4 - Ficheros con organización dispersa (hash).

4.1 - Concepto de dispersión. Algoritmos de dispersión.

Organización indexada ⇒
- Mantener un índice (claves + referencias)
- Búsqueda en índice para acceso al dato

Objetivo: reducir al máximo el coste de las operaciones de *búsqueda*, a partir de la clave: idealmente *un sólo acceso*, *y eliminar el índice*

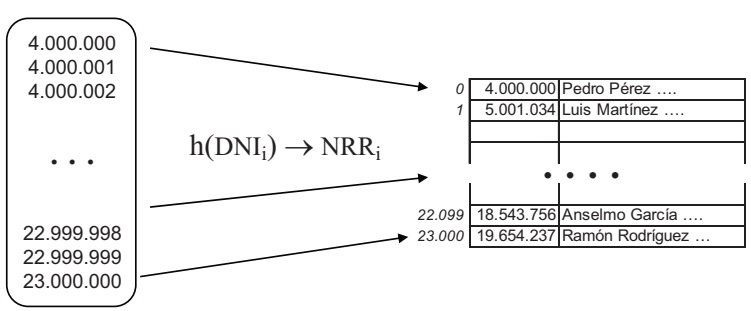
Solución: *diseñar una función (cálculo "simple"),* **h**, *de transformación de la clave en una referencia física* — *aritmético-lógico*

Organización dispersa ⇒
- no se almacenan las claves en un índice
- el acceso al dato a través *función de dispersión* aplicada a la clave

$h(clave_i) \rightarrow$ direcciónDato$_i$

Concepto de dispersión: planteamiento

problema: *dominio de posibles claves >> dominio de claves reales (datos)* — suele ser desconocido



dominio DNI >> *fichero de datos*

h ≡ función de *dispersión*, *hash*, *desmenuzamiento*, etc..

Concepto de dispersión: problemas de implementación

ejemplo de implementación : **h**(DNI) = DNI **mod** 30.001 — *> nº registros esperados*

problemas que aparecen:

- existen entradas en el fichero de datos no utilizadas (*huecos*)

  $\exists$ NRR$_i$ \ $\nexists$ K$_i$ con h(K$_i$) = NRR$_i$

- hay claves para las que la función de dispersión proporciona la misma entrada (sinónimos) → *colisión*

  $\exists$ K$_{i1}$, K$_{i2}$, con K$_{i1} \neq$ K$_{i2}$ \ h(K$_{i1}$) = h(K$_{i1}$) = NRR$_i$

☛ colisiones ⇒ ↑ tiempo y complejidad de la búsqueda

☛ huecos ⇒ ↑ ocupación fichero datos, ↓ probabilidad de colisiones

} *compromiso de diseño*

Concepto de dispersión: distribución de direcciones

el nº de huecos → *factor de carga* (*o densidad de empaquetamiento*) = $f_c = \frac{r}{N}$

- r → *nº de registros almacenados*
- N → *nº direcciones disponibles*

¿ cómo influye el factor de carga en el nº de colisiones ? ⟹ *depende de la función de dispersión*

hipótesis: distribución aleatoria uniforme, y un registro por dirección

función de POISSON : $P(x) = \frac{f_c^{\,x} * e^{-f_c}}{x!}$

$f_c \equiv$ *factor de carga (r/N)*
$x \equiv$ *nº registros asignados a una dirección dada*

→ *Probabilidad de que se asignen* ***x*** *registros a una dirección dada*
≡ proporción de direcciones con x registros asignados

N * P(x) *≡ nº direcciones esperadas con x registros*

ejemplo de cálculo de la distribución de direcciones

ejemplo: 1000 registros dispersados sobre 1000 direcciones ⇒ $f_c = 1$

direcciones sin registros asignados = 1000 * p(0) ≈ 368
direcciones con 1 registro asignado = 1000 * p(1) ≈ 368
direcciones con 2 registros asignados = 1000 * p(2) ≈ 184

direcciones con colisión = 1000 * [p(2)+p(3)+…] = 1000 * [1 - p(0) - p(1)] ≈ 264

(ubicados en otra dirección)

Si en cada dirección puede haber un solo registro ⇒ el resto en **saturación**

nº registros en saturación = N * [1 * p(2) + 2 * p(3) + 3 * p(4) + …]
= 1000 * [1 * p(2) + 2 * p(3) + 3 * p(4) + …] ≈ 368 = 36,8 %

→ *demasiado elevado*

¿ y para otros factores de carga ? → *tabla*

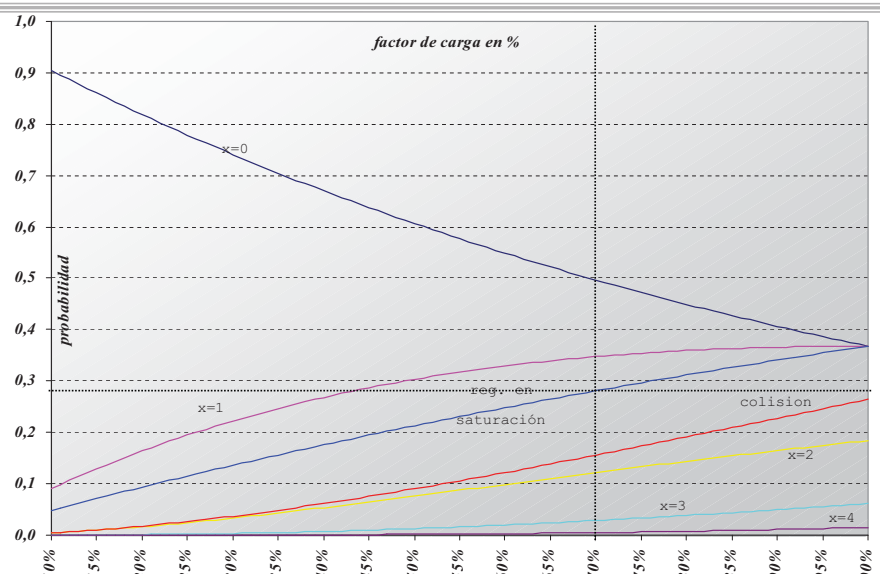
distribución de direcciones en función del factor de carga (1)

Factor de carga	*valores de* ***x (nº registros asignados a una dirección)***									*% direcc. colisión*	*% reg. en saturación*
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**		
0,1	0,90484	0,09048	0,00452	0,00015	0,00000	0,00000	0,00000	0,00000	0,00000	0,47%	4,84%
0,2	0,81873	0,16375	0,01637	0,00109	0,00005	0,00000	0,00000	0,00000	0,00000	1,75%	9,37%
0,3	0,74082	0,22225	0,03334	0,00333	0,00025	0,00002	0,00000	0,00000	0,00000	3,69%	13,61%
0,4	0,67032	0,26813	0,05363	0,00715	0,00072	0,00006	0,00000	0,00000	0,00000	6,16%	17,58%
0,5	0,60653	0,30327	0,07582	0,01264	0,00158	0,00016	0,00001	0,00000	0,00000	9,02%	21,31%
0,6	0,54881	0,32929	0,09879	0,01976	0,00296	0,00036	0,00004	0,00000	0,00000	12,19%	24,80%
0,7	0,49659	0,34761	0,12166	0,02839	0,00497	0,00070	0,00008	0,00001	0,00000	15,58%	28,08%
0,8	0,44933	0,35946	0,14379	0,03834	0,00767	0,00123	0,00016	0,00002	0,00000	19,12%	31,17%
0,9	0,40657	0,36591	0,16466	0,04940	0,01111	0,00200	0,00030	0,00004	0,00000	22,75%	34,06%
1	0,36788	0,36788	0,18394	0,06131	0,01533	0,00307	0,00051	0,00007	0,00001	26,42%	36,79%

☛ a partir del 70% de factor de carga, hay demasiados registros en saturación

→ *compromiso "razonable"*

distribución de direcciones en función del factor de carga (2)



diseño de la función de dispersión : criterios básicos

tipos de dispersión: { interna (en memoria) / externa (en fichero) } { estática (espacio direcc. y función fijos) / dinámica (espacio direcc. y función variables) }

Objetivo diseño: minimizar los huecos y las colisiones, manteniendo la eficiencia

⇩

criterios:

- el *cálculo* debe ser *eficiente* (simple)
- la *distribución* de claves debe ser *uniforme*
- la *distribución* de claves debe ser *aleatoria*
- debería *preservar el orden*

diseño de la función de dispersión : algoritmo básico

pasos: *(habituales)*

1) *transformación* en un valor numérico (considerar la clave como secuencia de bits)
   → ***sin pérdida de información***
2) aplicación de un *cálculo* aritmético sobre el valor obtenido
   → ***obtención de un nº mayor que el espacio de direcciones***
3) *normalización* del resultado
   → ***ajuste del valor al espacio de direcciones***

algoritmos de cálculo típicos:

- *Método del centro de los cuadrados*
  a) se multiplica la clave por sí misma
  b) se extraen los dígitos centrales

```
  4564      ≡ clave
20830096    ≡ clave²
  8300      ≡ dirección
```


algoritmos típicos de dispersión (1)

- *División*
  a) se calcula *clave* ***mod*** *N*

  N ≈ nº direcciones, y es primo o con factores primos

- *Desplazamiento*
  a) se desplazan los dígitos de los extremos hacia el centro en la medida de la longitud de la dirección y se suman

```
 17207359  ≡ clave
  1720
+ 7359
  1152     ≡ dirección
```

- *Plegado*
  a) se "pliegan" los dígitos de los extremos hacia el centro y se suman.

```
 17207359  ≡ clave
   953
+  710
   870    ≡ dirección
```

- *Análisis de dígitos*
  a) se eliminan de la clave los dígitos que más afectan a la no homogeneidad
  b) se aplica cualquiera de las otras técnicas

algoritmos típicos de dispersión (2)

- *Conversión de base*
  a) se supone la clave escrita en una base (normalmente nº primo) y se pasa a base 10
  b) se eliminan los primeros dígitos

clave ≡ 1 7 2 1 4 8 , en base 11 ⇒ ((((1*11+7)*11+2)*11+1)*11+4)*11+8 = 2 6 6 3 7 3

- *Método de Lin*
  a) se expresa la clave en una base ***p***, y el resultado se toma módulo q^m

  p y ***q*** son primos, y ***m*** un entero positivo

- *División polinómica*
  a) se interpreta cada dígito de la clave como un coeficiente de un polinomio
  b) se divide por otro polinomio fijo
  c) se toma como resultado el nº formado por los coeficientes del resto

• • • •

4.2 - Técnicas de resolución de colisiones.

ubicación de los sinónimos:

- en el mismo fichero:
  - • direcciones especiales: *áreas desbordamiento (overflow)*
  - • direcciones del espacio de direccionamiento (mezclados) → *pb. de elección de direcciones*
- en un fichero separado (secuencial ⇒ pb.búsqueda; directo ⇒ pb.ubicación)

a) *direccionamiento abierto*:

⇒ *se calcula una nueva posición* del sinónimo en cualquier lugar (disponible) del fichero mientras que la posición no esté libre

posición relativa en el paso i ≡ $Prel_i = FB(i, clave)$ → *función de búsqueda*

→ "encadenamiento" de posiciones calculadas

direccionamiento abierto : operación de inserción

ejemplo de algoritmo de inserción:

```
Procedimiento Insertar (ref. F: tpFD;  valor R: tpDato);
variables  i, posRel: entero;   dato: tpDato;
Principio
  i := 0;   posRel := H(R.clave);  leer(F, posRel, dato);
  Mientras Que NoLibre(posRel) Hacer
    i := i + 1;
    posRel := FB(i, R.clave);  leer(F, posRel, dato);
  FMQ;
  escribir(F, posRel, R);
Fin.
```

se supone que el dato a insertar no estaba ya

Pb. eliminación: *no se puede romper el encadenamiento de posiciones*

⇒ *cada posición puede estar en 3 estados (libre, ocupada, borrada)*

☛ ***en la creación, a todas las direcciones hay que asignarles el estado de libre***

direccionamiento abierto : búsqueda y eliminación

búsqueda ⇒ explorar hasta encontrar el dato o posición libre

```
Procedimiento Buscar (ref. F: tpFD;  valor clave: tpClave;
                      ref. R: tpDato; ref. éxito: booleano);
variables  i, posRel: entero;   encontrado: booleano;
Principio
  i := 0;   posRel := H(clave);  leer(F, posRel, R);
  encontrado := ocupada(posRel) and (R.clave = laClave);
  Mientras Que NoLibre(posRel) and not encontrado Hacer
    i := i + 1;  posRel := FB(i, R.clave);  leer(F, posRel, R);
    encontrado := ocupada(posRel) and (R.clave = laClave);
  FMQ;
  éxito := encontrado
Fin.
```

☛ *eliminación ⇒ pb. mantener la "cadena" → necesidad de marca de borrado*

☛ Pb.: *la longitud de búsqueda crece mucho a medida que se satura el fichero*

(los registros en saturación ocupan posiciones "destino" de otros registros)

direccionamiento abierto : saturación lineal progresiva

algunas técnicas de direccionamiento abierto son:

a.1 *prueba lineal* (saturación lineal progresiva):

⇒ los registros en saturación se colocan en posiciones obtenidas mediante un desplazamiento lineal de la anterior

posición relativa paso i ≡ $Prel_i = ((i-1)*d + H(clave)) \bmod N$ ∀ i>0

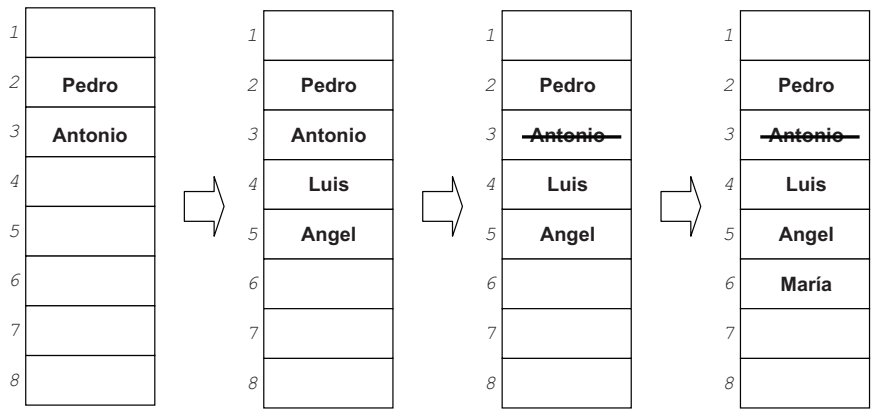
el caso más sencillo corresponde a: d = 1 ⇒ *pasar a la siguiente dirección*

ejemplo: *Sea* H(Pedro) = H(Luis) = H(Angel) = 2; H(Antonio) = 3; H(María) = 4;

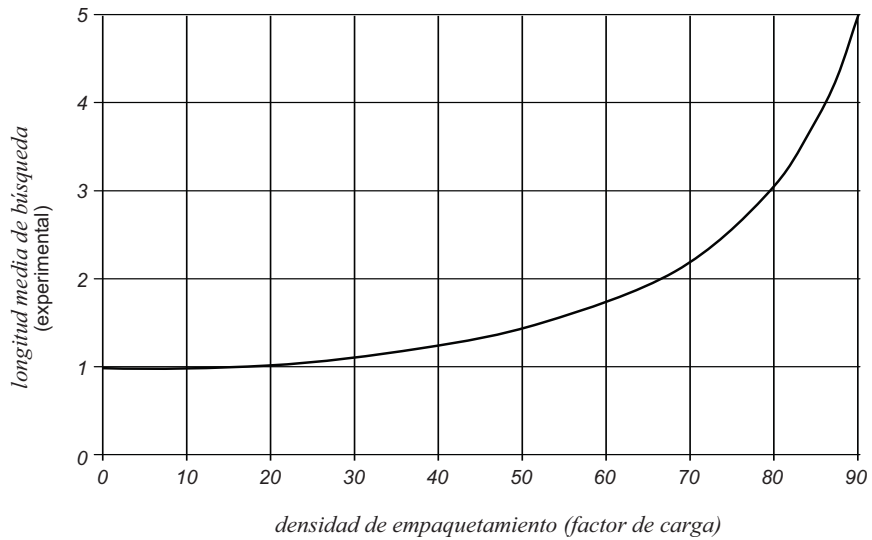
1) *insertar:* Pedro, Antonio, Luis, Angel
2) *eliminar:* Antonio
3) *insertar:* María

saturación lineal progresiva : ejemplo (1)

