

Programa de teoría

AED I. Estructuras de Datos

1. Abstracciones y especificaciones

→ 2. Conjuntos y diccionarios

3. Representación de conjuntos mediante árboles

4. Grafos

AED II. Algorítmica

1. Análisis de algoritmos

2. Divide y vencerás

3. Algoritmos voraces

4. Programación dinámica

5. Backtracking

6. Ramificación y poda

AED I: ESTRUCTURAS DE DATOS

Tema 2. Conjuntos y Diccionarios

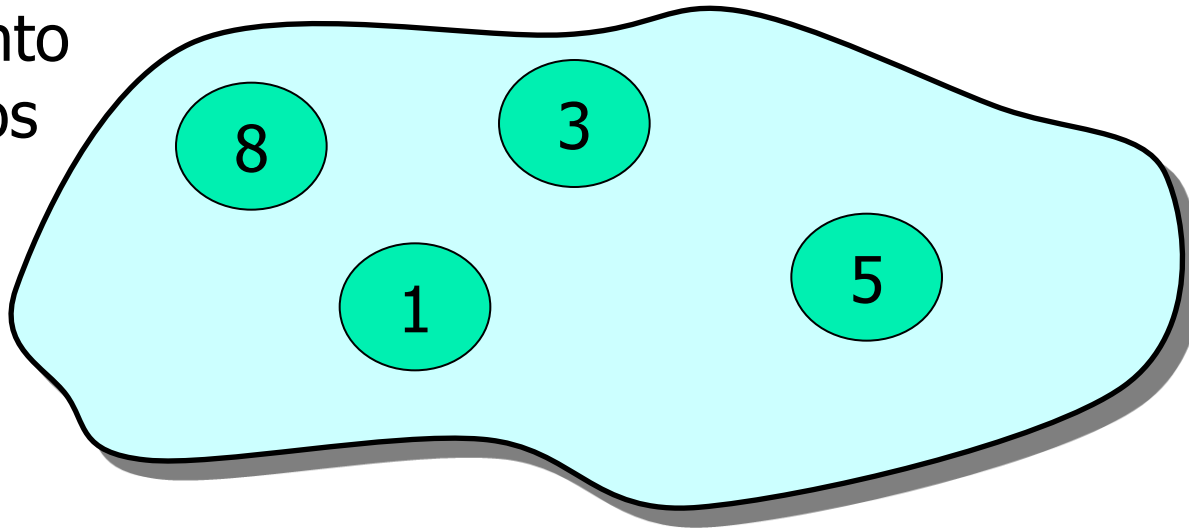
- 2.1. Repaso del TAD Conjunto
- 2.2. Implementaciones básicas
- 2.3. El TAD Diccionario
- 2.4. Las tablas de dispersión

2.1. Repaso del TAD Conjunto

Definición y propiedades

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

C: conjunto de enteros



2.1. Repaso del TAD Conjunto

- En programación se impone que todos los elementos sean del mismo tipo:
Conjunto[T] (conjuntos de enteros, de caracteres, de cadenas ...)
- ¿En qué se diferencia el TAD Conjunto del TAD Lista?
- ¿En qué se diferencia el TAD Conjunto del TAD Bolsa?

2.1. Repaso del TAD Conjunto

- Puede existir una relación de orden en el conjunto.
- **Relación “<” de orden en un conjunto C:**
 - **Propiedad transitiva:** para todo a, b, c , si $(a < b)$ y $(b < c)$ entonces $(a < c)$.
 - **Orden total:** para todo a, b , solo una de las afirmaciones $(a < b)$, $(b < a)$ o $(a = b)$ es cierta.
- ... colección no ordenada... → se refiere al orden de inserción de los elementos.

2.1. Repaso del TAD Conjunto

Repaso de Notación de Conjuntos

- **Definición:**

Por extensión

$$A = \{a, b, c, \dots, z\}$$

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

Mediante proposiciones

$$C = \{x \mid \text{proposición de } x\}$$

$$D = \{x \mid x \text{ es primo y menor que } 90\}$$

- **Pertenencia:** $x \in A$

- **No pertenencia:** $x \notin A$

- **Conjunto vacío:** \emptyset

- **Conjunto universal:** \mathcal{U}

- **Inclusión:** $A \subseteq B$

- **Intersección:** $A \cap B$

- **Unión:** $A \cup B$

- **Diferencia:** $A - B$

2.1. Repaso del TAD Conjunto

Operaciones más comunes

C: Conjunto de todos los Conjunto[T]

$a, b, c \in C; \quad x \in T$

- Vacío : $\rightarrow C$ $a := \emptyset$
- Unión : $C \times C \rightarrow C$ $c := a \cup b$
- Intersección : $C \times C \rightarrow C$ $c := a \cap b$
- Diferencia : $C \times C \rightarrow C$ $c := a - b$
- Combina : $C \times C \rightarrow C$ $c := a \cup b,$
con $a \cap b = \emptyset$
- Miembro : $T \times C \rightarrow B$ $x \in a$

2.1. Repaso del TAD Conjunto

Operaciones más comunes

- Inserta : $T \times C \rightarrow C$ $a := a \cup \{x\}$
- Suprime : $T \times C \rightarrow C$ $a := a - \{x\}$
- Min : $C \rightarrow T$ $\min_{\forall x \in a}(x)$
- Max : $C \rightarrow T$ $\max_{\forall x \in a}(x)$
- Igual : $C \times C \rightarrow B$ $a == b$

