Programa de teoría

Algoritmos y Estructuras de Datos I

1. Abstracciones y especificaciones
2. **Conjuntos y diccionarios**
3. Representación de conjuntos mediante árboles
4. Grafos

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

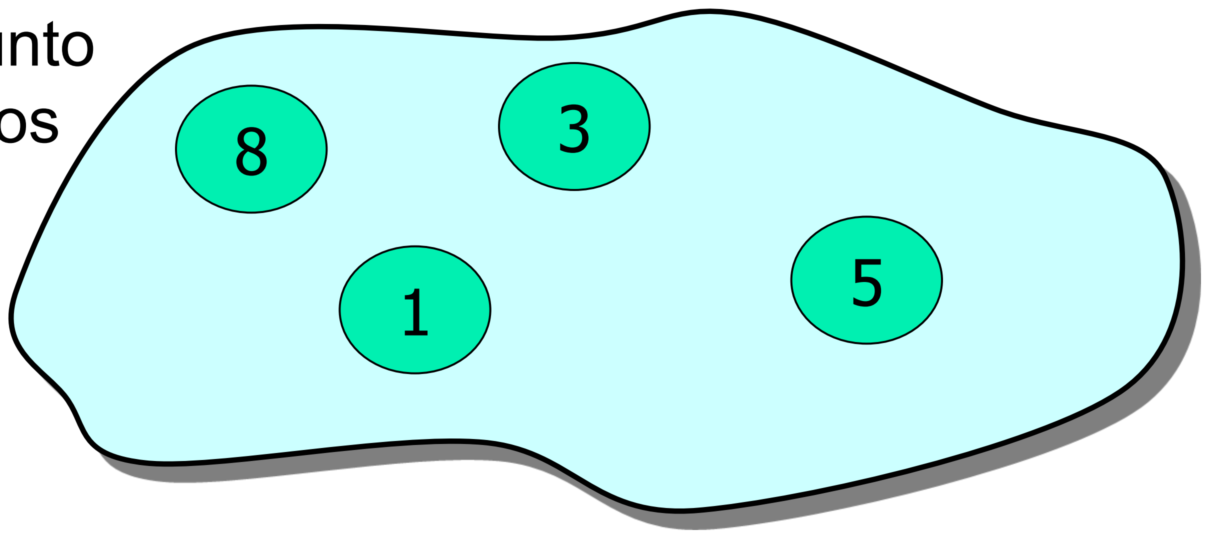


Diagrama de Venn o de patata

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	Mediante proposiciones
A= {a, b, c, ..., z}	C= {x | proposición de x}
B= {1, 4, 7} = {4, 7, 1}	D= {x | x 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;   x $\in$ T

• Vacío : $\rightarrow$ C	a:= $\emptyset$
• Unión : C x C $\rightarrow$ C	c:= a $\cup$ b
• Intersección : C x C $\rightarrow$ C	c:= a $\cap$ b
• Diferencia : C x C $\rightarrow$ C	c:= a – b
• Combina : C x C $\rightarrow$ C	c:= a $\cup$ b, con a $\cap$ b = $\emptyset$
• Miembro : T x C $\rightarrow$ Bool	x $\in$ a

2.1. Repaso del TAD Conjunto.

Operaciones más comunes.

- Inserta : T x C $\rightarrow$ C   a:= a $\cup$ {x}
- Suprime : T x 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 x C $\rightarrow$ Bool   a == b

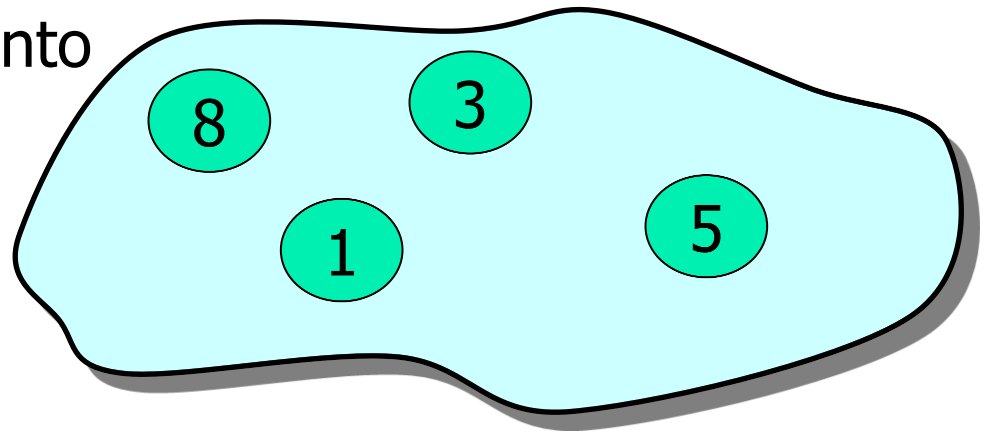
- … elementos distintos… → Si insertamos un elemento que ya pertenece, obtenemos el mismo conjunto.