H(Pedro) = H(Luis) = H(Angel) = 2; H(Antonio) = 3; H(María) = 4;



saturación lineal progresiva: longitud media de búsqueda



direccionamiento abierto : conclusiones

soluciones para reducir la longitud de búsqueda:

- ✓ reorganizaciones locales después de eliminar (costoso)
- ✓ reestructuraciones periódicas del fichero — cuando la longitud media supere un cierto valor, • • •
- ✓ utilización de algoritmos de resolución de colisiones diferentes

otras técnicas de direccionamiento abierto

a.2 ***prueba no-lineal*** *:* ⟶ *pretende "alejar" la ubicación de las zonas donde hay más colisiones*

⇒ los registros en saturación se colocan en posiciones obtenidas mediante un desplazamiento no-lineal de la anterior

$$Prel_i = ((i-1) * d1 + (i-1)^2 * d2 + H(clave)) \bmod N \quad \forall\ i>0$$

el caso más sencillo corresponde a: d1 = 0, *y* d2 = 1

- *i =1* ⇒ `increm. := 1`
- *en cada paso* ⇒ `increm. := increm. + 2`

a.3 ***doble dispersión*** *:* ⟶ *pretende modificar el comportamiento de los sinónimos*

⇒ si hay colisión, se reaplica una nueva función de dispersión:

$$Prel_1 = H(clave)$$
$$Prel_i = (Prel_{i-1} + H2(clave)) \bmod N, \quad \forall\ i>1$$

resolución de colisiones: direccionamiento cerrado

b) *direccionamiento cerrado*: (se usa poco en B.D. (Korth))

⇒ *mantener una lista con los sinónimos* → diversas soluciones de almacenamiento

⇒ *cada posición consta de dato + ref. siguiente + estado*

b.1 *encadenamiento sin reemplazamiento*:

⇒ si la posición está ocupada, se añade al final de la lista de registros asociada a dicha posición

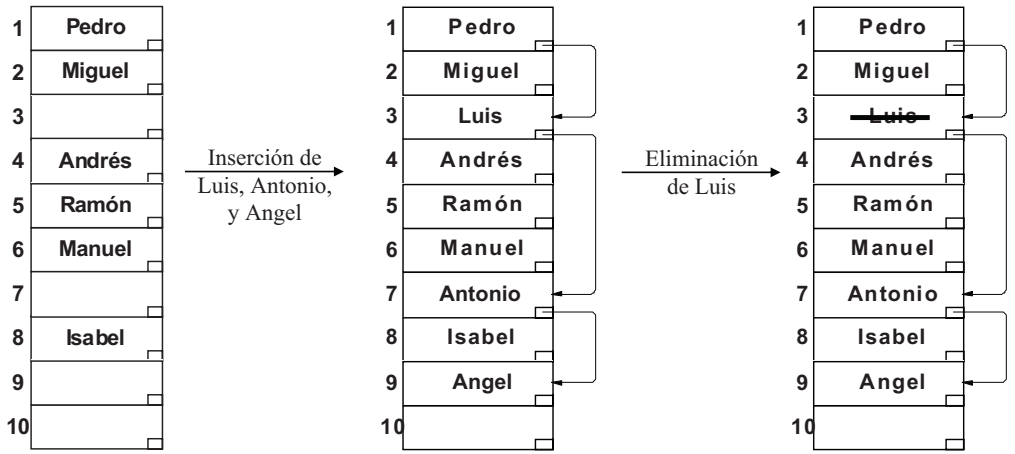
ejemplo: *Sea* H(Pedro) = H(Luis) = H(Angel) = 1; H(Miguel) = 2; H(Antonio) = 3;
H(Andrés) = 4; H(Ramón) = 5; H(Manuel) = 6; H(Isabel) = 8;

1) insertar: Pedro, Miguel, Isabel, Ramón, Andrés, Manuel, Luis , Antonio, Angel

2) eliminar: Luis → *sin problemas*

encadenamiento sin reemplazamiento: ejemplo (1)

H(Pedro) = H(Luis) = H(Angel) = 1; H(Miguel) = 2; H(Antonio) = 3;
H(Andrés) = 4; H(Ramón) = 5; H(Manuel) = 6; H(Isabel) = 8;



encadenamiento con reemplazamiento

eliminación → marca(estado) de borrado, o reestructuración (costoso)

Problema: *la longitud de la lista crece y puede ser larga*

b.2 *encadenamiento con reemplazamiento*:

si la posición está ocupada ⇒
- por un sinónimo: se añade al final de la lista asociada a dicha posición
- por otro dato: se reemplaza, y el dato que había se ubica en otra posición → *diversas soluciones*

ejemplo: *Sea* H(Pedro) = H(Luis) = H(Angel) = H(Andrés) = 1; H(Miguel) = 2;
H(Pablo) = 4; H(Antonio) = 5;

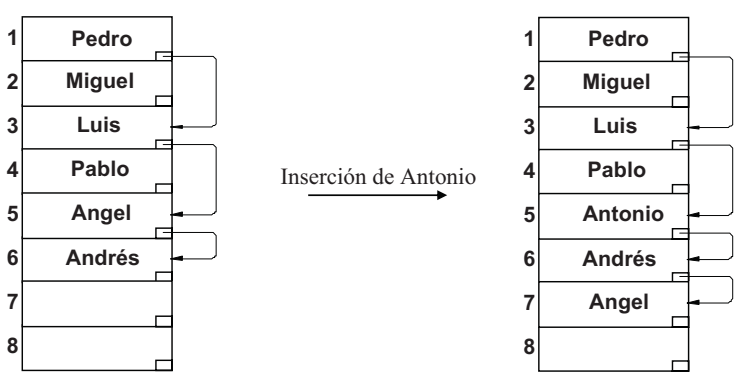
1) insertar: Miguel, Pablo, Pedro, Luis, Angel, Andrés, Antonio

2) eliminar: Luis, Antonio

encadenamiento con reemplazamiento: ejemplo (1a)

H(Pedro) = H(Luis) = H(Angel) = H(Andrés) = 1; H(Miguel) = 2; H(Pablo) = 4; H(Antonio) = 5;

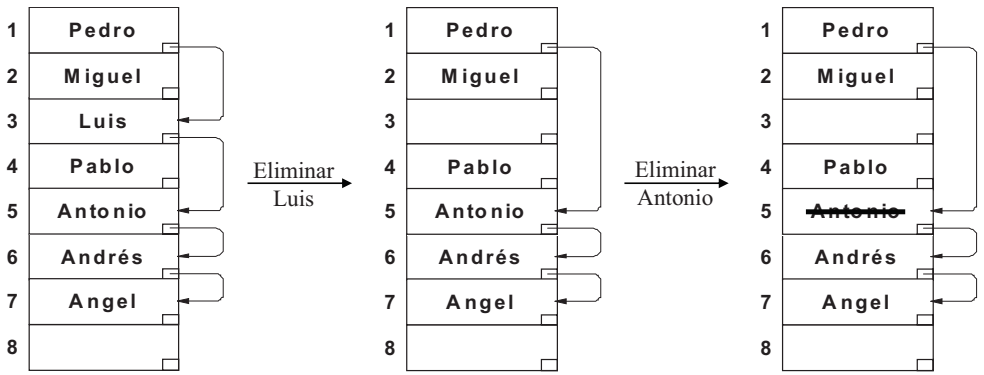
insertar: Miguel, Pablo, Pedro, Luis, Angel, Andrés, Antonio



encadenamiento con reemplazamiento: ejemplo (1b)

