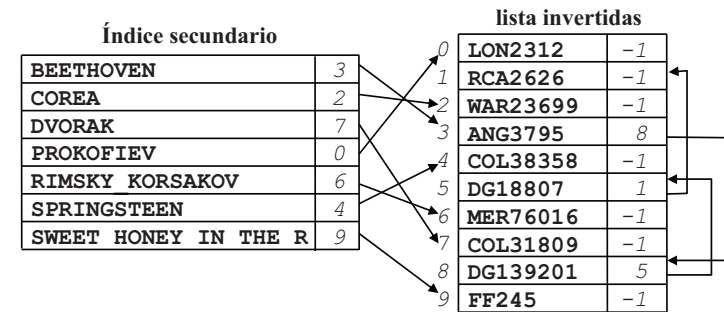
Índice secundario				
BEETHOVEN	ANG3795	DG139201	DG18807	RCA2626
COREA	WAR23699			
DVORAK	COL31809			
PROKOFIEV	LON2312			
RIMSKY KORSAKOV	MER75016			
SPRINGSTEEN	COL38358			
SWEET HONEY IN THE R	FF245			

• solución más interesante: separar las listas del índice

Ficheros invertidos: *implementación* (2)



Ficheros invertidos: *implementación con listas invertidas*



ventajas de las
listas invertidas:

- gestión más eficiente (op. mantenimiento)
- menor ocupación memoria

Ficheros invertidos: *referencia de los datos*

Utilización de la clave primaria como referencia:

- ventajas:
- mayor fiabilidad (*es una redundancia*)
 - no es necesario reorganizar el índice secundario en la eliminación
 - mayor independencia de la ubicación de los datos

- inconveniente:
- mayor coste de la búsqueda (hay que buscar en el índice primario)

3.4 - Estructuras de árbol y su utilización en las organizaciones indexadas: árboles B y B*.

Idea de *búsqueda binaria* en el índice → *organización indexada como árbol binario de búsqueda*

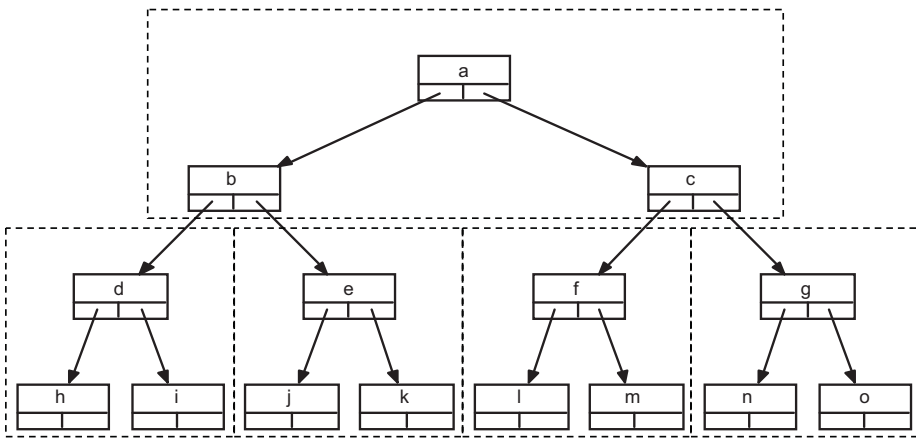


respecto al índice como vector

- ventajas:
- mayor eficiencia búsqueda (\cong)
 - mayor eficiencia en operaciones de mantenimiento

- Pb:
- op. mantenimiento \Rightarrow *desbalanceo* del árbol \Rightarrow pérdida de eficiencia en la búsqueda
 - **árboles 1-balanceados AVL** \Rightarrow gestión bastante simple y eficiente
{altura máx. $1,44 \log_2(N+2)$ y ≈ 1 reorg. local cada 2 inserciones y cada 4 eliminaciones y 1 reorganización no implica más de 5 reasignaciones}
 - no adecuado para tratamiento en bloques (ubicación no predecible del nodo en bloque)
 - *demasiados accesos a bloques* (hasta 1 por nodo)
 - {**árboles binarios (AVL) paginados**
árboles multicamino} \Rightarrow gestión eficiente (balanceo) bastante compleja

Estructuras de árbol: árboles binarios paginados.



ejemplo de árbol binario paginado (3 nodos/bloque)

Concepto de árbol B: definición (original)

solución: **árboles B** (\approx multicamino de orden variable y balanceados)

$m \equiv n^\circ$ mínimo de ítem por nodo

árbol B de orden m (Bayer y McCreight 1972)

- los ítem (datos o claves) de cada nodo están ordenados por clave
- la raíz tiene entre 1 y $2m$ ítem
- el resto de los nodos tiene entre m y $2m$ ítem
- un nodo con k ítem tiene $k+1$ descendientes (excepto las hojas)
 - $\left\{ \begin{array}{l} \text{el } i\text{-ésimo subárbol tiene todos los ítem con clave comprendida} \\ \text{entre las claves } (i-1)\text{-ésima e } i\text{-ésima del nodo considerado (si existen)} \end{array} \right.$
- todos los nodos terminales (hojas) están al mismo nivel (completamente balanceado)

$\leftarrow 1$ nodo por bloque \Rightarrow factor ocupación (f_c) $> 50\%$ ($\approx 70\%$)

Concepto de árbol B: definición (generalizada)

árbol B de orden m (Knuth)

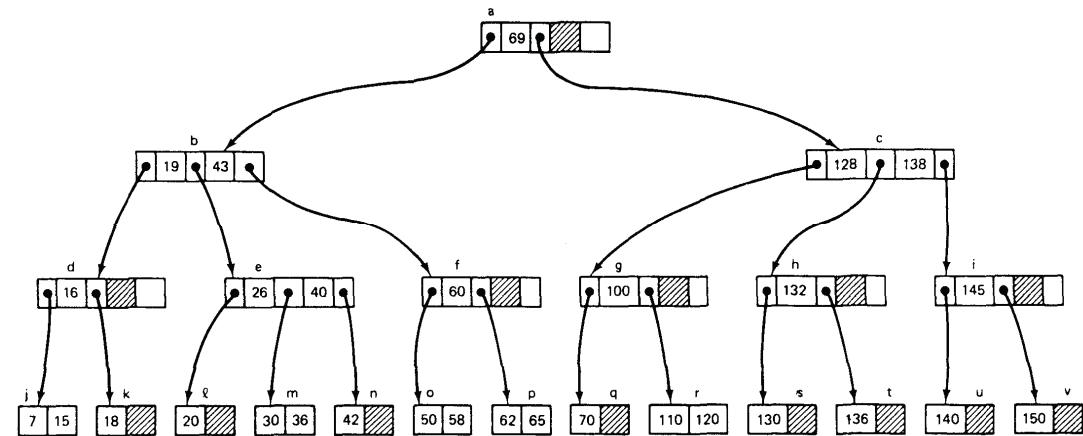
$m \equiv n^\circ$ máximo de descendientes por nodo

- los ítem (datos o claves) de cada nodo están ordenados por clave
- la raíz tiene entre 2 y m descendientes
- el resto de los nodos tiene entre $(m-1) \text{ div } 2 + 1$ y m descendientes (excepto las hojas)
- un nodo con k descendientes tiene $k-1$ ítem (excepto las hojas)
 - $\left\{ \begin{array}{l} \text{el } i\text{-ésimo subárbol tiene todos los ítem con clave comprendida} \\ \text{entre las claves } (i-1)\text{-ésima e } i\text{-ésima del nodo considerado (si existen)} \end{array} \right.$
- todos los nodos terminales (hojas) están al mismo nivel (completamente balanceado)