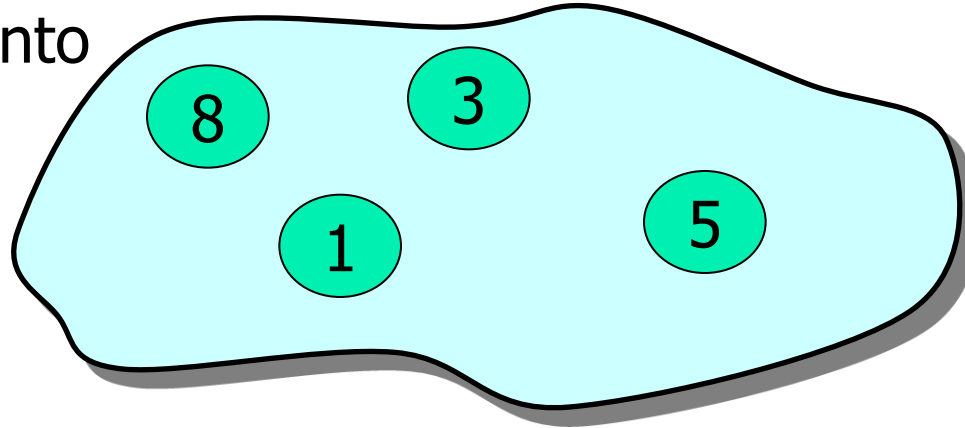
- ... elementos distintos... \rightarrow si insertamos un elemento que ya pertenece, obtenemos el mismo conjunto.

2.2. Implementaciones básicas

- **Problema:** ¿cómo representar el tipo conjunto de forma que las operaciones se ejecuten rápidamente, con un uso razonable de memoria?
- **Respuesta:** en este tema y el siguiente.
- Dos tipos de **implementaciones básicas:**
 - Mediante arrays de booleanos.
 - Mediante listas de elementos.
- La mejor implementación depende de cada aplicación concreta:
 - Operaciones más frecuentes en esa aplicación.
 - Tamaño y variabilidad de los conjuntos usados.
 - Etc.

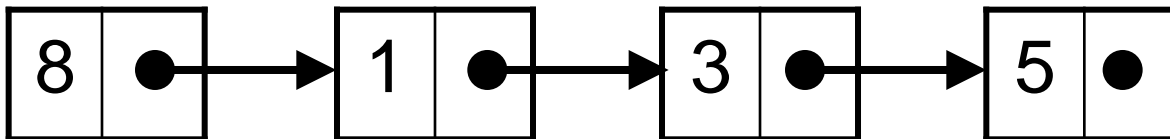
2.2. Implementaciones básicas

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:** $T = \{a, b, \dots, g\}$

$C = \text{Conjunto}[T]$

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

$B = \{c, e, f, g\}$

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**

$C[i] := A[i] \text{ OR } B[i]$

operación Intersección (A, B: Conjunto[T]; **var** C: Conjunto[T])

para cada i en Rango(T) **hacer**

$C[i] := A[i] \text{ AND } B[i]$

operación Diferencia (A, B: Conjunto[T]; **var** C: Conjunto[T])

para cada i en Rango(T) **hacer**

$C[i] := A[i] \text{ AND NOT } B[i]$

2.2.1. Mediante arrays de booleanos

operación Inserta ($x: T$; **var** $C: \text{Conjunto}[T]$)

$C[x] := 1$

operación Suprime ($x: T$; **var** $C: \text{Conjunto}[T]$)

$C[x] := 0$

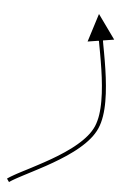
operación Miembro ($x: T$; $C: \text{Conjunto}[T]$): booleano

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.
 - No hace falta usar memoria dinámica.
 - El tamaño usado es **proporcional al tamaño del conjunto universal**, independientemente de los elementos que contenga el conjunto.
 - ¿Ventaja o inconveniente?
- 

2.2.1. Mediante arrays de booleanos

- **Ejemplo.** Implementación en C, con $T = \{1, 2, \dots, 64\}$

tipo

Conjunto[T] = long long

- Unión (A, B, C) $\rightarrow C = A | B;$
- Intersección (A, B, C) $\rightarrow C = A \& B;$
- Inserta (x, C) $\rightarrow C = C | (1 \ll (x-1));$
- ¡Cada conjunto ocupa 8 bytes, y las operaciones se hacen en 1 o 3 ciclos de la CPU!

2.2.1. Mediante arrays de booleanos

- **Ejemplo.** Implementación con

$T = \text{enteros de 32 bits} = \{0, 1, \dots, 2^{32}-1\}$

tipo

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

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

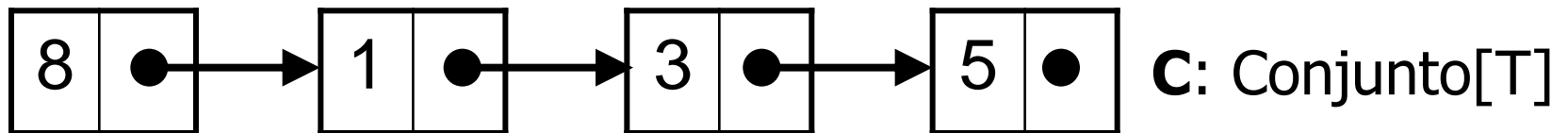
2.2.2. Mediante listas de elementos

- **Idea:** guardar en una lista los elementos que pertenecen al conjunto.
- **Definición:**

tipo

$\text{Conjunto}[T] = \text{Lista}[T]$

- $C = \{1, 5, 8, 3\}$



2.2.2. Mediante listas de elementos

Ventajas:

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

Inconvenientes:

- Las operaciones son menos eficientes si el conjunto universal es reducido.
- Gasta más memoria y tiempo si los conjuntos están muy llenos.
- Más complejo de implementar.

2.2.2. Mediante listas de elementos

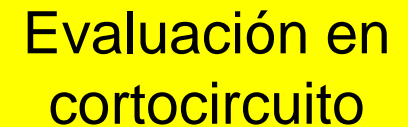
operación Miembro (x: T; C: Conjunto[T]): booleano

Primero(C)

mientras NOT EsUltimo(C) AND Actual(C) \neq x **hacer**

Avanzar(C)

devolver NOT EsUltimo(C)



Evaluación en
cortocircuito

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)

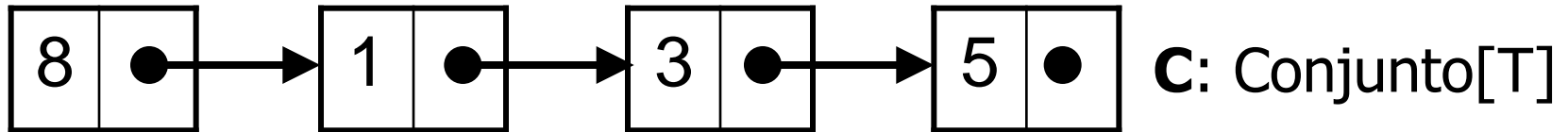
finmientras

2.2.2. Mediante listas de elementos

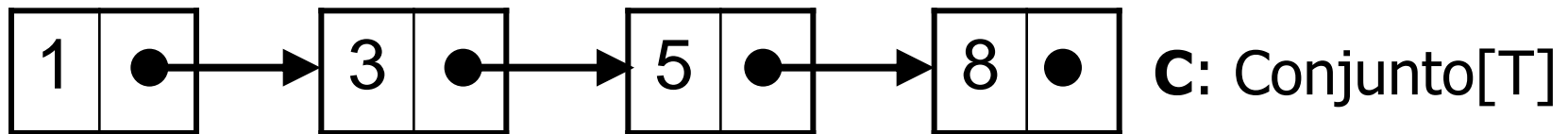
- ¿Cuánto tiempo tardan las operaciones anteriores? Suponemos una lista de tamaño n y otra m (o ambas de tamaño n).
- ¿Cómo sería Unión, Diferencia, Inserta, Suprime, etc.?
- **Inconveniente:** Unión, Intersección y Diferencia recorren la lista B muchas veces (una por cada elemento de A).
- Se puede mejorar usando listas ordenadas.

2.2.2. Mediante listas de elementos

- Listas no ordenadas



- Listas ordenadas



- **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

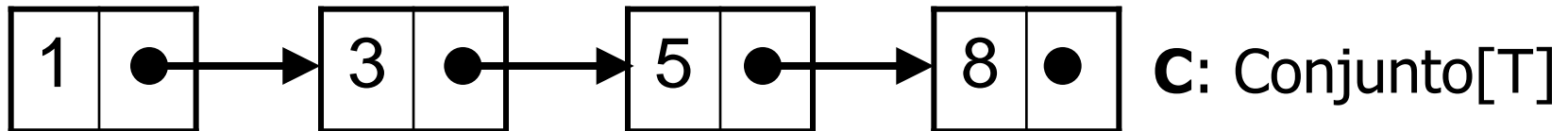
operación Miembro ($x: T; C: \text{Conjunto}[T]$): booleano

Primero(C)

mientras NOT EsUltimo(C) AND Actual(C) < x **hacer**

Avanzar(C)

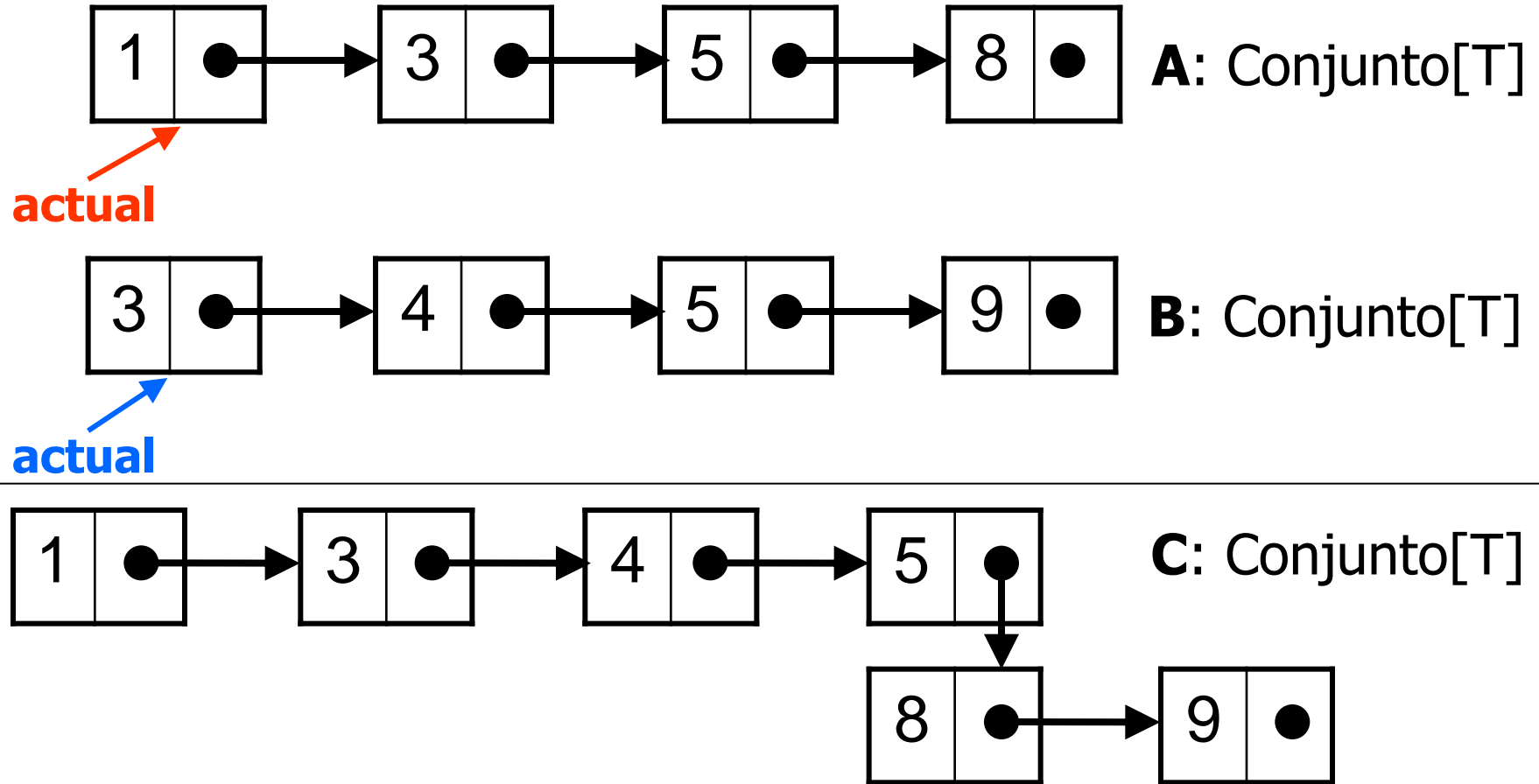
devolver NOT EsUltimo(C) AND Actual(C) == x



- ¿Cuánto es el tiempo de ejecución ahora?

2.2.2. Mediante listas de elementos

- **Unión:** idea parecida al procedimiento de mezcla, en la ordenación por mezcla.



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 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

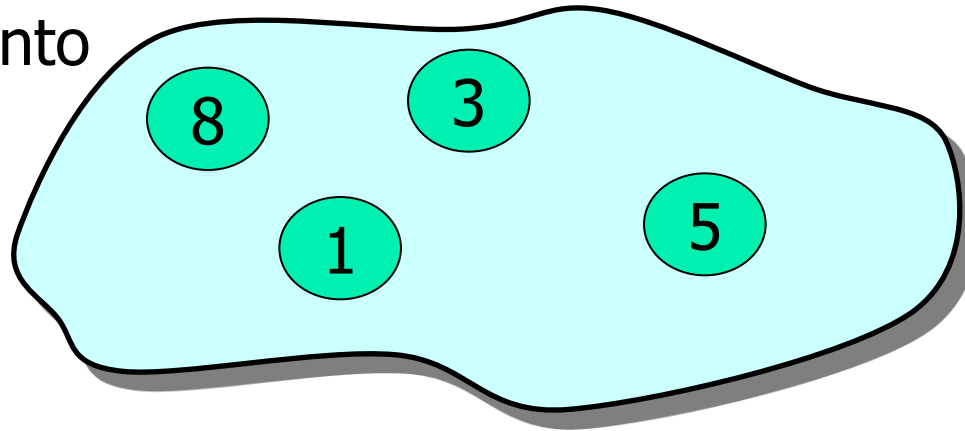
- **Arrays de booleanos:** muy rápida para las operaciones de inserción y consulta.
- Inviabile si el tamaño del conjunto universal es muy grande.
- **Listas de elementos:** uso razonable de memoria, proporcional al tamaño usado.
- Muy ineficiente para la inserción y consulta de un elemento.
- **Solución:**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

2.3. El TAD Diccionario

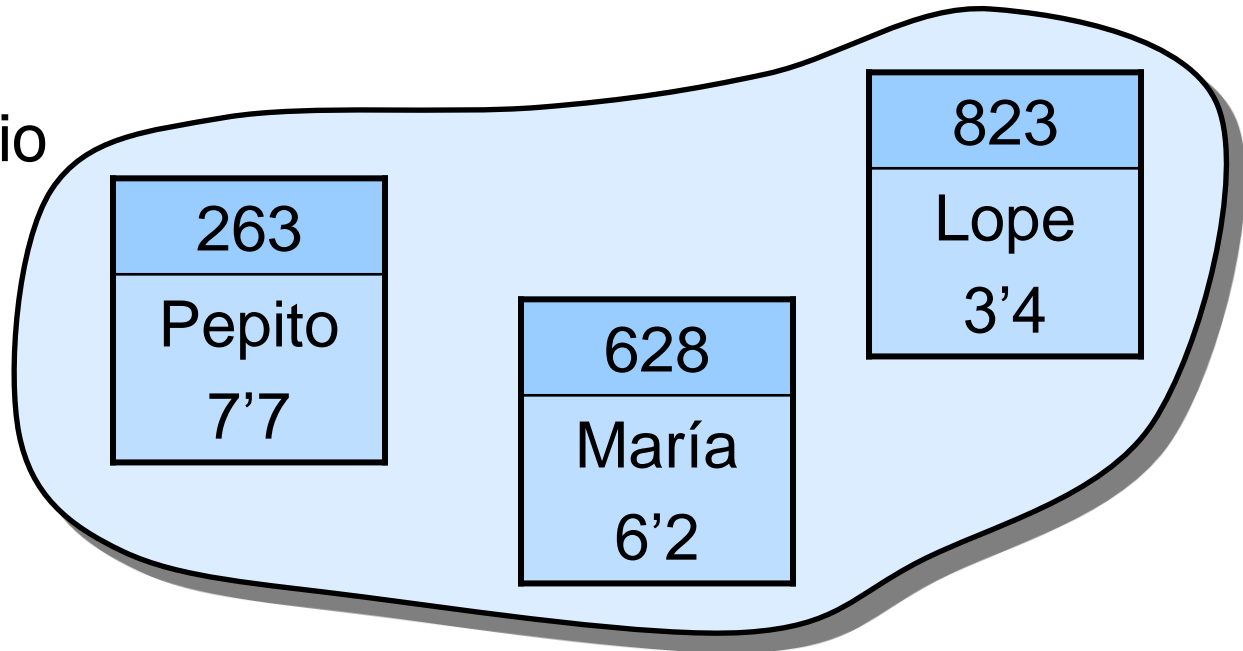
- Muchas aplicaciones usan **conjuntos** de datos, que pueden variar en tiempo de ejecución.
- Cada elemento tiene una **clave**, y asociado a ella se guardan una serie de **valores**.
- Las operaciones de **consulta son por clave**.
- **Ejemplos:** agenda electrónica, diccionario de sinónimos, base de datos de empleados, notas de alumnos, etc.
- Normalmente, no son frecuentes las operaciones de unión, intersección o diferencia, sino inserciones, consultas y eliminaciones.

2.3. El TAD Diccionario

C: Conjunto



D: Diccionario



2.3. El TAD Diccionario

- **Definición: asociación.** Una asociación es un par (clave: tipo_clave; valor: tipo_valor).

clave

263

valor

Pepito

7'7

- Un **diccionario** es, básicamente, un conjunto de asociaciones con las operaciones Inserta, Suprime, Miembro y Vacío.
- 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])
Vacío (**var D**: Diccionario[tcl,tval])

2.3. El TAD 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 TAD Diccionario

operación Vacío (**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 TAD 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 \neq 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

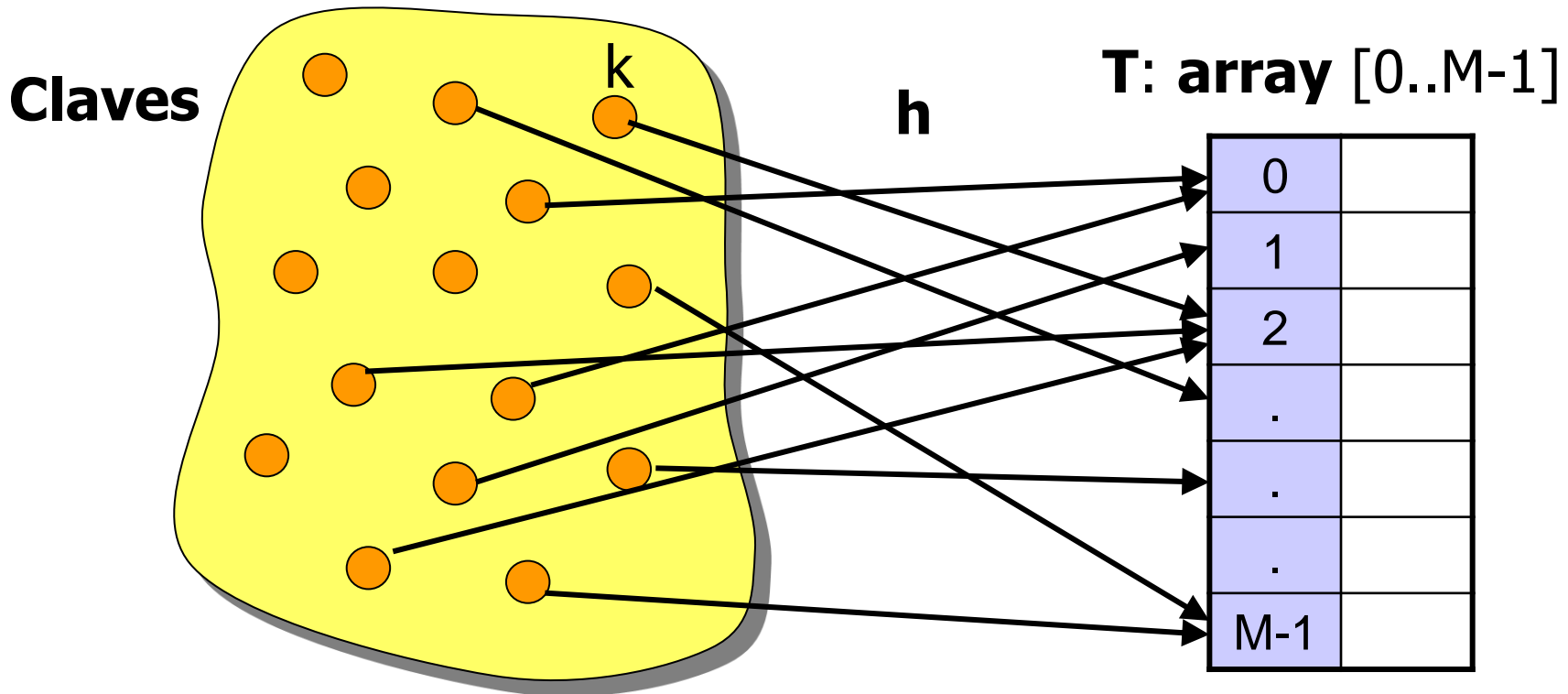
finsi

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 arrays de booleanos 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 ($0, \dots, M-1$).
- 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

$$h : \text{tipo_clave} \rightarrow [0, \dots, M-1]$$

- **Insertar (clave, valor, T):** aplicar $h(\text{clave})$ y almacenar en esa posición **valor**.

$$T[h(\text{clave})] := \text{valor}$$

- **Consultar (clave, T): valor:** devolver la posición de la tabla en $h(\text{clave})$.

$$\text{devolver } T[h(\text{clave})]$$

- Se consigue **$O(1)$** , en teoría...

2.4. Las tablas de dispersión

- **Ejemplo.** tipo_clave = entero de 32 bits.
Fun. de disp.: $h(k) = (37 \cdot k^2 + 61 \cdot k \cdot \text{sqrt}(k)) \bmod M$
Más sencilla: $h(k) = k \bmod M$
- Sea $M=10$, $D= \{9, 25, 33, 976, 285, 541, 543, 2180\}$
- $h(k) = k \bmod 10$

D

0	1	2	3	4	5	6	7	8	9

Habemos problema



2.4. Las tablas de dispersión

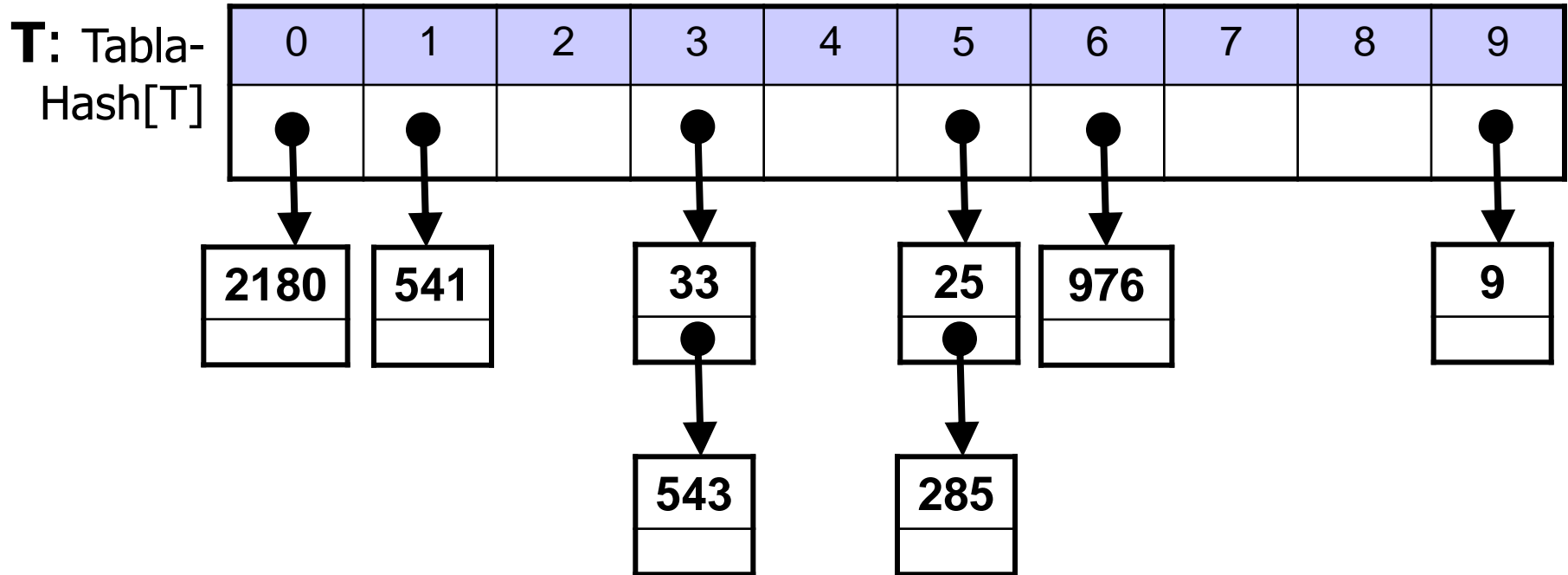
- ¿Qué ocurre si para dos elementos distintos x e y , ocurre que $h(x) = h(y)$?
- **Definición:** si $(x \neq y)$ y $(h(x) = h(y))$ entonces se dice que x e y son **sinónimos**.
- Los distintos métodos de dispersión difieren en el tratamiento de los sinónimos.
- **Tipos de dispersión (hashing):**
 - **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 $M = 10$, $D = \{9, 25, 33, 976, 285, 541, 543, 2180\}$
- $h(k) = k \bmod 10$



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 un puntero = k_1 bytes, un elemento = k_2 bytes.
- En las celdas: $(k_1 + k_2)n$
- En la tabla: $k_1 M$

Conclusión

Menos cubetas: se gasta menos memoria.

Más cubetas: operaciones más rápidas.

2.4.2. Dispersión cerrada

- Las celdas de la tabla 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)

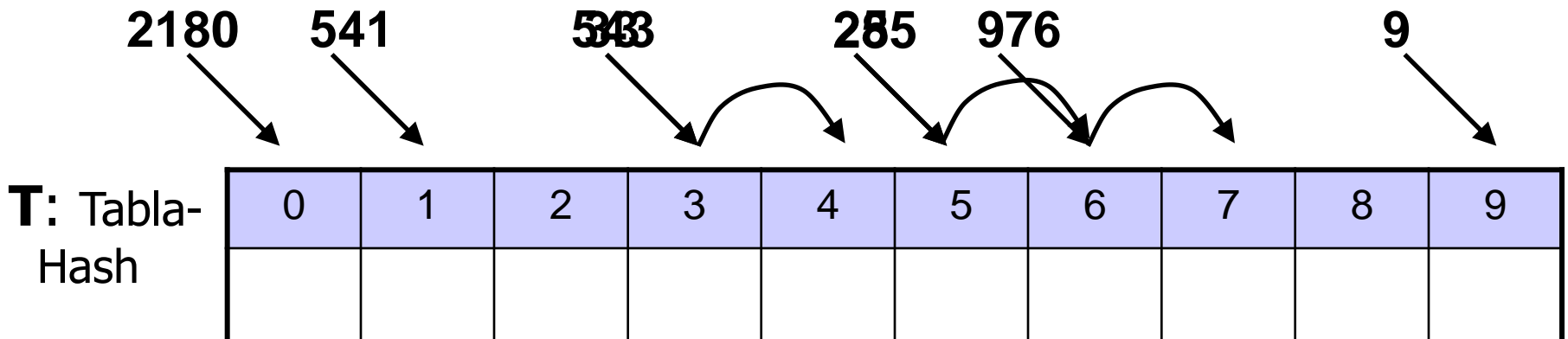
- Si al insertar un elemento nuevo **k**, ya está ocupado **h(k)**, se dice que ocurre una **colisión**.
- En caso de colisión se hace **redispersión**: buscar una nueva posición donde meter el elemento **k**.

2.4.2. Dispersión cerrada

- **Redispersión:** si falla $h(k)$, aplicar $h_1(k)$, $h_2(k)$, ... hasta encontrar una posición libre.
- Definir la familia de funciones $h_i(k)$.
- **Ejemplo. Redispersión lineal:**

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

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



- ¿Dónde iría a para el 99? ¿Y luego el 12? ¿Y...?

2.4.2. Dispersión cerrada

- La secuencia de posiciones recorridas para un elemento se suele denominar **cadena** o **secuencia de búsqueda**.

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

p:= h(clave)

i:= 0

mientras T[p].clave ≠ VACIO AND T[p].clave ≠ clave

AND i<M **hacer**

i:= i + 1

p:= h_i(clave)

finmientras

si T[p].clave == clave **entonces**

devolver T[p].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.

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

- **Moraleja:** 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.

285

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

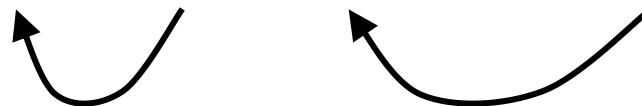
Resultado: ¡¡Encontrado 285 en la tabla!!

- ¿Cómo sería consultar 13? ¿E insertar 13?

2.4.2. Dispersión cerrada

- En la operación Consulta, la búsqueda continúa 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.

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



2.4.2. Dispersión cerrada

Utilización de memoria en disp. cerrada

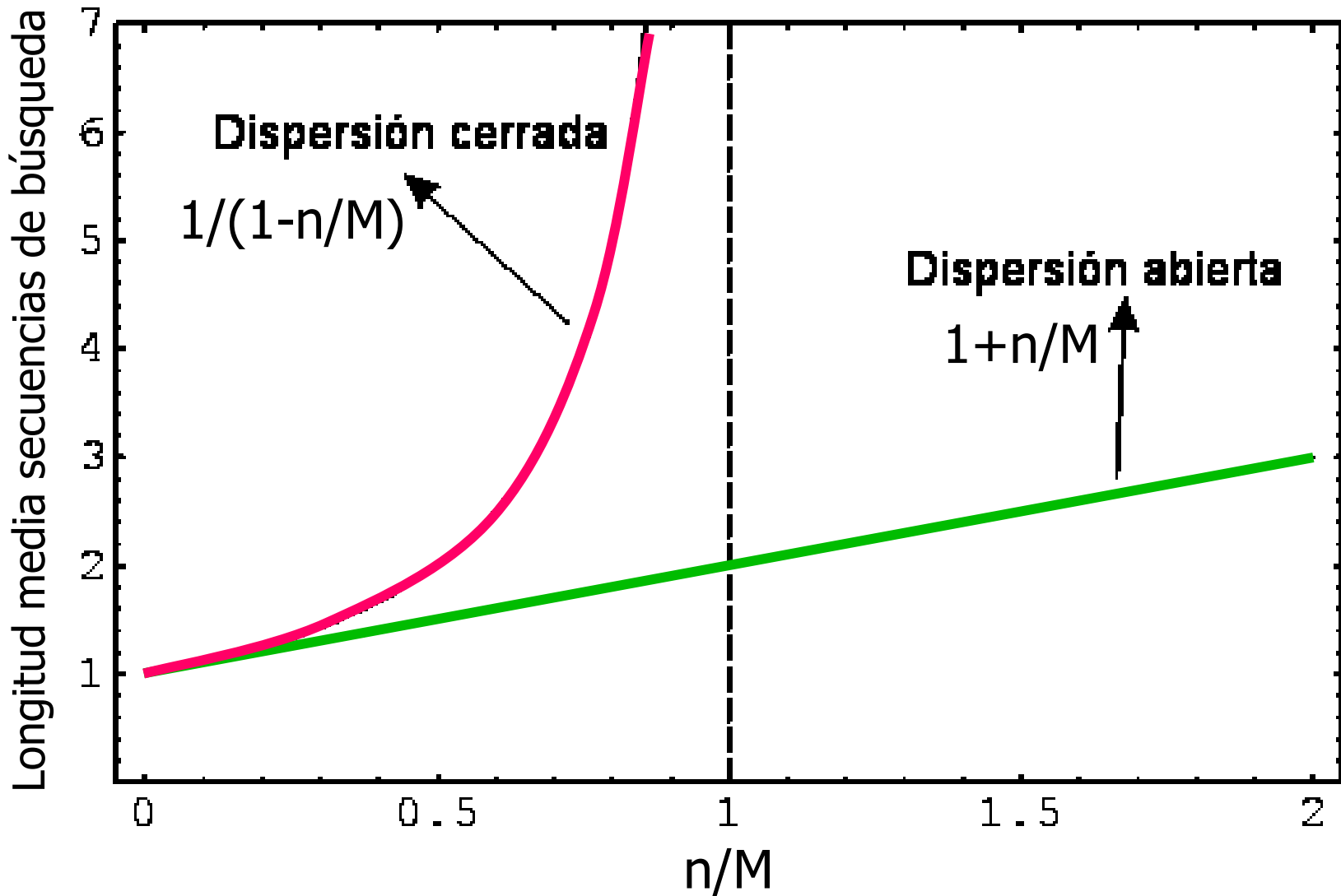
- Si un puntero = k_1 bytes, un elemento = k_2 bytes.
- Memoria en la tabla: $k_2 M$
- O bien: $k_1 M + k_2 n$
- En **dispersión abierta** teníamos:
 $k_1 M + (k_1 + k_2)n$
- ¿Cuál es mejor?

2.4.2. Dispersión cerrada

Eficiencia de las operaciones

- La tabla nunca se puede llenar con más de M elementos.
- La probabilidad de colisión crece cuantos más elementos haya, 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**.
- Por ejemplo:
 - Dispersión abierta: reestructurar si $n > 2 M$
 - Dispersión cerrada: reestructurar si $n > 0'75 M$

2.4.3. Funciones de dispersión

- En ambos análisis se supone una “buena” función de dispersión.
- Si no es buena, el tiempo puede ser mucho mayor...
- **Propiedades de una buena función de dispersión**
 - Repartir los elementos en la tabla de manera **uniforme**: debe ser lo más “aleatoria” posible.
 - La función debe ser fácil de calcular (eficiente).
 - **Ojo**: $h(k)$ es función de k , devuelve siempre el mismo valor para un mismo valor de k .

2.4.3. Funciones de dispersión

Funciones de dispersión con números:

Sea la clave k un número entero.

- **Método de división.**

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

- Cuidado. ¿Qué valores son adecuados para M ?
- Ejemplo. $k = 261, 871, 801, 23, 111, 208, 123, 808$
- $M = 10, \quad h(k) = k \bmod 10 \quad \text{XYZ} \rightarrow Z$
- $M = 100, \quad h(k) = k \bmod 100 \quad \text{XYZ} \rightarrow YZ$
- $M = 11, \quad h(k) = k \bmod 11 \quad \text{XYZ} \rightarrow \text{¿?}$

k	261	871	801	23	111	208	123	808
k mod 11	8	2	9	1	1	10	2	5

- **Conclusión:** es aconsejable que M sea un número primo.

2.4.3. Funciones de dispersión

¿Cómo hacer que la función dependa de todos los dígitos?

- **Método de multiplicación**

$$h(k) = \lfloor C \cdot k \rfloor \bmod M$$

- C debe ser un número real (si es entero, no hace nada)
- $k = 261, 871, 801, 23, 111, 208, 123, 808$
- Ejemplo: $C = 1,11; M = 10$ $\boxed{X}\boxed{Y}\boxed{Z} \rightarrow (X+Y+Z)\%10$

k	261	871	801	23	111	208	123	808
h(k)	9	6	9	5	3	0	6	6

- **Variante:** $h(k) = \lfloor \text{frac}(A \cdot k/W) \cdot M \rfloor$

1. k/W = convertir resultado al rango $[0...1]$
2. $A \cdot$ = convertir al rango $[0...A]$
3. $\text{frac}()$ = quedarse con los decimales $[0...1]$
4. $\lfloor \cdot M \rfloor$ = convertir a un número entero $[0...M-1]$

$\text{frac}()$ = parte decimal
 W = valor máximo del tipo
 A = número coprimo con W

2.4.3. Funciones de dispersión

- $k = 261, 871, 801, 23, 111, 208, 123, 808$
- Ejemplo: $W = 1000$; $A = 3$; $M = 10$

k	261	871	801	23	111	208	123	808
k/1000	0,261	0,871	0,801	0,023	0,111	0,208	0,123	0,808
3·k/1000	0,783	2,613	2,403	0,069	0,333	0,624	0,369	2,424
h(k)	7	6	4	0	3	6	3	4

- **Método de centro del cuadrado**

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

- Ejemplo: $C = 100$. $261^2 = 68121$; $871^2 = 758641$
- Para que funcione bien, se sugiere elegir un C que cumpla aprox.: $C \approx W/\sqrt{M}$

2.4.3. Funciones de dispersión

Funciones de dispersión con secuencias:

- En ocasiones la clave no es un solo número, sino una secuencia de números u otros datos.

- Ejemplo: clave = cadena de caracteres

$$k = \mathbf{x}_1 \mathbf{x}_2 \mathbf{x}_3 \mathbf{x}_4 \mathbf{x}_5 \mathbf{x}_6 \dots$$

0	1	2	3
H	O	L	A
72	79	76	65

Número entero de 32 bits

$$k = \mathbf{x}_1 \mathbf{x}_2 \mathbf{x}_3 \mathbf{x}_4$$

Número real double (64 bits)

$$k = \mathbf{x}_1 \mathbf{x}_2 \mathbf{x}_3 \mathbf{x}_4 \mathbf{x}_5 \mathbf{x}_6 \mathbf{x}_7 \mathbf{x}_8$$

- En general, cualquier tipo de dato se puede tratar como una secuencia de números (los bytes que lo forman).
- Las anteriores funciones no se pueden aplicar directamente (porque solo usan un número). Objetivos:
 - que la función dependa de todos (o muchos) valores \mathbf{x}_i .
 - que genere muchos valores distintos.
 - que no tarde mucho tiempo en calcularse.

2.4.3. Funciones de dispersión

- **Método de suma.** ¡¡Muy malo!! Muchas colisiones.

$$h(k) = \left(\sum_{\forall i} x_i \right) \bmod M$$

- **Método de suma posicional.** Mucho mejor que antes.

$$h(k) = \left(\sum_{\forall i} E^i \cdot x_i \right) \bmod M \quad E = \text{base del exponente}$$

- Ejemplo, $E = 10$. $h(k) = x_1 + 10 \cdot x_2 + 100 \cdot x_3 + 1000 \cdot x_4$
- E puede ser un número primo, no muy grande.

- **Método de plegado (folding).** Suma posic. de trozos.

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

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

$$E \cdot x_5 + x_6$$

$$E^2 \cdot x_6 + E \cdot x_5 + x_4$$

- **Método de extracción.**

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

2.4.3. Funciones de dispersión

- **Métodos hash iterativos**

operación $h(k : \text{tipo_clave}) : \text{entero}$

res := VALOR_INICIAL

para $i := 1$ hasta longitud(k) hacer

res := res · BASE

res := res COMB x_i

finpara

devolver res mod M

- COMB = normalmente es suma o XOR
- BASE = valor multiplicativo (puede hacerse con desplazamiento de bits)
- **Ejemplos:**
 - **Suma posicional** → VALOR_INIC = 0; COMB = + .
 - **djb2** → VALOR_INIC = 5381; BASE = 33; COMB = + .
 - **djb2a** → VALOR_INIC = 5381; BASE = 33; COMB = XOR .
 - **FNV-1** → VALOR_INIC = 2166136261; BASE = 16777619; COMB = XOR .

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 $i = 1, \dots, M-1$.
- **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.

0	1	2	3	4	5	6	7	8	9	M-2	M-1

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**

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

- $D(i) = (+1, -1, +2^2, -2^2, +3^2, -3^2, \dots)$
- Funciona cuando $M = 4R + 3$, para $R \in \mathbf{N}$.
- ¿Resuelve el problema del agrupamiento?

2.4.3. Funciones de redistribución

- **Redistribución doble**

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

- **Idea:** es como una redistribución con saltos de tamaño $C(k)$, donde el tamaño del salto depende de cada k .
- Si M es un número primo, $C(k)$ es una función:
 $C : \text{tipo_clave} \rightarrow [1, \dots, M-1]$
- Se resuelve el problema del agrupamiento si los sinónimos (con igual valor $h(k)$) producen distinto valor de $C(k)$.
- **Ejemplo.** Sea $x = x_1 x_2 x_3 x_4$
 $h(k) = x_1 x_4 \bmod M$
 $C(k) = 1 + (x_3 x_2 \bmod (M-1))$

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 en orden, etc.?