H(Pedro) = H(Luis) = H(Angel) = H(Andrés) = 1; H(Miguel) = 2; H(Pablo) = 4; H(Antonio) = 5;

eliminar: Luis, Antonio



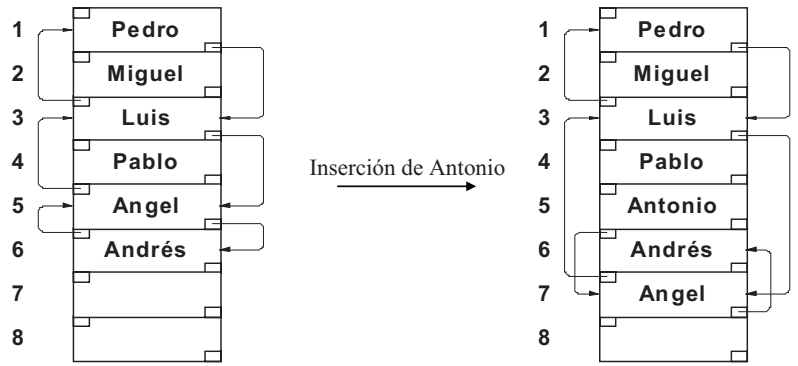
se pueden hacer reestructuraciones parciales, pero se incrementa el nº de accesos

p.e. cambiar Andrés a la posición 5

encadenamiento con reemplazamiento: ejemplo (2a)

H(Pedro) = H(Luis) = H(Angel) = H(Andrés) = 1; H(Miguel) = 2; H(Pablo) = 4; H(Antonio) = 5;

insertar: Miguel, Pablo, Pedro, Luis, Angel, Andrés, Antonio



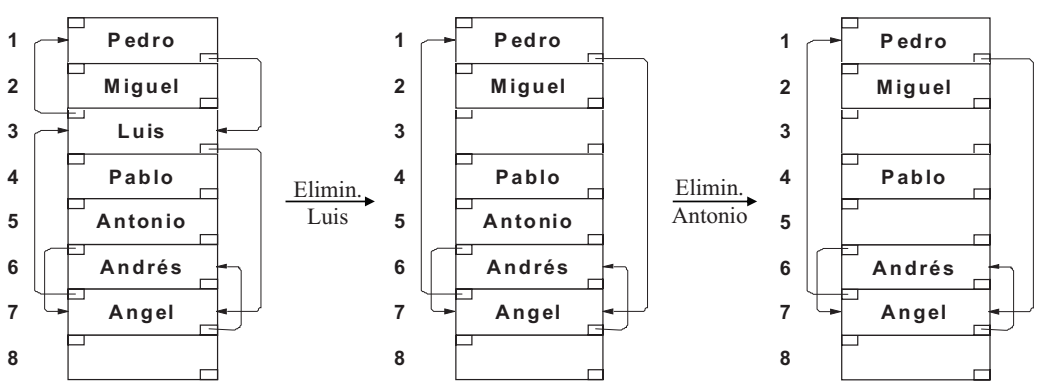
☛ con **listas doblemente encadenadas** disminuye la longitud de búsqueda

(se incrementan los accesos en la inserción)

encadenamiento con reemplazamiento: ejemplo (2b)

H(Pedro) = H(Luis) = H(Angel) = H(Andrés) = 1; H(Miguel) = 2; H(Pablo) = 4; H(Antonio) = 5;

eliminar: Luis, Antonio



☛ con listas doblemente encadenadas no son necesarias "marcas" de borrado

sólo si libre/ocupada

(se incrementan los accesos en la eliminación)

resolución de colisiones: estructuración en bloques

c) *posiciones relativas múltiples:*

⇒ *colocar varios datos en una misma entrada* ≡ cubo, cubeta, cajón, *(página o bloque)*

¿ influye el tamaño de la cubeta en el nº de registros en saturación ?

ejemplo: 750 registros dispersados sobre 1000 direcciones ⇒ $f_c = 0{,}75$

750 registros dispersados sobre 500 cubetas (2 direcciones/cubeta) ⇒ $f_c = 0{,}75$

función de POISSON :

$$P(x) = \frac{(r/N)^{x} * e^{-(r/N)}}{x!}$$

r ≡ nº de registros
N≡ nº de direcc. posibles (cubetas)
x ≡ nº registros asignados a una dirección dada

sí cambia, pues (r/N) es diferente **(0,75** y **1,5)**

distribución de direcciones en función del factor de bloque

factor de carga $f_c = \frac{\text{nº registros reales}}{\text{nº dir.registros tot.}} = \frac{r}{N * f_b} = \frac{\textit{nº medio de registros reales por bloque}}{\textit{nº de registros posibles por bloque}}$ → f_b

$\frac{r}{N} = f_c * f_b$

función de POISSON :

$$P(f_c, f_b, x) = \frac{(f_c * f_b)^x * e^{-(f_c * f_b)}}{x!}$$

el ***nº de registros en saturación disminuye*** al aumentar f_b (caben hasta f_b registros por cubeta)

1) registros en saturación = 1000 * [1 * p(2) + 2 * p(3) + 3 * p(4) + ….] ≈ 222
2) registros en saturación = 500 * [1 * p(3) + 2 * p(4) + 3 * p(5) + ….] ≈ 140

$$\%\ registros\ en\ saturación = \frac{\sum_{x=fb+1}^{\infty}(x - f_b) * P(fc, fb, x)}{fc * fb} * 100$$

distribución de direcciones en función del factor r/N

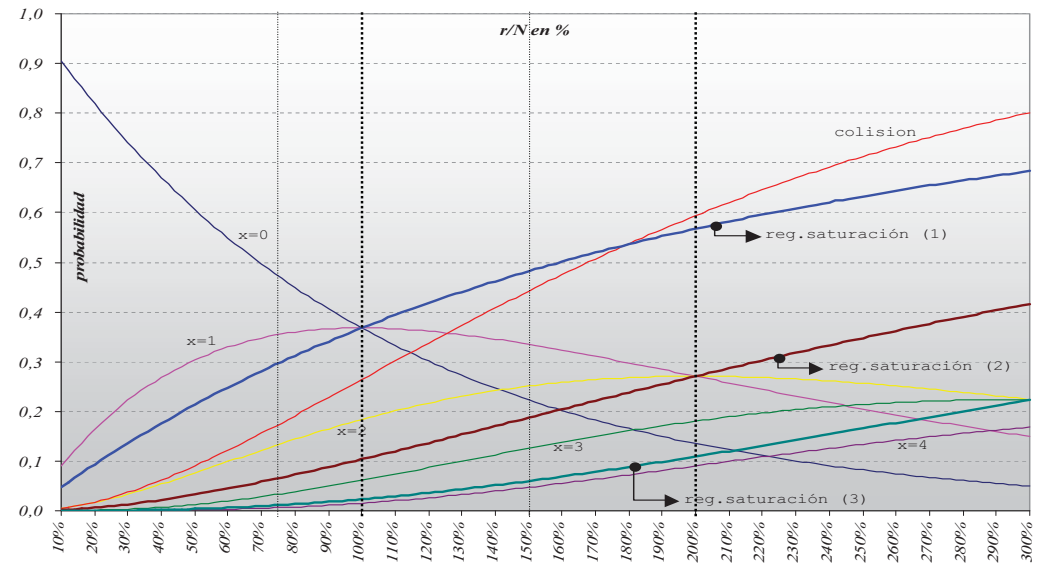
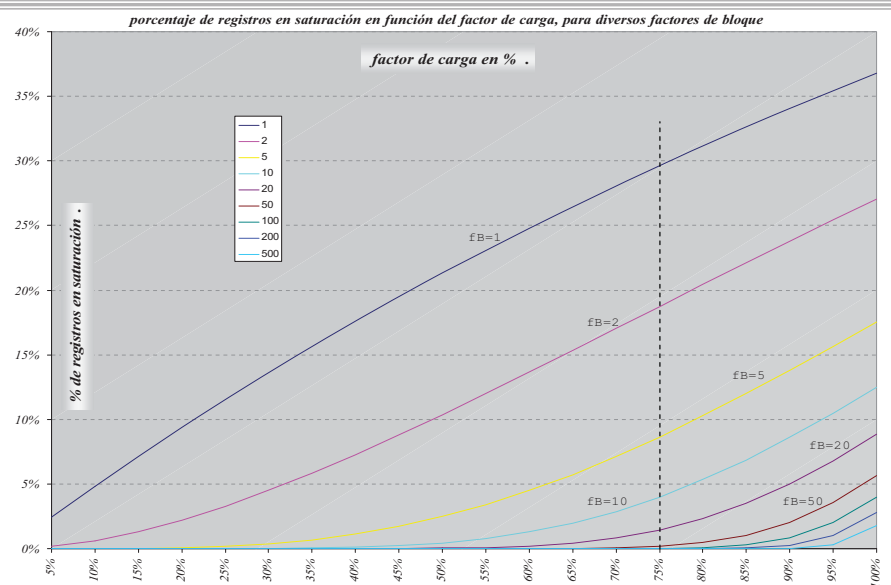