$= (m + 1) \text{ div } 2$

$\leftarrow 1$ nodo por bloque \Rightarrow factor ocupación (f_c) $> 50\%$ ($\approx 70\%$)

altura ($\approx n^\circ$ de accesos) $\approx 1 + \log_{\lceil p * m \rceil} ((N+1)/2)$ siendo p el factor de ocupación (bloque)



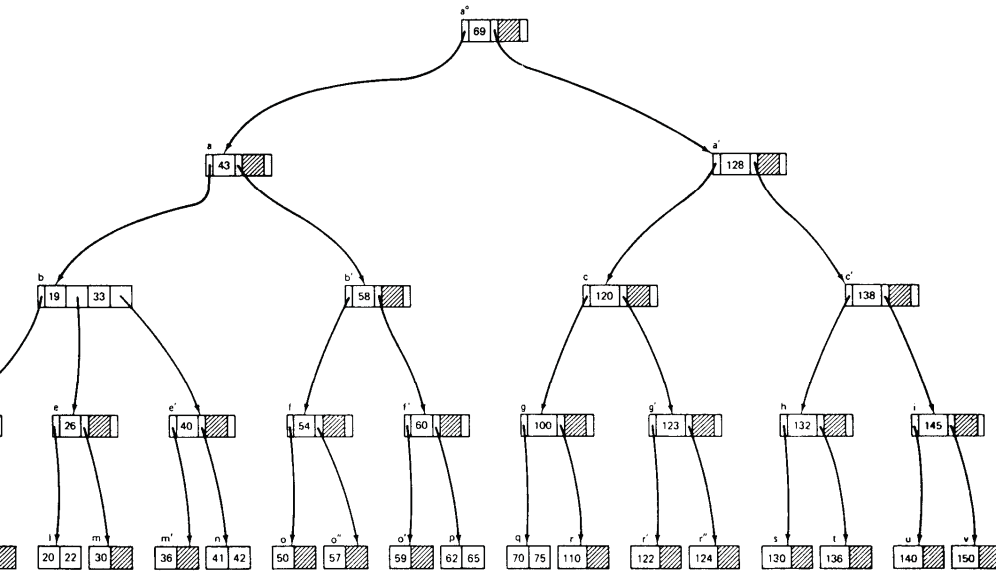
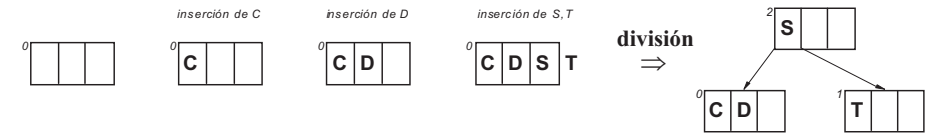
ejemplo de un árbol-B de orden 3

ejemplo de inserción en un árbol B. (1)

implementación (ejemplo 1): INSERCIÓN

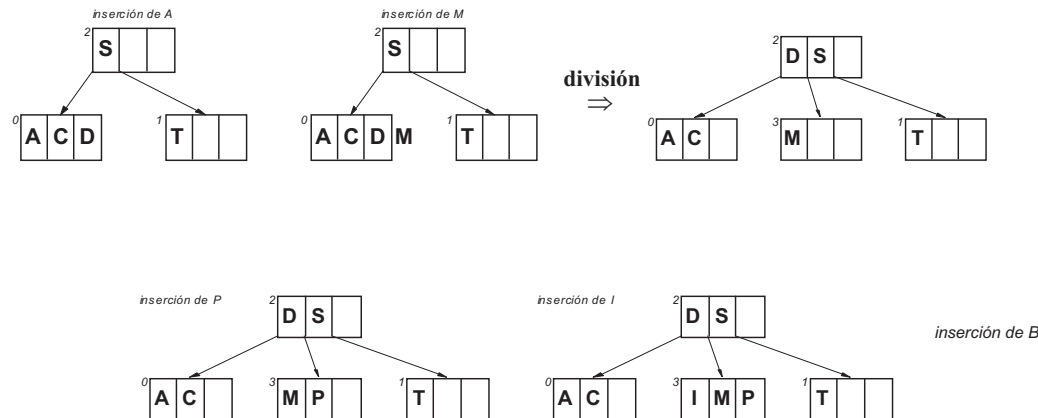
suponiendo un árbol B de orden 4 (knuth), realizar las operaciones de inserción de:
C, D, S, T, A, M, P, I, B, W, N, G, U, R, K, E, H, O, L, J, Y, Q, V, X, Z

$$\left\{ \begin{array}{l} \maxClavesNodo = 3; \quad \minClavesNodo = (4-1) \div 2 = 1; \\ nClavesNodoIzqDiv = 4 \div 2 = 2 \text{ (claves en el nodo de la izquierda al dividir un nodo)} \end{array} \right.$$

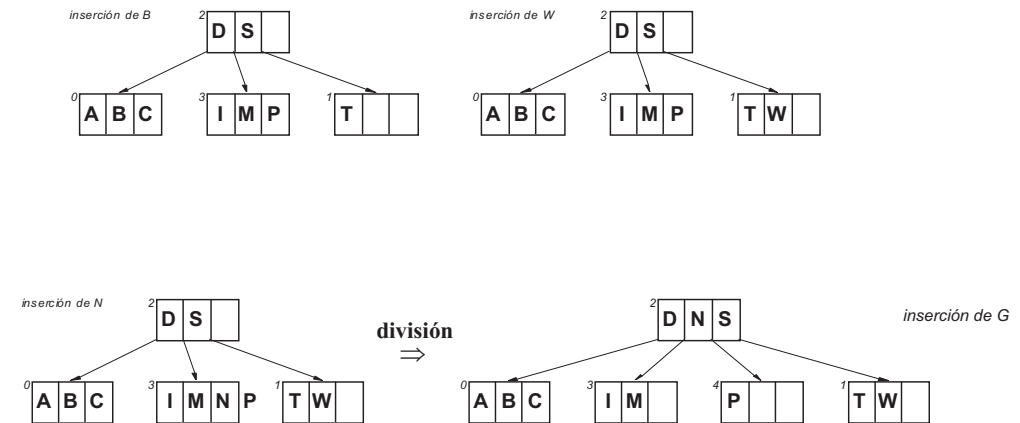


árbol-B tras la inserción de 22, 41, 59, 57, 54, 33, 75, 124, 122, 123 en el árbol de la figura anterior

