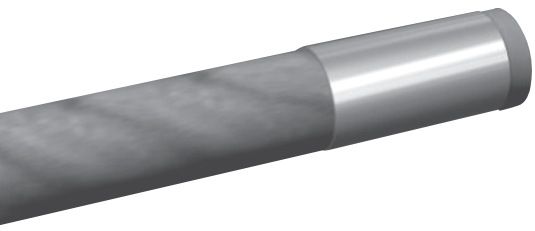
Parte VI

Estructura de datos en C, C++ y Java

Capítulo 31

Organización de datos en un archivo



Contenido

Introducción

Grandes cantidades de datos se almacenan normalmente en dispositivos de memoria externa. En este capítulo se realiza una introducción a la organización y gestión de datos estructurados sobre dispositivos de almacenamiento secundario, tales como discos magnéticos, CD... Las técnicas requeridas para gestionar datos en archivos son diferentes de las técnicas que estructuran los datos en memoria principal, aunque se construyen con ayuda de estructuras utilizadas en esta memoria.

Los algoritmos de ordenación de arrays no se pueden aplicar, normalmente, si la cantidad de datos a ordenar no caben en la memoria principal de la computadora y están en un dispositivo de almacenamiento externo. Es necesario aplicar nuevas técnicas de ordenación que se complementen con las ya estudiadas. Entre las técnicas más importantes destaca la fusión o mezcla. Mezclar significa combinar dos (o más) secuencias en una sola secuencia ordenada por medio de una selección repetida entre los componentes accesibles en ese momento.

Conceptos clave

- Archivo secuencial
- Archivo secuencial indexado
- Colisión
- Dirección dispersa
- *Hash*
- Mezcla
- Ordenación externa
- Registro
- Secuencia
- Transformación de claves

Registros

Un **archivo** o **fichero** es un conjunto de datos estructurados en una colección de registros, que son de igual tipo y constan a su vez de diferentes entidades de nivel más bajo denominadas **campos**.

Un *registro* es una colección de campos lógicamente relacionados, que pueden ser tratados como una unidad por algún programa. Un ejemplo de un registro puede ser la información de un determinado libro que contiene los campos de título, autor, editorial, fecha de edición, número de páginas, ISBN, etc. Los registros organizados en campos se denominan **registros lógicos**.

El concepto de registro es similar al concepto de estructura (*struct*) de C. Una posible representación en C del registro *libro* es la siguiente:

```
struct libro
{
  char titulo [46];
  char autor [80];
  char editorial [35];
  struct fecha fechaEdicion;
  int numPags;
  long isbn;
};
```

A recordar

El número de registros lógicos que puede contener un registro físico se denomina factor de bloqueo. Las operaciones de entrada/salida que se realizan en programas C se hacen por bloques a través de un área de memoria principal denominada *buffer*; esto hace que mejore el rendimiento de los programas.

La constante predefinida `BUFSIZ (stdio.h)` contiene el tamaño del *buffer*.

Clave

Una **clave** es un campo de datos que identifica el registro y lo diferencia de otros registros. Normalmente los registros de un archivo se organizan según un campo clave. Claves típicas son números de identificación, nombres; en general puede ser una clave de cualquier campo que admita *relaciones de comparación*. Por ejemplo, un archivo de libros puede estar organizado por autor, por editorial, etcétera.

Registro físico (bloque)

Un **registro físico** o **bloque** es la cantidad de datos que se transfieren en una operación de entrada/salida entre la memoria central y los dispositivos periféricos.

Un bloque puede contener uno o más registros lógicos. También puede ser que un registro lógico ocupe más de un registro físico o bloque.

Organización de archivos

La organización de un archivo define la forma en que los registros se disponen sobre el dispositivo de almacenamiento. La organización determina cómo estructurar los registros en un archivo. Se consideran tres organizaciones fundamentales:

- Organización secuencial.
- Organización directa.
- Organización secuencial indexada.

Organización secuencial

Un archivo con organización secuencial (**archivo secuencial**) es una sucesión de registros almacenados consecutivamente, uno detrás de otro, de tal modo que para acceder a un registro dado es necesario pasar por todos los registros que le preceden.

En un *archivo secuencial* los registros se insertan en orden de llegada, es decir, un registro se almacena justo a continuación del registro anterior. Una vez insertado el último registro y cerrado el archivo, el sistema añade la marca *fin de archivo.*

Las operaciones básicas que se realizan en un archivo secuencial son: *escribir los registros, consultar los registros* y *añadir un registro al final del archivo.*

Un archivo con organización secuencial se puede procesar tanto en modo texto como en modo binario. En C para crear estos archivos se abren (`fopen ( )`) especificando en el argumento *modo*: `"w"`, `"a"`, `"wb"` o `"ab"`; a continuación se escriben los registros utilizando, normalmente, las funciones `fwrite ( )`, `fprintf ( )` y `fputs ( )`.

La consulta del archivo se realiza abriendo éste con el modo `"r"` y, generalmente, leyendo todo el archivo hasta el indicador o marca *fin de archivo.* La función `feof ( )` es muy útil para detectar el fin de archivo, devuelve `1` si se ha alcanzado el final del archivo. El siguiente bucle permite leer todos los registros del archivo:

```
while (!feof(pf))
{
  < leer registro >
}
```

Ejercicio 31.1

Se realizan votaciones para elegir al presidente de la federación de Petanca. Cada distrito envía a la oficina central un sobre con los votos de cada uno de los tres candidatos. En un archivo con organización secuencial se graban registros con la estructura de datos correspondiente a cada distrito, es decir: nombre del distrito y votos de cada candidato. Una vez terminado el proceso de grabación se han de obtener los votos de cada candidato.

Los campos de datos que tiene cada uno de los registros están descritos en el propio enunciado: *Nombre del distrito, Candidato1, votos, Candidato2, votos* y *Candidato3, votos.* El archivo es de tipo binario; de esa forma se ahorra convertir los datos de tipo entero (*votos*) a dígitos `ASCII`, y cuando se lea el archivo para contar votos hacer la conversión inversa.

La creación del archivo se hace abriendo éste en modo *añadir al final* (`"a"`); de esta forma pueden añadirse nuevos registros en varias sesiones.

Para realizar la operación de *cuenta de votos* es necesario, en primer lugar, abrir el archivo en modo lectura (`"rb"`), y en segundo lugar leer todos los registros hasta llegar a la marca de *fin de archivo.* Con cada registro leído se incrementa la cuenta de votos de cada candidato.

La declaración del registro, en este ejercicio, se denomina `Distrito`, el nombre del archivo y la definición del puntero a `FILE` se realiza en el archivo *`Petanca.h`*:

```
typedef struct
{
  char candidato1[41];
  long vot1;
  char candidato2 [41];
  long vot2;
  char candidato3 [41];
  long vot3;
} Distrito;

char* archivo = "Petanca.dat";
FILE *pf = NULL;
```

```
/*
  Código fuente del programa, petanca.c, que escribe secuencialmente los
  registros en el archivo Petanca.dat.
*/

void main ( )
{
  Distrito d;
  int termina;
  pf = fopen (archivo, "ab");
  if (pf == NULL)
  {
    puts ("No se puede crear el archivo.");
    exit(-1);
  }

  strcpy(d.candidato1, "Lis Alebuche");
  strcpy(d.candidato2, "Pasionis Cabitorihe");
  strcpy(d.candidato3, "Gulius Martaria");
  termina = 0;
  puts ("Introducir los votos de cada candidato, termina con 0 0 0");
  do {
    leeRegistro (&d);
    if ( (d.vot1 == 0) && (d.vot2 == 0) && (d.vot3 == 0))
    {
      termina = 1;
      puts ("Fin del proceso. Se cierra el archivo");
    }
    else
      fwrite(&d, sizeof(Distrito), 1, pf);
  } while (!termina);
  fclose(pf);
}

void leeRegistro(Distrito* d)
{
  printf ("Votos para %s : ", d -> candidato1);
  scanf("%ld", &(d -> vot1));
  printf ("Votos para %s : ", d -> candidato2);
  scanf("%ld", &(d -> vot2));
  printf ("Votos para %s : ", d -> candidato3);
  scanf("%ld", &(d -> vot3));
}

/*
  Código fuente del programa, cntavoto.c, que lee secuencialmente los registros
  del archivo Petanca.dat y cuenta los votos.
*/

#include <stdlib.h>
#include <stdio.h>
#include <string .h>
#include "petanca. h"
void main ( )
{
  Distrito d;
  int votos[3] = {0,0,0};

  pf = fopen(archivo, "rb");
  if (pf == NULL)
  {
    puts("No se puede leer el archivo.");
    exit(-1);
  }
  fread (&d, sizeof(Distrito),1, pf);
```

```
  while (!feof(pf))
  {
    votos[0] += d.vot1;
    votos[1] += d.vot2;
    votos[2] += d.vot3;
    fread(&d, sizeof(Distrito),1, pf);
  }
  fclose(pf);
  puts (" \n\tVOTOS DE CADA CANDIDATO\n");
  printf (" %s %ld: \n", d.candidato1, votos [0] );
  printf (" %s %ld: \n", d.candidato2, votos [1] );
  printf (" %s %ld: \n", d.candidato3, votos [2] );
}
```

Organización directa

Un archivo con organización directa (aleatoria), o sencillamente **archivo directo**, se caracteriza porque el acceso a cualquier registro es directo mediante la especificación de un índice, que da la posición ocupada por el registro respecto al origen del archivo.

