Tema 4. Ficheros

`http://aulavirtual.uji.es`

José M. Badía, Begoña Martínez, Antonio Morales y José M. Sanchiz

{badia, bmartine, morales, sanchiz}@icc.uji.es

Estructuras de datos y de la información

Universitat Jaume I

Índice

Bibliografía

- *File Structures, an object-oriented approach with C++*. M.J. Folk, B. Zoellick. Addison-Wesley, 1998. Capítulos 2, 4, 7 y 11, capítulo 6 (Índices), capítulo 10 (Tablas de dispersión).
- (Orallo'02), capítulo 18.
- *Estructuras de datos y algoritmos*. M.A. Weiss. Addison Wesley Iberoamericana, 1995. Capítulo 5.
- *Fundamentals of data structures in C++*. E. Horowitz, S. Sahni, D. Mehta. Freeman and Co., 1995. Capítulo 8.
- *Programación en C++. Algoritmos, estructuras de datos y objetos*. L. Joyanes. McGraw-Hill, 2000. Capítulo 17.
- (Nyhoff'06), apartados 5.1 y D.1

Objetivos

- Conocer el concepto de fichero.
- Saber utilizar las operaciones básicas para manejar ficheros con información binaria y texto.
- Saber utilizar los tipos de acceso secuencial y directo.
- Conocer posibles organizaciones de los datos en un fichero.
- Conocer y saber utilizar la organización de un fichero indexado.
- Conocer y saber utilizar la técnica de acceso mediante tablas de dispersión.

1 Conceptos y definiciones

- **Fichero** $\Rightarrow$ colección unidimensional de datos donde se puede:
  - leer bloques de información,
  - modificar bloques de información,
  - añadir bloques de información incrementando el tamaño del fichero.
- Los ficheros tienen asociados unos **punteros** que indican la posición donde se lee y escribe.
- Aplicación principal: guardar datos en disco para que sean permanentes.
- Un objeto de tipo *fichero* se suele asociar con un archivo de disco, pero también puede asociarse con una pantalla, un teclado, o una cadena de caracteres.

1 Conceptos y definiciones (II)

Un fichero conceptualmente es similar a un vector, pues se puede acceder a cualquier posición. Las diferencias son:

- El tamaño del fichero puede modificarse en tiempo de ejecución.
- Se pueden leer o escribir, a partir de una posición del fichero, bloques de información de cualquier tamaño.
- Un fichero se puede asociar con un archivo de disco. El sistema operativo se encarga de actualizar la información del objeto fichero al archivo.

1 Conceptos y definiciones (III)

Formatos de almacenamiento

- **Texto**: secuencia de caracteres.
- **Binarios**: secuencia de bytes que pueden representar distintos tipos de datos.

Ejemplo:

```
char cad[5] = "Hola";
int i = 132;
```

Codificación en octal:

- texto: 110 157 154 141 000 061 063 062
- binario: 110 157 154 141 000 204 000 000 000

2 Acceso: secuencial y directo

- **Acceso secuencial**: Se accede a los elementos desde el principio del fichero y en el orden en el que están escritos.
- **Acceso directo o aleatorio**: el índice se sitúa en un posición determinada antes de leer o escribir. En las implementaciones en C/C++ la posición se da en bytes.

3 Operaciones básicas. C/C++

Las operaciones básicas sobre ficheros son:

- Abrir y/o crear.
- Cerrar.
- Leer y escribir.
- Posicionar el índice del fichero.

C y C++ definen tipos y funciones para operar con ficheros:

- Acceso a bajo nivel en C con llamadas al Sistema Operativo.
- Flujos de datos, `stream`, en C. Más posibilidades de acceso.
- Clases específicas para ficheros en C++, que incluyen todos los métodos (funciones) de acceso necesarios.

Un primer ejemplo

```
...
int main() {
  fstream fich;
  int i;
  fich.open("datos.dat", ios::in);
  if (!fich) {
    cout << "Error de apertura de 'datos.dat' " << endl;
    exit(1);
  }
  if (!fich.eof()) fich >> i;
  fich.close();
}
```

4 Clases de flujos de datos en C++

Flujo de datos (*stream*): Fuente y/o destino de la lectura y/o escritura de datos.

El concepto de *stream* es útil para:

- ficheros,
- canales en redes,
- comunicación entre procesos,
- buffers y colas de espera.

4 Clases de flujos de datos en C++ (II)

- En C++, la librería *iostream* agrupa las clases para trabajar con flujos de datos.
- Clases que trabajan con el tipo `char`:
  - `ios`: operaciones generales de E/S,
  - `istream`: *streams* de entrada, clase derivada de `ios`,
  - `ostream`: *streams* de salida, clase derivada de `ios`,
  - `iostream`: *streams* de entrada/salida, derivada de `ios`.
- Objetos *stream* predefinidos:
  - `cin`: entrada estándar, generalmente el teclado,
  - `cout`: salida estándar, generalmente la pantalla,
  - `cerr y clog`: salidas de error, generalmente la pantalla.