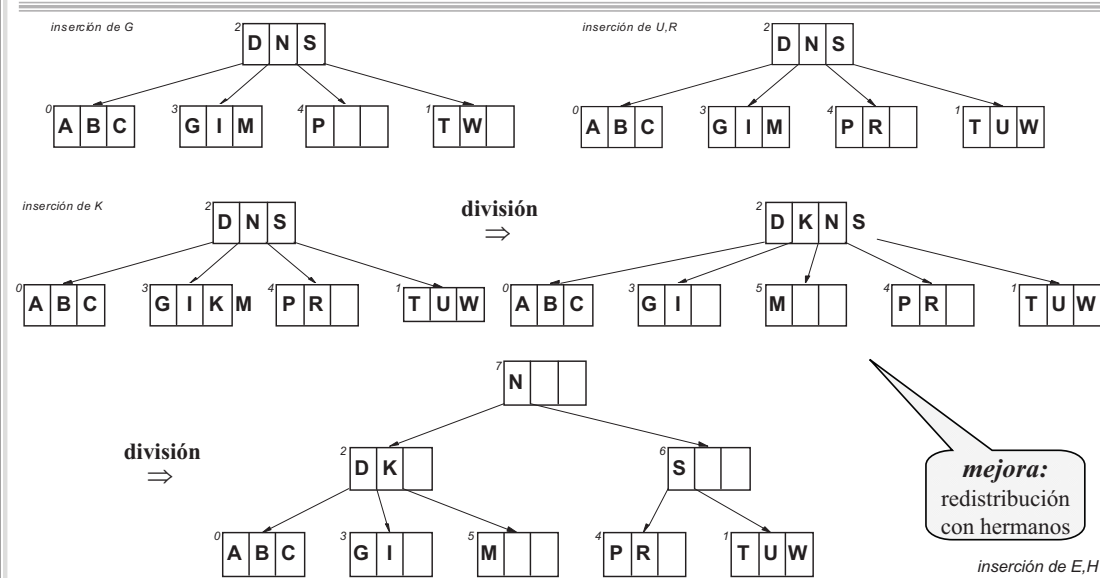
ejemplo de inserción en un árbol B. (2)



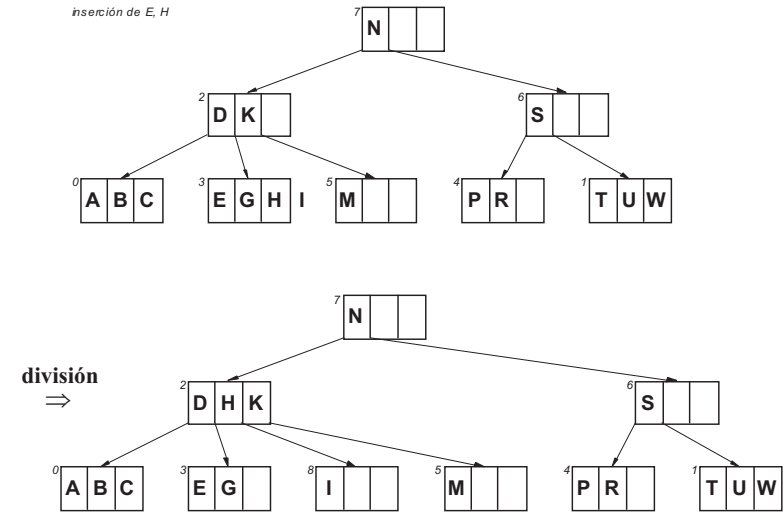
ejemplo de inserción en un árbol B. (3)



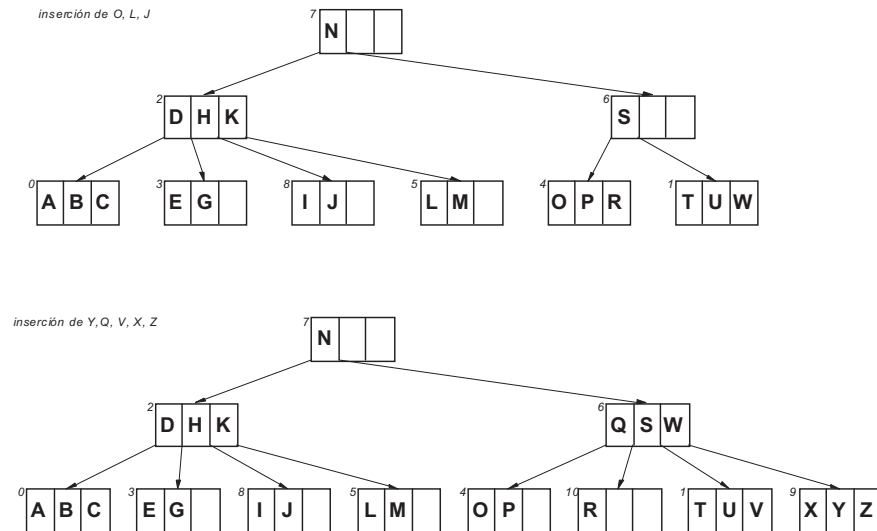
ejemplo de inserción en un árbol B. (4)



ejemplo de inserción en un árbol B. (5)



ejemplo de inserción en un árbol B. (6)



inserción en un árbol B : algoritmo básico

resumen de ideas: pasos en la INSERCIÓN

1) buscar el dato (clave) \Rightarrow descender hasta las hojas

Si *datoEncontrado* entonces *fin* {error, el dato ya existe} Fsi

siempre se comienza la inserción en las hojas

2) insertar en nodo:

Si $n^{\circ}datosNodo < (m-1)$

entonces *guardar en nodo* {colocar en posición correspondiente}

si no Si *hueco en hermano* entonces *redistribuir* {con nodo hermano}

si no {división del nodo} \rightarrow afecta al nodo padre

crear_nuevo_nodo;

redistribuir_con_nuevo_nodo;

insertar_en_nodo_padre(nuevo_dato); {promoción dato "central"}

Fsi

\rightarrow Si es la raíz \Rightarrow nueva raíz

ejemplo de algoritmo de inserción en un árbol B. (1)

```
constantes
ordenArbol = 6;                               {nº máximo de descendientes de un nodo}
nMinClvNodo = (ordenArbol - 1) div 2;         {nº mínimo de claves en un nodo}
nClvDivNodo = (ordenArbol) div 2;           {nº claves de la pag.(nodo) Izqda. al dividir un nodo}
nMaxClvNodo = ordenArbol - 1;               {nº máximo de claves en un nodo}

tipos
refNodo = ^tp_Nodo;                           { implementación en memoria, para simplificar }
tp_Item = registro
    clave: tp_Clave;
    valDato: tp_Dato;
    p: refNodo;
fReg;
tp_Nodo = registro
    contItems: 0..nMaxClvNodo;
    p0: refNodo;
    item: vector[1..nMaxClvNodo] de tp_Item
fReg;
```

ejemplo de algoritmo de inserción en un árbol B. (2)

```
procedimiento insertarItem (ref. a: refNodo; valor u: tp_Item);
variables q: refNodo;
           promocion: booleano;
principio
    promocion := false;
    ponEnArbol(a, u.clave, promocion, u);
    si promocion entonces {crear nueva raíz con item u}
        q := a; nuevoDato(a);
        a^.contItems := 1; a^.p0 := q; a^.item[1] := u
    fsi;
fin;
```

ejemplo de algoritmo de inserción en un árbol B. (3)

```
procedimiento ponEnArbol(a: refNodo; laClave: tp_Clave;
                        ref. promocion: booleano; ref. v: tp_Item);
variables k: entero; encontrado: booleano; q: refNodo;
principio {la clave no está en la página a^; promocion=falso}
    si a = nil
        entonces promocion := true {el item no está en el árbol }
    si no
        buscaEnNodo(a, laClave, k, encontrado);
        si encontrado
            entonces promocion := falso
            si no {el item no está en la página}
                q := sucesor(item[k - 1]);
                ponEnArbol(q, laClave, promocion, v);
                si promocion entonces insertar(a, k, v, promocion) fsi
        fsi
    fsi
fin; {ponEnArbol}
```

ejemplo de algoritmo de inserción en un árbol B. (4)

```
procedimiento insertar (a:refNodo; pos:entero;
                        ref. u:tp_Item; ref. promocion:booleano);
variables b: refNodo;
principio
    si a^.contItems < nMaxClvNodo
        entonces { insertar u en la posición pos} promocion := false;
    si no {la pagina a^ está llena => dividirla y promocionar item }
        nuevoDato(b);
        redistribuir_Items(a,b,u); {los de a^ y u en las páginas a^ y b^}
    fsi
fin; {de insertar}
```