Los archivos directos tienen una gran rapidez para el acceso a los registros comparados con los secuenciales. La lectura/escritura de un registro es rápida, ya que se accede directamente al registro y no se necesita recorrer los anteriores, como ocurre en los archivos secuenciales.

En C estos archivos pueden ser de texto o binarios. Normalmente se crean de tipo binario para que las operaciones de entrada/salida sean más eficientes. C dispone de funciones para situarse, o conocer la posición en el archivo: `fseek ( )`, `fsetpos ( )`, `ftell ( )` y `fgetpos ( )`.

Los registros de los archivos directos disponen de un campo de tipo entero (por ejemplo, `indice`) con el que se obtiene la posición en el archivo. Esta sencilla expresión permite obtener el desplazamiento de un registro dado respecto al origen del archivo:

```
desplazamiento (indice) = (indice – 1) * tamaño (registro)
```

En los registros de un archivo directo se suele incluir un campo adicional, *ocupado*, que permite distinguir un registro dado de baja o de alta. Entonces, dentro del proceso de creación se puede considerar una inicialización de cada uno de los posibles registros del archivo con el campo `ocupado = 0`, para indicar que no se han dado de alta:

```
registro.ocupado = 0;
for (i = 1; i <= numRegs; i++)
  fwrite(&registro, sizeof (tipoRegistro), 1, pf);
```

Tanto para dar de *alta* como para dar de *baja, modificar* o *consultar* un registro se accede a la posición que ocupa, calculando el desplazamiento respecto al origen del archivo. La función C de posición (`fseek ( ) fsetpos ( )`) sitúa el apuntador del archivo directamente sobre el byte a partir del cual se encuentra el registro.

Ejercicio 31.2

Las reservas de un hotel de n habitaciones se van a gestionar con un archivo directo. Cada reserva tiene los campos: nombre del cliente, ***NIF*** *(número de identificación fiscal) y número de habitación asignada. Los números de habitación son consecutivos, desde 1 hasta el número total de habitaciones (por ello, se utiliza como índice de registro el número de habitación).*

Las operaciones que se podrán realizar son: inaguración, entrada de una reserva, finalización de estancia, consulta de habitaciones.

Cada registro del archivo va a corresponder a una reserva y a la vez al estado de una habitación. Si la *habitación n* está ocupada, el registro de índice *n* contendrá el nombre del cliente y su *NIF.* Si está vacía (libre) el campo *NIF* va a tener un asterisco (`'*'`). Por consiguiente, se utiliza como indicador de habitación libre que `nif == *`.

La operación *inauguración* inicializa el archivo, escribe tantos registros como habitaciones; cada registro con el campo *NIF* igual a la clave `*`, para indicar habitación libre y su número de habitación.

La operación *entrada* busca en el archivo la primera habitación libre y en su registro escribe uno nuevo con los datos de la reserva.

La *finalización* de una reserva consiste en asignar al campo *NIF* la clave (`*`) que indica habitación libre.

También se añade la operación *ocupadas* para listar todas las habitaciones ocupadas en ese momento.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#define numhab 55
FILE *fb = NULL;
const char fich [ ] = "fichero.dat";
typedef struct
{
  int num;
  char nif [13];
  char nombre[45];
} Habitacion;

#define desplazamiento(n) ((n – 1) * sizeof (Habitacion))

void inauguracion (void);
void entrada (void);
void salida (void);
void ocupadas (void);
void leerRes (Habitacion * ph);
void escribirRes (Habitacion h);
void main ( )
{
  char opcion;
  do {
    puts ("1. Inauguracion");
    puts ("2. Entrada ");
    puts ("3. Salida  ");
    puts ("4. Ocupadas ");
    puts ("5. Salir   ");
    do {
      printf ("Elige una opción ");
      scanf ("%c%*c", &opcion);
    } while (opcion < '1' || opcion > '5');
    switch (opcion)
    {
      case '1':
        inauguracion ( );
        break;
      case '2':
        entrada ( );
        break;
      case '3':
        salida ( );
        break;
      case '4':
        ocupadas ( );
        system ("pause");
```

```
        break;
    }
  }
  while (opcion != '5');
  if (fb != NULL) fclose (fb);
}

void inauguracion (void)
{
  Habitacion h;
  int i;
  if (fb != NULL)
  {
    char opcion;
    printf ("Archivo ya creado, ¿ desea continuar(S/N) ?: ");
    scanf ("%c%*c", &opcion);
    if (toupper(opcion) != 'S') return;
  }
  fb = fopen (fich, "wb+");
  for (i = 1; i <= numhab; i++)
  {
    h.num = i;
    strcpy (h.nif, "*");
    fwrite (&h, sizeof (h), 1, fb);
  }
  fflush (fb);
}

void entrada (void)
{
  Habitacion h;
  int encontrado, nh;
    /* Búsqueda secuencial de primera habitación libre */
  encontrado = 0;
  nh = 0;
  if (fb == NULL) fb = fopen (fich, "rb+");
  fseek (fb, 0L, SEEK_SET);
  while ((nh < numhab) && !encontrado)
  {
    fread (&h, sizeof (h), 1, fb);
    nh++;
    if (strcmp (h.nif ,"*") == 0)    /* Habitación libre */
    {
      encontrado = 1;
      leerRes (&h);
      fseek (fb, desplazamiento (h.num), SEEK_SET);
      fwrite (&h, sizeof (h), 1, fb);
      puts ("Datos grabados");
    }
  }
  if (!encontrado) puts ("Hotel completo ");
  fflush (fb);
}

void salida (void)
{
  Habitacion h;
  int n;
  char r;
  if (fb == NULL) fb = fopen (fich, "rb+");
  printf ("Numero de habitacion: ");
  scanf ("%d%*c",&n);
  fseek (fb, desplazamiento (n), SEEK_SET);
  fread (&h, sizeof (h), 1, fb);
```

```
  if (strcmp (h.nif,"*") != 0)
  {
    escribirRes (h);
    printf ("¿Son correctos los datos?(S/N) ");
    scanf ("%c%*c" ,&r);
    if (toupper (r) == 'S')
    {
      strcpy (h.nif, "*");
      fseek (fb, -sizeof (h), SEEK_CUR);/* se posiciona de nuevo */
      fwrite (&h, sizeof (h), 1, fb);
    }
  }
  else
    puts ("La habitacion figura como libre");
  fflush (fb);
}

void ocupadas (void)   {
  Habitacion h;
  int n;
  if (fb == NULL) fb = fopen (fich, "rb+");
  fseek (fb, 0L, SEEK_SET);
  puts ("   Numero \t NIF \t\t Nombre");
  puts (" habitacion                    ");
  for (n = 1 ; n <= numhab; n++)
  {
    fread (&h, sizeof (h), 1, fb);
    if (strcmp (h.nif ,"*") != 0)
      escribirRes (h);
  }
}

void leerRes (Habitacion *ph)
{
  printf ("Nif: ");
  gets (ph -> nif);
  printf ("Nombre ");
  gets (ph -> nombre);
}

void escribirRes (Habitacion h)
{
  printf ("\t %d", h.num);
  printf ("\t%s\t", h.nif);
  printf ("\t%s\n", h.nombre);
}
```

A recordar

La fórmula o algoritmo que transforma una clave en una dirección se denomina *función hash*. Las posiciones de los registro en un archivo se obtienen transformando la clave del registro en un índice entero. A los archivos con esta organización se les conoce como *archivos hash* o *archivos con transformación de claves*.

Archivos con función de direccionamiento *hash*

La organización directa tiene el inconveniente de que no hay un campo del registro que permita obtener posiciones consecutivas y, como consecuencia, quedan muchos huecos libres entre registros. En-

tonces la organización directa necesita programar una relación entre un campo clave del registro y la posición que ocupa.

Funciones *hash*

Una *función hash,* o *de dispersión,* convierte un campo clave (un entero o una cadena) en un valor entero dentro del rango de posiciones que puede ocupar un registro de un archivo.

Si *x* es una clave, entonces *h*(*x*) se denomina direccionamiento *hash* de la clave *x*, además es el índice de la posición donde se debe guardar el registro con esa clave. Por ejemplo, si están previstos un máximo de `tamIndex = 199` registros, la *función hash* debe generar índices en el rango `0 .. tamIndex-1`.

Por ejemplo, la clave puede ser el número de serie de un artículo (hasta 6 dígitos) y si están previstos un máximo de `tamIndex` registros, la función de direccionamiento tiene que ser capaz de transformar valores pertenecientes al rango `0 .. 999999`, en un conjunto de rango `0 .. tamIndex-1`. La clave también puede ser una cadena de caracteres, en cuyo caso se hace una transformación previa a valor entero.

La función *hash* más utilizada por su sencillez se denomina *aritmética modular.* Genera valores dispersos calculando el resto de la división entera entre la clave `x` y el número máximo de registros previstos.

```
x % tamIndex = genera un número entero de 0 a tamIndex-1
```

La función *hash* o función de transformación debe reducir al máximo las colisiones. Se produce una *colisión* cuando dos registros de claves distintas, `c1` y `c2`, producen la misma dirección, `h(c1) = h(c2)`. Nunca existirá una garantía plena de que no haya colisiones, y más sin conocer de antemano las claves y las direcciones. La experiencia enseña que siempre habrá que preparar la *resolución de colisiones* para cuando se produzca alguna.

Para que la función *aritmética modular* disperse lo más uniformemente posible se recomienda elegir como tamaño máximo, `tamIndex`, un número primo que supere el número previsto de registros.

Ejemplo 31.1

Considerar una aplicación en la que se deben almacenar n = 900 registros. El campo clave elegido para dispersar los registros en el archivo es el número de identificación. Elegir el tamaño de la tabla de dispersión y calcular la posición que ocupan los registros cuyo número de identificación es: 245643, 245981 y 257135.

En este supuesto, una buena elección de `tamIndex` es `997` al ser un número primo mayor que el número de registros que se van a grabar, `900`. Se aplica la función *hash* de *aritmética modular* y se obtienen estas direcciones:

```
h(245643) = 245643 % 997 = 381
h(245981) = 245981 % 997 = 719
h(257135) = 257135 % 997 = 906
```

Características de un archivo con direccionamiento *hash*

Para el diseño del archivo se deben considerar dos áreas de memoria externa: el área principal y el área de sinónimos o colisiones. Aproximadamente el archivo se crea con 25% más que el número de registros necesarios.

Un archivo *hash* se caracteriza por:

- Acceder a las posiciones del archivo a través del valor que devuelve una función *hash.*
- La función *hash* aplica un algoritmo para transformar uno de los campos llamado campo clave en una posición del archivo.

- El campo elegido para la función debe ser único (no repetirse) y conocido fácilmente por el usuario, porque a través de ese campo va a acceder al programa.
- Todas las funciones *hash* provocan colisiones o sinónimos. Para solucionar estas repeticiones se definen dos zonas:
  — *Zona de datos* o *principal* en la que el acceso es directo al aplicarle la función *hash.*
  — *Zona de colisiones* o *sinónimos* en la que el acceso es secuencial. En esta zona se guardan las estructuras o registros en los que su campo clave ha producido una posición repetida. Y se van colocando secuencialmente, es decir, en la siguiente posición que esté libre.

En el ejercicio 31.3 se diseña un archivo con direccionamiento *hash.* Implementa las operaciones más importantes que se pueden realizar con archivos: *creación, consulta, alta* y *baja.* El campo clave elegido es una cadena (formada por caracteres alfanuméricos); para obtener la posición del registro, primero se transforma la cadena (según el código `ASCII`) a un valor entero; a continuación se aplica la *aritmética modular* para obtener la posición del registro.

Ejercicio 31.3

Los libros de una pequeña librería van a guardarse en un archivo para poder realizar accesos tan rápido como sea posible. Cada registro (libro) tiene los campos código (cadena de 6 caracteres), autor y título. El archivo debe estar organizado como acceso directo con transformación de claves (archivo hash*), la posición de un registro se obtendrá aplicando aritmética modular al campo clave: código. La librería tiene capacidad para 190 libros.*

Para el diseño del archivo se crearán `240` registros que se distribuirán de la siguiente forma:

1. Posiciones `0 – 198` constituyen el área principal del archivo.
2. Posiciones `199 – 239` constituyen el área de desbordamiento o de colisiones.

El campo clave, cadena de 6 caracteres, se transforma considerando su secuencia de valores numéricos (ordinal `ASCII` de cada carácter) en base `27`. Por ejemplo, el código `2R545` se transforma en:

$$'2'*27^4 + 'R'*27^3 + '5'*27^2 + '4'*27^1 + '5'*27^0$$

En C, un carácter se representa como un valor entero que es, precisamente, su ordinal. La transformación da lugar a valores que sobrepasan el máximo entero (incluso con enteros largos), generando números negativos. No es problema, simplemente se cambia de signo.

La creación del archivo escribe `240` registros, con el campo `código = '*'`, para indicar que están disponibles (de baja).

Para dar de alta un registro, se obtiene primero la posición (función *hash*); si se encuentra dicha posición ocupada, el nuevo registro deberá ir al área de colisiones (sinónimos).

El proceso de consulta de un registro debe comenzar con la entrada del código; la transformación de la clave permite obtener la posición del registro. A continuación se lee el registro; la comparación del código de entrada con el código del registro determina si se ha encontrado. Si son distintos, se explora secuencialmente el área de colisiones.

La baja de un registro también comienza con la entrada del código, se realiza la búsqueda, de igual forma que en la consulta, y se escribe la marca `'*'` en el campo código (se puede elegir otro campo) para indicar que ese hueco (registro) está libre.

La función *hash* devuelve un número entero *n* de `0` a `(199-1)`; por esa razón el desplazamiento desde el origen del archivo se obtiene multiplicando *n* por el tamaño de un registro.

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

#define principal 199
#define total 240
```

```
const char fich [12] = "fichash.dat";

FILE *fh = NULL;

typedef struct
{
  char codigo [7];
  char autor [41];
  char titulo [41];
} Libro;

#define desplazamiento (n) ((n) * sizeof (Libro))

  /* prototipo de las funciones */

void creación (void);
void compra (void);                          /* operación dar de Alta */
void venta (void);                           /* operación dar de Baja */
void consulta (void);
void colisiones (Libro lib);
int indexSinonimo (const char c [ ]);
int hash(char c [ ]);
long transformaClave (const char c [ ]);
void escribir (Libro lib);
void main ( )
{
  char opcion;
    /* comprueba si el archivo ya ha sido creado */
  fh = fopen (fich, "rb");
  if (fh == NULL)
  {
    puts ("EL ARCHIVO VA A SER CREADO");
    creacion ( );
  }
  else
    fh = NULL;
  do
  {
    puts ("1. Compra  ");
    puts ("2. Venta   ");
    puts ("3. Consulta ");
    puts ("5. Salir   ");

    do {
      printf ("Elige una opción ");
      scanf ("%c%*c", &opcion);
    } while (opcion < '1' || opcion > '5' || opcion == '4 ');
    switch (opcion)
    {
      case '1':
        compra ( ); break;
      case '2':
        venta ( ); break;
      case '3':
        consulta ( ); break;
    }
  } while (opcion != '5');
  if (fh != NULL) fclose (fh);
}

  /*
    Creación: escribe consecutivamente total registros, todos con el campo código
    igual a '*' para indicar que están libres.
  */

void creación (void)
```

```
{
  Libro lib;
  int i;
  fh = fopen (fich, "wb+");
  strcpy (lib.codigo, "*");
  for (i = 1; i <= total; i++)
    fwrite (&lib, sizeof (lib), 1, fh);
  fclose (fh);
  fh = NULL;
}

/*
    Alta de un registro: pide al usuario los campos código, título y autor. Llama a
    la función hash ( ) para obtener la posición en la cual leer el registro, si
    está libre graba el nuevo registro. Si está ocupado busca en el área de
    colisiones la primera posición libre que será donde escribe el registro.
*/

void compra (void)
{
  Libro lib, libar;
  long posicion;

  if (fh == NULL) fh = fopen (fich, "rb+");

  printf ("Código: ");gets (lib.codigo);
  printf ("Autor: ");gets (lib.autor);
  printf ("Título: ");gets (lib.titulo);
  posicion = hash (lib.codigo);
  posicion = desplazamiento(posicion);

  fseek(fh, posicion, SEEK_SET);
  fread(&libar, sizeof(Libro), 1, fh);
  if (strcmp(libar.codigo, "*") == 0) /* registro libre */
  {
    fseek(fh, -sizeof(Libro), SEEK_CUR);
    fwrite(&lib, sizeof(Libro), 1, fh);
    printf("Registro grabado en la dirección: %ld\n",posicion);
  }
  else if (strcmp(lib.codigo, libar.codigo) == 0)    /* duplicado */
  {
    puts("Código repetido, revisar los datos.");
    return;
  }
  else
    colisiones(lib);
  fflush(fh);
}

/*
    Baja de un registro: pide el código del registro. Se lee el registro cuya
    posición está determinada por la función hash ( ). Si los códigos son iguales,
    se da de baja escribiendo '*' en el campo código. En caso contrario se busca en
    el área de colisiones y se procede igual.
*/

void venta( )
{
  Libro libar;
  char codigo [7], r;
  long posicion;

  if (fh == NULL) fh = fopen(fich, "rb+");

  printf("Código: ");gets(codigo);
  posicion = hash(codigo);
  posicion = desplazamiento(posicion);
```

```
  fseek(fh, posicion, SEEK_SET);
  fread(&libar, sizeof(Libro), 1, fh);

  if (strcmp(libar.codigo, codigo) != 0)
    posicion = indexSinonimo (codigo);

  if (posicion != -1)
  {
    escribir(libar);
    printf("¿Son correctos los datos? (S/N): ");
    scanf("%c%*c" ,&r);
    if (toupper(r) == 'S')
    {
      strcpy(libar. codigo, " *");
      fseek(fh, -sizeof(Libro), SEEK_CUR);
      fwrite(&libar, sizeof(Libro), 1, fh);
    }
}
else
  puts("No se encuentra un registro con ese código.");
fflush(fh);
}

/*
  Consulta de un registro: pide el código del registro. Se lee el registro cuya
  posición está determinada por la función hash ( ). Si los códigos son iguales
  se muestra por pantalla. En caso contrario se busca en el área de colisiones.
*/

void consulta( )
{

  Libro lib;
  char codigo[7];
  long posicion;

  if (fh == NULL) fh = fopen(fich, "rb+");

  printf("Código: ");gets(codigo);
  posicion = hash(codigo);
  posicion = desplazamiento(posicion);

  fseek(fh, posicion, SEEK_SET);
  fread(&lib, sizeof(Libro), 1, fh);

  if (strcmp(lib.codigo, codigo) == 0)
    escribir(lib);
  else
  {
    int posicion;
    posicion = indexSinonimo (codigo);
    if (posicion != -1)
    {
      fseek(fh, -sizeof(Libro), SEEK_CUR);
      fread(&lib, sizeof(Libro), 1, fh);
      escribir(lib);
    }
    else
      puts("No se encuentra un ejemplar con ese código.");
  }
}

/*
  Inserta en área de sinónimos: busca secuencialmente el primer registro libre
  (codigo=='*') para grabar el registro lib.
*/
```

```
void colisiones(Libro lib)
{
  Libro libar;
  int pos = desplazamiento(principal);
  int j = principal;
  int encontrado;

  fseek(fh, pos, SEEK_SET);          /* se sitúa en área de sinónimos */
  encontrado = 0;
  while ((j < total) && !encontrado)
  {
    fread(&libar, sizeof(Libro), 1, fh);
    j++;
    if (strcmp(libar.codigo ,"*") == 0)             /* libre */
    {
      encontrado = 1;
      fseek(fh, -sizeof(Libro), SEEK_CUR);
      fwrite(&lib, sizeof(Libro), 1, fh);
      puts("Datos grabados en el área de sinónimos.");
    }
  }

  if (!encontrado) puts("Área de sinónimos completa. ");
  fflush(fh);
}

/*
  Búsqueda en área de sinónimos: búsqueda secuencial, por código, de un registro.
  Devuelve la posición que ocupa, o bien -1 si no se encuentra.
*/

int indexSinonimo (const char c [ ])
{
  Libro libar;
  int pos = desplazamiento(principal);
  int j = principal;
  int encontrado;

  fseek(fh, pos, SEEK_SET);          /* se sitúa en área de sinónimos */
  encontrado = 0;
  while ((j < total) && !encontrado)
  {
    fread(&libar, sizeof(Libro), 1, fh);
    j++;
    if (strcmp(libar.codigo, c) == 0)
      encontrado = 1;
  }
  if (!encontrado) j = -1;
  return j;
}

/*
  Aritmética modular: transforma cadena a un entero en el rango [0, principal).
  En primer lugar pasa los caracteres del código a mayúsculas. A continuación, llama
  a la función que convierte la cadena a entero largo. Por último, aplica el módulo
  respecto a principal. El módulo produce un entero de 0 a principal-1.
*/

int hash(char c [ ])
{
  int i, suma = 0;

  for (i = 0; i < strlen(c); i++)
    c[i] = toupper(c[i]);
  return transformaClave(c) % principal;
}
```

```
long transformaClave(const char* clave)
{
  int j;
  long d;
  d = 0;
  for (j = 0; j < strlen(clave); j+
  {
    d = d * 27 + clave[j];
  }
    /*
      Si d supera el máximo entero largo, genera número negativo
    */
  if (d < 0) d = -d;
  return d;
}

void escribir(Libro lib)
{
  printf("Código: %s\t", lib.codigo);
  printf("Autor: %s\t", lib.autor);
  printf("Título: %s\n" ,lib.titulo);
}
```

Archivos secuenciales indexados

Para buscar el teléfono de una persona en la guía no se busca secuencialmente desde los nombres cuya inicial es `"a"` hasta la `"z"`, sino que se abre la guía por la letra inicial del nombre. Si se desea buscar `"zalabastro"`, se abre la guía por la letra `"z"` y se busca la cabecera de página hasta encontrar la página más próxima al nombre, buscando a continuación nombre a nombre hasta encontrar `"zalabastro"`. La guía es un ejemplo típico de *archivo secuencial indexado* con dos niveles de índices, el nivel superior para las letras iniciales y el nivel menor para las cabeceras de página. Por consiguiente, cada archivo secuencial indexado consta de un archivo de índices y un archivo de datos. La figura 31.1 muestra un archivo con organización secuencial indexada.

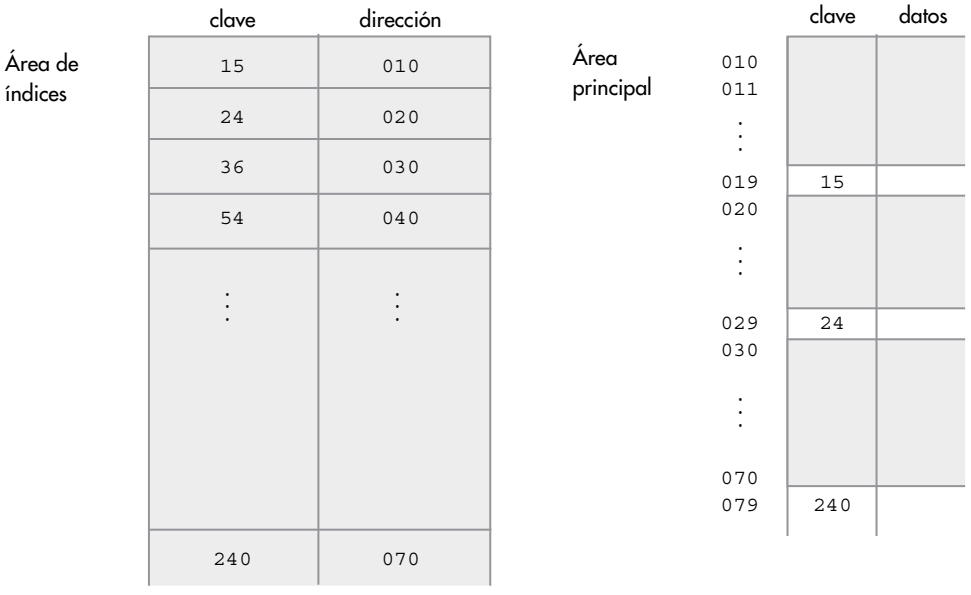


Figura 31.1 Organización secuencial indexada.

Partes de un archivo secuencial indexado

Para que un archivo pueda organizarse en forma secuencial indexada el tipo de los registros debe contener un campo clave identificador. La clave se asocia con la dirección (posición) del registro de datos en el archivo principal.

Un archivo con organización secuencial indexada consta de las siguientes partes:

- *Área de datos.* Contiene los registros de datos en forma secuencial, sin dejar huecos intercalados.
- *Área de índices.* Es una tabla que contiene la clave identificativa y la dirección de almacenamiento. Puede haber índices enlazados.

El área de índices normalmente está en memoria principal para aumentar la eficiencia en los tiempos de acceso. Ahora bien, debe existir un archivo donde guardar los índices para posteriores explotaciones del archivo de datos. Entonces, al diseñar un archivo indexado hay que pensar que se manejarán dos tipos de archivos, el de datos y el de índices, con sus respectivos registros.

Por ejemplo, si se quieren grabar los atletas federados en un archivo secuencial indexado, el campo índice que se puede elegir es el nombre del atleta (también se puede elegir el número de *carnet de federado*). Habría que declarar dos tipos de registros:

```
typedef struct
{
  int edad;
  char carnet[15];
  char club[29];
  char nombre[41];
  char sexo;
  char categoria[21];
  char direccion[71];
} Atleta;
typedef struct
{
  char nombre[41];
  long posicion;
} Indice;
```

A recordar

Los archivos secuenciales indexados presentan dos *ventajas* importantes: rápido acceso, y que la gestión de archivos se encarga de relacionar la posición de cada registro con su contenido mediante la tabla de índices. El principal *inconveniente* es el espacio adicional para guardar el área de índices.

Proceso de un archivo secuencial indexado

Al diseñar un archivo secuencial indexado, lo primero que hay que decidir es cuál va a ser el campo clave. Los registros han de ser grabados en orden secuencial, y simultáneamente a la grabación de los registros, el sistema crea los índices en orden secuencial ascendente del contenido del campo clave.

A continuación se desarrollan las operaciones (*altas, bajas, consultas*...) para un archivo con esta organización. También es necesario considerar el inicio y la salida de la aplicación que procesa un archivo indexado, para cargar y descargar, respectivamente, la tabla de índices.

Los registros tratados se corresponden con artículos de un supermercado. Los campos de cada registro son: *nombre del artículo, identificador, precio, unidades.* Un campo clave adecuado para este tipo de registro es el *nombre del artículo.*

Tipos de datos

Se declaran dos tipos de estructura para representar el registro de datos y el de índice respectivamente:

```
typedef struct
{
  char nombre[21];
  char identificador[14];
  double precio;
  int unidades;
  int estado;
} Articulo;
typedef struct
{
```

```
  char nombre[21];
  long posicion;
} Indice;
```

Por razones de eficiencia, la tabla de índices está en la memoria principal. Se pueden seguir dos alternativas: definir un *array* de tipo `Indice` con un número de elementos que cubra el máximo previsto, o crear una tabla dinámica que se amplíe, cada vez que se llene, en *N* elementos. La tabla dinámica tiene la ventaja de ajustarse al número de artículos y la desventaja de que aumenta el tiempo de proceso ya que cuando la tabla se llena hay que pedir memoria extra para *N* nuevos elementos.

La definición de la tabla con el máximo de elementos:

```
#define MXAR 200
Indice tabla[MXAR];
```

La definición de la tabla dinámica:

```
#define N 15
Indice *tab = (Indice*)malloc(N*sizeof(Indice));
```

Creación

El archivo de datos y el archivo de índices, inicialmente, se abren en modo `wb`. La cuenta de registros se pone a cero. Es importante verificar que se está inicializando la aplicación del archivo indexado; los datos de un archivo que se abre en modo `w` se pierden.

```
FILE *fix;
FILE *findices;
int numReg = 0;
fix = fopen("mitienda.dat", "wb+");
findice = fopen("inxtienda.dat", "wb");
```

Altas

Para añadir registros al archivo de datos, éste se abre (puede ser que previamente se haya abierto) en *modo* lectura/escritura, se realizarán operaciones de lectura para comprobar datos. El proceso sigue estos pasos:

1. Leer el campo clave y el resto de campos del artículo.
2. Comprobar si existe, o no, en la tabla de índices. Se hace una búsqueda binaria de la clave en la tabla.
3. Si existe en la tabla, se lee el registro del archivo de datos según la dirección que se obtiene de la tabla. Puede ocurrir que el artículo, previamente, se hubiera dado de baja, o bien que se quiera reescribir; en cualquier caso se deja elegir al usuario la acción que desee.
4. Si no existe, se graba en el siguiente registro vacío del archivo de datos. A continuación, se inserta ordenadamente en la tabla de índices el nuevo campo clave junto a su dirección en el archivo de datos.

A continuación se escriben las funciones necesarias para realizar esta operación.

```
/*
  Búsqueda binaria de clave. Devuelve la posición; -1 si no está.
*/

int posicionTabla(Indice *tab, int n, const char* cl)
{
  int bajo, alto, central;
  bajo = 0;
  alto = n - 1;                    /* Número de claves en la tabla */
  while (bajo <= alto)
  {
    central = (bajo + alto)/2;
    if (strcmp(cl, tab[central].nombre) == 0)
      return central;              /* encontrado, devuelve posición */
    else if (strcmp(cl, tab[central].nombre) < 0)
```

```
      alto = central -1;        /* ir a sublista inferior */
    else
      bajo = central + 1;       /* ir a sublista superior */
  }
  return -1;
}

/*
  Inserción ordenada. Inserta, manteniendo el orden ascendente, una nueva clave y
  la dirección del registro.
*/
void inserOrden(Indice *tab, int *totreg, Indice nv)
{
  int i, j;
  int parada;
  parada = 0;
  /* busqueda de posición de inserción, es decir,
     el elemento mayor más inmediato */
  i = 0;
  while (!parada && (i < *totreg))
  {
    if (strcmp(nv.nombre , tab[i].nombre) < 0)
      parada = 1;
    else
      i++;
  }
  (*totreg)++;
  if (parada)
  {
    /* mueve elementos desde posición de inserción hasta el final*/
    for (j = *totreg -1; j >= i+1; j--)
    {
      strcpy(tab[j].nombre, tab[ j-1].nombre);
      tab[j].posicion = tab[ j-1].posicion;
    }
  }
     /* se inserta el índice en el "hueco" */
  strcpy(tab[i].nombre, nv.nombre);
  tab[i].posicion = nv.posicion;
}
void leeReg(Articulo* at)
{
  printf(" Nombre del artículo: ");
  gets (at -> nombre);
  printf(" Identificador: ");
  gets(at -> identificador);
  printf(" Precio: ");
  scanf("%lf", &(at -> precio));
  printf(" Unidades: ");
  scanf("%d%*c", &(at -> unidades));
}

#define desplazamiento(n) ((n - 1) * sizeof(Articulo))

void Altas(Indice* tab, int* nreg)
{
  Articulo nuevo;
  int p;
  if (fix == NULL) fix = fopen("mitienda.dat", "rb+");
  leeReg(&nuevo);
  nuevo.estado = 1;             /* estado del registro: alta */
                                /* busca si está en la tabla de índices */
  p = posicionTabla(tab, *nreg, nuevo.nombre);
  if (p == -1)                  /* registro nuevo, se graba en el archivo */
  {
    int dsp;
```

```
    Indice nv;
    nv.posicion = dsp = desplazamiento(*nreg +1);
    strcpy(nv.nombre, nuevo.nombre);
    inserOrden(tab, nreg, nv);            /* inserta e incrementa el número
                                             de registros */
    fseek(fix, dsp, SEEK_SET);
    fwrite(&nuevo, sizeof(Articulo), 1, fix);
  }
  else
    puts("Registro ya dado de alta");
}
```

Bajas

Para dar de baja un registro (en el ejemplo, un artículo) del archivo de datos, simplemente se marca el campo *estado* a cero que indica borrado, y se elimina la entrada de la tabla de índices. El archivo de datos estará abierto en *modo* lectura/escritura. El proceso sigue estos pasos:

1. Leer el campo clave del registro a dar de baja.
2. Comprobar si existe, o no, en la tabla de índices (*búsqueda binaria*).
3. Si existe en la tabla, se lee el registro del archivo de datos según la dirección que se obtiene de la tabla para confirmar la acción.
4. Si el usuario confirma la acción, se escribe el registro en el archivo con la marca *estado* a cero. Además, en la tabla de índices se elimina la entrada del campo clave.

La función de búsqueda ya está escrita, a continuación se escriben el resto de funciones que permiten dar de baja a un registro.

```
/*
  Elimina entrada. Borra una entrada de la tabla de índices moviendo a la
  izquierda los índices, a partir del índice p que se da de baja.
*/

void quitaTabla(Indice *tab, int *totreg, int p)
{
  int j;
  for (j = p; j < *totreg - 1; j++)
  {
    strcpy(tab[j].nombre, tab[j+1].nombre);
    tab[j].posicion = tab[j+1].posicion;
  }
  (*totreg) --;
}
void escribeReg(Articulo a)
{
  printf("Artículo: ");
  printf("%s, %s, precio: %.1f\n", a.nombre, a.identificador, a.precio);
}
void Bajas(Indice* tab, int* nreg)
{
  Articulo q;
  char nomArt[21] ;
  int p;

  if (fix == NULL) fix = fopen("mitienda.dat", "rb+");

  printf("Nombre del artículo: ");
  gets(nomArt);
  p = posicionTabla(tab, *nreg, nomArt);
  if (p != -1)                 /* encontrado en la tabla */
  {
    char r;
    fseek(fix, tab[p].posicion, SEEK_SET);
    fread(&q, sizeof(Articulo), 1, fix);

    escribeReg(q);
```

```
    printf("Pulsa S para confirmar la baja: ");
    scanf("%c%*c", &r);
    if (toupper(r) == 'S')
    {
      q.estado = 0;
      fseek(fix, -sizeof(Articulo), SEEK_CUR);
      fwrite(&q, sizeof(Articulo), 1, fix);
      quitaTabla(tab, nreg, p);
    }
    else
      puts("Acción cancelada. ");
  }
  else
    puts("No se encuentra un registro con ese código.");
}
```

Consultas

La consulta de un registro (un artículo) sigue los pasos:

1. Leer el campo clave (en el desarrollo, el nombre del artículo) del registro que se desea consultar.
2. Buscar en la tabla de índices si existe o no (*búsqueda binaria*).
3. Si existe, se lee el registro del archivo de datos según la dirección que se obtiene de la tabla para mostrar el registro en pantalla.

La operación *modificar,* típica de archivos, sigue los mismos pasos que los expuestos anteriormente. Se debe añadir el paso de escribir el registro que se ha leído, con el campo modificado. A continuación se escribe la función `consulta ( )` que implementa la operación:

```
void Consulta(Indice* tab, int nreg)
{
  Articulo q;
  char nomArt[21];
  int p;

  if (fix == NULL) fix = fopen("mitienda.dat", "rb+");

  printf("Nombre del artículo: ");
  gets(nomArt);
  p = posicionTabla(tab, nreg, nomArt);
  if (p != -1)               /* encontrado en la tabla */
  {
    fseek(fix, tab[p].posicion, SEEK_SET);
    fread(&q, sizeof(Articulo), 1, fix);
    escribeReg(q);
  }
  else
    puts("No se encuentra un registro con ese código.");
}
```

Salida del proceso

Una vez realizadas las operaciones con el archivo de datos, se da por terminada la aplicación cerrando el archivo de datos y, muy importante, guardando la tabla de índices en su archivo. El primer registro de este archivo contiene el número de elementos de la tabla (*nreg*), los siguientes registros son los *nreg* elementos de la tabla. La función `grabaIndice ( )` implementa estas acciones:

```
void grabaIndice(Indice *tab, int nreg)
{
  int i;

  findices = fopen("inxtienda.dat", "wb");

  fwrite(&nreg, sizeof(int), 1, findices);
  for (i = 0; i < nreg; i++)
```

```
      fwrite(tab++, sizeof(Indice), 1, findices);
    fclose (findices);
  }
```

Inicio del proceso

La primera vez que se ejecuta la aplicación se crea el archivo de datos y el de índices. Cada vez que se produce un alta se graba un registro y a la vez se inserta una entrada en la tabla. Cuando se dé por terminada la ejecución se grabará la tabla en el archivo de índices llamando a `grabaIndice( )`. Nuevas ejecuciones han de leer el archivo de índices y escribirlos en la tabla (memoria principal). El primer registro del archivo contiene el número de entradas, el resto del archivo son los índices. Como se grabaron en orden del campo clave, también se leen en orden y entonces la tabla de índices queda ordenada.

```
void recuperaIndice(Indice *tab, int *nreg) {
  int i;

  findices = fopen("inxtienda.dat", "rb");
  fread(nreg, sizeof (int), 1, f indices);
  for (i = 0; i < *nreg; i++)
    fread(tab++, sizeof (Indice), 1, findices);
  fclose(findices);
}
```

> ***Nota de programación***
> No es necesario que el primer registro del archivo de índices sea el número de entradas de la tabla (*`nreg`*) si se cambia el bucle `for` por un bucle *mientras no fin de fichero* y se cuentan los registros leídos. Se ha optado por grabar el número de registros porque de utilizar una tabla de índices dinámica se conoce el número de entradas y se puede reservar memoria para ese número de índices.

Ejercicio 31.4

Escribir la función principal para gestionar el archivo secuencial indexado de artículos de un supermercado. Los campos que describen cada artículo son: nombre del artículo, identificador, precio, unidades.

Únicamente se escribe la codificación de la función `main ( )` con un sencillo menú para que el usuario elija la operación que quiere realizar. El archivo *`articulo.h`* contiene la declaración de las estructuras `Articulo` e `Indice`, la macro `desplazamiento` y la declaración de los punteros a `FILE`, `fix` y `findices`. Los prototipos de las funciones desarrolladas en el anterior apartado se encuentran en el archivo *`indexado.h`*; la implentación de las funciones está en el archivo *`indexado.c`*

```
#include <stdlib.h>
#include <stdio.h>
#include "articulo.h"
#include "indexado.h"

#define MXAR 200
Indice tabla[MXAR];

void escribeTabla(Indice *tab, int nreg);

void main ( )
{
  int numReg;
  char opcion;
```

```
  if ((f ix = fopen("mitienda.dat", "rb+")) == NULL)
  {
    fix = fopen("mitienda.dat", "wb+");
    findices = fopen("inxtienda.dat", "wb");
    numReg = 0;
  }
  else               /* archivos ya existen. Se vuelcan los índices a tabla */
    recuperaIndice (tabla, &numReg);

  escribeTabla(tabla, numReg);
  do
  {
    puts("1. Alta     ");
    puts("2. Baja     ");
    puts("3. Consulta ");
    puts("5. Salir    ");
    do {
        printf("Elige una opción ");
        scanf("%c%*c", &opcion);
    } while (opcion < '1' || opcion > '5' || opcion == '4 ');
    switch (opcion)
    {
      case '1':
        Altas(tabla, &numReg); break;
      case '2':
        Bajas(tabla, &numReg); break;
      case '3':
        Consulta(tabla, numReg); break;
      case '5':
        grabaIndice (tabla, numReg); break;
    }
  } while (opcion != '5');
  if (fix != NULL) fclose(fix);
}
void escribeTabla(Indice *tab, int nreg)
{
  int i = 0;
  if (nreg > 0)
  {
    puts("Tabla de índices actual");
    for (;i < nreg; i++)
      printf ("%s %d\n",tabla[i].nombre,tabla[i].posicion);
    system("Pause");
    system("cls");
  }
}
```

Ordenación de archivos: ordenación externa

Los algoritmos de ordenación estudiados hasta ahora utilizan *arrays* para contener los elementos a ordenar, por lo que es necesario que la memoria interna tenga capacidad suficiente. Para ordenar secuencias grandes de elementos que se encuentran en soporte externo (posiblemente no pueden almacenarse en memoria interna) se aplican los *algoritmos de ordenación externa.*

El tratamiento de archivos secuenciales exige que éstos se encuentren ordenados respecto a un campo del registro, denominado *campo clave K*. El archivo está ordenado respecto a la clave si:

```
i < j → K( i ) < K( j )
```

Los distintos algoritmos de ordenación externa utilizan el esquema general de *separación* y *fusión* o *mezcla*. Por separación se entiende la distribución de secuencias de registros ordenados en varios

archivos; por fusión la mezcla de dos o más secuencias ordenadas en una única secuencia ordenada. Variaciones de este esquema general dan lugar a diferentes algoritmos de ordenación externa.

A recordar

El tiempo de un algoritmo de ordenación de registros de un archivo, *ordenación externa,* depende notablemente del dispositivo de almacenamiento. Los algoritmos utilizan el esquema general de *separación* y *mezcla,* y consideran sólo el tratamiento secuencial.

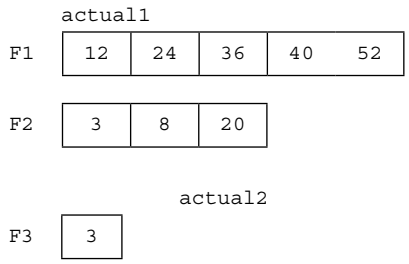
Fusión de archivos

La fusión (mezcla) consiste en juntar en un archivo los registros de dos o más archivos ordenados respecto a un campo clave. El archivo resultante también estará ordenado. `F1` y `F2`, archivos ordenados; `F3`, archivo generado por mezcla.

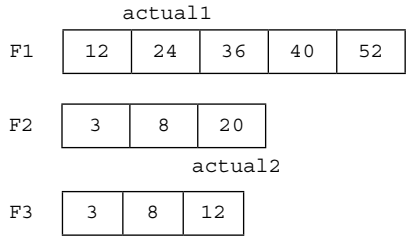
```
F1      12  24  36  37  40  52
F2       3   8   9  20
```

Para realizar la fusión es preciso acceder a los archivos `F1` y `F2`, en cada operación sólo se lee un elemento del archivo dado. Es necesario una variable de trabajo por cada archivo (`actual1, actual2`) para representar el elemento actual de cada archivo.

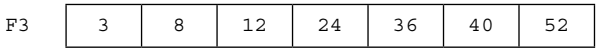
Se comparan las claves `actual1` y `actual2` y se sitúa la más pequeña `3` (`actual2`) en el archivo de salida (`F3`). A continuación, se avanza un elemento el archivo `F2` y se realiza una nueva comparación de los elementos situados en las variables *actual.*



La nueva comparación sitúa la clave más pequeña `8 (actual2)` en `F3`. Se avanza un elemento `(20)` el archivo `F2` y se realiza una nueva comparación. Ahora la clave más pequeña es `12 (actual1)` que se sitúa en `F3`. A continuación, se avanza un elemento el archivo `F1` y se vuelven a comparar las claves actuales.



Cuando uno u otro archivo de entrada ha terminado, se copia el resto del archivo sobre el archivo de salida. El resultado final será:



La codificación correspondiente de fusión de archivos (se supone que el campo clave es de tipo `int`) es:

```
void fusion(FILE* f1, FILE* f2, FILE* f3)
{
  Registro actual1, actual2;

  fread(&actual1, sizeof(Registro), 1, f1);
  fread(&actual2, sizeof(Registro), 1, f2);
while (!feof(f1) && !feof(f2))
  {
```

```
    Registro d;
    if (actual1.clave < actual2.clave)
    {
      d = actual1;
      fread(&actual1, sizeof(Registro), 1, f1);
    }
    else
    {
      d = actual2;
      fread(&actual2, sizeof(Registro), 1, f2);
    }
    fwrite(&d, sizeof(Registro), 1, f3);
  }
  /*
    Lectura terminada de f1 o f2. Se escriben los registros no procesados
  */
while (!feof(f 1))
  {
    fwrite(&actual1, sizeof(Registro), 1, f3);
    fread(&actual1, sizeof(Registro), 1, f1);
  }
while (!feof(f2))
  {
    fwrite(&actual2, sizeof(Registro), 1, f3);
    fread(&actual2, sizeof(Registro), 1, f1);
  }
  fclose (f3);fclose(f1);fclose(f2);
}
```

Antes de llamar a la función se han de abrir los archivos:

```
f3 = fopen("fileDestino", "wb");
f1 = fopen("fileOrigen1", "rb");
f2 = fopen("fileOrigen2", "rb");
if (f1 == NULL || f2 == NULL || f3 == NULL)
{
  puts("Error en los archivos.");
  exit (-1);
}
```

Por último, la llamada a la función:

```
fusion(f1, f2, f3);
```

Clasificación por mezcla directa

El método de ordenación externa más fácil de comprender es el denominado *mezcla directa.* Utiliza el esquema iterativo de *separación* y *mezcla.* Se manejan tres archivos: el archivo original y dos archivos auxiliares.

El proceso consiste en:

1. Separar los registros individuales del archivo original *O* en dos archivos *F1* y *F2.*
2. Mezclar los archivos *F1* y *F2* combinando registros individuales (según sus claves) y formando pares ordenados que son escritos en el archivo *O.*
3. Separar pares de registros del archivo original *O* en dos archivos *F1* y *F2.*
4. Mezclar *F1* y *F2* combinando pares de registros y formando cuádruplos ordenados que son escritos en el archivo *O.*

Cada separación (partición) y mezcla duplica la longitud de las secuencias ordenadas. La primera *pasada* (*separación* + *mezcla*) se hace con secuencias de longitud 1 y la mezcla produce secuencias de longitud 2; la segunda pasada produce secuencias de longitud 4. Cada pasada duplica la longitud de las secuencias; en la pasada *n* la longitud será 2^n. El algoritmo termina cuando la longitud de la secuencia supera el número de registros del archivo a ordenar.

Ejemplo 31.2

Un archivo está formado por registros que tienen un campo clave de tipo entero. Suponiendo que las claves del archivo son:

```
34 23 12 59 73 44 8 19 28 51
```

Se van a realizar los pasos que sigue el algoritmo de mezcla directa para ordenar el archivo. Se considera el archivo `O` *como el original,* `F1` *y* `F2` *archivos auxiliares.*

Pasada 1
Separación:

```
F1:    34    12    73    8    28
F2:    23    59    44    19   51
```

Mezcla formando duplos ordenados:

```
O: 23 34 12 59 44 73 8 19 28 51
```

Pasada 2
Separación:

```
F1:    23    34    44    73    28    51
F2:    12    59     8    19
```

Mezcla formando cuádruplos ordenados:

```
O: 12 23 34 59 8 19 44 73 28 51
```

Pasada 3
Separación:

```
F1:    12    23    34    59    28    51
F2:     8    19    44    73
```

Mezcla formando óctuplos ordenados:

```
O: 8 12 19 23 34 44 59 73 28 51
```

Pasada 4
Separación:

```
F1:     8    12    19    23    34    44    59    73
F2:    28    51
```

Mezcla con la que ya se obtiene el archivo ordenado:

```
O: 8 12 19 23 28 34 44 51 59 73
```

Codificación del algoritmo mezcla directa

La implementación del método se basa, fundamentalmente, en dos rutinas: `distribuir( )` y `mezclar( )`. La primera *separa* secuencias de registros del archivo original en los dos archivos auxiliares. La segunda *mezcla* secuencias de los dos archivos auxiliares y la escribe en el archivo original. Las *pasadas* que da el algoritmo son iteraciones de un bucle *mientras longitud_secuencia menor numero_registros;* cada iteración consiste en llamar a `distribuir( )` y `mezclar( )`.

El número de registros del archivo se determina dividiendo *posición_fin _archivo* por *tamaño_registro*:

```
int numeroReg(FILE* pf)
{
```

```
  if (pf != NULL)
  {
    fpos_t fin;
    fseek(pf, 0L, SEEK_END);
    fgetpos(pf,&fin);
    return fin/sizeof (Registro);
  }
  else
    return 0;
}
```

La implementación que se escribe a continuación supone que los registros se ordenan de acuerdo con un campo clave de tipo `int`:

```
typedef int TipoClave;
typedef struct
{
  TipoClave clave;

} Registro;
void mezclaDirecta(FILE *f)
{
  int longSec;
  int numReg;
  FILE *f1 = NULL, *f2 = NULL;

  f = fopen("fileorg","rb");

  numReg = numeroReg(f);
  longSec = 1;
  while (longSec < numReg)
  {
    distribuir(f, f1, f2, longSec, numReg);
  mezclar(f1, f2, f, &longSec, numReg);
  }
}
void distribuir(FILE* f, FILE* f1, FILE* f2, int lonSec, int numReg)
{
  int numSec, resto, i;

  numSec = numReg/(2*lonSec);
  resto = numReg%(2*lonSec);
  f = fopen("fileorg","rb");
  f1 = fopen("fileAux1","wb");
  f2 = fopen("fileAux2","wb");
  for (i = 1; i <= numSec;   i++)
  {
    subSecuencia(f, f1, lonSec);
    subSecuencia(f, f2, lonSec);
  }
   /*
    Se procesa el resto de registros del archivo
   */
  if (resto > lonSec)
    resto -= lonSec;
  else
  {
    lonSec = resto;
  resto = 0;
  }
  subSecuencia(f, f1, lonSec);
  subSecuencia(f, f2, resto);
  fclose(f1); fclose(f2); fclose(f);
}
void subSecuencia(FILE* f, FILE* t, int longSec)
{
  Registro r;
  int j;
```

```
  for (j = 1; j <= longSec; j++)
  {
    fread(&r, sizeof(Registro), 1, f);
    fwrite(&r, sizeof(Registro), 1, t);
  }
}
void mezclar(FILE* f1, FILE* f2, FILE* f, int* lonSec, int numReg)
{
  int numSec, resto, s, i, j, k, n1, n2;
  Registro r1, r2;

  numSec = numReg/(2*(*lonSec)); /* número de subsecuencias */
  resto = numReg%(2*(*lonSec));

  f = fopen("fileorg","wb");
  f1 = fopen("fileAux1","rb");
  f2 = fopen("fileAux2","rb");

  fread(&r1, sizeof(Registro), 1, f1);
  fread(&r2, sizeof(Registro), 1, f2);
  for (s = 1; s <= numSec+1; s++)
  {
    n1 = n2 = (*lonSec);
    if (s == numSec+1)
    {    /* proceso de los registros de la subsecuencia incompleta */
      if (resto > (*lonSec))
        n2 = resto - (*lonSec);
      else
      {
        n1 = resto;
        n2 = 0;
      }
    }
    i = j = 1;
    while (i <= n1 && j <= n2)
    {
      Registro d;
      if (r1.clave < r2.clave)
      {
        d = r1;
        fread(&r1, sizeof(Registro), 1, f1);
        i++;
      }
      else
      {
        d = r2;
        fread(&r2, sizeof(Registro), 1, f2);
        j++;
      }
      fwrite(&d, sizeof(Registro), 1, f);
    }
    /*
      Los registros no procesados se escriben directamente
    */
    for (k = i; k <= n1; k-I--I-)
    {
      fwrite(&r1, sizeof(Registro), 1, f);
      fread(&r1, sizeof(Registro), 1, f1);
    }
    for (k = j; k <= n2; k-I--I-)
    {
      fwrite(&r2, sizeof(Registro), 1, f);
      fread(&r2, sizeof(Registro), 1, f2);
    }
  }
    (*lonSec) *= 2;
    fclose (f);fclose(f1);fclose(f2);
}
```

Resumen

- Un archivo de datos es un conjunto de datos relacionados entre sí y almacenados en memoria externa. Estos datos se encuentran estructurados en una colección de entidades denominadas registros. La organización de archivos define la forma en la que los registros se disponen sobre el soporte de almacenamiento y puede ser secuencial, directa o secuencial indexada.
- La organización secuencial sitúa los registros unos al lado de otros en el orden en el que van siendo introducidos. Para efectuar el acceso a un determinado registro es necesario pasar por los que le preceden.
- En la organización directa el orden físico de los registros puede no corresponderse con aquel en el que han sido introducidos y el acceso a un determinado registro no obliga a pasar por los que le preceden. La función de librería `fseek ( )` permite situarse directamente en un registro determinado y es la más utilizada para procesar este tipo de archivos.
- Los archivos *hash* son archivos de organización directa, con la particularidad de que el índice de un registro se obtiene transformando un campo del registro (campo clave) en un entero perteneciente a un rango de valores predeterminados. Una función *hash* realiza la transformación. Es necesario establecer una estrategia para tratar colisiones.
- La organización secuencial indexada requiere la existencia de un área de datos y un área de índices. El área de datos está siempre en memoria externa, es el archivo con los registros. Cada registro se corresponde unívocamente con un índice; éste consta de dos elementos: clave del registro y posición del registro en el dispositivo externo. El área de índices, normalmente, está en memoria interna formando una tabla de índices.
- La ordenación de archivos se denomina ordenación externa y requiere algoritmos apropiados. Una manera trivial de realizar la ordenación de un archivo secuencial consiste en copiar los registros a otro archivo de acceso directo, o bien secuencial indexado, usando como clave el campo por el que se desea ordenar.
- Si se desea realizar la ordenación de archivos, utilizando solamente como estructura de almacenamiento auxiliar otros archivos secuenciales de formato similar al que se desea ordenar, hay que trabajar usando el esquema de *separación* y *mezcla.*
- En el caso del algoritmo de *mezcla simple* se opera con tres archivos análogos: el original y dos archivos auxiliares. El proceso consiste en recorrer el archivo original y copiar secuencias de sucesivos registros en, alternativamente, cada uno de los archivos auxiliares. A continuación se mezclan las secuencias de los archivos y se copia la secuencia resultante en el archivo original. El proceso continúa, de tal forma que en cada pasada la longitud de la secuencia es el doble de la longitud de la pasada anterior. Todo empieza con secuencias de longitud 1, y termina cuando se alcanza una secuencia de longitud igual al número de registros.

Ejercicios

31.1. Un archivo secuencial contiene registros con un campo clave de tipo entero en el rango de 0 a 777. Escribir la función volcado( ), que genere un archivo directo de tal forma que el número de registro coincida con el campo clave.

31.2. El archivo secuencial F almacena registros con un campo clave de tipo entero. Supóngase que la secuencia de claves que se encuentra en el archivo es la siguiente:

```
14 27 33 5 8 11 23 44 22 31 46 7 8 11 1 99 23 40 6 11 14 17
```

Aplicando el algoritmo de mezcla directa, realizar la ordenación del archivo y determinar el número de pasadas necesarias.

31.3. Los registros de un archivo secuencial indexado tienen un campo, estado, para conocer si el registro está dado de baja. Escribir una función para compactar el archivo de datos, de tal forma que se eliminen físicamente los registros dados de baja.

31.4. Realizar los cambios necesarios en la función `fusión ( )` que mezcla dos archivo secuenciales ordenados ascendentemente respecto al campo fecha de nacimiento. La fecha de nacimiento está formada por los campos de tipo `int`: mes, día y año.

31.5. Escribir una función que distribuya los registros de un archivo no ordenado, F, en otros dos F1 y F2, con la siguiente estrategia: leer M (por ejemplo, 31) registros a la vez del archivo, ordenarlos utilizando un método de ordenación interna y a continuación escribirlos, alternativamente, en F1 y F2.

31.6. Modificar la implementación de la *mezcla directa* de tal forma que inicialmente se distribuya el fichero origen en secuencias de M registros ordenados, según se explica en el ejercicio 31.5. Y a partir de esa distribución, repetir los pasos del algoritmo *mezcla directa*: fusión de *M-uplas* para dar lugar a *2M* registros ordenados, *separación...*

31.7. Un archivo secuencial F contiene registros y quiere ser ordenado utilizando 4 archivos auxiliares. Suponiendo que la ordenación se desea hacer respecto a un campo de tipo entero, con estos valores:

```
22 11 3 4 11 55 2 98 11 21 4 3 8 12 41 21 42 58 26 19 11 59 37 28 61 72 47
```

aplicar el esquema seguido en el algoritmo de mezcla directa (teniendo en cuenta que se utilizan 4 archivos en vez de 2) y obtener el número de pasadas necesarias para su ordenación.

31.8. Un archivo está ordenado alfabéticamente respecto de un campo clave que es una cadena de caracteres. Diseñar un algoritmo e implementarlo para que la ordenación sea en sentido inverso.

31.9. Un archivo secuencial no ordenado se quiere distribuir en dos ficheros F1 y F2 siguiendo estos pasos:

1. Leer *n* registros del archivo origen y ponerlos en una lista secuencial. Marcar cada registro de la lista con un estatus, por ejemplo *activo = 1*.
2. Obtener el registro *t* con clave más pequeña de los que tienen el estatus *activo* y escribirlo en el archivo destino F1.
3. Sustituir el registro *t* por el siguiente registro del archivo origen. Si el nuevo registro es menor que *t*, se marca como *inactivo,* es decir *activo = 0*; en caso contrario se marca *activo.* Si hay registros en la lista *activos,* volver al paso 2.
4. Cambiar el fichero destino. Si el anterior es F1, ahora será F2 y viceversa. Activar todos los registros de la lista y volver al paso 2.

Problemas

31.1. Los registros que representan a los objetos de una perfumería se van a guardar en un archivo *hash.* Se prevén como máximo 1 024 registros. El campo clave es una cadena de caracteres, cuya máxima longitud es 10. Con este supuesto codificar la función de dispersión y mostrar 10 direcciones dispersas.

31.2. Diseñar un algoritmo e implementar un programa que permita crear un archivo secuencial `PERFUMES` cuyos registros constan de los siguientes campos:

Nombre
Descripción
Precio
Código
Creador

31.3. Realizar un programa que copie el archivo secuencial del ejercicio anterior en un archivo *hash* `PERME_DIR`; el campo clave es el código del perfume que tiene como máximo 10 caracteres alfanuméricos.

31.4. Diseñar un algoritmo e implementar un programa para crear un archivo secuencial indexado denominado `DIRECTORIO`, que contenga los datos de los habitantes de una población que actualmente está formada por 5 590 personas. El campo clave es el número de `DNI` .

31.5. Escribir un programa que liste todas las personas del archivo indexado `DIRECTORIO` que pueden votar.

31.6. Realizar un programa que copie los registros de personas con edad ente 18 y 31 años, del archivo `DIRECTORIO` del ejercicio 31.4, en un archivo secuencial `JOVENES`.

31.7. Se desea ordenar alfabéticamente el archivo `JOVENES` (ejercicio 31.6). Aplicar el método *mezcla directa.*

31.8. Dado un archivo *hash,* diseñar un algoritmo e implementar el código para compactar el archivo después de dar de baja un registro. Es decir, un registro del área de sinónimos se mueve al área principal si el registro del área principal, con el que colisionó el registro del área de sinónimos, fue dado de baja.