tabla de distribución en función del factor de bloque

% registros en saturación / factor de carga	tamaño de la cubeta (= factor de bloque) 1	2	5	10	20	50	100	200	500
5%	2,46%	0,16%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
10%	4,84%	0,60%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
15%	7,14%	1,29%	0,02%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
20%	9,37%	2,19%	0,07%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
25%	11,52%	3,27%	0,18%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
30%	13,61%	4,49%	0,37%	0,01%	0,00%	0,00%	0,00%	0,00%	0,00%
35%	15,63%	5,83%	0,68%	0,04%	0,00%	0,00%	0,00%	0,00%	0,00%
40%	17,58%	7,27%	1,12%	0,10%	0,00%	0,00%	0,00%	0,00%	0,00%
45%	19,47%	8,78%	1,72%	0,23%	0,01%	0,00%	0,00%	0,00%	0,00%
50%	21,31%	10,36%	2,48%	0,44%	0,03%	0,00%	0,00%	0,00%	0,00%
55%	23,08%	11,99%	3,40%	0,79%	0,08%	0,00%	0,00%	0,00%	0,00%
60%	24,80%	13,65%	4,49%	1,29%	0,20%	0,00%	0,00%	0,00%	0,00%
65%	26,47%	15,33%	5,73%	1,98%	0,42%	0,01%	0,00%	0,00%	0,00%
70%	28,08%	17,03%	7,11%	2,88%	0,81%	0,05%	0,00%	0,00%	0,00%
75%	**29,65%**	**18,73%**	**8,63%**	**3,99%**	**1,42%**	**0,17%**	**0,01%**	**0,00%**	**0,00%**
80%	31,17%	20,43%	10,26%	5,32%	2,30%	0,45%	0,06%	0,00%	0,00%
85%	32,64%	22,11%	11,98%	6,86%	3,48%	1,04%	0,27%	0,03%	0,00%
90%	34,06%	23,79%	13,78%	8,59%	4,99%	2,04%	0,83%	0,24%	0,02%
95%	35,45%	25,44%	15,64%	10,48%	6,80%	3,57%	2,02%	1,01%	0,29%
100%	36,79%	27,07%	17,55%	12,51%	8,88%	5,63%	3,99%	2,82%	1,78%

registros en saturación en función del factor de bloque



nº medio de accesos en función del factor de bloque

Estimación del nº medio de accesos:

factor de Carga (%)

factor de bloque	*10*	*30*	*40*	*50*	*60*	*70*	*80*	*90*	*95*
1	1,06	1,21	1,33	1,5	1,75	2,17	3	5,5	10,5
2	1,01	1,06	1,1	1,18	1,29	1,49	1,9	3,15	5,6
5	1	1	1,01	1,03	1,07	1,14	1,29	1,78	2,7
10	1	1	1	1	1,01	1,04	1,11	1,35	1,8
50	1	1	1	1	1	1	1,01	1,04	1,1

Número medio de accesos para un fichero disperso utilizando saturación progresiva lineal

factor de Carga (%)

factor de bloque	*10*	*20*	*30*	*40*	*50*	*60*	*70*	*80*	*85*	*90*	*95*	*98*	*99*
1	1,04	1,09	1,14	1,19	1,24	1,3	1,35	1,4	1,42	1,45	1,47	1,48	1,49
2	1,006	1,022	1,05	1,07	1,12	1,16	1,21	1,27	1,3	1,33	1,36	1,37	1,38
3	1	1,002	1,018	1,03	1,07	1,1	1,14	1,2	1,23	1,26	1,29	1,31	1,32
5	1	1	1,005	1,011	1,023	1,05	1,09	1,14	1,17	1,2	1,23	1,25	1,26
10	1	1	1	1,001	1,01	1,014	1,04	1,07	1,09	1,12	1,15	1,17	1,18
20	1	1	1	1	1	1	1,002	1,014	1,03	1,05	1,08	1,1	1,11
50	1	1	1	1	1	1	1	1,002	1,01	1,026	1,05	1,07	1,08

Número medio de accesos para un fichero disperso utilizando encadenamiento con reemplazamiento

elección del factor de bloque óptimo

¿ Cual es el tamaño ideal de cubeta ?

→ compromiso entre *nº medio de accesos* y *coste de lectura de la cubeta*

(en el caso extremo de *1 cubeta* ⇒ *1 único acceso*, pero hay que leer todo el fichero)

☛ ***es razonable tomar como tamaño de cubeta un múltiplo del tamaño de bloque del S.O.***

¿ es interesante esta técnica para acceso a tablas en memoria ?

→ ***NO**, porque . . . .*

☛ ***Interesante para construir INDICES DISPERSOS (f_B puede ser grande, >> f_B F_datos)***

4.3 - Técnicas de dispersión dinámica.

dispersión estática ⇒ *espacio de direcciones (nº de cubetas) fijo* ⇒ estimar el nº de datos máximo al principio

→ *Pb. desperdicio de espacio + coste inicialización*
ventaja inicial de f_C bajo

¿ y si el nº de datos supera las previsiones iniciales ? ⟹ *se degradan enormemente las prestaciones, (hasta, incluso, resultar imposible insertar)*

soluciones

- *utilizar zonas desbordamiento especiales ... ("apaños" temporales)*
- *replantear el pb.*
  - ➡ definir una nueva función de dispersión
  - ➡ iniciar el nuevo espacio de direcciones
  - ➡ recolocar todos registros del fichero original

→ ***muy costoso***

técnicas de ***dispersión dinámica*** ⇒ *adaptación "continua" a las diversas situaciones con un coste reducido*

a) dispersión dinámica virtual.

a) *dispersión virtual*

idea básica: duplicar el espacio de direccionamiento al desbordarse una cubeta, reorganizando sólo los datos de dicha cubeta

→ adaptar la función de cálculo de dirección

se utiliza una familia de funciones de cálculo de dirección:

$Prel_{S,i} = clave_i \bmod (2^S * N)$

$N \equiv$ nº inicial de cubetas
$S \equiv$ nº de reorganizaciones (>=0)

ó $H(clave_i)$

→ los registros de la cubeta i se distribuyen entre i, y $i + 2^S * N$

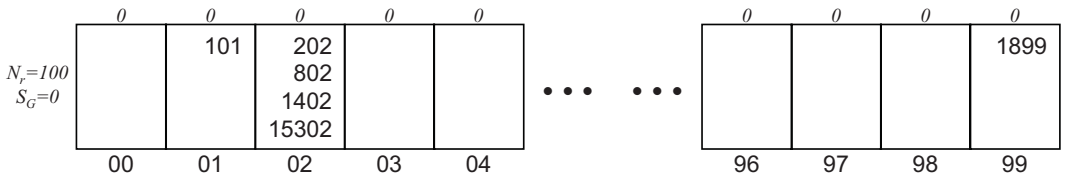
- *cada cubeta tiene asociado el nº de reorganizaciones (divisiones) efectuadas, S_C*
- *el fichero tiene asociado el nº de duplicaciones de espacio S_G*

☛ *inconvenientes:* ***se desaprovecha mucho espacio y se incrementa el nº de accesos***

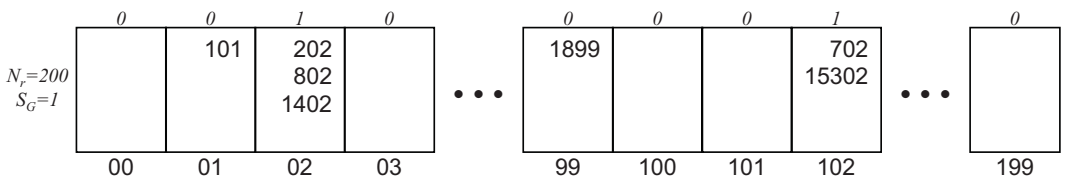
→ *puede ser necesario (conveniente) reestructurar periódicamente*

a) dispersión dinámica virtual: ejemplo de inserción (1)