sólo aspectos básicos

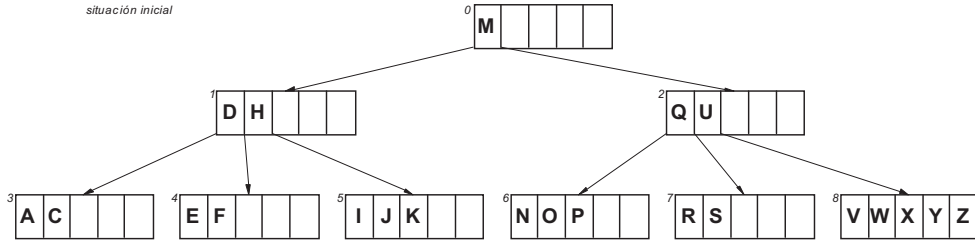
ejemplo de eliminación en un árbol B. (1)

implementación (ejemplo 2): ELIMINACIÓN

suponiendo un árbol B de orden 6 (knuth), realizar las operaciones de eliminación de: J, M, R, A

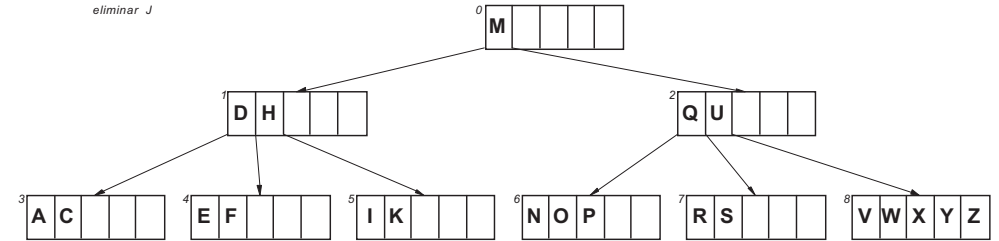
$$\left\{ \begin{array}{l} \text{maxClavesNodo} = 5; \text{minClavesNodo} = (6-1) \text{ div } 2 = 2; \\ \text{nClavesNodoIzqDiv} = 6 \text{ div } 2 = 3 \text{ (claves en el nodo de la izquierda al dividir un nodo)} \end{array} \right.$$

situación inicial



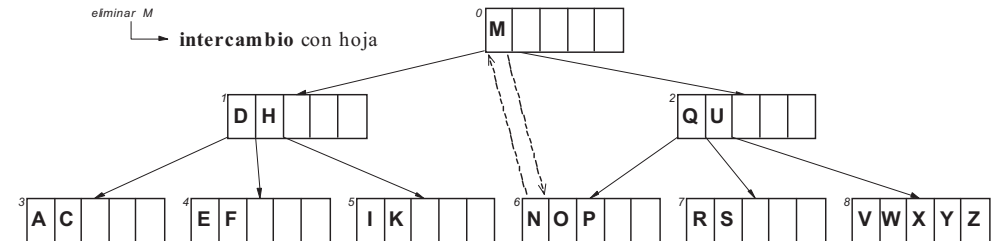
ejemplo de eliminación en un árbol B. (2)

eliminar J

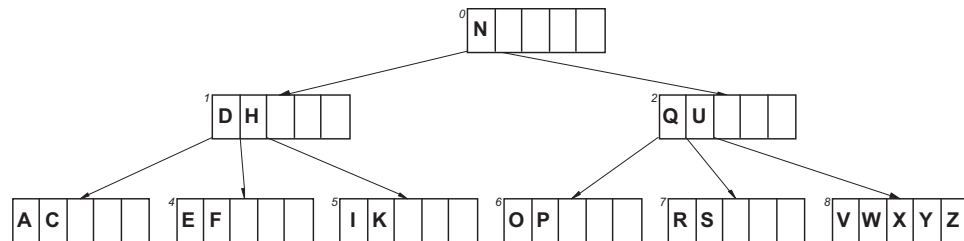


eliminar M

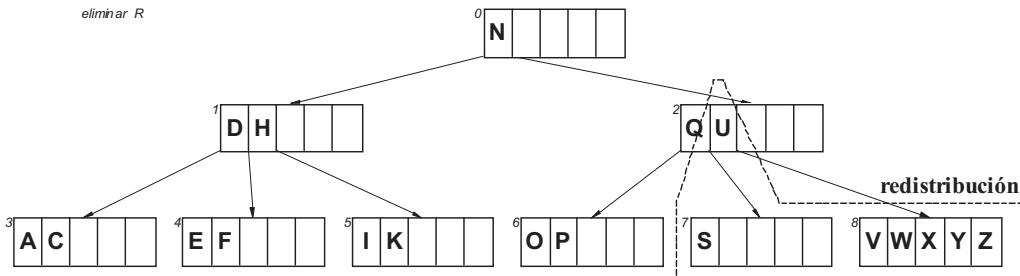
intercambio con hoja



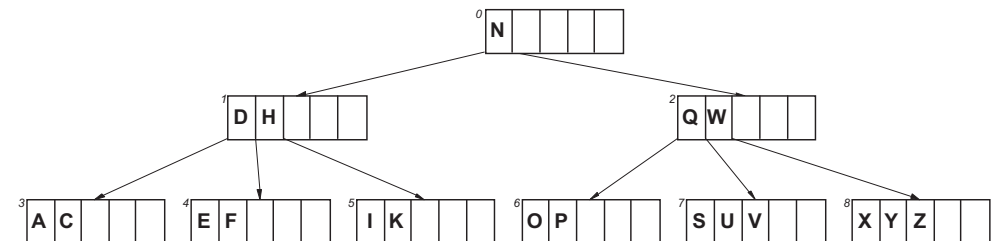
ejemplo de eliminación en un árbol B. (3)



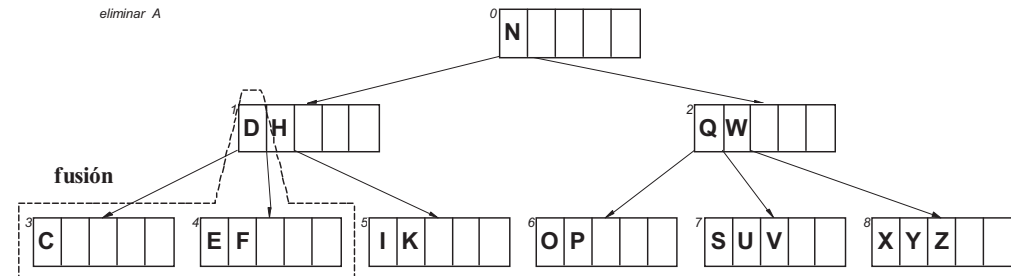
eliminar R



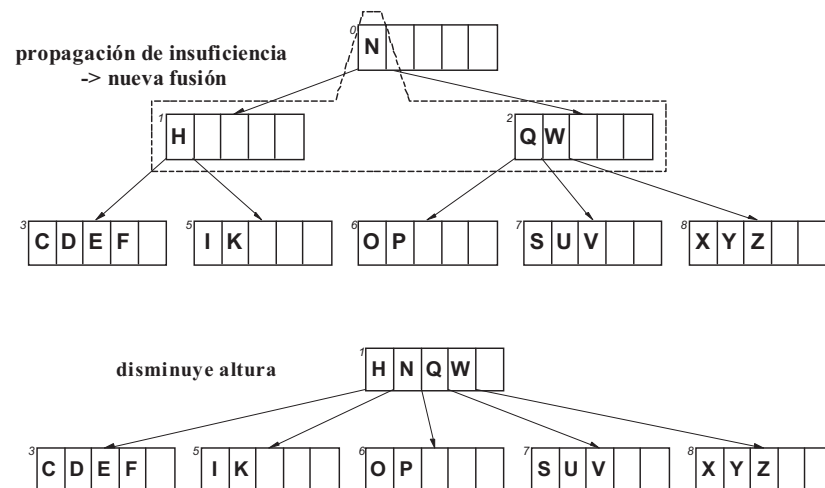
ejemplo de eliminación en un árbol B. (4)



