Tema 2. Conjuntos y Diccionarios

2.1. Repaso del TAD Conjunto

2.2. Implementaciones básicas

2.3. El TAD Diccionario

2.4. Las tablas de dispersión

2.5. Relaciones muchos a muchos

2.1. Repaso del TAD conjunto

Definiciones y propiedades

- **Conjunto:** colección no ordenada de elementos (o miembros) distintos.
- **Elemento:** cualquier cosa, puede ser un conjunto o un elemento primitivo (átomo).

En programación:

- Puede haber repetición de elementos (**bolsas**).
- Todos los elementos suelen ser del mismo tipo (enteros, caracteres, cadenas ...)
- Los elementos pueden estar ordenados.

  Relación "<" de orden de un conjunto S:
  - Orden total: para todo a, b, sólo una de las afirmaciones (a<b), (b<a) o (a=b) es cierta.
  - Propiedad transitiva: para todo a, b, c, si (a<b) y (b<c) entonces (a<c).

2.1. Repaso del TAD conjunto

Notación de conjuntos

- **Definición**:

  Por extensión

  A= {a, b, c, .., z}

  B= {1, 4, 7} = {4, 7, 1}

  Mediante proposiciones

  C= {x | proposición de x}

  D= {x | x es primo y menor que 90}

- **Pertenencia**: $x \in A$
- **No pertenencia**: $x \notin A$
- **Conjunto vacío**: $V = \emptyset$
- **Conjunto universal**: $U = U$
- **Inclusión**: $A \subseteq B$
- **Intersección:** $A \cap B$
- **Unión**: $A \cup B$
- **Diferencia**: $A - B$

2.1. Repaso del TAD conjunto

El TDA Conjunto: Operaciones más comunes

A, B, C y S conjuntos; x de tipo elemento

- Unión (A, B, C) — C:= $A \cup B$
- Intersección (A, B, C) — C:= $A \cap B$
- Diferencia (A, B, C) — C:= A - B
- Combina (A, B, C) — C:= $A \cup B$, con $A \cap B = \emptyset$
- bool= Miembro (x, A) — Verdad si $x \in A$. Falso si $x \notin A$
- Anula (A) — A:= $\emptyset$
- Inserta (x, A) — A:= $A \cup \{x\}$
- Suprime (x, A) — A:= A - {x}
- Asigna (A, B) — A:= B
- Min (A) — Devuelve el menor elemento de A
- Max (A) — Devuelve el mayor elemento de A
- bool= Igual (A, B) — Verdad si A = B. Falso si $A \neq B$
- S= Encuentra (x) — Devuelve el conjunto al que pertenece x

2.1. Repaso del TAD conjunto

Ejemplos

A:= {3, 5, 6, 8} B:= {2, 3, 5, 7, 9}

• Unión (A, B, C)	C:= {3, 5, 6, 8, 2, 7, 9}
• Intersección (A, B, C)	C:= {3, 5}
• Diferencia (A, B, C)	C:= {6, 8}
• Combina (A, B, C)	Error, ya que $A \cap B \neq \emptyset$
• Miembro (6, A)	Verdad
• Anula (A)	A:= Ø
• Inserta (5, A)	A:= {3, 5, 6, 8}
• Inserta (2, A)	A:= {2, 3, 5, 6, 8}
• Suprime (1, A)	A:= {3, 5, 6, 8}
• Asigna (A, B)	A:= {2, 3, 5, 7, 9}
• Min (A)	3
• Max (A)	8
• Igual (A, B)	Falso
• Encuentra (7)	B
• Encuentra (3)	Error, ya que $3 \in A$, $3 \in B$.

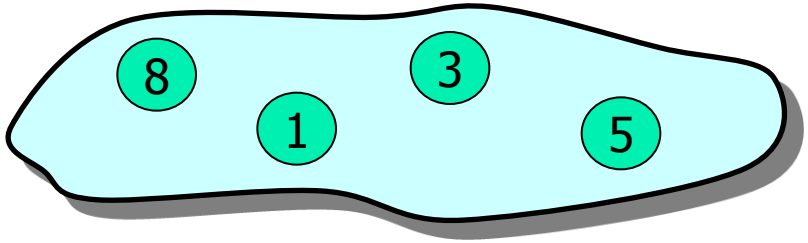
2.2. Implementaciones básicas

- Dos tipos de implementaciones básicas:
  - Mediante arrays de booleanos.
  - Mediante listas enlazadas.

La mejor implementación (en términos de eficiencia) depende del uso que hagamos de los conjuntos:

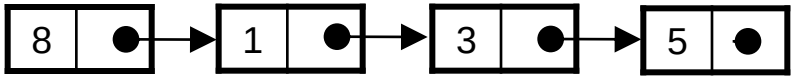
- Operaciones más frecuentes.
- Tamaño y variabilidad de los conjuntos usados.

C: Conjunto



1	2	3	4	5	6	7	8	9	10
1	**0**	**1**	**0**	**1**	**0**	**0**	**1**	**0**	**0**

Array de booleanos



Lista de elementos

2.2. Implementaciones básicas

2.2.1. Mediante arrays de booleanos

- **Idea:** Cada elemento del conjunto universal se representa con 1 bit. Para cada conjunto concreto A, el bit asociado a un elemento vale:

  1 - Si el elemento pertenece al conjunto A

  0 - Si el elemento no pertenece a A

- **Definición**:

  tipo

  Conjunto[T] = **array** [1..Rango(T)] **de** booleano

  Donde Rango(T) es el tamaño del conjunto universal.

2.2.1. Mediante arrays de booleanos

- **Ejemplo**: T = {a, b, …, g}

  C= Conjunto[T]

  A = {a, c, d, e, g}

  B = {c, e, f, g}

a	b	c	d	e	f	g
1	0	1	1	1	0	1

A: Conjunto[a..g]

a	b	c	d	e	f	g
0	0	1	0	1	1	1

B: Conjunto[a..g]

- Unión, intersección, diferencia: se transforman en las operaciones booleanas adecuadas.

2.2.1. Mediante arrays de booleanos

```
operación Unión (A, B: Conjunto[T]; var C: Conjunto[T])
    para cada i en Rango(T) hacer
        C[i]:= A[i] OR B[i]

operación Intersección (A, B: Conjunto[T]; var C: Conjunto[T])
    para cada i en Rango(T) hacer
        C[i]:= A[i] AND B[i]

operación Diferencia (A, B: Conjunto[T]; var C: Conjunto[T])
    para cada i en Rango(T) hacer
        C[i]:= A[i] AND NOT B[i]
```

2.2.1. Mediante arrays de booleanos

```
operación Inserta (x: T; var C: Conjunto[T])
    C[x]:= 1

operación Suprime (x: T; var C: Conjunto[T])
    C[x]:= 0

operación Miembro (x: T; C: Conjunto[T]): booleano
    devolver C[x]==1
```

- ¿Cuánto tardan las operaciones anteriores?
- ¿Cómo serían: Igual, Min, Max, ...?

2.2.1. Mediante arrays de booleanos

Ventajas:

- Operaciones muy sencillas de implementar. Se pueden realizar sin necesidad de usar memoria dinámica.
- **Miembro**, **Inserta** y **Suprime** tienen un tiempo constante.
- **Unión**, **Intersección** y **Diferencia** se pueden realizar en un tiempo proporcional al tamaño del conjunto universal.
- Si el conjunto universal es tan pequeño como el número de bits de una palabra de la máquina, las operaciones anteriores se pueden realizar con una simple operación lógica.

Inconvenientes:

- Utiliza espacio proporcional al tamaño del conjunto universal.
- El conjunto universal no puede ser muy grande ni infinito.
- Cada elemento debe tener un índice (¿Que pasa si tenemos cadenas?).

2.2.1. Mediante arrays de booleanos

- **Ejemplo**. Implementación en C, con
  T = {1, 2, …, 64}
  tipo
  Conjunto[T] = long long
- Unión (A, B, C) → C = A | B;
- Intersección (A, B, C) → C = A & B;
- Inserta (x, C) → C = C | (1<<(x-1));

- ¡Cada conjunto ocupa 8 bytes, y las operaciones se hacen en 1 ó 3 ciclos!

- **Ejemplo**. Implementación con
  T = enteros de 32 bits = {0, 1, …, 2^{32}-1}

  tipo
  Conjunto[T] = array [4.294.967.296] de bits = array [536.870.912] de bytes

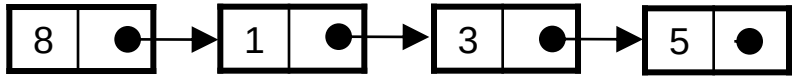
- ¡Cada conjunto ocupa 0,5 Gygabytes, independientemente de que contenga sólo uno o dos elementos…!
- ¡El tiempo es proporcional a ese tamaño!