Ejemplo : Sea N=100 y f_b=4. Insertar los datos (obtenidos al aplicar la función de dispersión) siguientes: 802, 1899, 15302, 1402, 101, 202; 702; 24602, 1102, 8902; 402

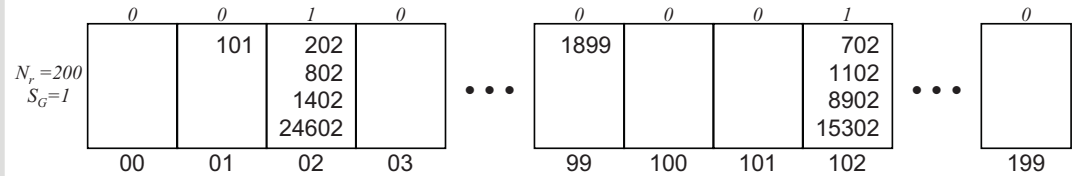


Inserción de 702 ⇒ duplicar el espacio

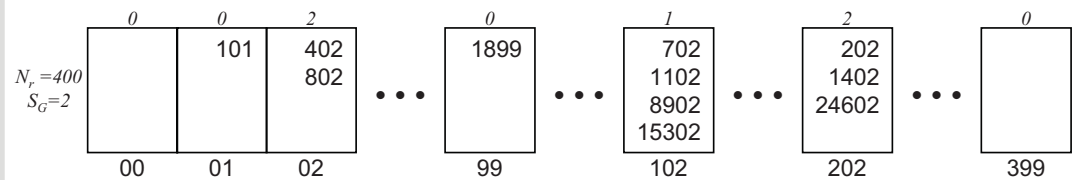


a) dispersión dinámica virtual: ejemplo de inserción (2)

Inserción de 24602, 1102, 8902 ⇒ sin problemas



Inserción de 402 ⇒ duplicar el espacio



ejercicio buscar 702, 1602, 202
insertar 302

a) dispersión dinámica virtual: algoritmo de búsqueda

ejemplo de implementación (muy simplificado)

```
tpCubeta = registro
             nDiv:   entero; {nº de divisiones}
             numDat: entero; {nº de datos}
             valDat: vector [1.. factorBloque] de tpDato
           fRegistro
```

- $N \equiv$ *nº de cubetas inicial*
- $S_G \equiv$ *nº de duplicaciones de espacio de direcciones*

Búsqueda

```
Procedimiento Buscar (clave: tpClave; ref. dato: tpDato; ref. encontrado: booleano);
Variables  i, j: entero;  {NRR de las cubetas a explorar}
           buff: tpCubeta;
Principio
  i := H(clave) mod N;  LeerCubeta(i, buff);
  dim := N;   nivel := 0;
  Mientras Que buff.nDiv <> nivel hacer {explorar las divisiones de cubetas}
    dim := 2 * dim;   j := H(clave) mod dim;     nivel := nivel + 1;
    Si j <> i entonces i := j; LeerCubeta(i, buff) FSi
  FMQ;
  BuscarEnCubeta (buff, clave, dato, encontrado)
Fin
```

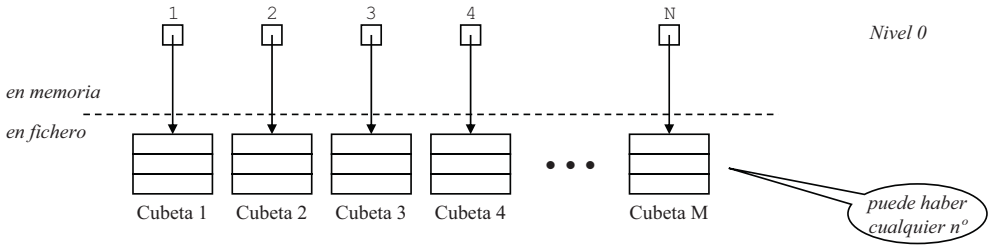

a) dispersión dinámica virtual: algoritmo de inserción

Inserción (sólo idea básica)

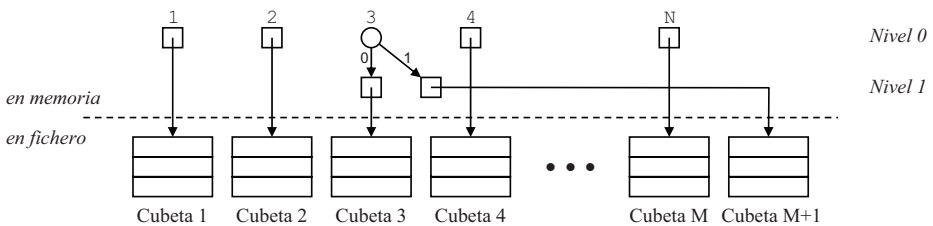
```
Procedimiento Insertar (clave: tpClave; ref. dato: tpDato; ref. encontrado: booleano);
Variables  i, j: entero;  {NRR de las cubetas a explorar}
           buff, nwCub: tpCubeta;
Principio
  i := H(clave) mod N;  LeerCubeta(i, buff);
  dim := N;   nivel := 0;
  Mientras Que buff.nDiv <> nivel hacer {explorar las divisiones de cubetas}
    dim := 2 * dim;   j := H(clave) mod dim;     nivel := nivel + 1;
    Si j <> i entonces i := j; LeerCubeta(i, buff) FSi
  FMQ;
  BuscarEnCubeta (buff, clave, dato, encontrado);
  Si not encontrado entonces
    Si buff.numDat = factorBloque entonces {la cubeta está llena → división}
      Si buff.nDiv = SG entonces DuplicarEspacioDirecc; SG := SG + 1 Fsi;
      buff.nDiv := buff.nDiv + 1;  nwCub.nDiv := buff.nDiv;  nwCub.numDat := 0;
      {distribuir el nuevo dato y los datos de cubeta i del siguiente modo:
       si (H(clave) mod 2*dim) = i entonces dato → buff si no dato → nwCub Fsi}
      EscribirCubeta(i, buff); EscribirCubeta(i+dim, nwCub);
    si no {añadir el dato al buffer y} EscribirCubeta(i, buff); Fsi
  si no ERROR {el dato ya existe} FSi
Fin
```

¡¡Se está suponiendo posible la distribución!!
Si no es así, hay que repetir el proceso

b) dispersión dinámica BT : introducción



Si el dato a insertar no cabe en la cubeta 3 ⇒ nueva cubeta, redistribución de datos, y crecimiento del árbol



b) dispersión dinámica BT : concepto

b) *dispersión dinámica (árbol trie binario BT)*

idea:
- mantener un "directorio" (N entradas en memoria) con referencias a las cubetas
- la división de cubetas genera un árbol binario de referencias asociado a la entrada

*el árbol se gestiona como un **trie** basado en una función, B, de binarización de la clave*

`B(clave) = b1b2b3•••bn` *secuencia digitos binarios*

o de H(clave)

✓ la búsqueda consiste en el recorrido del árbol utilizando la función B, hasta llegar a una hoja

✓ si el registro a insertar no cabe en la cubeta asociada
- *se hace crecer un nivel la rama del árbol (⇒ nueva hoja)*
- *se asocia una nueva cubeta a la nueva hoja*
- *se redistribuyen los registros utilizando un bit más de la función*

b) dispersión dinámica BT: ejemplo de inserción (1)

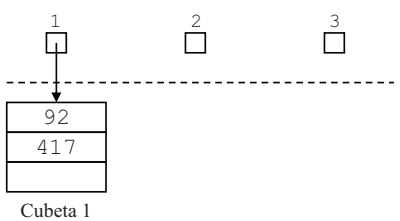
Ejemplo: Suponiendo que $f_b = 3$, N = 3, y la tabla:

clave	*H(clv)*	*B(clave)*
23	1	101010
35	2	010100
67	2	001110
92	1	110110
312	2	011110
395	2	110110
417	1	001110
798	2	000110
1243	3	101101

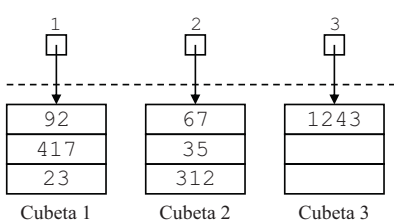
insertar los siguientes datos:

92, 417, 67, 23, 1243, 35, 312; 395; 798

Inserción de 92, 417



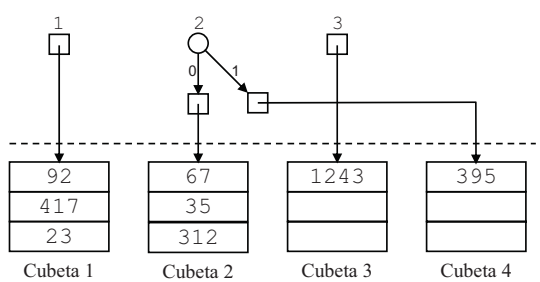
Inserción de 67, 23, 1243, 35, 312



b) dispersión dinámica BT: ejemplo de inserción (2)

clave	*H(clv)*	*B(clave)*
23	1	101010
35	2	010100
67	2	001110
92	1	110110
312	2	011110
395	2	110110
417	1	001110
798	2	000110
1243	3	101101

Inserción de 395 → división de la cubeta 2

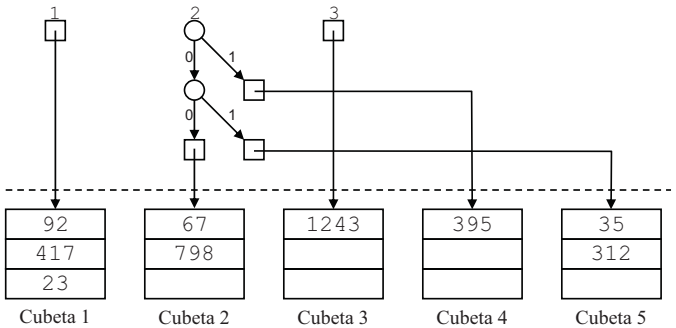


Inserción de 798 → división de la cubeta 2

b) dispersión dinámica BT: ejemplo de inserción (3)

clave	*H(clv)*	*B(clave)*
23	1	101010
35	2	010100
67	2	001110
92	1	110110
312	2	011110
395	2	110110
417	1	001110
798	2	000110
1243	3	101101

Inserción de 798 → división de la cubeta 2



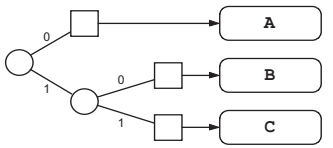
factor de carga ≈ ***69%***

ejercicio*: implementar (los aspectos básicos) de las operaciones de búsqueda, inserción y eliminación*

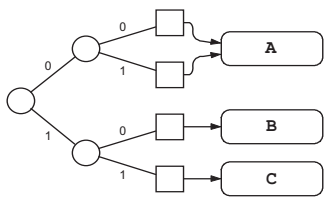
c) dispersión dinámica extensible: introducción (1)

clave	H(clave)
A	0 0 0
B	1 0 0
C	1 1 0
D	1 0 1
E	0 1 1

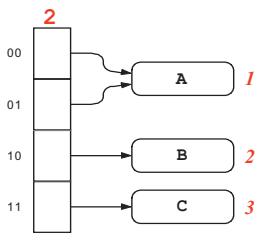
Insertar: A, B, C ⇒



trie que proporciona acceso a las cubetas

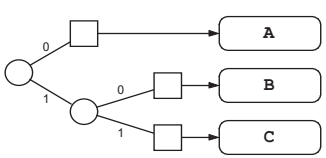


transformación del **trie** en árbol binario completo ⇒

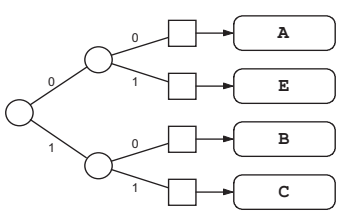


representación del **trie** como directorio

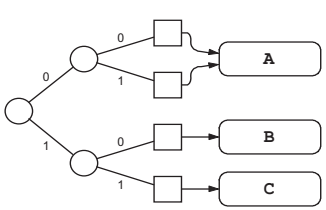
c) dispersión dinámica extensible: introducción (2)



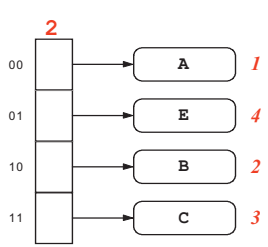
Insertar: E ⇒



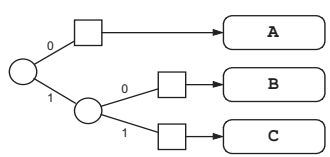
clave	H(clave)
A	0 0 0
B	1 0 0
C	1 1 0
D	1 0 1
E	0 1 1



⇒

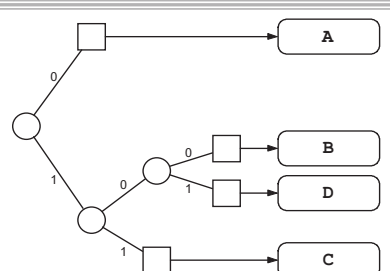


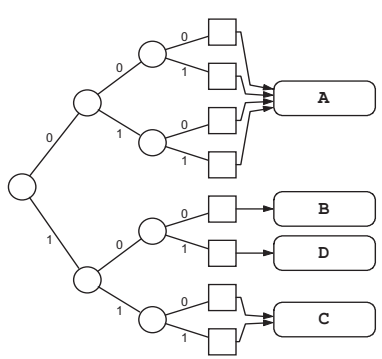
c) dispersión dinámica extensible: introducción (3)



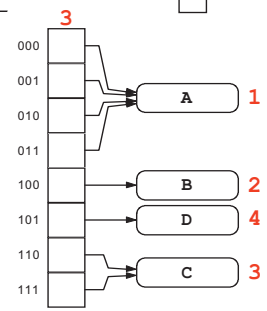
Insertar: D ⇒

clave	H(clave)
A	0 0 0
B	1 0 0
C	1 1 0
D	1 0 1
E	0 1 1





⇒



c) dispersión dinámica extensible: conceptos básicos (1)

c) *dispersión dinámica extensible*

objetivo: aprovechar la idea del método anterior, mejorando su rendimiento

idea: { • mantener un "directorio" con referencias a las cubetas; • acceso directo a la referencia de la cubeta a partir de la clave } ⇒ 2 accesos (a lo sumo)

→ *acceso con nº formado por* d_p *bits* ⇒ *tamaño directorio potencia de 2*

profundidad de directorio

`H(clave) = b1b2b3•••bn` *secuencia digitos binarios*

`B(clave,dp) = b1b2•••bdp` *entero ≡ secuencia de los* d_p *primeros digitos de* `H(clave)`

mejor en orden inverso

✓ se construye un directorio con nº d_p bits y 2^{d_p} entradas (inicialmente $d_p = 0$)

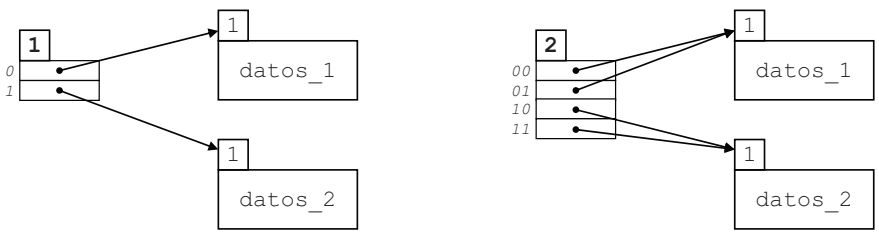
✓ la entrada i-ésima contiene la referencia (NRR) de la cubeta con los registros / B(clave,d_p) = i

c) dispersión dinámica extensible: conceptos básicos (2)

✓ para evitar que haya cubetas vacías ⇒ en cada cubeta *i* { • los registros | B(clave, d_c) = i ; • el número d_c | $d_c <= d_p$ }

profundidad local

↳ duplicar el directorio (≡ $d_p := d_p + 1$) ⇒ { • *duplicar el contenido* ; • *no afecta a las cubetas* }



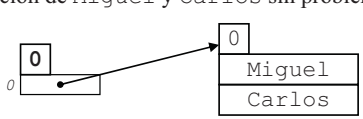
c) dispersión dinámica extensible: ejemplo de inserción (1)

Ejemplo: Suponiendo que $f_b = 2$, y la función de binarización:
insertar los siguientes datos:

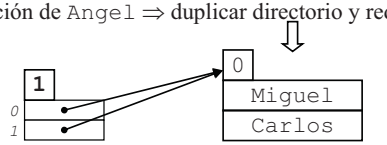
`Miguel, Carlos, Angel, Pepe, Pedro, Luis, y Alfredo`

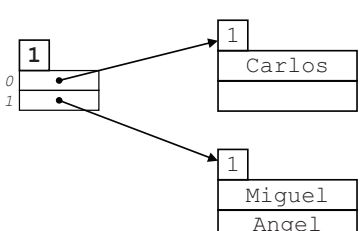
B(clave)	clave	Info.
00101…	Pedro	. . .
11010…	Luis	. . .
10100…	Angel	. . .
10000…	Alfredo	. . .
11110…	Miguel	. . .
10110…	Pepe	. . .
01011…	Carlos	. . .