eliminar A



ejemplo de eliminación en un árbol B. (5)



eliminación en un árbol B : algoritmo básico

resumen de ideas: pasos en la ELIMINACIÓN

- 1) buscar el dato (clave) → nodo en que está ubicado
Si nodo = nil entonces fin {error, el dato no existe} Fsi ó Menor_Hijo_Post
- 2) Si nodo no-terminal entonces intercambiar_Con_Mayor_Hijo_Ant Fsi

siempre se comienza la eliminación en las hojas
- 3) eliminar de nodo:
 Si $n^{\circ}\text{datosNodo} > (m-1) \text{ div } 2$
 entonces quitar_del_nodo {recolocar el resto de los datos}

si no Si dato en hermano entonces redistribuir {con nodo hermano}

 si no {fusión con nodo hermano} → afecta al nodo padre
 añadir_datos_hermano; →
 eliminar_nodo_hermano;
 eliminar_de_nodo_padre(nuevo_dato); {promoción de "hueco"}
 Fsi → Si es la raíz ⇒ disminuye un nivel

ejemplo de algoritmo de eliminación en un árbol B. (1)

```

procedimiento eliminarItem (ref. a: refNodo; laClave: tp_Clave);
variables   q: refNodo;
              promocion: booleano;
principio
  promocion := falso;
  borrar(a, laClave, promocion);
  si promocion entonces {se ha eliminado dato de la raíz}
    si a^.contItems = 0 entonces {si no hay datos, disminuir nivel}
      q := a; a := q^.p0; disponer(q)
    fsi
  fsi
fin;
    
```

ejemplo de algoritmo de eliminación en un árbol B. (2)

```

procedimiento borrar (a: refNodo; clv: tp_Clave; ref. promocion: booleano);
variables   posClv: entero; encontrado: booleano;
              rfHijoAnt: refNodo;
principio { de borrar }
  si a = nil entonces promocion := falso {la clave no está en el arbol}
  si no
    buscaEnNodo(a, clv, posClv, encontrado);
    rfHijoAnt := sucesor(item[posClv - 1]);
    si encontrado
      entonces {encontrada, ahora se borra item[posClv] }
        si rfHijoAnt = nil entonces { a es una página terminal }
          eliminar_Item; promocion := a^.contItems < nMinClvNodo;
        si no
          ponMayorHijo(rfHijoAnt, a, posClv, promocion);
          si promocion entonces subocupacion(a, rfHijoAnt, posClv-1, promocion) fsi
        fsi
      si no {no encontrada, buscar en descendientes}
        borrar(clv, rfHijoAnt, promocion);
        si promocion entonces subocupacion(a, rfHijoAnt, posClv-1, promocion) fsi
      fsi
    fsi
  fin; {de borrar}
    
```

árbol B : ejemplo de implementación.

ejercicio: Transformar los algoritmos anteriores (E.D. en memoria) para implementar un árbol B representado sobre un fichero.

ideas:

- ✓ E.D. Soporte → fichero directo (con operadores nuevoDato (nodo) y eliminarDato, . . .)
- ✓ acceso a dato dinámico → cargar dato en memoria (leerDato(f, nodo, pos))
+ guardarlo si se ha modificado (o siempre para simplificar)

mejoras:

- ✓ guardar en memoria varios nodos (“pool” de páginas) con bit de modificación
↳ política de reemplazo . . .
- ✓ eliminación de recursividad → referencias al padre y hermanos
+ pb.variables locales (bloque activación)
- . . .

árbol B : evaluación.

evaluación:

- cada nodo tiene $\begin{cases} m \text{ ref. de bloque} + m-1 \text{ datos} + n^\circ \text{ datos (1 byte)} \\ m \text{ ref. de bloque} + m-1 \text{ claves} + m-1 \text{ ref. dato} + n^\circ \text{ datos (1 byte)} \end{cases}$
- n° medio de accesos (k descendientes por nodo): $A \approx h - (1/(k-1))$

ejercicio: deducir las expresiones de la ocupación y número de accesos para un árbol B cuya raíz tiene z descendientes y el resto de los nodos k descendientes, y aplicarlo a un ejemplo concreto.

mejora de prestaciones ⇒ reducir n° de accesos a nodos

mejora de las prestaciones del árbol B.

algunas ideas:

- mejorar redistribución en inserción y eliminación → pasar a $f_C \approx 85\%$
- cambiar las reglas de fusión y división → árboles B*
- **separar índice de datos** ⇒ ↑ factor de bloque de los nodos
- implementar índices no-densos ⇒ *tratamiento especial de las hojas*
↳ **secuencial indexada** ↳ sin ref. a nodos y ≠ orden
- árboles B de orden variable → *gestión más compleja* (≈ árboles B+ de prefijos simples)
- aprovechar la memoria central → *árboles B-virtuales* (árboles B con buffer en memoria)
- . . .
- otros tipos de árboles
↳ política de gestión páginas ↳
 - mantener niveles altos (raíz + ..)
 - últimos niveles LRU
 - soluciones intermedias LRU ponderando el nivel
- **reestructuración** periódica (copiar) ⇒ $\begin{cases} \uparrow \text{factor de ocupación} \\ \uparrow \text{ordenación de nodos (mejora acceso)} \end{cases}$

concepto de árbol B*.

objetivo: mantener un alto factor de ocupación (f_C) de los nodos

idea básica: $\begin{cases} \bullet \text{ retrasar la división de un nodo hasta tener 2 llenos} \rightarrow \text{redistribución} \\ \bullet \text{ en la división, repartir los items de los 2 nodos entre 3 (los 2 + el nuevo)} \end{cases}$
↳ mínimo (f_C) ≈ 70%

problema: la división del nodo raíz (no tiene hermanos)

↳ $\begin{cases} \bullet \text{ tratamiento especial como árbol B} \\ \bullet \text{ permitir que pueda ser más grande (hasta 4/3)} \rightarrow \text{pb. repres.} \end{cases}$

árbol B* (Knuth) de orden m (n° máximo descendientes)

≈ árbol B, excepto que:
• los nodos tienen entre $2^*(m-1) \text{ div } 3 + 1$ y m descendientes (excepto la raíz y las hojas)
• la raíz tiene tratamiento especial

otras organizaciones indexadas basadas en árboles.

otros tipos de árboles:

tries ≡ árbol *m*-ario (el orden *m* es la base empleada para representar la clave)
cada camino desde la raíz a las hojas representa una clave