2.2.2. Implementación mediante listas de elementos

- **Idea**: Guardar en una lista los elementos del conjunto.
- **Definición**:

  tipo Conjunto[Tipo] = Lista[Tipo];

C = {1, 5, 8, 3}



C: Conjunto[T]

Ventajas:

- Utiliza espacio proporcional al tamaño del conjunto representado (no al conjunto universal).
- El conjunto universal puede ser muy grande, o incluso infinito.

Inconvenientes:

- Operaciones más complejas de implementar.
- Utiliza más recursos si el conjunto está “muy lleno”.
- Algunas operaciones son menos eficientes.

2.2.2. Mediante listas de elementos

```
operación Miembro (x: T; C: Conjunto[T]): booleano
    Primero(C)
    mientras Actual(C) ≠  x AND NOT EsUltimo(C) hacer
            Avanzar(C)
    devolver Actual(C) == x

operación Intersección (A, B; Conjunto[T]; var C: Conjunto[T])
    C:= ListaVacía
    Primero(A)
    mientras NOT EsUltimo(A) hacer
            si Miembro(Actual(A), B) entonces
                    InsLista(C, Actual(A))
            Avanzar(A)
    finmientras
```

2.2.2. Mediante listas de elementos

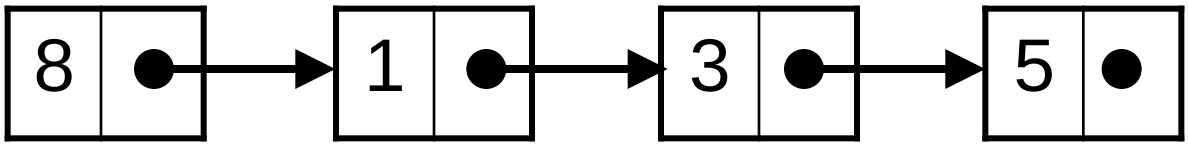
- ¿Cuánto tiempo tardan las operaciones anteriores? Suponemos una lista de tamaño **n** y otra **m** (o ambas de tamaño **n**).

- ¿Cómo sería Intersección, Diferencia, Inserta, Suprime, etc.?

- **Inconveniente:** Unión, Intersección y Diferencia recorren la lista B muchas veces (una por cada elemento de A).
- Se puede mejorar usando listas ordenadas.

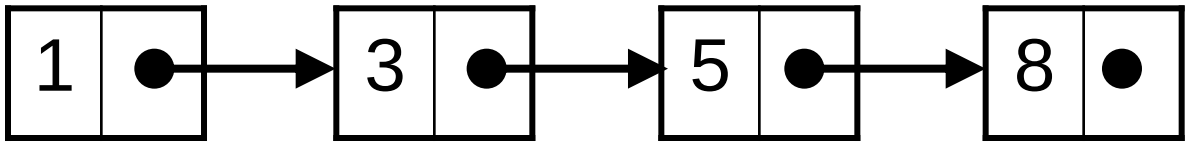
2.2.2. Mediante listas de elementos

- Listas no ordenadas.



C: Conjunto[T]

- Listas ordenadas.

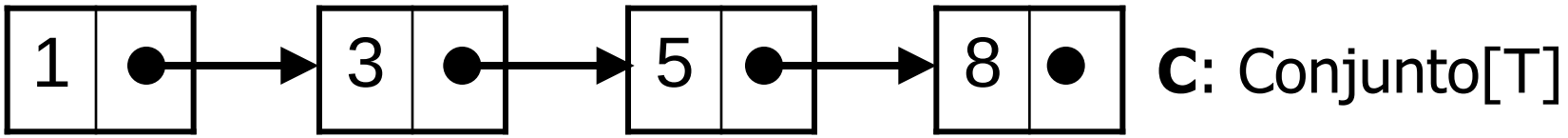


C: Conjunto[T]

- **Miembro, Inserta, Suprime**: Parar si encontramos un elemento mayor que el buscado.
- **Unión**, **Intersección**, **Diferencia**: Recorrido simultáneo (y único) de ambas listas.

2.2.2. Mediante listas de elementos

operación Miembro (x: T; C: Conjunto[T]): booleano
    Primero(C)
    mientras Actual(C) < x AND NOT EsUltimo(C) **hacer**
        Avanzar(C)
    devolver Actual(C) == x



- ¿Cuánto es el tiempo de ejecución ahora?

2.2.2. Mediante listas de elementos

```
operación Unión (A, B; Conjunto[T]; var C: Conjunto[T])
    C:= ListaVacía
    Primero(A)
    Primero(B)
    mientras NOT EsUltimo(A) AND NOT EsUltimo(B) hacer
        si EsUltimo(B) OR Actual(A)<Actual(B) entonces
                InsLista(C, Actual(A))
                Avanza(A)
        sino si EsUltimo(A) OR Actual(B)<Actual(A) entonces
                InsLista(C, Actual(B))
                Avanza(B)
        sino
                InsLista(C, Actual(A))
                Avanza(A)
                Avanza(B)
        finsi
    finmientras
```

2.2.2. Mediante listas de elementos

- ¿Cuánto es el tiempo de ejecución? ¿Es sustancial la mejora?
- ¿Cómo serían la Intersección y la Diferencia?
- ¿Cómo serían las operaciones Min, Max?
- ¿Cuánto es el uso de memoria para tamaño **n**? Supongamos que 1 puntero = k_1 bytes, 1 elemento = k_2 bytes.

2.2. Implementaciones básicas

Conclusiones

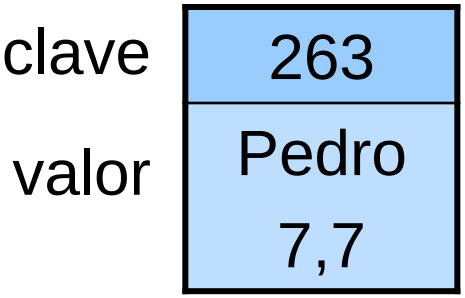
- **Arrays de booleanos**: muy rápida para las operaciones de inserción y consulta.
- Inviable si el tamaño del conjunto universal es muy grande.
- **Listas de elementos**: uso razonable de memoria, proporcional al tamaño usado.
- Muy ineficiente para la inserción y consulta de un elemento.
- **Solución:** Tablas de dispersión, estructuras de árbol, combinación de estructuras, etc.

2.3. El TDA diccionario

- En muchas aplicaciones necesitamos guardar datos de un conjunto de elementos, que pueden variar en tiempo de ejecución.
- **P. ej.:** agenda electrónica, diccionario de sinónimos, base de datos de empleados, notas de alumnos, etc.
- **Particularidades**:
  - Los datos se guardan en un solo sitio, no siendo necesarias las operaciones de unión, intersección o diferencia, pero sí inserciones, consultas y modificaciones.
  - Cada elemento tiene una clave, y asociado a ella se guardan una serie de valores.
  - Las operaciones de **consulta son por clave.**

2.3. El TDA diccionario

•**Definición**: Una **asociación** es un par (clave: tipo_clave, valor: tipo_valor).



•**Definición**: Un **diccionario** es un conjunto de asociaciones, con las operaciones Inserta, Suprime, Consulta y Crear.

TAD Diccionario[tclave, tvalor]

Inserta (**clave**: tclave; **valor**: tvalor, **var D**: Diccionario[tcl,tval])

Consulta (**clave**: tclave; **D**: Diccionario[tcl,tval]): **tvalor**

Suprime (**clave**: tclave; **var D**: Diccionario[tcl,tval])

Crear (**var D**: Diccionario[tcl,tval])

2.3. El TDA diccionario

Todo lo dicho sobre implementación de conjuntos se puede aplicar (extender) a diccionarios.

- **Implementación**:
  - **Con arrays de booleanos**: ¡Imposible! Conjunto universal muy limitado. ¿Cómo conseguir la asociación clave-valor?
  - **Con listas de elementos**: Representación más compleja y muy ineficiente para inserción, consulta, etc.