Salida sobre un *stream*: Escritura

Operador de inserción: <<.

```
cout << ''Manufacturas '' << ''Acme'' << endl;
```

<< está sobrecargado para cada tipo predefinido de C++.

```
ostream& ostream::operator << (const char *var);
ostream& ostream::operator << (int i);
...
```

```
cout << ''Compilador de C++ '' << 3.4 << ''marca Acme'' << endl;
```

Entrada desde un *stream*: Lectura

Operador de extracción: >>

Sobrecargado, se lee sólo lo necesario y se convierte al tipo de la variable destino.

```
istream& istream::operator >> (tipo& variable);
```

Ejemplo: introducimos “version 5a 1998” por teclado,

```
int i,j;
char s1[100],s2[100];
cin >> s1; // s1=''version''
cin >> i; // i=5
cin >> s2; // s2=''a''
cin >> j; // j=1998
```

Streams fichero

Además de la consola podemos definir *streams* asignados a ficheros de disco.

Clases:

- `ifstream`, deriva de `istream`, para solo lectura de ficheros.
- `ofstream`, deriva de `ostream`, para solo escritura.
- `fstream`, deriva de `iostream`, para lectura/escritura.

Modos de apertura

- `in`: Entrada. Lectura
- `out`: Salida. Escritura
- `binary`: Acceso a un fichero binario
- `ate`, `app`, `trunc`, ...

Se combinan con '|'

```
fstream fich, f2;
fich.open("facturas.dat", ios::in | ios::binary);
f2.open("clientes.dat", ios::out | ios::app);
```

4 Clases de flujos de datos en C++ (VII)

Lectura/Escritura de texto

```
Operador de extracción >>  // Lectura
istream& get(char &c);
istream& get(char *s, streamsize n, char t='\n');
istream& getline(char *s, streamsize n, char t='\n');
Operador de inserción <<  // Escritura
ostream& put(char c);
```

Lectura/Escritura de información binaria

```
istream& read(char *s, int n);
ostream& write(const char *s, int n);
```

Lectura/Escritura de strings

```
...
int main() {
  string s1, s2, s3;
  ...
  getline(cin, s1); // Otra forma de 'getline'
  getline(fich, s2, '|'); // Lee hasta encontrar '|'
  fich >> s3; // Lee una palabra

  fich2 << s1 << "#" << s2 << "#" << s3 << endl;
}
```

4 Clases de flujos de datos en C++ (IX)

- **Acceso directo a *streams* fichero**:

```
istream& seekg(long offset, seek_dir mode=ios::beg);
ostream& seekp(long offset, seek_dir mode=ios::beg);
long tellg(); long tellp();
```

Ejemplo:

```
#include <fstream>
void main() {
  int i;
  fstream f;
  f.open("fichero.dat", ios::in | ios::binary);
  f.seekg(4*sizeof(int), ios::beg);
  f.read( (char *) &i, sizeof(int));
  f.close();
}
```

5 Organización: campos y registros

Campo: unidad básica de datos en un registro. Unidad de información más pequeña que tiene significado.

Registro: conjunto de campos relacionados que forman una unidad de información.

Un fichero se organiza como una colección de registros ordenados.

Representación más sencilla de un fichero $\Rightarrow$ flujo (*stream*) de bytes.

Problema:

Si escribimos los campos como un flujo de bytes, perdemos la pista de la información: dónde empieza y acaba cada campo y registro.

Solución:

Organizar el fichero para mantener los campos y registros separados.

Formas de mantener la identidad de los campos:

```
struct Tpersonaf {
  char nombre[15];
  char apellidos[20];
  char direccion[15];
  float sueldo;
};
```

```
struct Tpersonav {
  string nombre, apellidos, direccion;
  float sueldo;
};
```

- Que cada campo sea de longitud fija.

  Desventaja: se puede desaprovechar mucho espacio en campos con mucha variabilidad de longitud.

- Empezar cada campo con un indicador de longitud (si no son muy largos, hasta 256, con 1 byte es suficiente).

5.1 Organización de campos (II)

- Poner un delimitador entre campos, un carácter especial (|,#).

```
void LeePersona (istream & stream, Tpersonav & p) {
  // leer campos delimitados
  getline(stream, p.nombre, '|');
  getline(stream, p.apellidos, '|');
  getline(stream, p.direccion, '|');
  stream.read( (char *) &p.sueldo, sizeof(float));
  stream.ignore();  // "Se salta" el último '|'
}
```

- Utilizar el formato *clave_campo* = *valor_campo*.

  Ventaja: cada campo da información sobre sí mismo. Soporta que un registro tenga más o menos campos.

  Este formato se suele usar combinado con un delimitador.

  Se pierde algo de espacio por la clave.

5.2 Organización de registros

Para organizar un fichero en registros se puede hacer:

- Que los registros tengan longitud fija.

Pepe	Pi	Avda. Diagonal 224-2	Barcelona	04082
Maruja	Mir	Lavapies 34-3	Madrid	28035

- Que tengan un número fijo de campos.

Pepe|Pi|Avda. Diagonal 224-2|Barcelona|04082|Maruja|Mir|Lavapies 34-3|Madrid|28035|

- Empezar cada registro con un indicador de longitud.

45Pepe|Pi|Avda. Diagonal 224-2|Barcelona|04082|**38**Maruja|Mir|Lavapies 34-3|Madrid|28035|

5.2 Organización de registros (II)

- Utilizar otro fichero para guardar la posición de inicio de cada registro.

  Pepe|Pi|Avda. Diagonal 224-2|Barcelona|04082|Maruja|Mir|Lavapies 34-3|Madrid|28035|

  00 45 83 . . .

- Poner un delimitador entre cada registro: por ejemplo el carácter #.

  Pepe|Pi|Avda. Diagonal 224-2|Barcelona|04082|#Maruja|Mir|Lavapies 34-3|Madrid|28035|#

Índices. Objetivos:

- Asimilar el concepto de índice.
- Conocer las aplicaciones de los índices en el acceso a ficheros.
- Ser capaz de implementar un índice simple.

¿Qué es un índice?

Ejemplo: índice de un libro

- Un índice proporciona un modo rápido de localizar un tema (alternativa a buscar secuencialmente por el libro).
- Los índices son pares (clave, campos de referencia). En el ejemplo, la clave es el término y los campos de referencia, las páginas del libro.

6.1 ¿Qué es un índice?

Los índices simples usan estructuras vectoriales simples para almacenar las claves y los campos de referencia.

Aplicación de los índices a ficheros:

- Un índice impone un orden en un fichero de datos sin necesidad de tener ordenado dicho fichero.
- Permiten el acceso ordenado a registros de ficheros no ordenados.
- Acceso rápido a registros de longitud variable.
- Permiten múltiples caminos de acceso a un fichero.

6.2 Tipos de índices

Una clasificación:

- **Índice denso**: si contiene una referencia a cada uno de los registros del fichero de datos.
- **Índice no denso**: si sólo contiene referencia a un subconjunto de registros (bloques). Se utilizan cuando el orden del índice y del fichero de datos coincide.

Otra clasificación:

- **Índice primario**: si la clave del índice coincide con la clave del registro (valor único).
- **Índice secundario**: si la clave del índice no coincide con la clave del registro.

Ejemplo: Acceso a registros de longitud variable

- Se tiene una colección extensa de registros musicales que se quiere mantener en un fichero.
- En cada registro se tiene la siguiente información: compañía discográfica, número de identificación, título, compositor e intérprete.
- La longitud de los registros es variable.
- Se construye una clave única para cada registro con el campo compañía discográfica y el campo número de identificación.

6.3 Un índice simple (II)

dir.reg.	Co.	N.ID	tema	compositor	intérprete
32	RCA	2626	Sinfonía n.9	Beethoven	Giulini
77	COL	38558	Nebraska	Springsteen	Springsteen
152	DG	12343	Sinfonía n.9	Beethoven	Karajan
204	MER	343	Sinfonía n.9	Dvorak	Bernstein
245	ANG	31809	Concierto de Violín	Beethoven	Ferras

¿Cómo podemos acceder de forma rápida a un determinado registro?

- Ordenar el fichero y usar búsqueda binaria
  - No es posible ir al registro central si los registros son de longitud variable.
- Alternativa: Construir un índice para el fichero.

6.3 Un índice simple (III)

Fichero índice

clave	referencia
ANG31809	245
COL38558	77
DG12343	152
MER343	204
RCA2626	32

- La clave es la clave primaria de cada registro (12 caracteres completados con blancos a la derecha).
- El campo de referencia es la posición del fichero donde está el primer byte del registro.

6.3 Un índice simple (IV)

- El índice es un fichero de registros con dos campos de longitud fija.
- Hay un registro en el fichero índice por cada registro del fichero de datos (índice denso).
- Los registros del índice están ordenados por la clave.
- Los registros del fichero de datos están por orden de inserción en el fichero.
- Si el índice se mantiene en memoria, la inserción, búsqueda y borrado de registros es sencilla y eficiente: algoritmo de búsqueda binaria sobre el índice.
- El índice se guarda en memoria utilizando una estructura vectorial.
- Si el índice no cabe en memoria se utilizan otras estructuras más complejas: B-árboles (Tema 10).

6.4 Múltiples caminos de acceso a un fichero

- Se desea realizar búsquedas de los discos por compositor, por tema y por intérprete.
- Soluciones:
  - Tener 3 copias de cada disco, cada una en una discoteca distinta (una ordenada por compositor, otra por tema y otra por intérprete).
  - Usar un catálogo con 3 índices. Cada índice tendrá una clave distinta (compositor, tema e intérprete) y el mismo campo de referencia (el número de catálogo del disco).

6.4 Múltiples caminos de acceso a un fichero (II)

- Definir índices secundarios para cada campo por el que se quiera realizar consultas.

  Ejemplo: por compositor.

clave	referencia
BEETHOVEN	ANG31809
BEETHOVEN	DG12343
BEETHOVEN	RCA2626
DVORAK	MER343
SPRINGSTEEN	COL38558

- Operaciones: búsqueda, inserción, borrado, actualización.

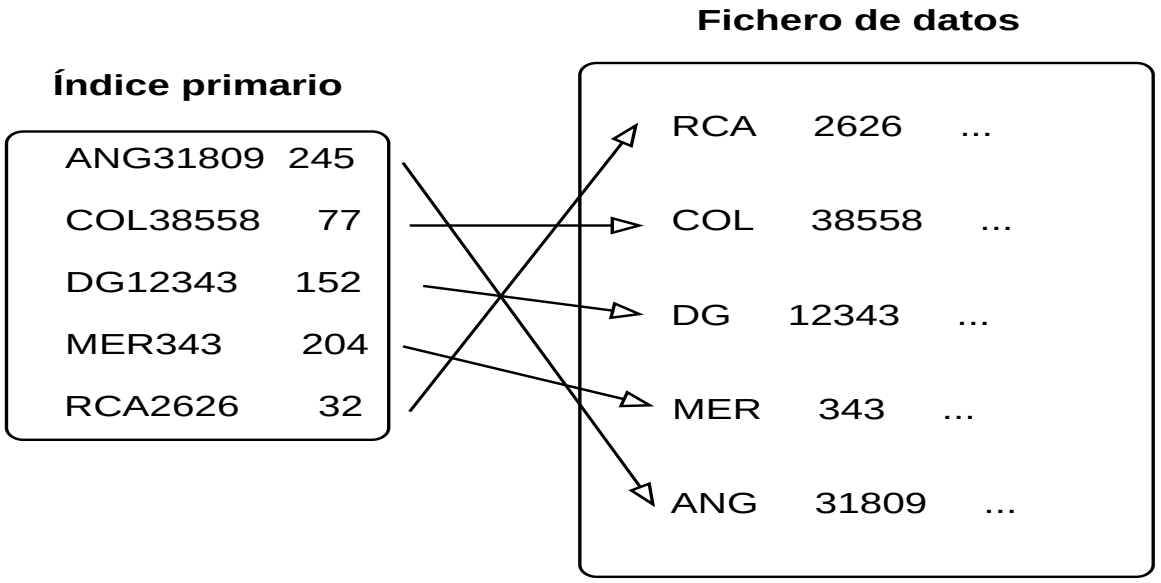
6.4 Múltiples caminos de acceso a un fichero (III)

Índices secundarios

Beethoven	ANG31809
Beethoven	DG12343
Beethoven	RCA2626
Dvorak	MER343
Springsteen	COL38558

Concierto ...	ANG31809
Nebraska	COL38558
Sinfonía n.9	DG12343
Sinfonía n.9	MER343
Sinfonía n.9	RCA2626

Bernstein	MER343
Ferras	ANG31809
Giulini	RCA2626
Karajan	DG12343
Springsteen	COL38558



6.4 Múltiples caminos de acceso a un fichero (IV)

El uso de índices secundarios permite tener múltiples vistas de los datos con una sóla copia de los mismos

Búsquedas por combinación de claves secundarias: compositor + tema:

1. Obtener la lista con todas las claves primarias que tengan un determinado compositor (índice secundario por clave *compositor*).
2. Obtener la lista con todas las claves primarias que tengan un determinado tema (índice secundario por clave *tema*).
3. Realizar la intersección de las dos listas resultantes.

Estructura de datos mejorada para índices sobre claves no primarias

Listas enlazadas

BEETHOVEN	1
DVORAK	4
SPRINGSTEEN	2

ANG31809	3
COL38558	-1
DG12343	5
MER343	-1
RCA2626	-1

Tablas de dispersión. Objetivos:

- Asimilar el concepto de tabla de dispersión.
- Conocer y razonar las aplicaciones de las tablas de dispersión.
- Conocer las características que debe tener una función de dispersión.
- Conocer las principales estrategias para el manejo de colisiones en las tablas de dispersión.
- Ser capaz de implementar una tabla de dispersión con alguna técnica de manejo de colisiones.

7 Tablas de dispersión (*Hashing*)

Técnica que se utiliza para implementar inserciones, eliminaciones y búsquedas en un tiempo medio constante.

TAD Tabla de símbolos

- Cada símbolo es una clave.
- Las operaciones básicas del TAD son:
  - Insertar un nuevo símbolo.
  - Buscar un símbolo.
  - Eliminar un símbolo.

7 Tablas de dispersión (*Hashing*) (II)

Aplicaciones:

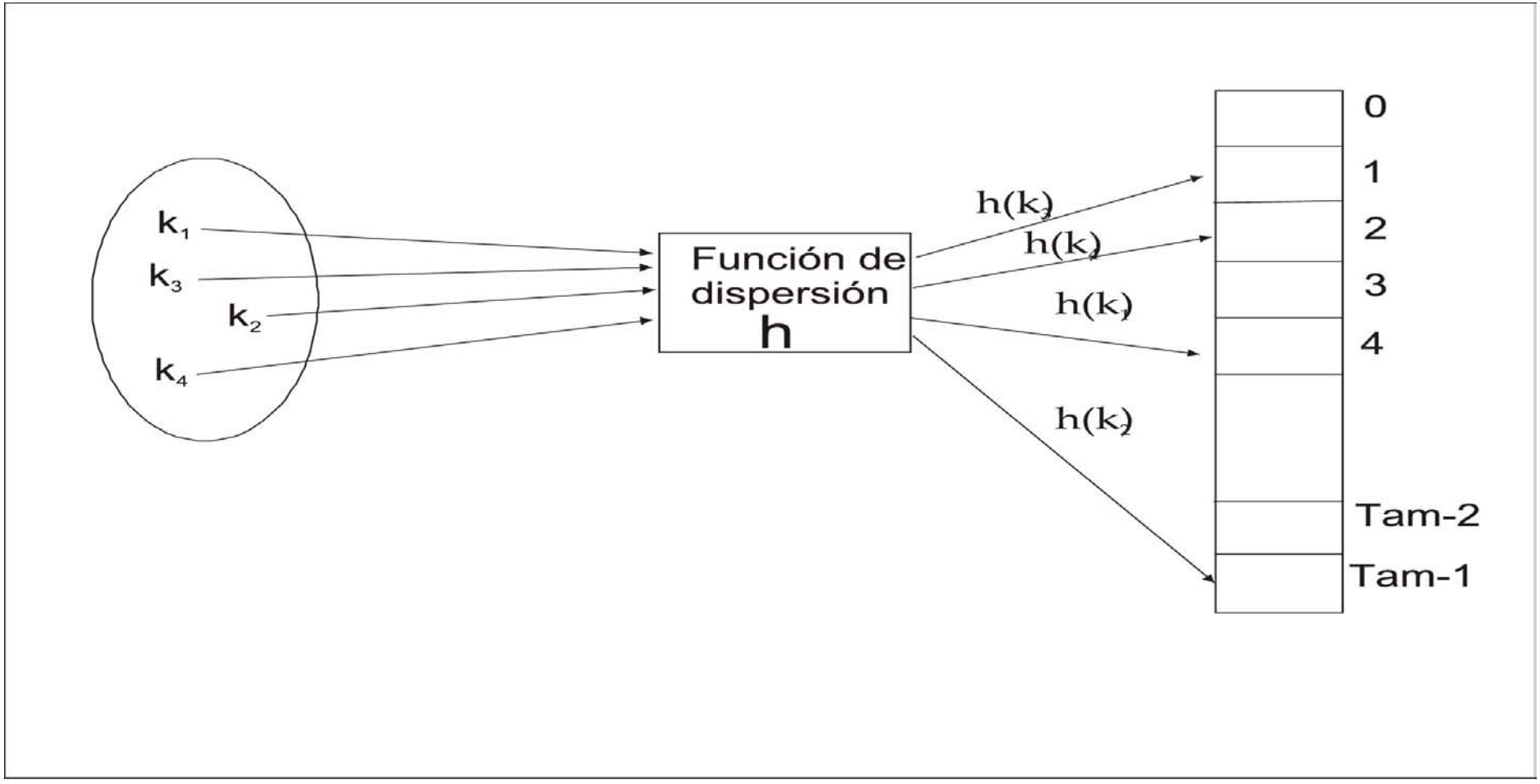
- TAD Diccionario.
- Tabla de símbolos de un compilador.
- Acceso rápido a registros de ficheros (cada registro está identificado unívocamente por una clave).
- Sin embargo, en general, no es eficiente para:
  - Obtener un algoritmo para recorrer ordenadamente todos los elementos de la tabla.
  - Implementar otras operaciones como *mínimo* y *máximo*.

7.1 Estructura de datos

- Las tablas de dispersión son pares **(clave, valor)**, donde no hay dos pares con las misma clave.
- La tabla de dispersión se implementa como una estructura secuencial de tamaño fijo: un vector.
- La posición donde almacenar una clave se calcula a partir del valor de dicha clave utilizando una función de dispersión, h.
- Si TAM es el tamaño de la tabla, la función de dispersión devolverá valores entre 0 y $TAM - 1$.

7.1 Estructura de datos (II)

Las claves están almacenadas en posiciones de un vector de acuerdo con el resultado de una función aplicada sobre la clave, llamada **función de dispersión**.



7.1 Estructura de datos (III)

Ejemplo:

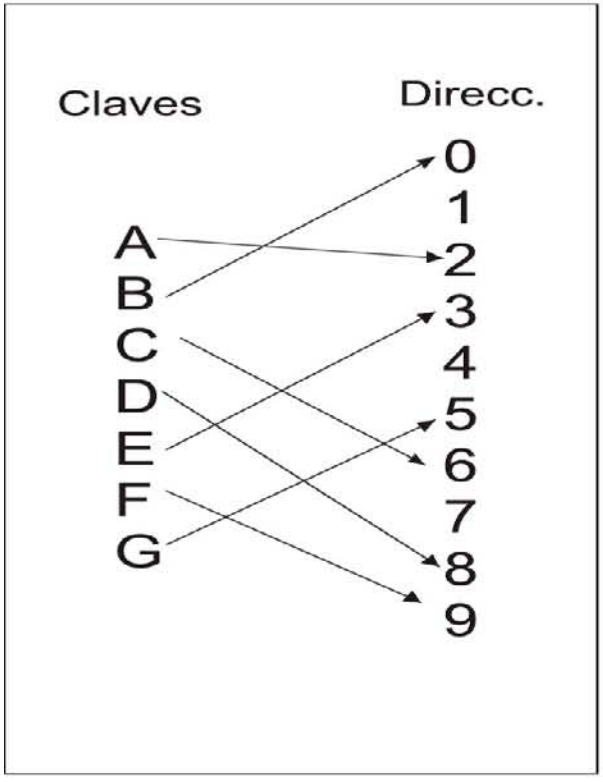
- Claves: apellidos de personas.
- Tamaño de la tabla: 0..999.
- Función de dispersión: tomar los 3 últimos dígitos del resultado de multiplicar los códigos ASCII de los 2 primeros caracteres de cada apellido.

Apellido	Cálculo función	Posición
BAUSA	$66 * 65 = 4290$	290
LÓPEZ	$76 * 79 = 6004$	004
TORRES	$84 * 79 = 6636$	636

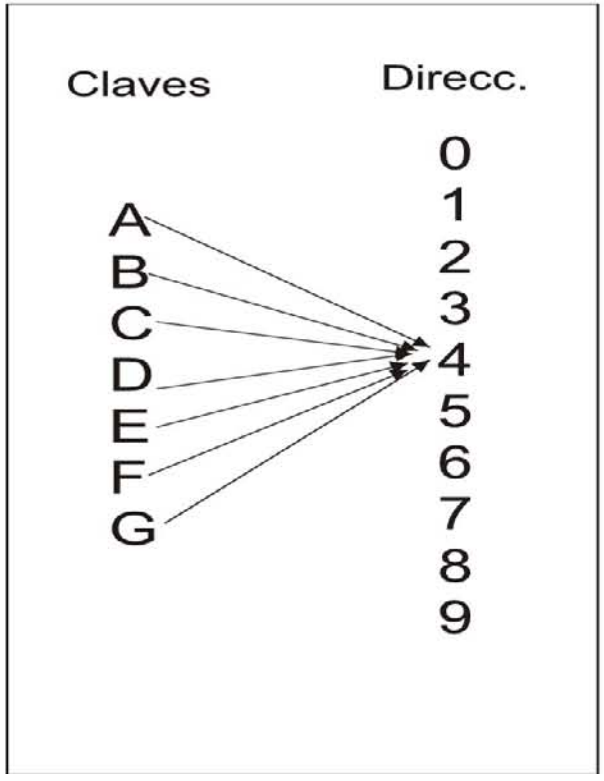
7.2 Características de las tablas de dispersión

- **Problema**: Espacio de la tabla limitado y conjunto de claves infinito.
- **Objetivo**: Distribuir uniformemente las claves entre las direcciones.

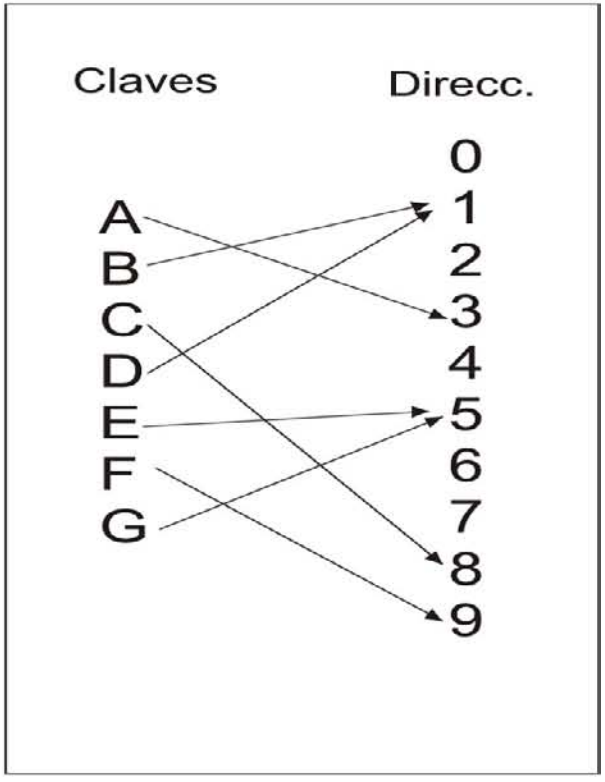
Uniforme



Peor caso



Aceptable



7.2 Características de las tablas de dispersión (II)

Colisión: Para dos claves distintas la función de dispersión devuelve el mismo valor.

Ejemplo:

Apellido	Cálculo función	Dirección
ABELLÁN	$65 * 66 = 4290$	290
BAUSA	$66 * 65 = 4290$	290
LÓPEZ	$76 * 79 = 6004$	004
TORRES	$84 * 79 = 6636$	636

7.2 Características de las tablas de dispersión (III)

- **Aspectos que considerar para implementar una tabla de dispersión**
  - Tamaño de la tabla.
  - Función de dispersión.
  - Técnica para resolver las colisiones:
    1. Direccionamiento abierto.
    2. Encadenamiento.

7.2 Características de las tablas de dispersión (IV)

Tamaño de la tabla

- Aumentar tamaño $\Rightarrow$ disminuyen las colisiones pero muchos huecos sin utilizar.
- Disminuir tamaño $\Rightarrow$ aumentan las colisiones.

Factor de carga de una tabla dispersa: razón entre el número de elementos almacenados y el tamaño de la tabla.

- Regla práctica: Escoger el tamaño de manera que el factor de carga nunca exceda de 0.8.

7.3 Funciones de dispersión

Condiciones que debe cumplir la función de dispersión:

1. Rápida de calcular.
2. Minimice las colisiones.

Cada clave tiene la misma probabilidad de dispersarse en cualquiera de los huecos de la tabla.

¿Es una buena función de dispersión la propuesta sobre los apellidos en el ejemplo?

7.3 Funciones de dispersión (II)

Algunas funciones de dispersión:

1. Método de la división

$$h(k) = k \; mod \; m$$

- Fácil de calcular.
- Muy dependiente del valor de m (tamaño de la tabla) $\Rightarrow$ elegir un número primo.

Algunas funciones de dispersión (cont.):

2. Método de la multiplicación

$$h(k) = \lfloor m\,(kA - \lfloor kA \rfloor)\rfloor$$

- k: clave
- m: tamaño de la tabla
- A: constante real, $0 < A < 1$

Elección del valor de A: inversa de la razón áurea

$$A = \Phi^{-1} \simeq 0{,}61803099$$

$$\Phi = \frac{1+\sqrt{5}}{2}$$

3. Mid-square:

- Consiste en elevar al cuadrado la clave y tomar r dígitos centrales (dependiendo del tamaño de la tabla).
- Se usa con frecuencia para la tabla de símbolos de un compilador (utilizando la representación octal de los identificadores).

4. Plegado:

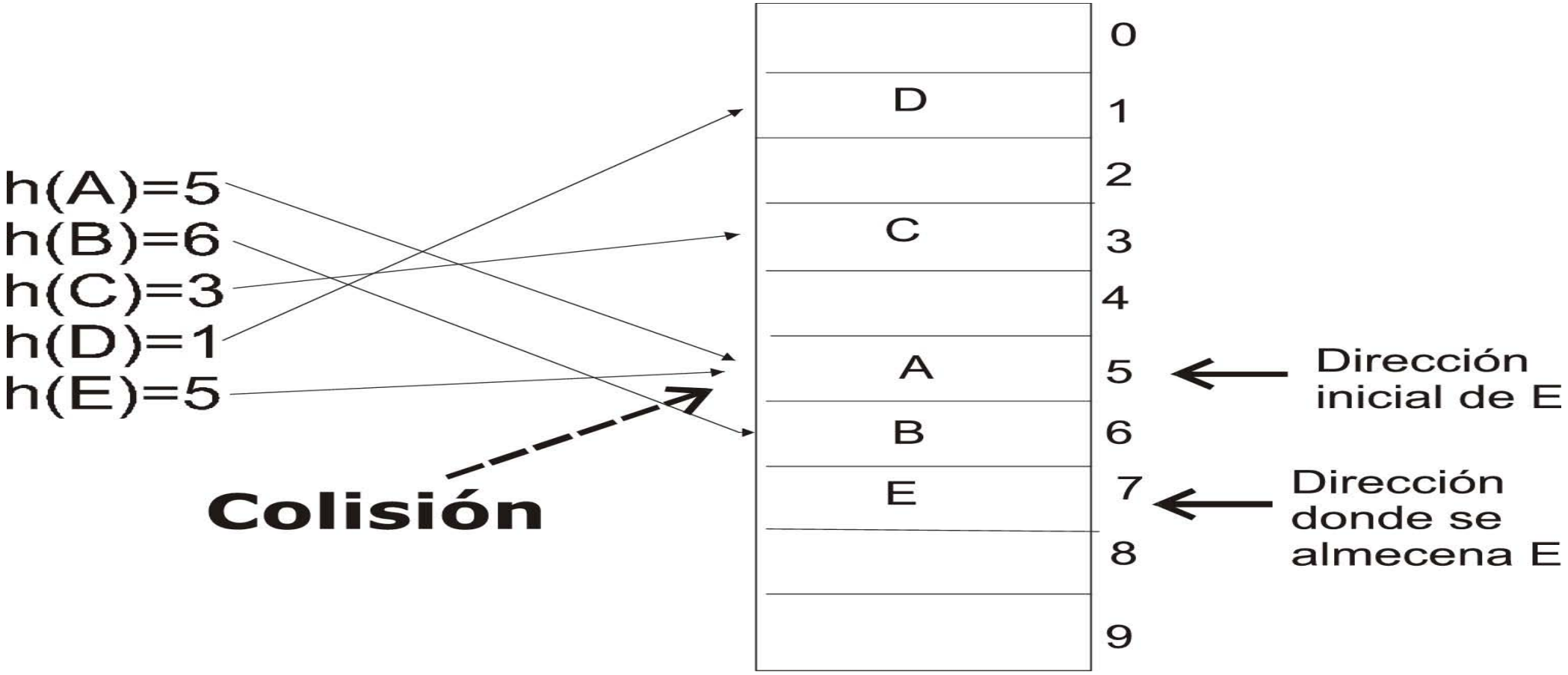
- La clave se divide en varias partes que se combinan entre ellas con una operación simple.
- Por ejemplo, para k=123402100231, dividiendo en grupos de 3 dígitos y sumándolos:

$$h(k) = 123 + 402 + 100 + 231 = 856$$

7.3 Funciones de dispersión (V)

- Cuando la clave es una cadena de caracteres:
  - No da buenos resultados considerar sólo parte de la clave.
  - Sumar los valores ASCII de cada uno de los caracteres.
  - Se suele utilizar el plegado agrupando la representación numérica de los caracteres.

Resolución de colisiones: 1. Direccionamiento abierto



Direccionamiento abierto

- Todos los datos se almacenan en la tabla.
- En la **inserción**, cuando se produce una colisión, la tabla es **explorada** hasta encontrar un hueco libre (considerar la tabla circular)
- Para **buscar** una clave, si no se encuentra en la dirección proporcionada por la función de dispersión, se explorará la tabla hasta:
  - encontrar la clave
  - encontrar una posición libre $\Rightarrow$ no está la clave.
- **Borrar** un elemento supone explorar la tabla igual que para buscar.
- Distinguir entre las posiciones vacías y las posiciones borradas.

7.4 Resolución de colisiones (III)

Distintas estrategias para explorar la tabla en el direccionamiento abierto:

1. Exploración lineal.
2. Exploración cuadrática.
3. Dispersión doble.

➤ **1. Exploración lineal**

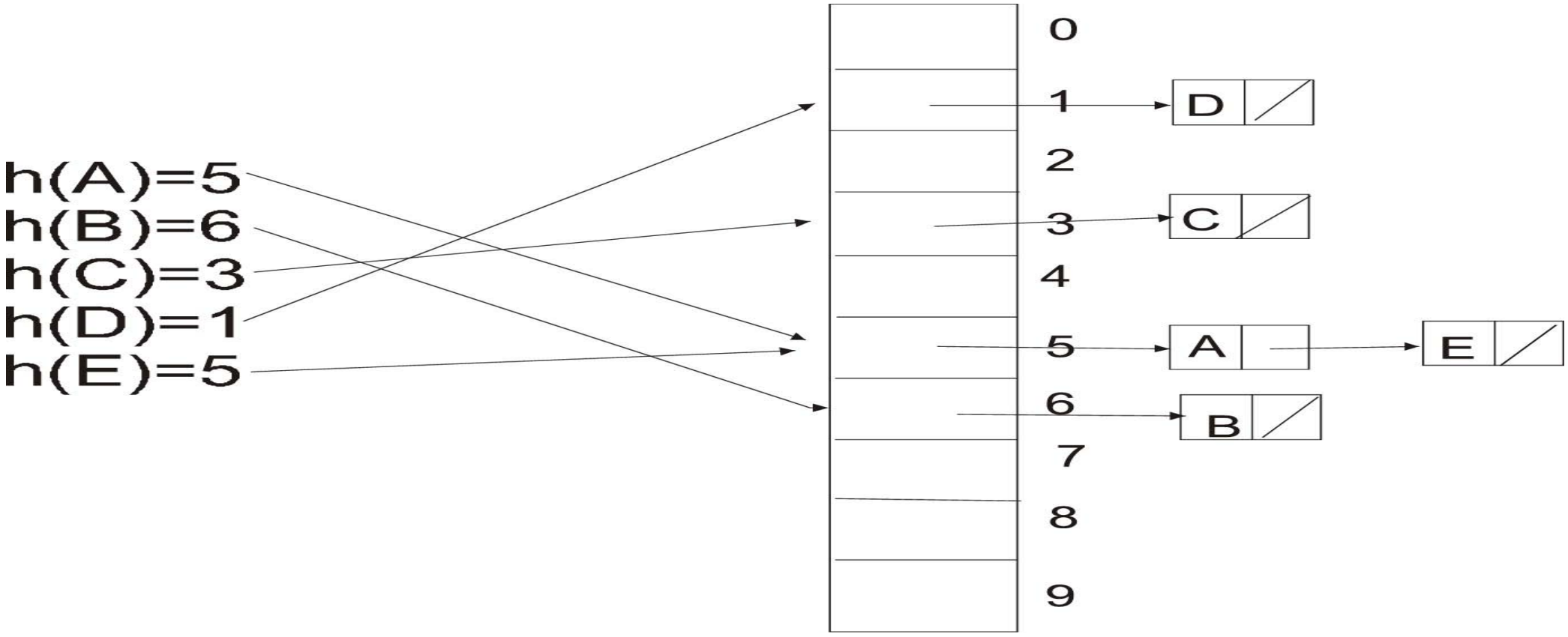
- Si la primera posición de la tabla explorada es j y c es una constante positiva, la secuencia explorada es: j, $(j + c * 1) mod\ m$, $(j + c * 2) mod\ m$, ..., donde m es el tamaño de la tabla.
- Problema de **agrupamiento**.
- Su rendimiento cae rápidamente al aumentar el factor de carga.

7.4 Resolución de colisiones (IV)

- **2. Exploración cuadrática**
  - Si j es la última posición explorada y c es una constante positiva, la secuencia explorada mientras no se encuentre una posición vacía será: $(j + (c * i)^2)\ mod\ m$
  - Inconveniente: si la ocupación de la tabla es el 50 % y el tamaño de la tabla es par, puede resultar imposible encontrar una posición vacía.
- **3. Dispersión doble**
  - Utilizar dos funciones de dispersión, h y h'.
  - Si se produce colisión, se utilizará la función $h + h'$ para calcular la nueva dirección. Si se vuelve a producir colisión, la función será $h + 2h'$ y así sucesivamente.

Resolución de colisiones: 2. Encadenamiento

Mantener una lista enlazada con todos los valores que se dispersan en la misma posición.



Encadenamiento

- La tabla contiene ahora punteros a la lista donde se guardan las claves que se dispersan en una determinada posición.
- Coste búsqueda = coste función de dispersión + coste de recorrer la lista.
- El número de claves que se pueden almacenar no depende del tamaño de la tabla.