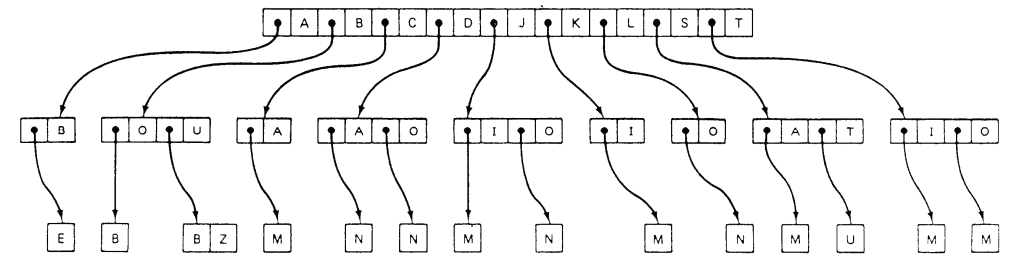


- cada nodo interior consta de *m* ref. de nodo
- cada hoja consta de *m* ref. a datos (índice denso)

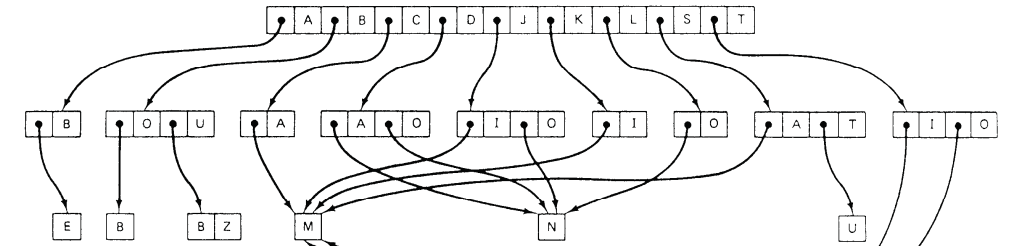
problemas: { claves de muy diferente longitud → { *✓* limitar altura del árbol
✓ representar nodo como lista
 • acomodación a los bloques

H-tree ≡ (dispersión + árbol B)

...



ejemplo de un **trie** para una colección de nombres de tres letras (en inglés) de niños



variante en la que se eliminan algunos nodos hoja



3.5 - Familia de árboles B+ y organización secuencial indexada

objetivo: aprovechar las ventajas de las organizaciones { indexada → búsqueda
 • secuencial → trat. secuencial

idea básica: árbol (≈ árbol B) sobre una secuencia (conjunto de secuencias)

árbol B+

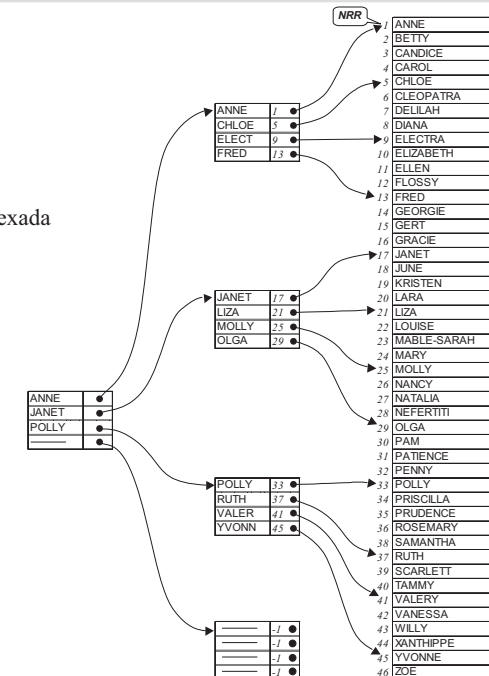
- toda la información (claves ò datos) está en un conjunto encadenado de bloques (hojas) ≡ conjunto secuencia
- el árbol de acceso (≈ árbol B) está formado con copia de las claves (pueden estar repetidas), que actúan como separadores ≡ conjunto índice

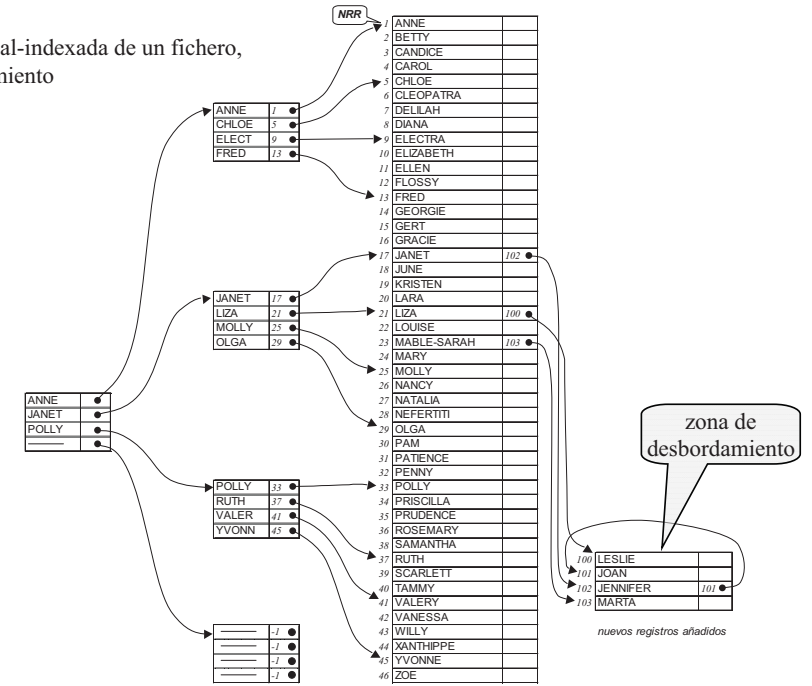
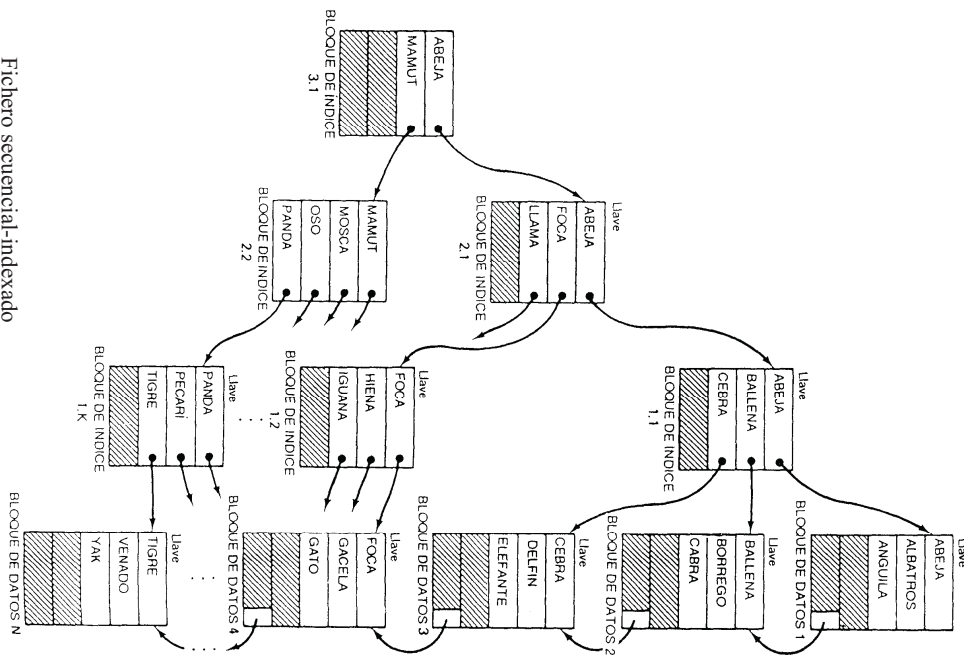