- Representación sencilla **mediante arrays**.

  tipo

  Diccionario[tclave, tvalor] = **registro**
  último: entero
  datos: **array** [1..máximo] **de** Asociacion[tclave, tvalor]
  finregistro

2.3. El TDA diccionario

```
operación Crear (var D: Diccionario[tclave, tvalor])
        D.último:= 0

oper Inserta (clave: tclave; valor: tvalor; var D: Diccionario[tc,tv])
        para i:= 1 hasta D.último hacer
                si  D.datos[i].clave == clave  entonces
                        D.datos[i].valor:= valor
                        acabar
        finpara
        si  D.último < máximo  entonces
                D.último:= D.último + 1
                D.datos[D.último]:= (clave, valor)
        sino
                Error (“El diccionario está lleno”)
        finsi
```

2.3. El TDA diccionario

```
operación Consulta (clave: tclave; D: Diccionario[tc,tv]): tvalor
        para i:= 1 hasta D.último hacer
                si  D.datos[i].clave == clave  entonces
                        devolver D.datos[i].valor
        finpara
        devolver NULO

operación Suprime (clave: tclave; var D: Diccionario[tc,tv])
        i:= 1
        mientras (D.datos[i].clave ≠ clave) AND (i < D.último) hacer
                 i:= i + 1
        finmientras
        si  D.datos[i].clave == clave  entonces
                D.datos[i]:= D.datos[D.último]
                D.último:= D.último – 1
        finsi
```

2.4. Las tablas de dispersión

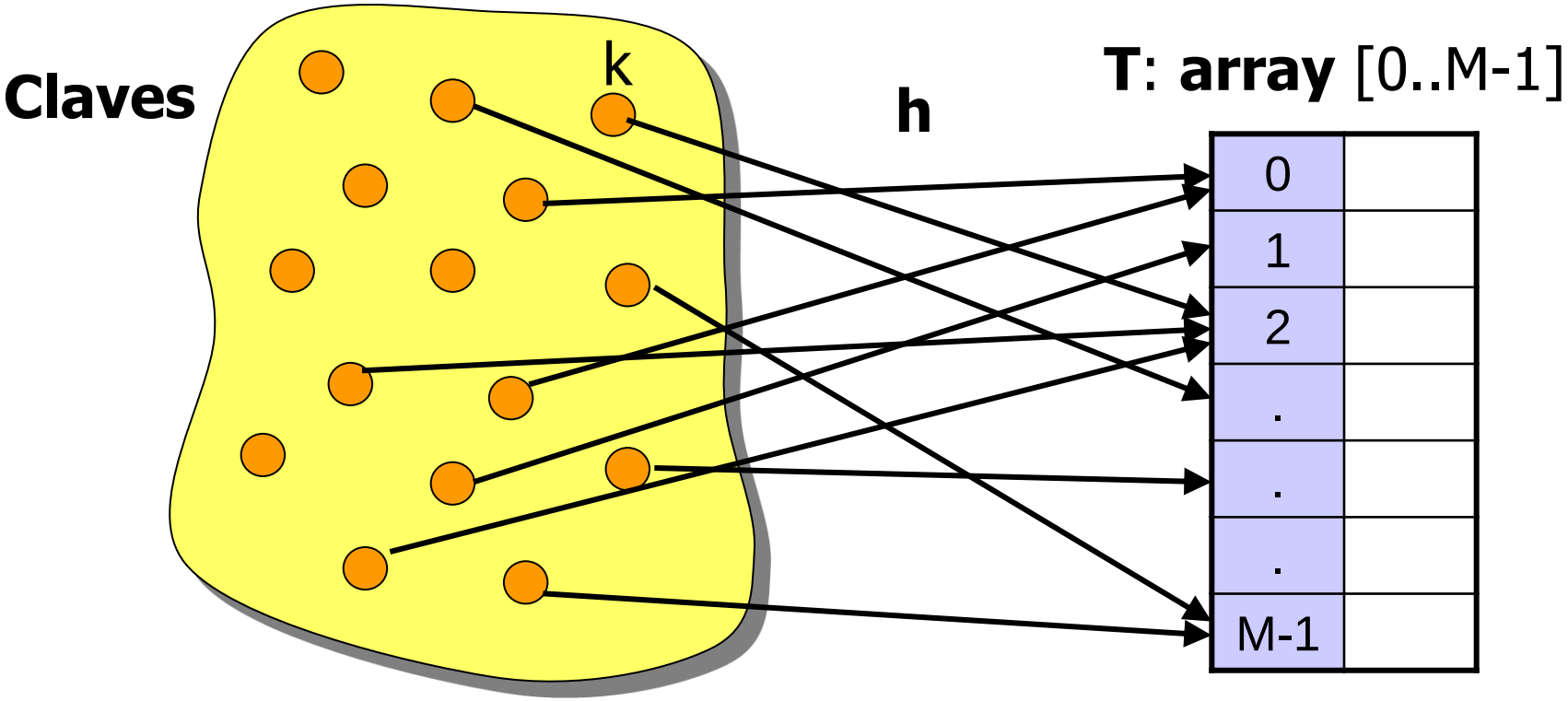
- La representación de conjuntos o diccionarios con listas o arrays tiene un tiempo de **O(n)**, para Inserta, Suprime y Miembro, con un uso razonable de memoria.

- Con vectores de bits el tiempo es **O(1)**, pero tiene muchas limitaciones de memoria.

- ¿Cómo aprovechar lo mejor de uno y otro tipo?

2.4. Las tablas de dispersión

- **Idea**: reservar un tamaño fijo, un array T con M posiciones (0, ..., M-1).
- Dada una clave k (sea del tipo que sea) calcular la posición donde colocarlo, mediante una función **h**.



2.4. Las tablas de dispersión

- **Función de dispersión (hash): h**

  h : tipo_clave $\rightarrow$ [0, …, M-1]

- **Insertar (clave, valor, T)**:
- Aplicar **h(clave)** y almacenar en esa posición **valor**.

  T[h(clave)]= valor
- **Consultar (clave, T): valor**:
- Devolver la posición de la tabla en h(clave).

  devolver T[h(clave)]
- Se consigue **O(1)**, en teoría…

2.4. Las tablas de dispersión

- El tipo del elemento no está restringido. Si es un registro, podemos tomar uno de sus campos como **clave**, y aplicar la función sobre él.
- Función de dispersión: **h: Clave $\rightarrow$ [0, ..., M-1]**
- Tamaño de la tabla M: aproximadamente sobre el número de elementos de los conjuntos (normalmente, M << rango de claves).

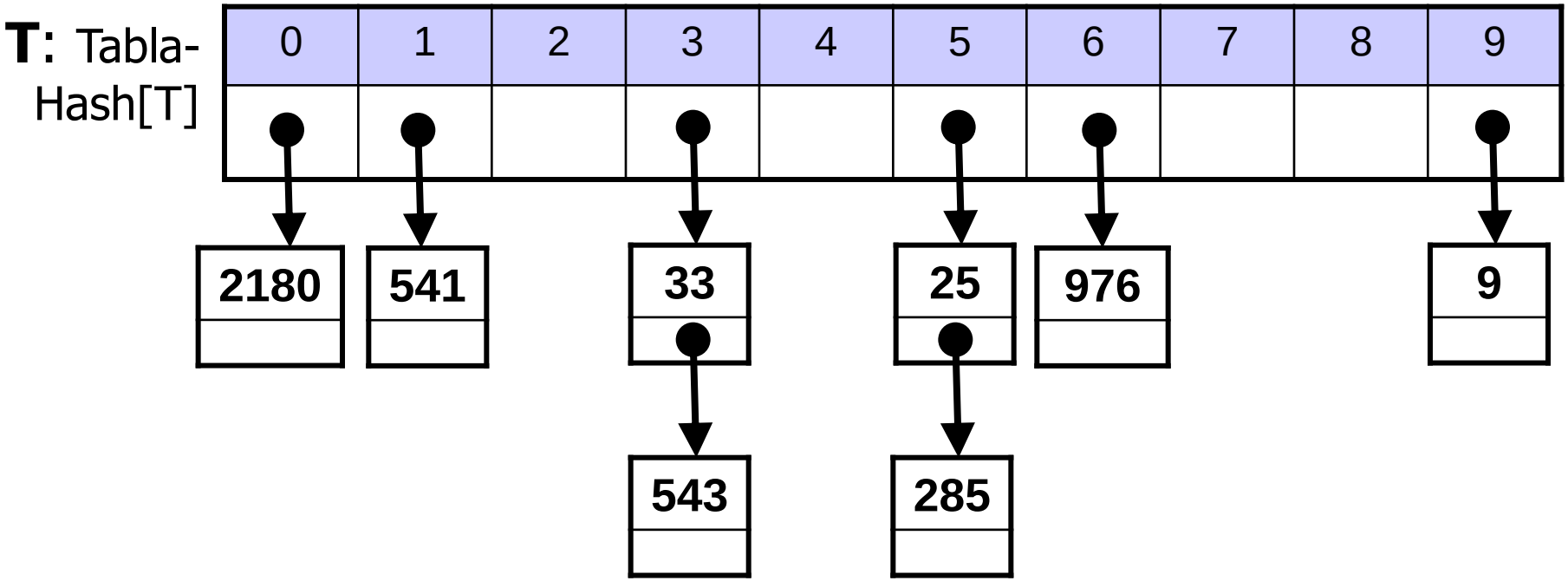
- ¿Qué ocurre si para dos claves distintas k_1, k_2, ocurre que $h(k_1) = h(k_2)$?
- **Definición**: Si $(k_1 \neq k_2) \wedge (h(k_1) = h(k_2))$ entonces se dice que k_1, k_2 son **sinónimos**.
- Los distintos métodos de dispersión difieren en el tratamiento de los sinónimos. Tipos de dispersión:
  - **Dispersión abierta.**
  - **Dispersión cerrada.**

2.4.1. Dispersión abierta

- Las celdas de la tabla no son elementos (o asociaciones), sino listas de elementos, también llamadas **cubetas**.

  tipo TablaHash[T]= **array** [0..M-1] **de** Lista[T]

- Sea M= 10, D= {9, 25, 33, 976, 285, 541, 543, 2180}



2.4.1. Dispersión abierta

- La tabla de dispersión está formada por M cubetas. Dentro de cada una están los sinónimos.
- El conjunto de sinónimos es llamado **clase**.

Eficiencia de la dispersión abierta

- El tiempo de las operaciones es proporcional al tamaño de las listas (cubetas).
- Supongamos **M** cubetas y **n** elementos en la tabla.
- Si todos los elementos se reparten uniformemente cada cubeta será de longitud: **1 + n/M**

2.4.1. Dispersión abierta

- Tiempo de Inserta, Suprime, Consulta: **O(1+n/M)**
- **Ojo:** ¿qué ocurre si la función de dispersión no reparte bien los elementos?

Utilización de memoria

- Si 1 puntero = p_1 bytes, 1 elemento = p_2 bytes.
- En las celdas: $(p_1 + p_2)n$
- En la tabla: $p_1 \cdot M$

Conclusión:

Menos cubetas: se gasta menos memoria.

Más cubetas: operaciones más rápidas.

2.4.2. Dispersión cerrada

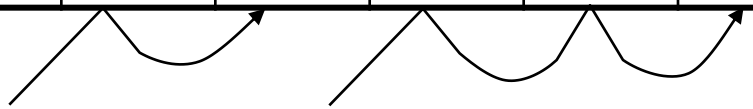
•Las celdas del array son elementos del diccionario (no listas). No se ocupa un espacio adicional de memoria en listas.

tipo TablaHash[tc, tv]= **array** [0..M-1] **de** (tc, tv)

•Si al insertar un elemento nuevo k, ya está ocupado h(k), se dice que ocurre una **colisión**.

•En este caso será necesario hacer **redispersión**: buscar una nueva posición para meter el elemento.

0	1	2	3	4	5	6	7	8	9
2180	541		33	**543**	285	976	**25**		9



- **Redispersión**: si falla h(k), aplicar $h_1(k)$, $h_2(k)$, ... hasta encontrar una posición libre.
- **Redispersión lineal**: $h_i(k)= (h(k) + i) \bmod M$
- La secuencia de posiciones recorridas para un elemento se suele denominar cadena o **secuencia de búsqueda**.

2.4.2. Dispersión cerrada

- Consulta (k, D):
  - Examinar la posición h(k).
  - Si x está en la posición h(k), entonces devolver Verdad (o Tabla[h(k)].valor).
  - Si está vacía, entonces no es miembro; devolver Falso.
  - En otro caso, la posición está ocupada pero por otro elemento. Debemos examinar las posiciones $h_1(k)$, $h_2(k)$, ... y así sucesivamente hasta encontrar k, vacío o examinar toda la tabla.

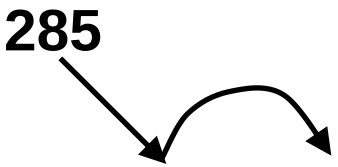
- **Consulta (clave, T): valor**

```
x:= h(clave)
i:= 0;
mientras T[x].clave ≠ clave AND T[x].clave ≠ VACIO
                    AND i<M hacer
      i:= i + 1
      x:= h_i(clave)
finmientras
si T[x].clave == clave entonces
    devolver T[x].valor
sino devolver NULO
```

2.4.2. Dispersión cerrada

- ¿Cómo sería la inserción?
- ¿Y la eliminación?
- **Ojo** con la eliminación.

- **Ejemplo:** eliminar 976 y luego consultar 285.



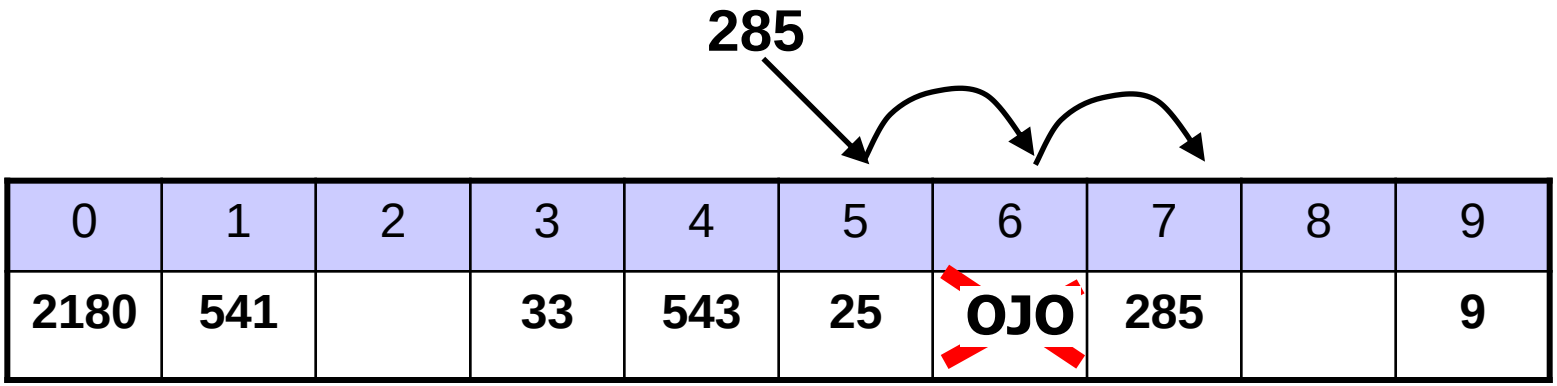
0	1	2	3	4	5	6	7	8	9
2180	**541**		**33**	**543**	**25**		**285**		**9**

Resultado: ¡¡285 no está en la tabla!!

2.4.2. Dispersión cerrada

- **Conclusión:** en la eliminación no se pueden romper las secuencias de búsqueda.
- **Solución:** usar una marca especial de “elemento eliminado”, para que siga la búsqueda.

- **Ejemplo:** eliminar 976 y luego consultar 285.



0	1	2	3	4	5	6	7	8	9
2180	**541**		**33**	**543**	**25**		**285**		**9**

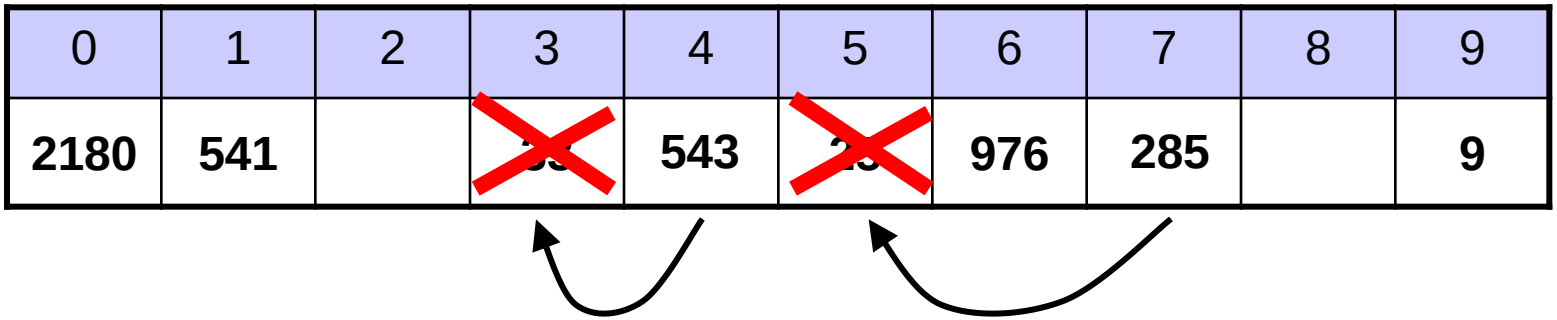
Resultado: ¡¡Encontrado 285 en la tabla!!

2.4.2. Dispersión cerrada

- En la operación Consulta, la búsqueda sigue al encontrar la marca de “elemento eliminado”.
- En Inserta también sigue, pero se puede usar como una posición libre.

- **Otra posible solución:** mover algunos elementos, cuya secuencia de búsqueda pase por la posición eliminada.

- **Ejemplo:** eliminar 25 y luego eliminar 33.



2.4.2. Dispersión cerrada

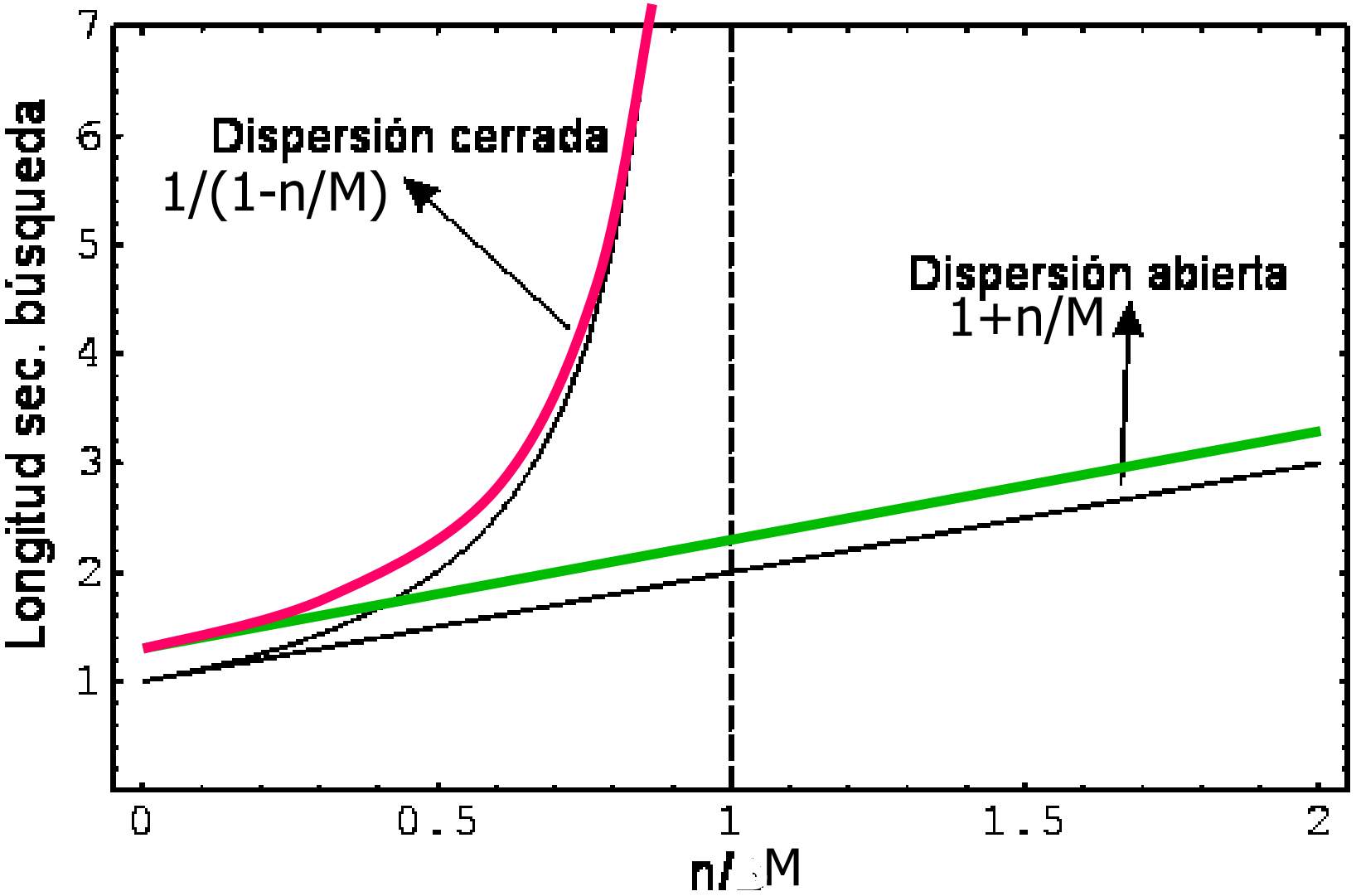
Utilización de memoria en dispersión cerrada

- Si 1 puntero = p_1 bytes, 1 elemento = p_2 bytes.
- Memoria en la tabla: $p_2 \cdot M$
- O bien: $p_1 \cdot M + p_2 \cdot n$
- En **dispersión abierta** teníamos:
  $p_1 \cdot M + (p_1 + p_2)n$
- ¿Cuál es mejor?

Eficiencia de las operaciones

- La tabla nunca se puede llenar con más de M elementos.
- La probabilidad de colisión crece cuantos más elementos hayan, disminuyendo la eficiencia.
- El costo de Inserta es **O(1/(1-n/M))**
- Cuando N $\rightarrow$ M, el tiempo tiende a infinito.
- En **dispersión abierta** teníamos: **O(1+n/M)**

2.4.2. Dispersión cerrada



2.4.2. Dispersión cerrada

Reestructuración de las tablas de dispersión

- Para evitar el problema de la pérdida de eficiencia, si el número de elementos, **n**, aumenta mucho, se puede crear una nueva tabla con más cubetas, M, **reestructurar**.

- Dispersión abierta: reestructurar si $n > 2 \cdot M$

- Dispersión cerrada: reestructurar si $n > 0{,}75 \cdot M$

2.4.3. Funciones de dispersión

- **Propiedades de una buena función de dispersión**
  - La función debe minimizar el número de sinónimos: debe ser lo más aleatoria posible y repartir los elementos en la tabla de manera uniforme.
  - La función debe ser fácil de calcular (buscamos eficiencia).
  - **Ojo:** h(k) es función de k, devuelve siempre el mismo valor para un mismo valor de k.

2.4.3. Funciones de dispersión

Ejemplos de funciones de dispersión

Sea la clave **k** un entero.

- **Método de la multiplicación.**

  $h(k) = \lfloor (((A/w)\cdot k) \bmod 1)\cdot M \rfloor$; con w tamaño palabra y

  A constante entera prima con w

- **Método de división.**

  $h(k) = k \bmod M$;

- **Método del centro del cuadrado.**

  $h(k) = \lfloor k^2 / 100 \rfloor \bmod M$

  $h(k) = \lfloor k^2 / C \rfloor \bmod M$

  Escoger un C, tal que $MC^2 \approx K^2$, para k en el intervalo (0, ..., K).

  Ej.: K= 1000; M= 8; C=354; h(456)= 3

2.4.3. Funciones de dispersión

Sea la clave **k= $x_1\, x_2\, x_3\, x_4\, x_5\, x_6$** un entero o cadena.

- **Método de plegado (folding).**

  $h(k) = (x_1\, x_2 + x_3\, x_4 + x_5\, x_6) \bmod M$

  $h(k) = (x_3\, x_2\, x_1 + x_6\, x_5\, x_4) \bmod M$

- **Método de extracción.**

  $h(k) = (x_4\, x_1\, x_6) \bmod M$

- **Combinación de métodos.**

  $h(k) = \lfloor (x_4\, x_1\, x_6)^2 / D + C(x_3\, x_5\, x_2) \rfloor \bmod M$

  $h(k) = (\lfloor C1 \cdot x^2 \bmod C2 \rfloor + x \cdot C3) \bmod M$

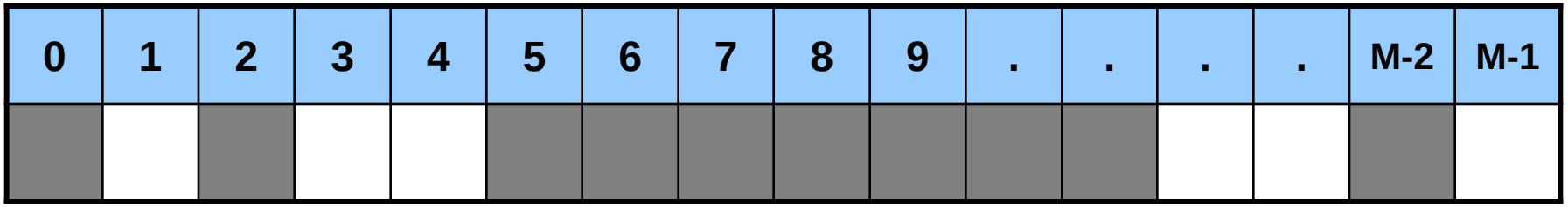
  …

2.4.3. Funciones de redispersión

- **Redispersión lineal.**

$$h_i(k) = h(i, k) = (h(k) + i) \bmod M$$

- Es sencilla de aplicar.
- Se recorren todas las cubetas para i= 1, …, M-1.
- **Problema de agrupamiento:** Si se llenan varias cubetas consecutivas y hay una colisión, se debe consultar todo el grupo. Aumenta el tamaño de este grupo, haciendo que las inserciones y búsquedas sean más lentas.