Inserción de `Miguel` y `Carlos` sin problemas



Inserción de `Angel` ⇒ duplicar directorio y redistribución

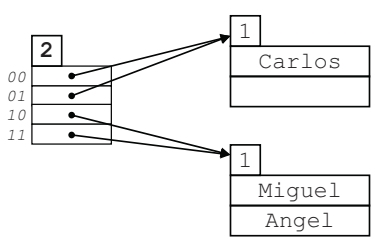


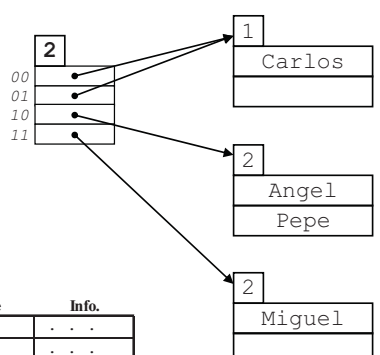


Inserción de `Pepe` ?

c) dispersión dinámica extensible: ejemplo de inserción (2)

Inserción de `Pepe` ⇒ duplicar directorio y redistribución





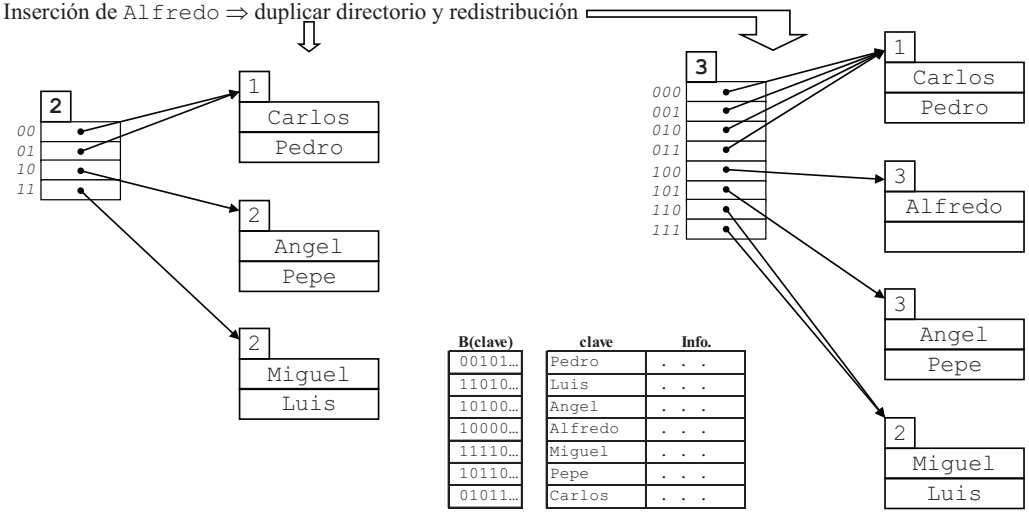
B(clave)	clave	Info.
00101…	Pedro	. . .
11010…	Luis	. . .
10100…	Angel	. . .
10000…	Alfredo	. . .
11110…	Miguel	. . .
10110…	Pepe	. . .
01011…	Carlos	. . .

Inserción de `Pedro` y `Luis` y `Alfredo`?

c) dispersión dinámica extensible: ejemplo de inserción (3)

Inserción de `Pedro` y `Luis` sin problemas

Inserción de `Alfredo` ⇒ duplicar directorio y redistribución



B(clave)	clave	Info.
00101…	Pedro	. . .
11010…	Luis	. . .
10100…	Angel	. . .
10000…	Alfredo	. . .
11110…	Miguel	. . .
10110…	Pepe	. . .
01011…	Carlos	. . .

c) dispersión dinámica extensible: algoritmo de inserción

Inserción:

1) i := B(clave,d_p); → se accede a la cubeta C1 en NRR1 = directorio[i]

2) si cabe → no hay problema: se coloca el nuevo registro; fin

3) si no cabe ⇒ *crear una nueva cubeta y redistribuir con ella los registros utilizando 1 bit más*

a) crear una nueva cubeta C2 en NRR2

b) si $d_{c1} = d_p$ *entonces* duplicar directorio → $d_p := d_p + 1$ *Fsi*

c) $d_{c1} = d_{c1} + 1$; $d_{c2} = d_{c1}$;

d) ajustar las entradas de directorio / los d_{c1} primeros bits = B(clave,d_{c1}) → C1 / C2

e) redistribuir los registros entre C1 y C2, según los d_{c1} primeros bits

f) si no fuese posible, → C1 ≡ la cubeta que desborda, y continuar en paso *b)*

duplicar directorio

```
Para i := 2^(dp-1) descendiendo hasta 0 hacer
    directorio[2*i+1] := directorio[i]; directorio[2*i] := directorio[i];
Fpara;
dp := dp + 1;
```


c) dispersión dinámica extensible: conclusiones

Eliminación: ⇒ búsqueda y, si encontrado, simple eliminación

→ *Posibilidad de fusión con cubeta "adyacente"* / *Si se queda vacía, liberar el espacio* } *no siempre conveniente*

↓

reestructurar el fichero "de vez en cuando"

Evaluación:

- *factor de carga ≈* ***ln(2) ≈ 69,3%***
- *nº de referencias asociadas a cada cubeta ≈* ***1/ln(2) ≈ 1,44***

☛ el rendimiento no disminuye al crecer el fichero

☛ el espacio adicional (directorio) es pequeño

→ un acceso adicional (2 en total)

✶ *ejemplo de organización dispersa: gdbm*

otras implementaciones (db):

Advanced programming in the UNIX environement **W. Richard Stevens**

c) ejemplo de implementación : *gdbm*

```
typedef struct {
          char *dptr;
          int  dsize;
        } datum;
```

algunas definiciones básicas de ***gdbm***

```
GDBM_FILE gdbm_open (name, block_size, flags, mode, fatal_func);
void   gdbm_close   (dbf);
int    gdbm_store   (dbf, key, content, flag);
datum  gdbm_fetch   (dbf, key);
int    gdbm_delete  (dbf, key);
datum  gdbm_firstkey(dbf);
datum  gdbm_nextkey (dbf);
int    gdbm_exists  (dbf, key);
int    gdbm_reorganize(dbf);
void   gdbm_sync    (dbf);
char *gdbm_strerror(errno);
int    gdbm_setopt  (dbf, option, value, size);
```

gdbm : función de dispersión

```
word_t gdbm_hash (datum key)
{
unsigned int value;       /* Used to compute the hash value.  */
int   index;              /* Used to cycle through random values. */

  /* Set the initial value from key. */
  value = 0x238F13AF * key.dsize;
  for (index = 0; index < key.dsize; index++)
    value = (value + (key.dptr[index] << (index*5 % 24))) & 0x7FFFFFFF;

  value = (1103515243 * value + 12345) & 0x7FFFFFFF;

  /* Return the value. */
  return((word_t) value);
}
```


comparación entre dispersión e indexación

	Indexación	Dispersión
☞ ventajas		
☞ inconvenientes		

Otras organizaciones de ficheros

☞ pb. acceso a los datos usando criterios más complejos (p.e. multiclave)

otras E.D. + sofisticadas →
- *estructura de "rejilla"*
- *dispersión dividida*
- *índices de mapas de bits*

....

mezcla de organizaciones . . .

☞ solución "óptima" => COMPROMISO complejo de resolver

☞ quedan otros problemas esenciales (concurrencia, seguridad, confidencialidad,...)

→ BASES DE DATOS