- tratamiento un poco diferente del árbol B (especialmente para las hojas)
- la altura es algo menor que para el árbol B

organización secuencial indexada ≡ organización basada en un árbol B+

Organización secuencial-indexada de un fichero





ejemplo de inserción en un árbol B+ (1)

implementación (ejemplo 1): INSERCIÓN

suponiendo un árbol B+ de orden 3 (knuth) para los nodos interiores, y de orden 4 para las hojas, realizar las operaciones de inserción de: **R, U, A, N, S, E, L, T, V**

- $maxClvNodoInt = 2; minClvNodoInt = (3-1) \div 2 = 1;$
- $nClvNodoIntIzqDiv = 3 \div 2 = 1$ (claves en el nodo de la izquierda al dividir un nodo)
- $maxClvNodoHoj = 3; minClvNodoHoj = (3+1) \div 2 = 2;$
- $nClvNodoHojIzqDiv = (3+2) \div 2 = 2$ (claves en el nodo de la izquierda al dividir un nodo)

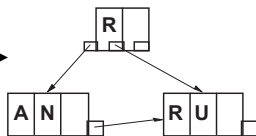
inserción de R, U, A



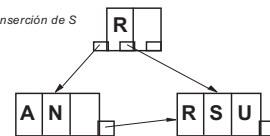
inserción de N



división hoja

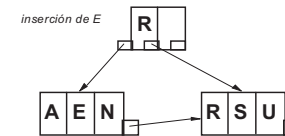


inserción de S

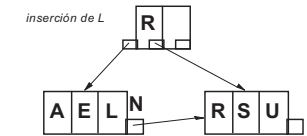


ejemplo de inserción en un árbol B+ (2)

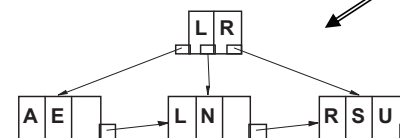
inserción de E



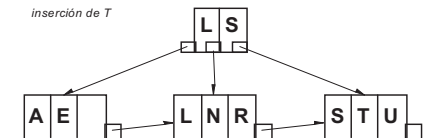
inserción de L



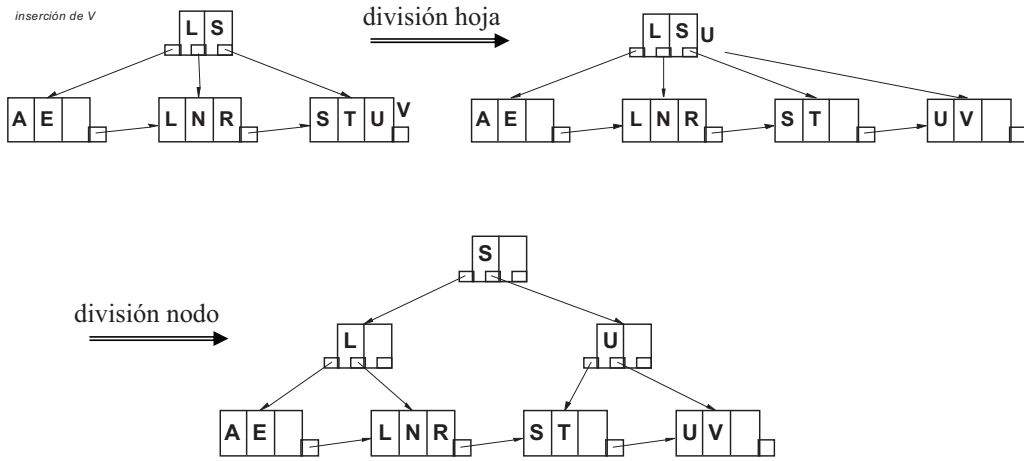
división hoja



inserción de T



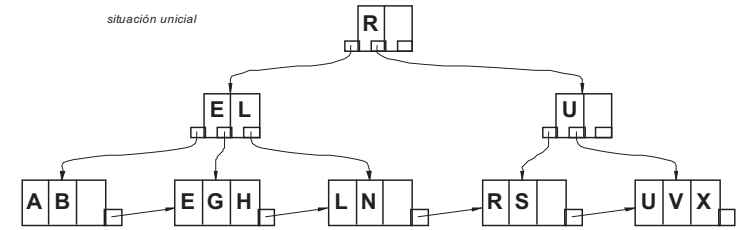
ejemplo de inserción en un árbol B+ (3)



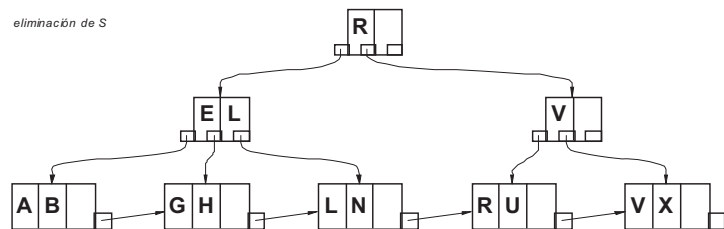
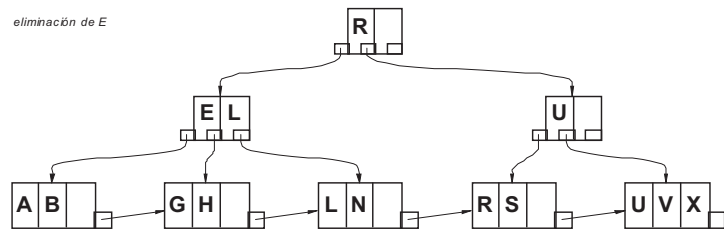
ejemplo de eliminación en un árbol B+ (1)

implementación (ejemplo 2): ELIMINACIÓN

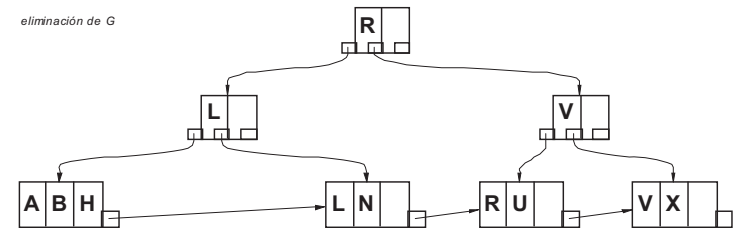
suponiendo un árbol B+ de orden 3 (knuth) para los nodos interiores, y de orden 4 para las hojas, realizar las operaciones de eliminación de: **E, S, G**



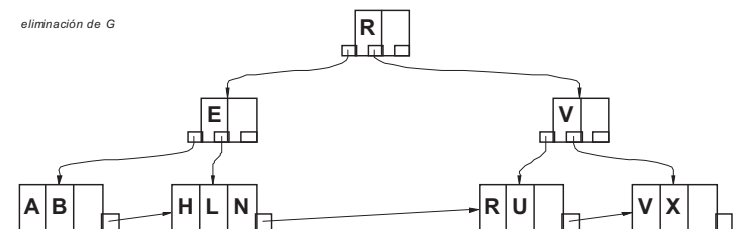
ejemplo de eliminación en un árbol B+ (2)



ejemplo de eliminación en un árbol B+ (3)



o también



árbol B+ : *evaluación.*

evaluación:

- cada nodo hoja : $m-1$ datos + 1 ref. de bloque(hermano) + n° datos y tipo (2 bytes)
 - cada nodo interior : $m-1$ claves + m ref. de bloque(hijos) + n° datos y tipo (2 bytes)
 - n° accesos : = altura árbol \approx árbol B
- ↓
tipo: nodo hoja o interior

ejercicio: deducir las expresiones de la ocupación y número de accesos para un árbol B+ cuya raíz tiene z descendientes y el resto de los nodos k descendientes, y aplicarlo al mismo ejemplo que el desarrollado para el árbol B. Comparar los resultados obtenidos.