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.

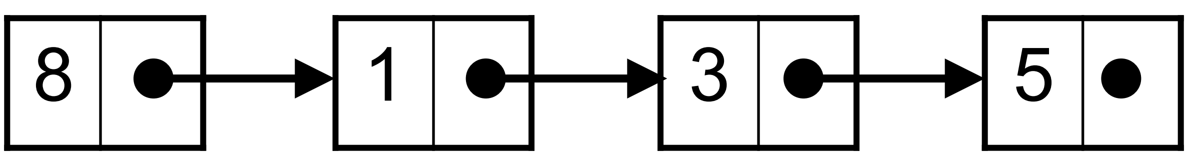
2.2. Implementaciones básicas.

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

± d

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 conj. 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.
- 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/C++, con T = {1, 2, …, 64}
  tipo
  Conjunto[T] = unsigned 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 o 3 ciclos de la CPU!

2.2.1. Mediante arrays de booleanos.

- **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 Gigabytes, independientemente de que contenga solo uno o dos elementos…!
- ¡El tiempo es proporcional a ese tamaño!

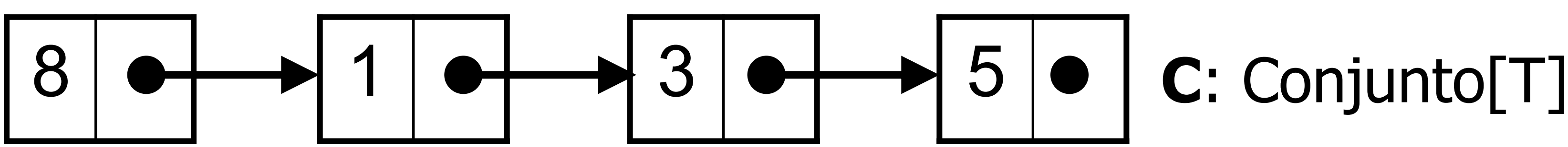
2.2.2. Mediante listas de elementos.

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

tipo

Conjunto[T] = 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 EsFinal(C) AND Actual(C) ≠ x hacer
        Avanzar(C)
    devolver NOT EsFinal(C)
```

Evaluación en cortocircuito

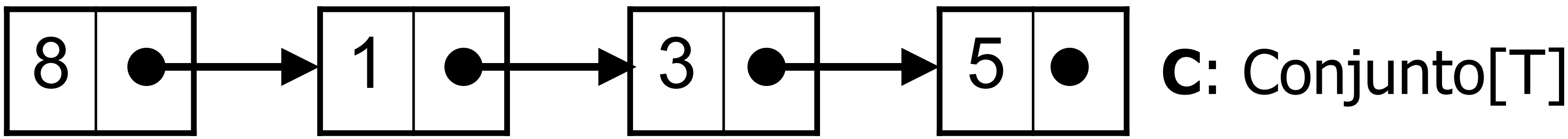
```
operación Intersección (A, B; Conjunto[T]; var C: Conjunto[T])
    C:= ListaVacía
    Primero(A)
    mientras NOT EsFinal(A) hacer
        si Miembro(Actual(A), B) entonces
            InsLista(C, Actual(A))
        Avanzar(A)
    finmientras
```

2.2.2. Mediante listas de elementos.

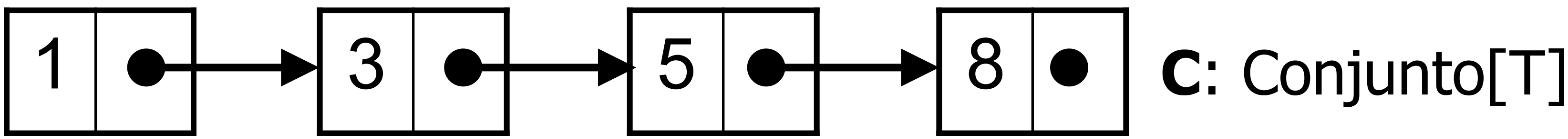
- ¿Cuánto tiempo tardan las operaciones anteriores? Suponemos una lista de tamaño **n** y otra **m** (o ambas de tamaño **n**).
- ¿Cómo serían Unión, Diferencia, Inserta, Suprime, etc.?
- **Inconveniente:** Unión, Intersección y Diferencia recorren la lista B muchas veces (una por cada elemento de A).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

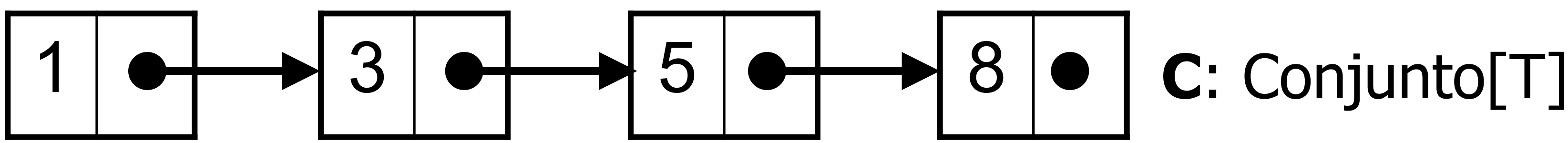
operación Miembro (x: T; C: Conjunto[T]): booleano

  Primero(C)

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

    Avanzar(C)

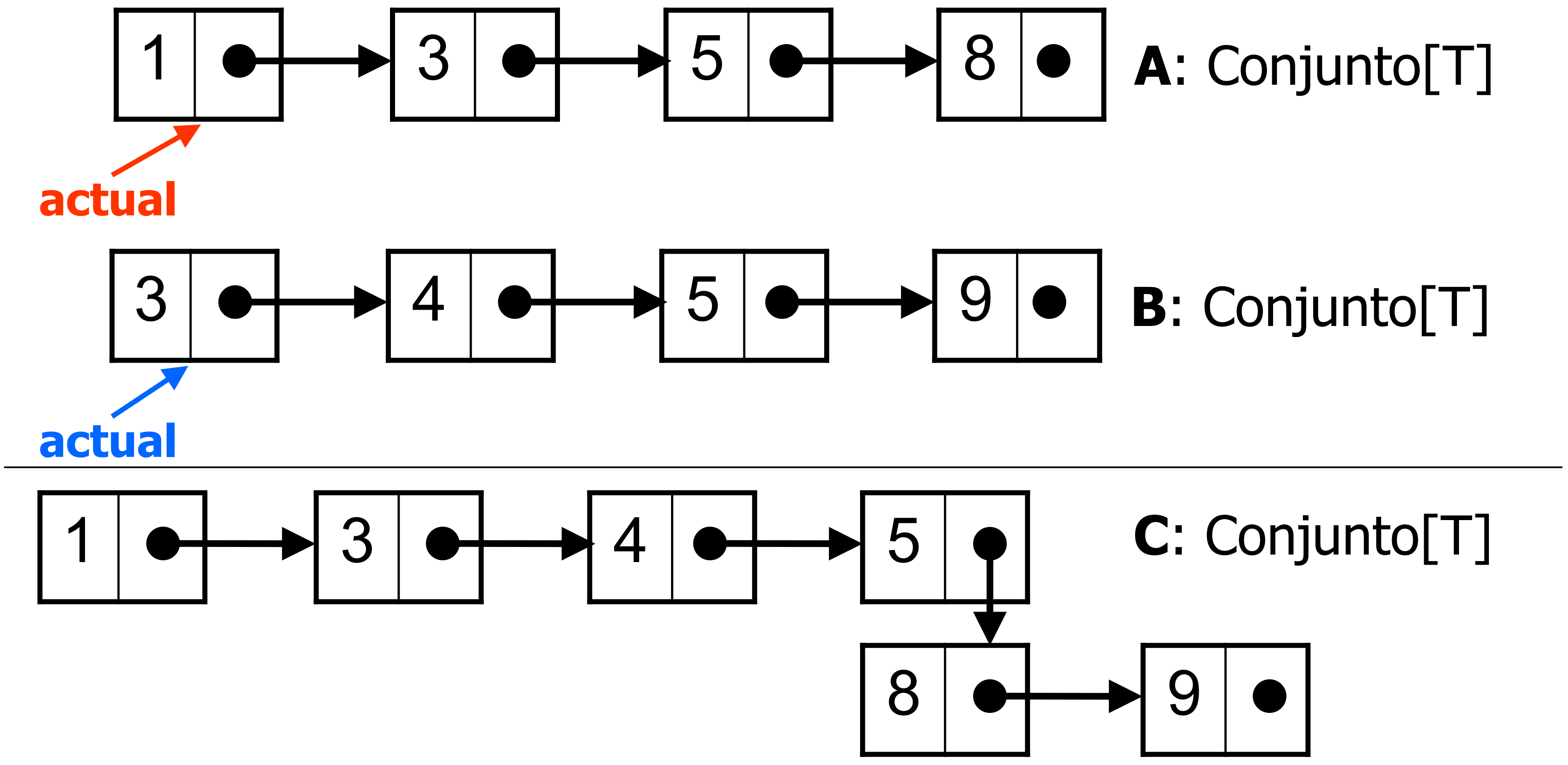
  devolver NOT EsFinal(C) AND Actual(C) == x



- ¿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 (EsFinal(A) AND EsFinal(B)) hacer
        si EsFinal(B) entonces
                InsLista(C, Actual(A))
                Avanza(A)
        sino si EsFinal(A) entonces
                InsLista(C, Actual(B))
                Avanza(B)
        sino si Actual(A) < Actual(B) entonces
                InsLista(C, Actual(A))
                Avanza(A)
        sino si Actual(A) > Actual(B) 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

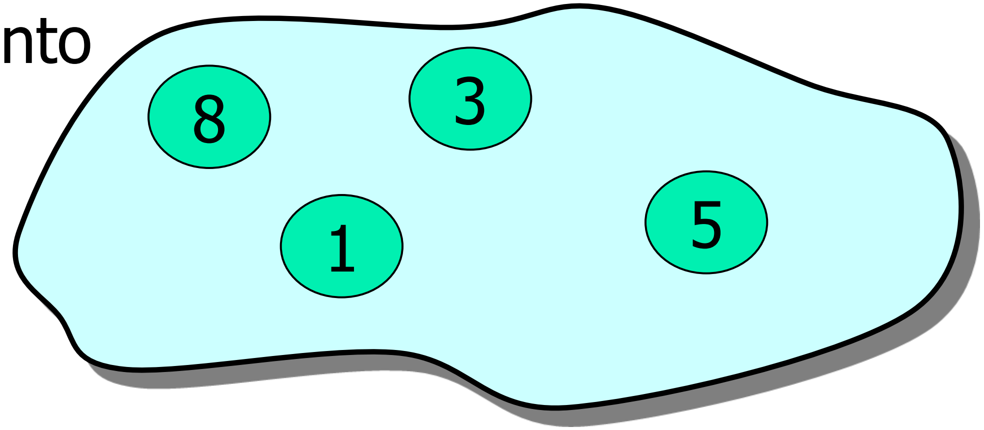
- **Arrays de booleanos**: muy rápida para las operaciones de inserción y consulta.
- Inviable si el tamaño del conjunto universal es muy grande.
- **Listas de elementos**: uso razonable de memoria, proporcional al tamaño usado.
- Muy ineficiente para la inserción y consulta de un elemento.
- **Solución:**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

2.3. El TAD Diccionario.

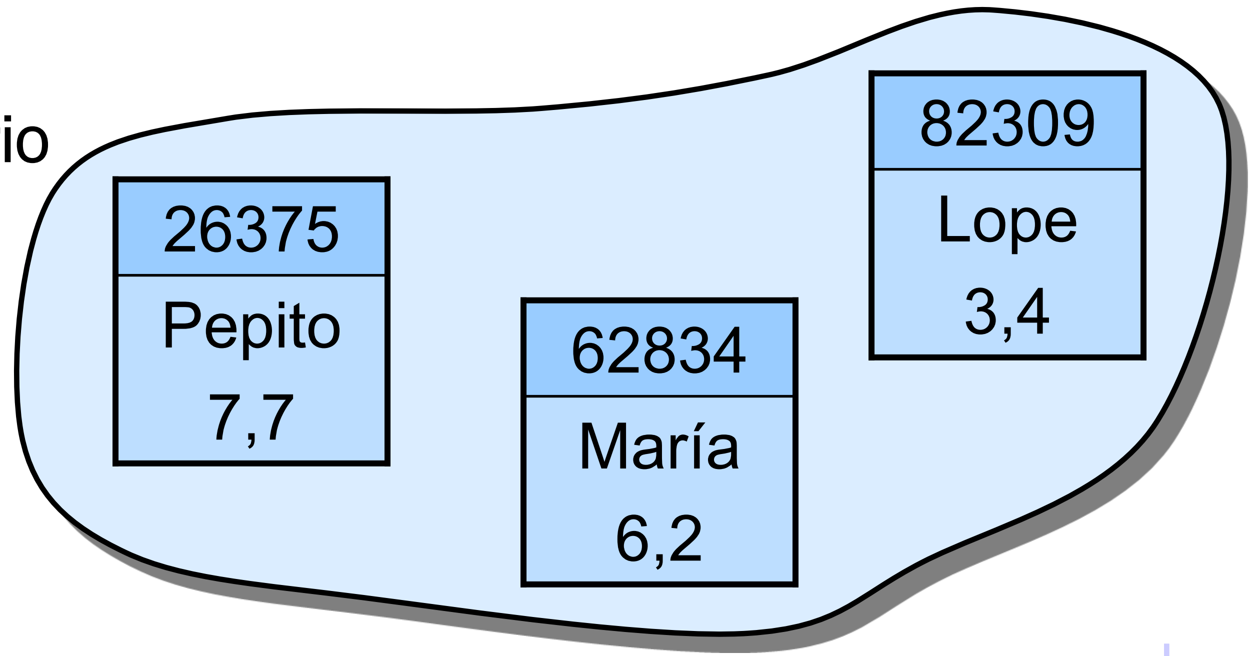
- Muchas aplicaciones usan **conjuntos** de datos, que pueden variar en tiempo de ejecución.
- Cada elemento tiene una **clave**, y asociado a ella se guardan una serie de **valores**.
- Las operaciones de **consulta son por clave**.
- **Ejemplos.** Agenda electrónica, diccionario de sinónimos, base 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	26375
valor	Pepito 7,7

- Un **diccionario** (también llamado **mapa**) es, básicamente, un conjunto de asociaciones con las operaciones Inserta, Suprime, Consulta 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

operac 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.

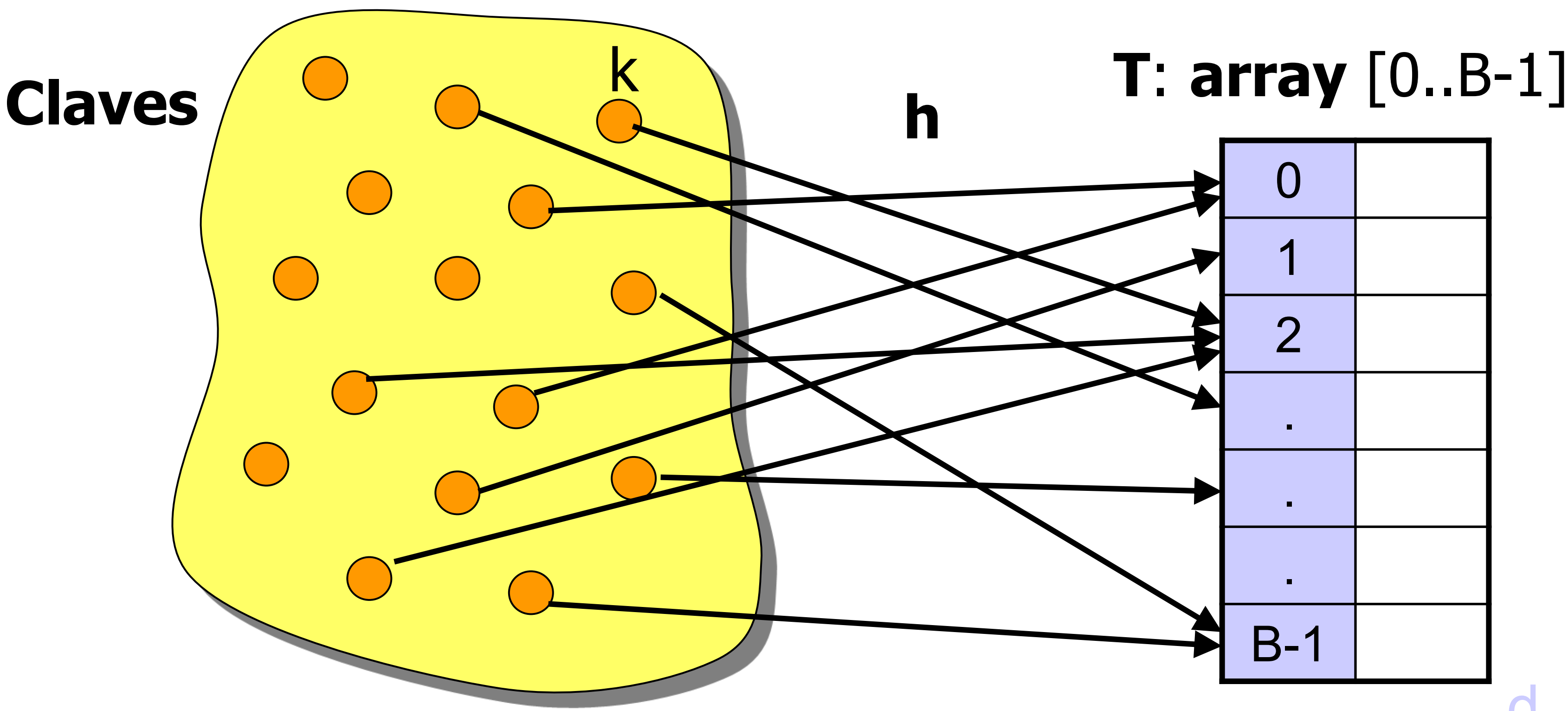
```
operación Consulta (clave: tclave; D: Diccionario[tc,tv]): tvalor
        para i:= 1 hasta D.último hacer
                si  D.datos[i].clave == clave  entonces
                        devolver D.datos[i].valor
        finpara
        devolver NULO

operación Suprime (clave: tclave; var D: Diccionario[tc,tv])
        i:= 1
        mientras (i < D.último) AND (D.datos[i].clave ≠ clave) hacer
                i:= i + 1
        finmientras
        si  i < D.último  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 B posiciones (0, ..., B-1).
- Dada una clave **k** (sea del tipo que sea) calcular la posición donde colocarlo, mediante una función **h**.



± d

2.4. Las tablas de dispersión.

- **Función de dispersión (hash): h**

  h : tipo_clave $\rightarrow$ [0, …, B-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.

- **Ejemplo**. tipo_clave = entero de 32 bits.

  Func. de disp.: $h(k) = (37 \cdot k^2 + 61 \cdot k \cdot sqrt(k)) \bmod B$

  Más sencilla: $h(k) = k$ módulo B

- Sea $B = 10$, $D = \{9, 25, 33, 976, 285, 541, 543, 2180\}$
- $h(k) = k \bmod 10$

D

0	1	2	3	4	5	6	7	8	9

Habemvs problema

2.4. Las tablas de dispersión.

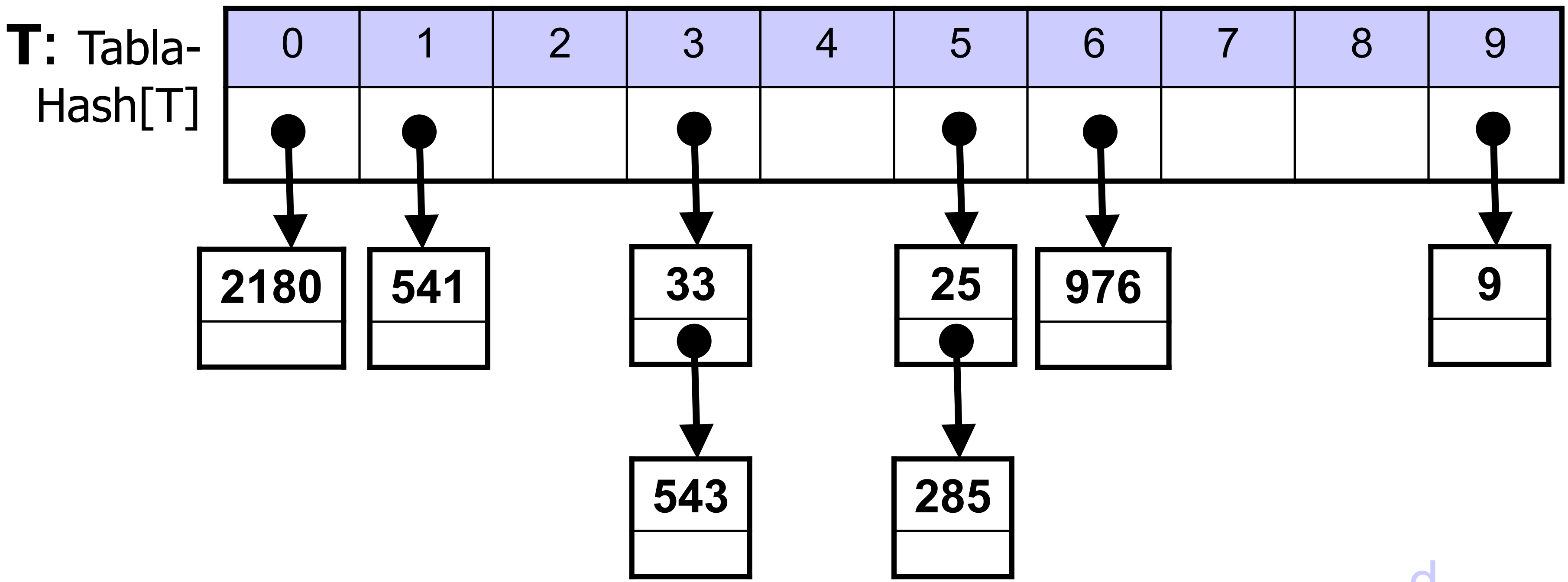
- ¿Qué ocurre si para dos elementos distintos **x** e **y**, ocurre que **h(x) = h(y)**?
- **Definición**: Si (**x** $\neq$ **y**) Y (**h(x) = h(y)**) entonces se dice que **x** e **y** son **sinónimos**.
- Los distintos métodos de dispersión difieren en el tratamiento de los sinónimos.
- **Tipos de 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..B-1] **de** Lista[T]

- Sea B= 10, D= {9, 25, 33, 976, 285, 541, 543, 2180}
- h(k) = k mod 10



2.4.1. Dispersión abierta.

- La tabla de dispersión está formada por B 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 **B** cubetas y **n** elementos en la tabla.
- Si todos los elementos se reparten uniformemente cada cubeta será de longitud: **1 + n/B**

2.4.1. Dispersión abierta.

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

Utilización de memoria

- Si 1 puntero = k_1 bytes, 1 elemento = k_2 bytes.
- En las celdas: $(k_1 + k_2)n$
- En la tabla: k_1 B

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..B-1] **de** (tc, tv)

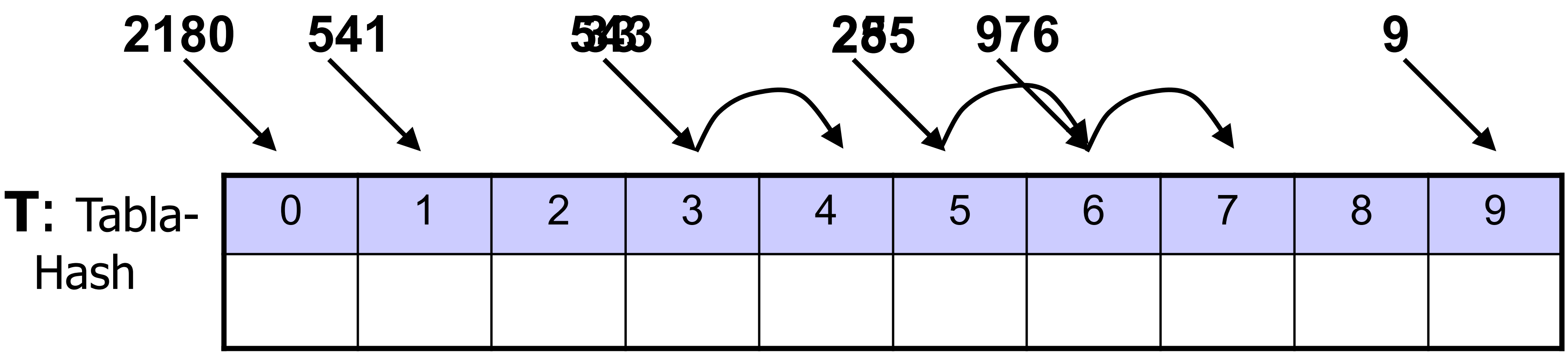
- Si al insertar un elemento nuevo **k**, ya está ocupado **h(k)**, se dice que ocurre una **colisión**.
- En caso de colisión se hace **redispersión**: buscar una nueva posición donde meter el elemento **k**.

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**:

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

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



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

d
±

2.4.2. Dispersión cerrada.

- La secuencia de posiciones recorridas para un elemento se denomina **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<B **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 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.

0	1	2	3	4	5	6	7	8	9
2180	**541**		~~**33**~~	~~**543**~~	~~**25**~~	**976**	~~**235**~~		**9**

2.4.2. Dispersión cerrada.

Utilización de memoria en disp. cerrada

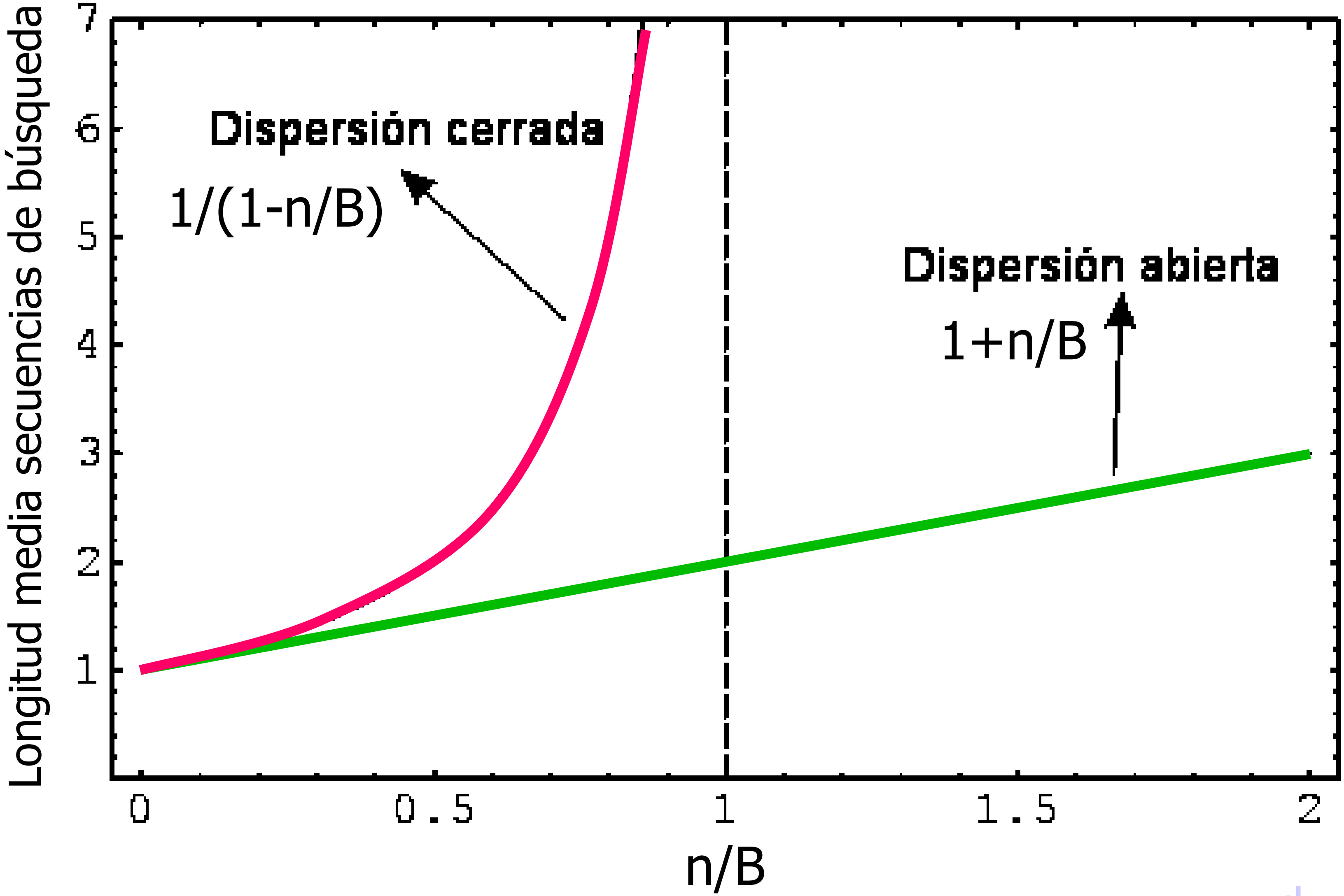
- Si 1 puntero = k_1 bytes, 1 elemento = k_2 bytes.
- Memoria en la tabla: k_2 B
- O bien: k_1 B + k_2 n (array de punteros al dato)
- En **dispersión abierta** teníamos:
  k_1 B + (k_1 + k_2)n
- ¿Cuál es mejor?

2.4.2. Dispersión cerrada.

Eficiencia de las operaciones

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

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, **B**, **reestructurar**.
- Por ejemplo:
  - Dispersión abierta: reestructurar si **n** > 2 **B**
  - Dispersión cerrada: reestructurar si **n** > 0,75 **B**

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 mod B

- Cuidado. ¿Qué valores son adecuados para B?
- Ejemplo. k= 261, 871, 801, 23, 111, 208, 123, 808
- B = 10, h(k) = k mod 10 XY$\boxed{\text{Z}}$ → Z
- B = 100, h(k) = k mod 100 X$\boxed{\text{YZ}}$ → YZ
- B = 11, h(k) = k mod 11 XYZ → ¿?

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

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

2.4.3. Funciones de dispersión.

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

- **Método de multiplicación.**

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

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

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

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; B = 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.**

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

- Ejemplo: C = 100. $261^2 = 6\boxed{81}21$; $871^2 = 75\boxed{86}41$
- Para que funcione bien, se sugiere elegir un C que cumpla aprox.: $C \approx W/\sqrt{B}$

2.4.3. Funciones de dispersión.

Funciones de dispersión con secuencias:

- A veces la clave no es un número, sino una secuencia de números, letras u otros datos.
- Ejemplo: clave = cadena de caracteres
  k = $\mathbf{x_1\, x_2\, x_3\, x_4\, x_5\, x_6}\ldots$

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

- Número entero de 32 bits
  k = $\mathbf{x_1\, x_2\, x_3\, x_4}$

  Número real double (64 bits)
  k = $\mathbf{x_1\, x_2\, x_3\, x_4\, x_5\, x_6\, x_7\, x_8}$
- 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.**

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

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

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

- Ejemplo, E = 10. $h(k) = x_1 + 10\cdot x_2 + 100\cdot x_3 + 1000\cdot x_4$
- E suele ser un número primo.

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

$$h(k) = (x_1\, x_2 + x_3\, x_4 + \boxed{x_5\, x_6}) \bmod B$$

$$h(k) = (x_3\, x_2\, x_1 + \boxed{x_6\, x_5\, x_4}) \bmod B$$

$E\cdot x_5 + x_6$

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

- **Método de extracción.**

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

2.4.3. Funciones de dispersión.

- ¿Cómo se implementaría en C/C++ la suma posicional?
- Ejemplo, suma posicional con base 67:

0	1	2	3	4	5	6
P	A	L	A	B	R	A
80	65	76	65	66	82	65

```
unsigned int h (string clave)
{
    unsigned int res= 0;
    for (int i= 0; i < clave.length(); i++)
        res+= pow(67, i)*clave[i];
    return res % B;
}
```

$67^5 = 1.350.125.107$

```
unsigned int h (string clave)
{
    unsigned int res= 0;
    for (int i= 0; i < clave.length(); i++)
        res= 67*res + clave[i];
    return res % B;
}
```

2.4.3. Funciones de dispersión.

- **Métodos hash iterativos.**

  operación h (k : tipo_clave) : entero

      res:= VALOR_INICIAL

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

          res:= res · BASE

          res:= res COMB x_i

      finpara

      devolver res mod B

- 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
  - **sdbm** → VALOR_INIC = 0; BASE = 65599; COMB = +
  - **FNV-1** → VALOR_INIC = 2166136261; BASE = 16777619; COMB = XOR

2.4.3. Funciones de redispersión.

- **Redispersión lineal.**

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

- Es sencilla de aplicar.
- Se recorren todas las cubetas para i= 1, …, B-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	.	.	.	.	B-2	B-1

2.4.3. Funciones de redispersión.

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

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

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

- **Redispersión cuadrática.**

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

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

2.4.3. Funciones de redispersión.

- **Redispersión doble.**

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

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

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

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

- **Ejemplo.** Sea $x = x_1\, x_2\, x_3\, x_4$

  $h(k) = x_1x_4 \bmod B$

  $C(k) = 1 + (x_3x_2 \bmod (B\text{-}1))$

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.?