2.4.3. Funciones de redispersión

- **Redispersión con saltos de tamaño C.**

$$h_i(k) = h(i, k) = (h(k) + C \cdot i) \bmod M$$

- Es sencilla de aplicar.
- Se recorren todas las cubetas de la tabla si C y M son primos entre sí.
- **Inconveniente:** no resuelve el problema del agrupamiento.

- **Redispersión cuadrática.**

$$h(i, k) = (h(k) + D(i)) \bmod M$$

- $D(i) = (+1, -1, +2^2, -2^2, +3^2, -3^2, \ldots)$
- Funciona cuando $M = 4 \cdot q + 3$, para $q \in N$
- ¿Resuelve el problema del agrupamiento?

2.4.3. Funciones de redispersión

- **Redispersión doble.**

  $h(i, k) = (h(k) + C(k) \cdot i) \bmod M$

- **Idea:** es como una redispersión con saltos de tamaño C(k), donde el tamaño del salto depende de k.
- Si M es un número primo, C(k) es una función:

  $C : \text{tipo\_clave} \rightarrow [1, \ldots, M-1]$

- Se resuelve el problema del agrupamiento si los sinónimos (con igual valor h(k)) producen distinto valor de C(k).

- **Ejemplo.** Sea $k = x_1x_2x_3x_4$

  $h(k) = x_1x_4 \bmod M$

  $C(k) = 1 + (x_3x_2 \bmod (M-1))$

2.4. Las tablas de dispersión

Conclusiones:

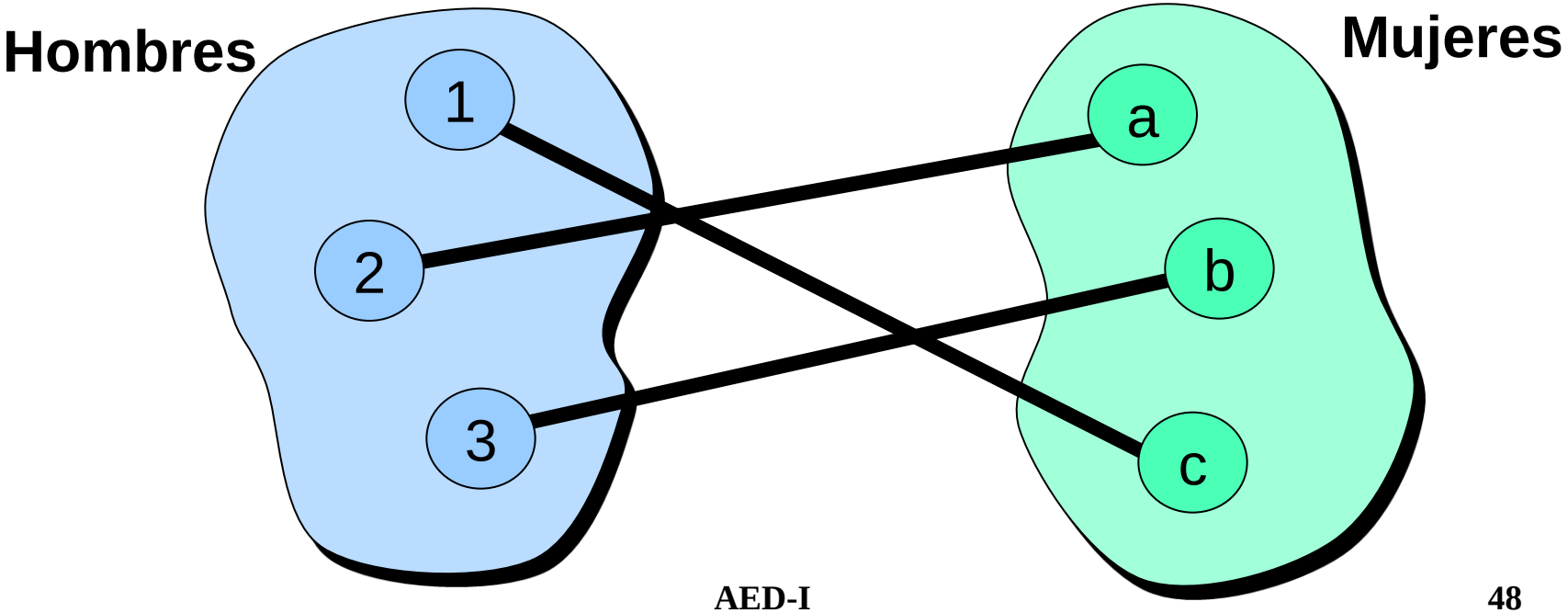
- **Idea básica:** la función de dispersión, **h**, dice dónde se debe meter cada elemento. Cada **k** va a la posición **h(k)**, en principio…
- Con suficientes cubetas y una buena función **h**, el tiempo de las operaciones sería **O(1)**.
- Una buena función de dispersión es esencial. ¿Cuál usar? Depende de la aplicación.
- Las tablas de dispersión son muy buenas para Inserta, Suprime y Consulta, pero…
- ¿Qué ocurre con Unión, Intersección, Máximo, Mínimo, listar los elementos por orden, etc.?

2.5. Relaciones muchos a muchos

- En muchas aplicaciones se almacenan **conjuntos** de dos tipos distintos y **relaciones** entre elementos de ambos.

Tipos de relaciones:

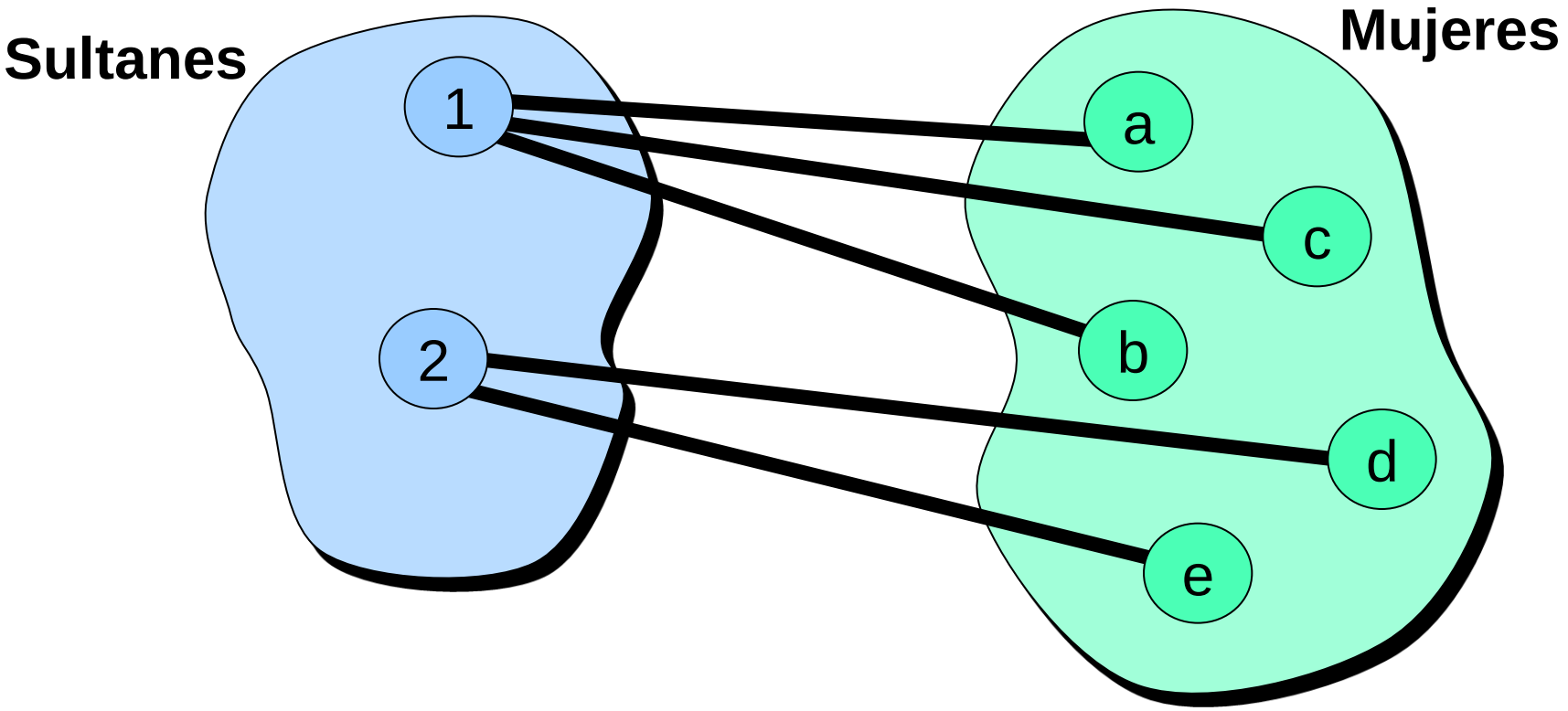
- **Relación uno a uno.** Ej. Relación marido-mujer.



2.5. Relaciones muchos a muchos

Tipos de relaciones:

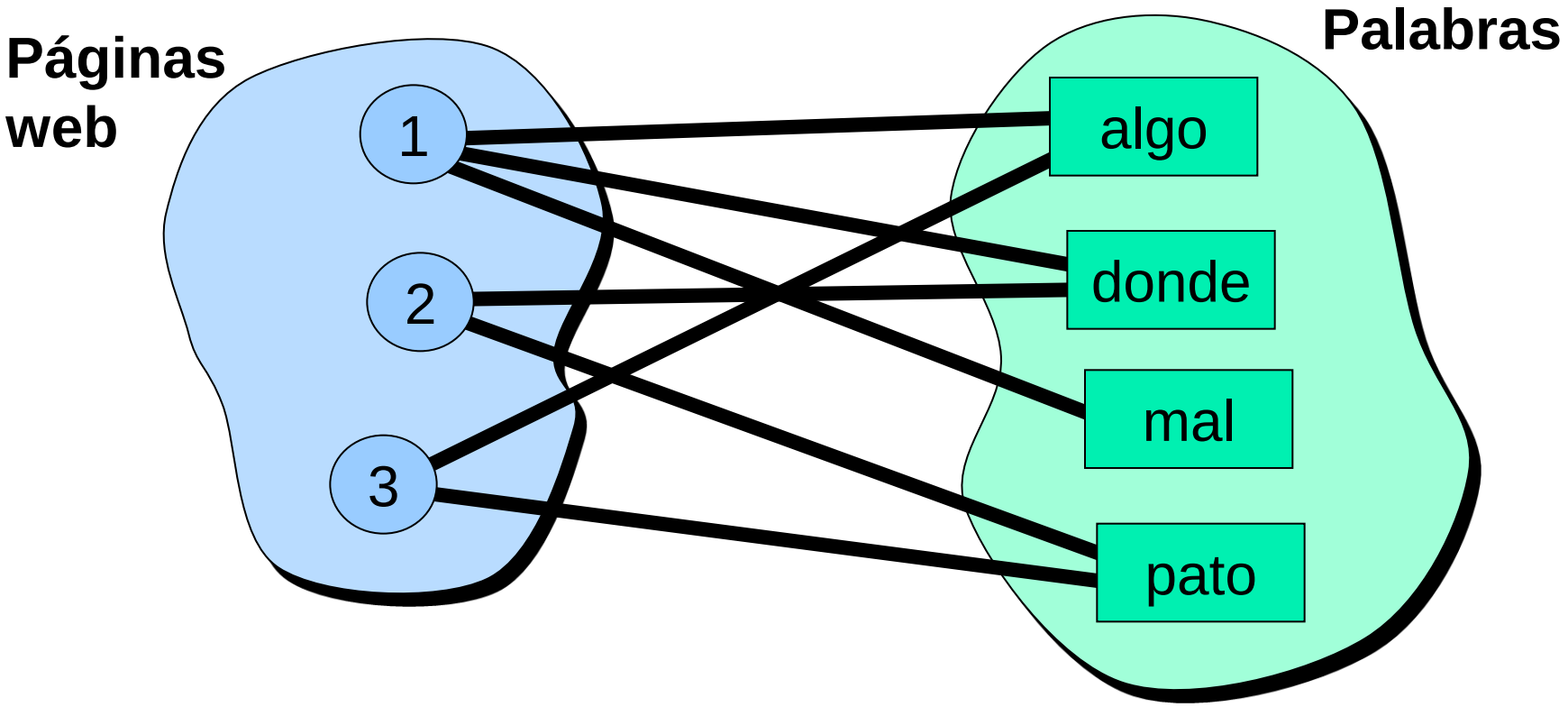
- **Relación uno a muchos.** Ej. Relación marido-mujer.



2.5. Relaciones muchos a muchos

Tipos de relaciones:

- **Relación muchos a muchos.** Ej. Relación "contenido en".



2.5. Relaciones muchos a muchos

- **Otros ejemplos** de relación muchos a muchos:
  - Alumnos de la universidad, cursos y matriculaciones de alumnos en cursos.
  - Personas, libros y préstamos de libros a personas.
  - Ciudades y carreteras entre ciudades.

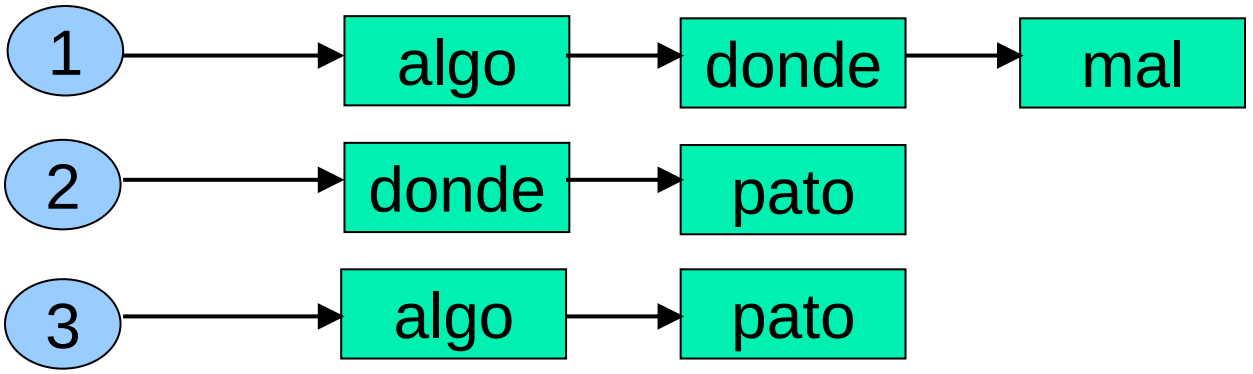
- **Cuestión:** ¿Cómo representar una relación de este tipo?
- **Objetivos:** uso de memoria razonable y tiempo de ejecución rápido.

2.5. Relaciones muchos a muchos

- Supongamos que existen 3 mil millones de páginas ($3 \cdot 10^9$), 20 millones de palabras distintas ($2 \cdot 10^7$) y cada página tiene 30 palabras diferentes.
- En total tenemos: $3 \cdot 10^9 \cdot 30 = 90$ mil millones de relaciones ($9 \cdot 10^{10}$)
- Cada palabra aparece de media en 4.500 páginas.

2.5.1. Representaciones básicas

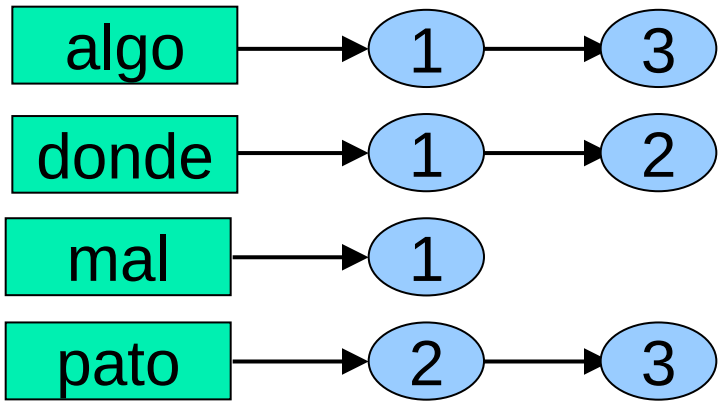
- **Opción 1:** Para cada página, almacenar una lista con las palabras que contiene (lista de punteros a palabras).



- Sea k_1 = tamaño de 1 puntero = 8 bytes
- Uso de memoria: $2 \cdot k_1 \cdot 9 \cdot 10^{10}$ bytes = 1,44 Terabytes
- **Buscar palabras en una página dada:** recorrer (de media) 10 asociaciones.
- **Buscar páginas dada una palabra:** habría que recorrer las $9 \cdot 10^{10}$ asociaciones. **Muy ineficiente.**

2.5.1. Representaciones básicas

- **Opción 2:** Para cada palabra, almacenar una lista con las páginas donde aparece (lista de identificadores).



- Sea k_1 = tamaño de 1 puntero o identificador = 8 bytes
- Uso de memoria: $2 \cdot k_1 \cdot 9 \cdot 10^{10}$ bytes = 1,44 Terabytes
- **Buscar páginas dada una palabra:** recorrer (de media) 4500 asociaciones.
- **Buscar palabras en una página dada:** habría que recorrer las $9 \cdot 10^{10}$ asociaciones. **Muy ineficiente.**

2.5.1. Representaciones básicas

- **Opción 3:** Matriz de booleanos. Una dimensión para las páginas y otra para las palabras.

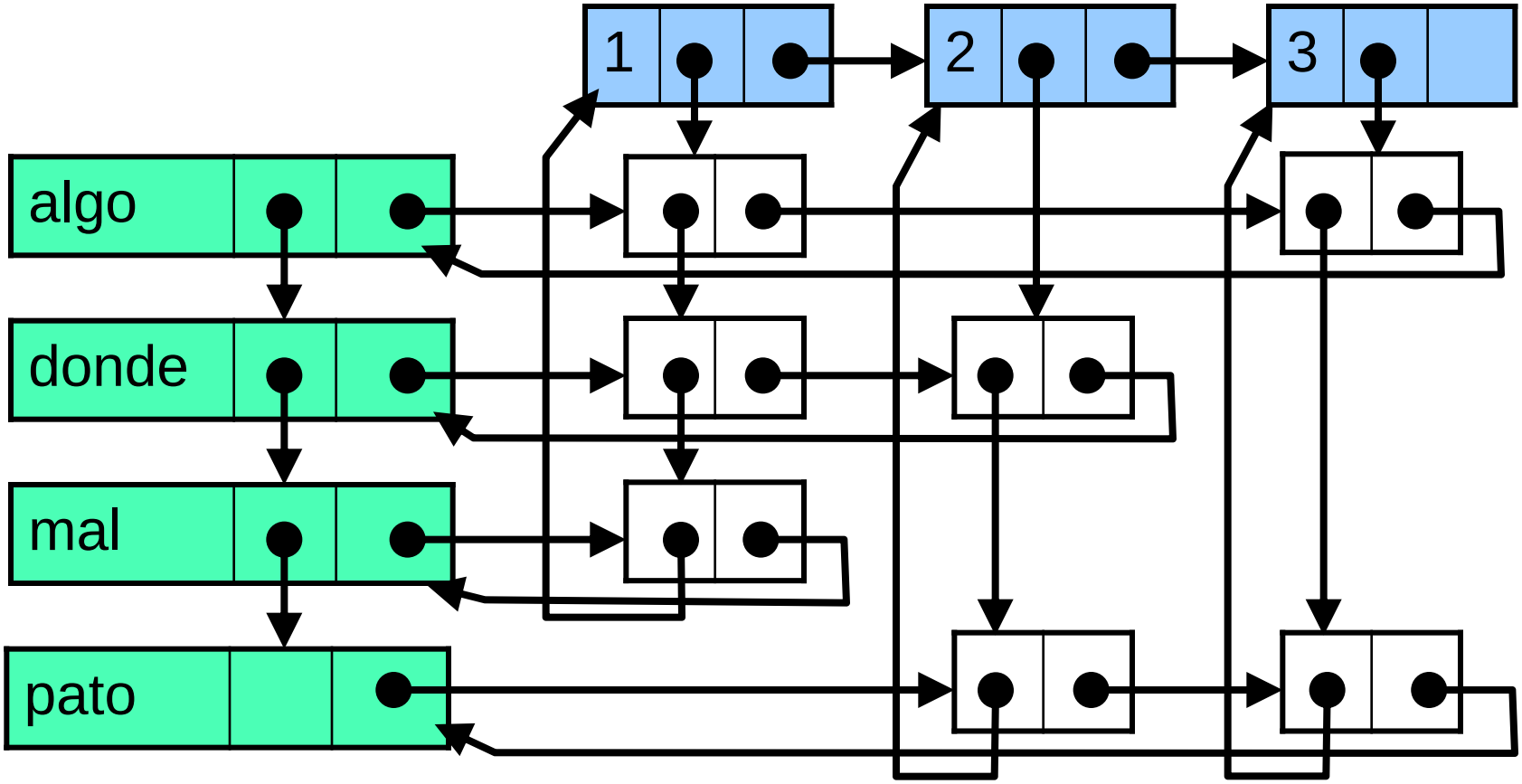
	1	2	3
algo	X		X
donde	X	X	
mal	X		
pato		X	X

- Uso de memoria: $3 \cdot 10^9$ x $2 \cdot 10^7$ bits = $6/8 \cdot 10^{16}$ bytes = ¡¡75.000 Terabytes!! ¡¡50.000 veces más memoria!!
- Sólo 1 de cada 700.000 celdas será true.
- **Buscar páginas dada una palabra:** recorrer una fila: $3 \cdot 10^9$ elementos.
- **Buscar palabras en una página dada:** recorrer una columna: $2 \cdot 10^7$ elementos.

2.5.2. Listas múltiples

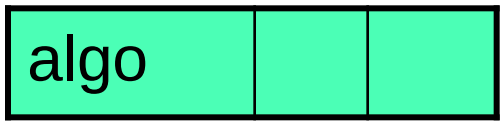
- Ninguna estructura, por sí misma, proporciona un buen tiempo de ejecución con un uso de memoria razonable.
- Con listas, una operación es rápida y la otra muy ineficiente.
- **Solución:** combinar las dos estructuras de listas en una sola.
  - Listas de palabras en una página: sig_pal.
  - Lista de páginas en una palabra: sig_pag.
  - Las listas son circulares.

2.5.2. Listas múltiples



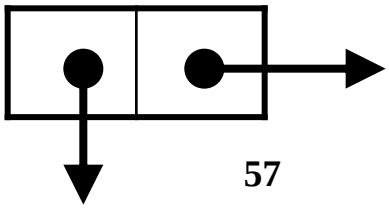
- Celdas de la estructura de listas múltiples:

palabra sig_pal pri_pag



pagina pri_pal sig_pag

sig_pal sig_pag



2.5.2. Listas múltiples

- **Buscar páginas dada una palabra, pal**
  - Recorrer la lista horizontal empezando en **pal.pri_pag**.
  - Para cada elemento recorrer verticalmente hasta llegar (circularmente) a una página.
- **Buscar palabras en una página dada, pag**
  - Recorrer la lista vertical empezando en **pag.pri_pal**.
  - Para cada elemento recorrer horizontalmente hasta llegar (circularmente) a una palabra.

2.5.2. Listas múltiples

Uso de memoria

- Sea k_1 = tamaño de 1 puntero = 8 bytes
- Cada celda ocupa: $2 \cdot k_1$ = 16 bytes
- En total hay $9 \cdot 10^{10}$ bytes.
- Memoria necesaria: $2 \cdot k_1 \cdot 9 \cdot 10^{10}$ bytes = 1.44 Terabytes = Lo mismo que con listas simples

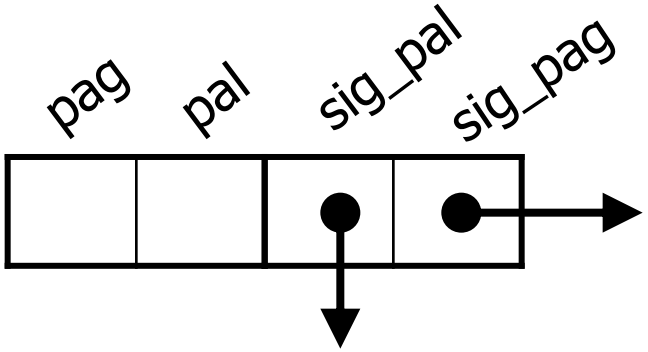
Eficiencia de las operaciones

- **Buscar páginas dada una palabra:** Tamaño lista horizontal (promedio) * Tamaño lista vertical (promedio) = 50*4.500 = 225.000 celdas recorridas.
- **Buscar palabras en una página dada:** Tamaño lista vertical (promedio) * Tamaño lista horizontal (promedio) = 4.500*50 = 225.000 celdas recorridas.

2.5.2. Listas múltiples

Conclusiones

- Las listas simples por separado presentan problemas en una u otra operación.
- Usando listas múltiples conseguimos operaciones eficientes, con un uso de memoria razonable.
- Problema general: representación de matrices *escasas*.
- Añadiendo información redundante en las listas es posible mejorar la eficiencia, a costa de usar más memoria.



2.5.2. Listas múltiples

Conclusión general

- Es algunas aplicaciones es posible, y adecuado, **combinar** varias estructuras de datos en una sola.
- Sobre unos mismos datos podemos tener diferentes estructuras de acceso: **estructuras de datos múltiples**.
- Normalmente, estas estructuras mejoran la eficiencia a costa de usar más memoria.