→ Similar al árbol B. Se parte del cálculo de las hojas.

árbol B+ : *evaluación.*

- Implementación de operadores de recorrido (primero, siguiente) **más eficiente que en árbol-B**
- en general es mejor el índice separado (fichero datos + fichero índice)
 - mayor eficiencia en organización secuencial (incluso contigua)
 - mayor seguridad (recuperabilidad frente a errores)
 - reconstrucción simple y eficiente del índice a partir del fichero de datos

pb.: las referencias del último nivel son a otro fichero
- reestructuración periódica para mantener (o mejorar) prestaciones
 - fichero índice
 - fichero datos
- implementación de índices secundarios simple

árboles B+ de prefijos simples

mejora de prestaciones \Rightarrow reducir n° de accesos a nodos



representar parte de la clave (prefijo) \Rightarrow aumenta el factor de bloque

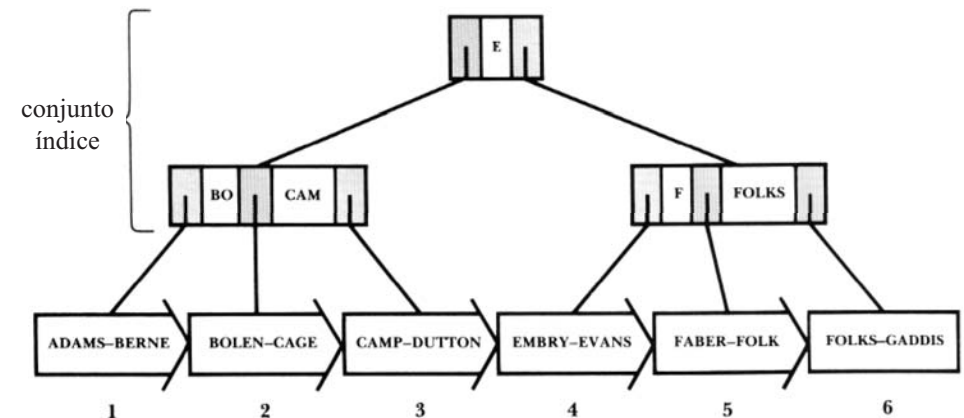
➤ **árboles B+ de prefijos simples** \equiv orden variable (gestión un poco + compleja)

→ implementación de operaciones similar

Implementación de índices con claves repetidas (secundarios, agrupamiento, etc.): relativamente simple

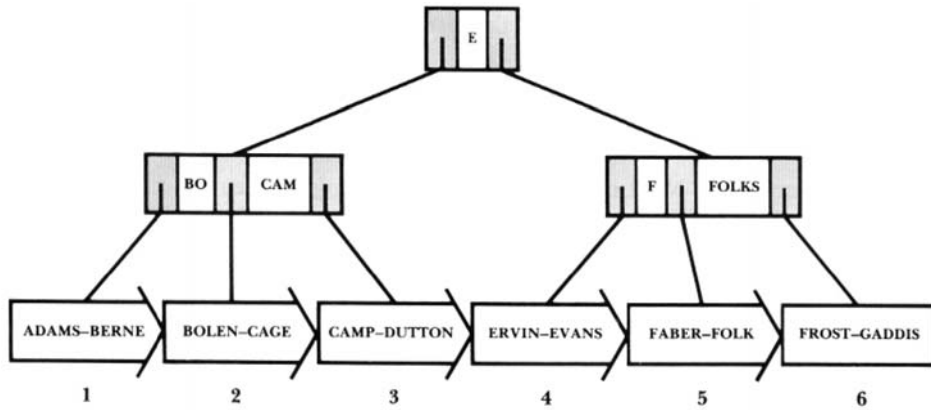
→ diferentes soluciones (compromiso prestaciones-coste)

árboles B+ de prefijos simples: *ejemplo (1)*



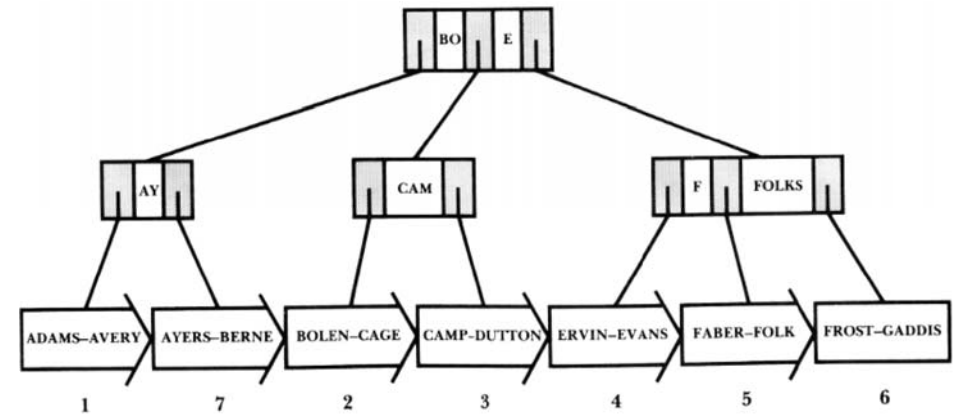
Un conjunto índice en forma de árbol B para el conjunto de secuencias, que forma un **árbol B+ de prefijos simples**

árboles B+ de prefijos simples : *ejemplo* (2)



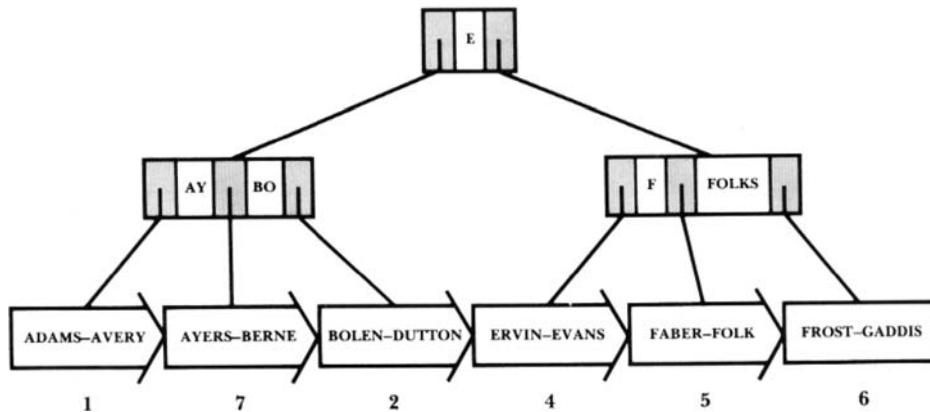
La eliminación de los registros EMBRY y FOLKS del conjunto de secuencias **no altera el conjunto índice**

árboles B+ de prefijos simples : *ejemplo* (3)



Una inserción dentro del bloque 1 provoca una división y la consecuente adición del bloque 7. La adición de un bloque en el conjunto de secuencias requiere un nuevo separador en el conjunto índice (AY). Este separador provoca la división del nodo y la promoción de BO a la raíz.

árboles B+ de prefijos simples : *ejemplo* (4)



Una eliminación en el bloque 2 provoca insuficiencia y la consecuente concatenación de los bloques 2 y 3 (el bloque 3 se puede colocar en la lista de disponibles). Esta eliminación de un bloque conlleva eliminar el separador correspondiente (CAM), lo que provoca una insuficiencia y la consecuente reorganización del conjunto índice.

árboles B+ de prefijos simples : *implementación*

ejemplo de implementación de un nodo:

